# [笔记][Learning Python][2. 类型和运算]

Python

---

# 4. 介绍 `Python` 的对象类型

在 `Python` 中，数据 `data` 以对象的形式存在。
对象实际上就是内存中的一些片段，拥有值和与之对应的一系列操作。
在 `Python` 脚本中，一切皆对象。 `Everything is an object.`
即使是最简单的数字也符合这一特征，它们有值，比如 `99` ，也有支持的操作，比如加减乘除。
本章会粗略过一遍 `Python` 的内置对象类型，后续章节会详细讲解。

---

## 4.1 `Python` 概念层级

具体来讲， `Python` 程序可以分解为模块、语句、表达式、对象：

1. 程序由模块组成
2. 模块包含语句
3. 语句包含表达式
4. 表达式创建和处理对象

> 传统编程书籍通常强调三个支柱 `pillar` ，序列 `sequence` ，选择 `selection` 和重复 `repetition` 。 `Python` 具备这三种类型的工具，另外还有定义 `definition` 工具，用

> 来定义函数和类。
> 但是，在 `Python` 中，更统一的主题是对象 `object` ，和我们可以对它们做什么。

---

# 4.2 为什么要使用内置类型？

- 内置对象让程序更容易编写。
- 内置对象是组成扩展的部件。比如可以通过对列表的管理或者定制来实现堆栈类。
- 内置对象通常比自定义的数据结构更高效。因为内置类型都是用 `C` 实现的。
- 内置对象是语言的一个标准组成部分。

---

# 4.3 `Python` 的核心数据类型

*Table 4-1. Built-in objects preview*

| Object type | Example literals/creation |
|---|---|
| Numbers | `1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()` |
| Strings | `'spam', "Bob's", b'a\x01c', u'sp\xc4m'` |
| Lists | `[1, [2, 'three'], 4.5], list(range(10))` |
| Dictionaries | `{'food': 'spam', 'taste': 'yum'}, dict(hours=10)` |
| Tuples | `(1, 'spam', 4, 'U'), tuple('spam'), namedtuple` |
| Files | `open('eggs.txt'), open(r'C:\ham.bin', 'wb')` |
| Sets | `set('abc'), {'a', 'b', 'c'}` |
| Other core types | Booleans, types, None |
| Program unit types | Functions, modules, classes (Part IV, Part V, Part VI) |
| Implementation-related types | Compiled code, stack tracebacks (Part IV, Part VII) |

列表提供了其他对象的有序集合 `ordered collection` ，而字典按键存储对象，列表和字典都可以嵌套，可以按需增长或缩小，可以包含任意类型的对象。
类似函数，模块和类这样的程序单元 `program units` 在 `Python` 里也是对象。
`Python` 还提供了与实现相关的类型，比如编译好的代码对象，工具制造者对这些会比应用开发者更感兴趣。
然而表 `4-1` 并不完整，因为在 `Python` 程序中，我们处理的所有东西都是对象。
在 `Python` 中没有类型声明，你采用的表达式语法决定了你创造和使用的对象的类型。
`Python` 是动态类型的 `dynamically typed` ，不需要声明类型； `Python` 又是强类型的 `strongly typed` ，意味着你只能对某一对象进行它相应类型的操作。

---

# 4.4 数字

数字包括：
整数 `integer` ：没有小数部分 `fractional part` 的数。
浮点数 `floating-point number` ：有小数点的数。
复数 `complex number` ：有虚部。
固定精度的十进制数 `decimal`
有理数 `rational` ：有分子 `numerator` 和分母 `denominator`
集合 `set`
布尔 `Boolean`
第三方插件提供了更多的类型

它们支持基本的数学运算，比如加 `+` ，乘 `*` ，乘方 `**`（幂运算） `exponentiation` 。

比如：

```
>>> 123 + 222 # Integer addition
345
>>> 1.5 * 4 # Floating-point multiplication
6.0
>>> 2 ** 100 # 2 to the power 100, again
1267650600228229401496703205376
```

在 `Python 3.X` 中，整数类型支持大数运算，但是在 `Python 2.X` 中，有一个单独的长整型 `long integer type` 来处理大数。

在 `Python 2.7` 和 `Python 3.1` 之前，浮点数的运算结果可能会有点奇怪：

```
>>> 3.1415 * 2 # repr: as code (Pythons < 2.7 and 3.1)
6.2830000000000004
>>> print(3.1415 * 2) # str: user-friendly
6.283
```

在 `Python` 里面，每个对象都有两种 `print` 形式。
第一种形式是代码形式的 `repr` （ `as-code` ）。
第二种形式是用户友好的 `str` （ `user-friendly` ）。
如果某个东西看起来很奇怪，就用 `print` 打印它。

但是 `Python 2.7` 和最新的 `Python 3` ，浮点数已经可以智能显示了，不会出现多余的数字：

```
>>> 3.1415 * 2 # repr: as code (Pythons >= 2.7 and 3.1)
6.283
```

除了表达式，还有很多算术模块，**模块**就是我们导入进来使用的额外工具组成的包。

```
>>> import math
```

```
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

`math` 模块包含了很多高级的算术工具， `random` 模块可以产生随机数、进行随机选择。

```
>>> import random
>>> random.random()
0.7082048489415967
>>> random.choice([1, 2, 3, 4])
1
```

---

# 4.5 字符串

字符串属于**序列** `sequence` ：对象的有序集合。
字符串是单个字符组成的序列，其它的序列还有列表 `list` 和元组 `tuple` 。

---

**序列操作**
通过 `len` 获得字符串的长度，通过索引操作 `indexing` 获取元素。

```
>>> S = 'Spam' # Make a 4-character string, and assign it to a name
>>> len(S) # Length
4
>>> S[0] # The first item in S, indexing by zero-based position
'S'
>>> S[1] # The second item from the left
'p'
```

索引是从头部的偏移量，所以从 `0` 开始。

当你给变量赋值的时候就创建了一个变量。
当变量出现在表达式中，会把它的名字替换成它的值。

在 `Python` 中也可以反向索引，正索引是从左到右，负索引是从右到左。

```
>>> S[-1] # The last item from the end in S
'm'
>>> S[-2] # The second-to-last item from the end
'a'
```

实际上，负索引的机制是给负索引加上了序列的长度，例如下面两个是等价的：

```
>>> S[-1] # The last item in S
'm'
>>> S[len(S)-1] # Negative indexing, the hard way
'm'
```

在索引的方括号里面可以放任意表达式。

除了索引操作 `indexing` ，序列还支持切片操作 `slicing` 。

```
>>> S # A 4-character string
'Spam'
>>> S[1:3] # Slice of S from offsets 1 through 2 (not 3)
'pa'
```

`X[I:J]` 从 `X` 中取从 `I` 到 `J` （不含）的所有元素。

切片操作中，左边界默认为 `0` ，右边界默认为序列长度，这带来一些应用变体：

```
>>> S[1:] # Everything past the first (1:len(S))
'pam'
>>> S # S itself hasn't changed
'Spam'
>>> S[0:3] # Everything but the last
'Spa'
>>> S[:3] # Same as S[0:3]
'Spa'
>>> S[:-1] # Everything but the last again, but simpler (0:-1)
'Spa'
>>> S[:] # All of S as a top-level copy (0:len(S))
'Spam'
```

你其实没有必要拷贝字符串，但是这种形式对于列表这样的序列比较有用。

作为序列的一员，字符串还支持用加号 `+` 实现的拼接操作 `concatenation` ，以及用乘号 `*` 实现的重复操作 `repetition` 。

```
>>> S
'Spam'
>>> S + 'xyz' # Concatenation
'Spamxyz'
>>> S # S is unchanged
'Spam'
>>> S * 8 # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

注意到加号 `+` 对于不同的对象展现出了不同的行为：对于数字进行加法运算，对于字符串进行拼接操作。

这是 `Python` 语言多态 `polymorphism` 性质的体现，即**一项操作的意义取决于它作用的对象。**
当学到动态类型 `dynamic typing` 就知道，多态给 `Python` 代码带来了很大的简洁性 `conciseness` 和灵活性 `flexibility` 。
因为对类型不做限制，一个通过 `Python` 实现的操作自动支持很多不同种类的对象，只要它们提供一个兼容接口。

---

**不可变性** `immutability`
每一项字符串操作都产生一个新字符串，原来的字符串不受任何影响，因为在 `Python` 中字符串是不可变的 `immutable` 。
换句话说，你永远无法覆盖 `overwrite` 不可变对象的值。
比如你无法给字符串的某一个位置赋值，但是你总是可以创建一个新的字符串。
因为 `Python` 会清理旧对象，所以并不是像想象的那样没有效率。

```
>>> S
'Spam'
>>> S[0] = 'z' # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment
>>> S = 'z' + S[1:] # But we can run expressions to make new objects
>>> S
'zpam'
```

`Python` 中的每个对象都可以被归为可变对象或者不可变对象中的一种。
在核心类型中，数字、字符串和元组不可变；列表、字典和集合可变。
不可变性可以保证一个对象在整个程序执行的过程中都保持固定不变，而可变对象的值可以随时随地被改变。

你可以通过一些方法改变基于文本的数据，比如将字符串展开成列表，然后用空分隔符连接它们，或者使用 `Python 2.6` 和 `Python 3.0` 及之后版本提供的 `bytearray` 新类型。

```
>>> S = 'shrubbery'
>>> L = list(S) # Expand to a list: [...]
>>> L
['s', 'h', 'r', 'u', 'b', 'b', 'e', 'r', 'y']
>>> L[1] = 'c' # Change it in place
>>> ''.join(L) # Join with empty delimiter
'scrubbery'
>>> B = bytearray(b'spam') # A bytes/list hybrid (ahead)
>>> B.extend(b'eggs') # 'b' needed in 3.X, not 2.X
>>> B # B[i] = ord(c) works here too
bytearray(b'spameggs')
>>> B.decode() # Translate to normal string
'spameggs'
```

字节数组 `bytearray` 支持文本的原地改变，但是仅支持那些最多为八位宽的字符（比如 `ASCII` 字符）。

其它的字符串仍然是不可变的。
字节数组是不可变的字节串和可变的列表的独特的混合类型。
在 `3.X` 中字节串要以 `b'...'` 表示，而在 `2.X` 中则是可选的。

---

**类型特有的方法**
所有的序列操作，对于其它的序列类型，比如列表和元组也适用。
字符串也有属于自己的操作，叫做方法。
方法就是附属于一种特定对象，并可以作用于该类对象的函数，可以通过调用表达式 `call expression` 触发。
比如字符串的 `find` 方法是基本的子串查找操作，查找传入的子串的偏移量并返回，如果找不到返回 `-1` 。
而字符串的 `replace` 方法进行查找和替换操作。

```
>>> S = 'Spam'
>>> S.find('pa') # Find the offset of a substring in S
1
>>> S
'Spam'
>>> S.replace('pa', 'XYZ') # Replace occurrences of a string in S with an
other
'SXYZm'
>>> S
'Spam'
```

注意这些操作并不改变原始字符串，只不过生成了新字符串。

一些其它的字符串方法：

```
>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',') # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']
>>> S = 'spam'
>>> S.upper() # Upper- and lowercase conversions
'SPAM'
>>> S.isalpha() # Content tests: isalpha, isdigit, etc.
True
>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line.rstrip() # Remove whitespace characters on the right side
'aaa,bbb,ccccc,dd'
>>> line.rstrip().split(',') # Combine two operations
['aaa', 'bbb', 'ccccc', 'dd']
```

注意 `line.rstrip().split(',')` ， `Python` 是从左到右运行的。

字符串还支持格式化 `formatting` 操作。

```
>>> '%s, eggs, and %s' % ('spam', 'SPAM!') # Formatting expression (all)
'spam, eggs, and SPAM!'
>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!') # Formatting method (2.6
+, 3.0+)
'spam, eggs, and SPAM!'
>>> '{}, eggs, and {}'.format('spam', 'SPAM!') # Numbers optional (2.7+,
 3.1+)
'spam, eggs, and SPAM!'
```

格式化有很多特性：

```
>>> '{:,.2f}'.format(296999.2567) # Separators, decimal digits
'296,999.26'
>>> '%.2f | %+05d' % (3.14159, -42) # Digits, padding, signs
'3.14 | -0042'
```

序列操作是通用的，方法则不是，尽管方法名一样。
`Python` 的工具箱是分层的：

- 通用操作：对多种类型有效，以内置函数或者表达式的形式出现。比如 `len(X)` ， `X[0]`
- 特定类型的操作：方法调用的形式。比如 `aString.upper()` 。

---

**获得帮助**
使用内置的 `dir` 函数，返回传入对象的所有属性组成的列表。
方法是函数属性，它们也会在这个列表中出现。
`Python 3.3` 中，对于 `S` 字符串有这些属性：

```
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc
__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce
__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__si
zeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'coun
t',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'inde
x',
'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'lju
st',
'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex',
```

```
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'starts
with',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

暂时不需要关心双下划线方法，我们会在类的运算符重载 `operator overloading` 中学到。
下面这两个是等价的，但是不要用 `__add__()` ，因为不直观，而且速度可能更慢：

```
>>> S + 'NI!'
'spamNI!'
>>> S.__add__('NI!')
'spamNI!
```

`dir` 给出了方法的名字，要知道它们是干什么的，把它们传入 `help` 函数：

```
>>> help(S.replace)
Help on built-in function replace:
replace(...)
S.replace(old, new[, count]) -> str
Return a copy of S with all occurrences of substring
old replaced by new. If the optional argument count is
given, only the first count occurrences are replaced.
```

`help` 是 `PyDoc` 的一个接口，用来从对象中提取文档。
`PyDoc` 同样可以渲染 `HTML` 格式的报告。

`dir` 和 `help` 同时接受真实对象，比如我们的字符串 `S` ，或者数据类型 `data type` 的名字，比如 `str` ， `list` ， `dict` 。
对于 `dir` ，两者输出的列表内容是一样的。
对于 `help` ，后者显示完整的类型细节，并可以通过 `类型名.方法名` 的方式来查询，比如 `help(str.replace)` 。

你还可以看 `Python` 的标准库参考手册，或者其它商业印刷书籍。

---

**编写字符串的其它方式**
特殊字符可以用反斜杠 `backslash` 转义序列 `escape sequence` 来表示。

```
>>> S = 'A\nB\tC' # \n is end-of-line, \t is tab
>>> len(S) # Each stands for just one character
5
>>> ord('\n') # \n is a byte with the binary value 10 in ASCII
10
>>> S = 'A\0B\0C' # \0, a binary zero byte, does not terminate string
>>> len(S)
5
>>> S # Non-printables are displayed as \xNN hex escapes
'a\x00B\x00C
```

`Python` 中可以用双引号或者单引号包围字符串，大多数程序员喜欢单引号。
多行字符串用三引号包围，它可以用来做多行注释。

```
>>> msg = """
aaaaaaaaaaaaa
bbb'''bbbbbbbbbb""bbbbbbb'bbbb
cccccccccccccc
"""
>>> msg
'\naaaaaaaaaaaaa\nbbb\'\'\'bbbbbbbbbb""bbbbbbb\'bbbb\ncccccccccccccc\n'
```

`Python` 同样支持原始字符串 `raw string` ，它会关闭反斜杠的转义机制 `backslash escape mechanism` 。这样的字符串以 `r` 开头，用于表示 `Windows` 平台的目录路径很好用，比如 `r'C:\text\new'` 。

---

**`Unicode` 字符串**
在 `Python 3.X` 中，普通的 `str` 字符串就可以处理 `Unicode` 文本（包括 `ASCII` ，它本身就是 `Unicode` 的简化版），`bytes` 字符串类型用来表示原始字节值（包括媒体和编码后的文本）。
为了照顾兼容性， `Python 2.X` 的 `Unicode` 表示法被 `3.X` 所支持，只不过它们被当作普通的 `str` 字符串。

```
>>> 'sp\xc4m' # 3.X: normal str strings are Unicode text
'spÄm'
>>> b'a\x01c' # bytes strings are byte-based data
b'a\x01c'
>>> u'sp\u00c4m' # The 2.X Unicode literal works in 3.3+: just str
'spÄm'
```

在 `2.X` 和 `3.X` 中，非 `Unicode` 字符串是 `8` 位字节 `8-bit bytes` ，而 `Unicode` 字符串是 `Unicode` 代码点的序列，不一定对应单一字节。
其实 `Unicode` 没有字节的概念，有些编码方式的字符代码点一个字节完全装不下。

```
>>> 'spam' # Characters may be 1, 2, or 4 bytes in memory
'spam'
>>> 'spam'.encode('utf8') # Encoded to 4 bytes in UTF-8 in files
b'spam'
>>> 'spam'.encode('utf16') # But encoded to 10 bytes in UTF-16
b'\xff\xfes\x00p\x00a\x00m\x00'
```

`2.X` 和 `3.X` 同样支持字节数组 `bytearray` ，实际上就是字节串 `bytes string` （在 `2.X` 中是 `str` ），字节数组支持大多数列表对象的原地改变操作。

`2.X` 和 `3.X` 也支持非 `ASCII` 字符的三种表示方式：

- `\x` 十六进制数
- `\u` 短 `Unicode`
- `\U` 长 `Unicode`

下面是在 `3.X` 中的三种非 `ASCII` 字符的表示方式：

```
>>> 'sp\xc4\u00c4\U000000c4m'
'spÄÄÄm'
```

在 `2.X` 中：

```
>>> print u'sp\xc4\u00c4\U000000c4m'
'spÄÄÄm'
```

另外 `2.X` 和 `3.X` 都支持在源代码中对文件范围的编码声明。

文本字符串 `text string` ，在 `3.X` 就是普通字符串，在 `2.X` 中是 `unicode` 字符串。
字节字符串 `byte string` ，在 `3.X` 中是字节 `bytes` ，在 `2.X` 中是普通字符串。

`2.X` 允许普通字符串和 `Unicode` 字符串混合在表达式里， `3.X` 没有显式转换，不允许普通字符串和字节字符串混合。

```
u'x' + b'y' # Works in 2.X (where b is optional and ignored)
u'x' + 'y' # Works in 2.X: u'xy'
u'x' + b'y' # Fails in 3.3 (where u is optional and ignored)
u'x' + 'y' # Works in 3.3: 'xy'
'x' + b'y'.decode() # Works in 3.X if decode bytes to str: 'xy'
'x'.encode() + b'y' # Works in 3.X if encode str to bytes: b'xy
```

存储在文件中的时候，文本被编码成字节 `bytes` ，读取到内存中的时候，文本被解码成字符。

---

**模式匹配**
模式匹配需要导入 `re` 模块：

```
>>> import re
>>> match = re.match('Hello[ \t]*(.*)world', 'Hello Python world')
>>> match.group(1)
'Python '
```

再比如：

```
>>> match = re.match('[/:](.*)[/:](.*)[/:](.*)', '/usr/home:lumberjack')
>>> match.groups()
('usr', 'home', 'lumberjack')
```

```
>>> re.split('[/:]', '/usr/home/lumberjack')
['', 'usr', 'home', 'lumberjack']
```

---

# 4.6 列表

列表对象是 `Python` 语言提供的最常见的序列类型。
列表是任意类型对象的有序集合，没有固定的长度。
列表是可变的。

---

**序列操作**
列表支持所有序列操作，就像之前介绍的字符串一样，只不过得到的结果是列表而不是字符串。

```
>>> L = [123, 'spam', 1.23] # A list of three different-type objects
>>> len(L) # Number of items in the list
3
>>> L[0] # Indexing by position
123
>>> L[:-1] # Slicing a list returns a new list
[123, 'spam']
>>> L + [4, 5, 6] # Concat/repeat make new lists too
[123, 'spam', 1.23, 4, 5, 6]
>>> L * 2
[123, 'spam', 1.23, 123, 'spam', 1.23]
>>> L # We're not changing the original list
[123, 'spam', 1.23]
```

---

**特定类型的操作**
`Python` 的列表让人想起别的编程语言中的数组 `array` ，但是它更加强大。
比如它没有类型的限制，也没有固定的长度。

```
>>> L.append('NI') # Growing: add object at end of list
>>> L
[123, 'spam', 1.23, 'NI']
>>> L.pop(2) # Shrinking: delete an item in the middle
1.23
>>> L # "del L[2]" deletes from a list too
[123, 'spam', 'NI']
```

大多数列表的方法都会改变列表，比如 `append` 在末尾添加一项， `pop` 或者 `del` 删除给定偏移量的一项， `insert` 在任意位置插入一项， `remove` 根据值删除一项， `extend` 在表尾追加多

项。

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

---

**边界检查**
列表有边界检查，引用不存在的项或者给不存在的项赋值都会报错。

```
>>> L
[123, 'spam', 'NI']
>>> L[99]
...error text omitted...
IndexError: list index out of range
>>> L[99] = 1
...error text omitted...
IndexError: list assignment index out of range
```

`C` 语言不仅不报错，还静默的增长列表。

---

**嵌套**
最直接的应用就是来表示矩阵，或者多维数组。

```
>>> M = [[1, 2, 3], # A 3 × 3 matrix, as nested lists
[4, 5, 6], # Code can span lines if bracketed
[7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1] # Get row 2
[4, 5, 6]
>>> M[1][2] # Get row 2, then get item 3 within the row
6
```

对于大型应用，最好使用 `NumPy` 和 `SciPy` 。
`NumPy` 被认为把 `Python` 变成了免费的而且更强大的 `Matlab` 系统。

---

**解析** `comprehensions`
除了序列操作和列表方法， `Python` 还提供了更加高级的操作，叫做列表解析 `list comprehension expression` 。
对于类似矩阵的结构很实用，比如取矩阵的一列：

```
>>> col2 = [row[1] for row in M] # Collect the items in column 2
>>> col2
[2, 5, 8]
>>> M # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

列表解析起源于集合的记法。
列表解析由一个表达式和循环架构组成，两者共享一个变量名。
列表解析对于一个序列中的每个元素都运行一次表达式，然后形成一个新的列表。

实际的列表解析可以很复杂：

```
>>> [row[1] + 1 for row in M] # Add 1 to each item in column 2
[3, 6, 9]
>>> [row[1] for row in M if row[1] % 2 == 0] # Filter out odd items
[2, 8]
```

列表解析可以迭代任何可迭代对象 `iterable object` 。

```
>>> diag = [M[i][i] for i in [0, 1, 2]] # Collect a diagonal from matrix
>>> diag
[1, 5, 9]
>>> doubles = [c * 2 for c in 'spam'] # Repeat characters in a string
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

表达式可以用来接受多个值，比如：

```
>>> list(range(4)) # 0..3 (list() required in 3.X)
[0, 1, 2, 3]
>>> list(range(-6, 7, 2)) # -6 to +6 by 2 (need list() in 3.X)
[-6, -4, -2, 0, 2, 4, 6]
>>> [[x ** 2, x ** 3] for x in range(4)] # Multiple values, "if" filters
[[0, 0], [1, 1], [4, 8], [9, 27]]
>>> [[x, x / 2, x * 2] for x in range(-6, 7, 2) if x > 0]
[[2, 1, 4], [4, 2, 8], [6, 3, 12]]
```

列表解析和它的亲戚 `map` 和 `filter` 在这里不做过多介绍。
用括号包围解析式可以创建一个生成器 `generator` 。

```
>>> G = (sum(row) for row in M) # Create a generator of row sums
```

```
>>> next(G) # iter(G) not required here
6
>>> next(G) # Run the iteration protocol next()
15
>>> next(G)
24
```

使用 `map` 内置函数也可以有类似效果：

```
>>> list(map(sum, M)) # Map sum over items in M
[6, 15, 24]
```

注意在 `2.X` 中不用 `list()` 进行格式转换。

在 `2.7` 和 `3.X` 中，解析式语法可以被用来创建集合和字典：

```
>>> {sum(row) for row in M} # Create a set of row sums
{24, 6, 15}
>>> {i : sum(M[i]) for i in range(3)} # Creates key/value table of row su
ms
{0: 6, 1: 15, 2: 24}
```

在 `2.7` 和 `3.X` 中，列表，字典，集合，生成器都可以用解析式来构建：

```
>>> [ord(x) for x in 'spaam'] # List of character ordinals
[115, 112, 97, 97, 109]
>>> {ord(x) for x in 'spaam'} # Sets remove duplicates
{112, 97, 115, 109}
>>> {x: ord(x) for x in 'spaam'} # Dictionary keys are unique
{'p': 112, 'a': 97, 's': 115, 'm': 109}
>>> (ord(x) for x in 'spaam') # Generator of values
<generator object <genexpr> at 0x000000000254DAB0>
```

---

# 4.7 字典

`Python` 的字典根本不是序列，而是映射 `mapping` 。
映射同样是对象的集合，只不过它按照键来存储对象，而不是按照相对位置。
字典是可变的，就像列表，也可以原地改变，长度也可以按需变化。
跟列表一样，是集合的很好的表现工具，但是字典的更容易记忆的键让它更适合于那些有名字或者标签的项，比如数据库记录的字段。

---

**映射操作**
字典被花括号包围，包含一系列的键值对（ `key:value` ）

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

字典的索引操作的语法和序列的一样，只不过方括号里的不是相对位置，而是键。

```
>>> D['food'] # Fetch value of key 'food'
'Spam'
>>> D['quantity'] += 1 # Add 1 to 'quantity' value
>>> D
{'color': 'pink', 'food': 'Spam', 'quantity': 5}
```

在列表中，越界赋值是被禁止的，但是在字典中，对新的键进行赋值会新建它。

```
>>> D = {}
>>> D['name'] = 'Bob' # Create keys by assignment
>>> D['job'] = 'dev'
>>> D['age'] = 40
>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}
>>> print(D['name'])
Bob
```

这里把字典的键用作记录 `record` 的字段名 `field name` 。
有时字典也被用来代替搜索操作，通过键索引字典通常是在 `Python` 中最容易编写的搜索方法。

我们还可以给 `dict` 类型传递关键字参数 `keyword arguments` （函数调用中的一种语法，类似 `name=value` ）来创建字典。
或者用 `zip` 函数把键序列和值序列链接起来。

```
>>> bob1 = dict(name='Bob', job='dev', age=40) # Keywords
>>> bob1
{'age': 40, 'name': 'Bob', 'job': 'dev'}
>>> bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40])) # Zippin
g
>>> bob2
{'job': 'dev', 'name': 'Bob', 'age': 40}
```

> 总结一下，一共有四种创建字典的方式。
> 花括号配合键值对创建。
> 花括号创建空字典，然后用字典的索引操作给键赋值。
> 使用 `dict` 配合关键字参数创建。
> 使用 `zip` 链接键序列和值序列创建。

**嵌套**

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
'jobs': ['dev', 'mgr'],
'age': 40.5}
```

嵌套字典的访问：

```
>>> rec['name'] # 'name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}
>>> rec['name']['last'] # Index the nested dictionary
'Smith'
>>> rec['jobs'] # 'jobs' is a nested list
['dev', 'mgr']
>>> rec['jobs'][-1] # Index the nested list
'mgr'
>>> rec['jobs'].append('janitor') # Expand Bob's job description in place
>>> rec
{'age': 40.5, 'jobs': ['dev', 'mgr', 'janitor'], 'name': {'last':
'Smith',
'first': 'Bob'}}
```

上面展示了 `Python` 核心数据类型的灵活性 `flexibility` 。
在低级语言中设计这样复杂的信息结构很麻烦。
另外我们不需要关心清除对象的空间。
在 `Python` 中，当我们失去了对象的最后一个引用时，该对象所占用的内存空间被自动清理。

```
>>> rec = 0 # Now the object's space is reclaimed
```

`Python` 的这种特性叫做垃圾回收 `garbage collection` 。

两点应用注意：

- 通过 `Python` 的对象持久化系统 `object persistence system` ， `rec` 可以真正的变成为一条数据库的记录。该系统可以把对象和串行字节流 `serial byte stream` 进行相互转换。先记住有 `pickle` 和 `shelve` 这两个模块。
- `JSON` （ `JavaScript Object Notation` ）： `Python` 的 `json` 模块支持创建和解析 `JSON` 文本。更大的用例，见 `MongoDB` 和它的 `Python` 接口 `PyMongo` 。

---

**缺失的键： `if` 测试**
字典还有一些特定的操作，以方法调用的形式存在。
虽然可以给不存在的键赋值来新建项目，但是获取不存在的键仍然会报错：

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> D['e'] = 99 # Assigning new keys grows dictionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}
>>> D['f'] # Referencing a nonexistent key is an error
...error text omitted...
KeyError: 'f'
```

使用字典的 `in` 成员表达式查询一个键是否存在。

```
>>> 'f' in D
False
>>> if not 'f' in D: # Python's sole selection statement
        print('missing')
missing
```

`if` 语句的完整形式还包括 `else` ，用于默认情况，和一个或多个 `elif` 。
它是 `Python` 中的主要选择语句。
还有三元表达式 `if/else` 和 `if` 解析式过滤器。
假如在一个语句块中你有多条命令需要执行，那么就让这些语句缩进一样，这种方法不仅产生了易读的代码，还减少了打字次数。

```
>>> if not 'f' in D:
        print('missing')
        print('no, really...') # Statement blocks are indented
missing
no, really...
```

除了 `in` 测试，我们还有很多方法避免获取不存在的键。
`get` 方法，带有缺省值的条件索引。
`2.X` 的 `has_key` 方法，类似 `in` ，但是在 `3.X` 中不可用。
`try` 语句，可以捕获异常、从异常中恢复。
`if/else` 三元表达式，其实就是把 `if` 语句压缩到一行中去。

```
>>> value = D.get('x', 0) # Index but with a default
>>> value
0
>>> value = D['x'] if 'x' in D else 0 # if/else expression form
>>> value
0
```

**对键排序：`for` 循环**
因为字典是无序的，新建一个字典然后打印，它的键顺序可能会跟创建时的不一样。

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

假如我们必须要对字典排序要怎么办？
常见的做法是：

- 用 `keys` 方法获取一个键列表
- 用列表的 `sort` 方法对键列表排序
- 最后用 `for` 循环遍历

```
>>> Ks = list(D.keys()) # Unordered keys list
>>> Ks # A list in 2.X, "view" in 3.X: use list()
['a', 'c', 'b']
>>> Ks.sort() # Sorted keys list
>>> Ks
['a', 'b', 'c']
>>> for key in Ks: # Iterate though sorted keys
        print(key, '=>', D[key]) # <== press Enter twice here (3.X print)
a => 1
b => 2
c => 3
```

在最新的 `Python` 中，直接用 `sorted` 内置函数可以一步完成。
`sorted` 可以对很多对象类型进行排序并返回：

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

`for` 循环和 `while` 循环是完成重复性任务 `repetitive task` 的主要方式。

```
>>> for c in 'spam':
print(c.upper())
S
P
A
M
```

`Python` 的 `while` 循环是更通用的循环工具，不仅仅用于遍历序列，但是代码量一般更多：

```
>>> x = 4
>>> while x > 0:
        print('spam!' * x)
        x -= 1
spam!spam!spam!spam!
spam!spam!spam!
spam!spam!
spam!
```

`for` 循环不仅仅是序列操作，它还是迭代操作 `iterable operation` 。

---

**迭代和优化**
只要具备下列两点之一，一个对象就是可迭代的：

- 它在内存中是按照序列来存储的
- 在迭代操作中，该对象每次生成一个元素（类似一种虚拟的序列）

这两种对象都是可迭代的，因为它们支持迭代协议 `iteration protocol` ：先用一个对象对 `iter` 做出响应，然后对 `next` 进行响应；当完成生成值的过程之后抛出异常。

> 【?】这句英文其实没看太懂
> More formally, both types of objects are considered iterable because they support the iteration protocol—they respond to the iter call with an object that advances in response to next calls and raises an exception when finished producing values.

生成器解析表达式其实是一个对象，它的值没有一次都存储在内存中，而是按需生成，通常被迭代工具所使用。
`Python` 的文件对象被迭代工具使用时，会按行迭代，文件内容不在一个列表里，而是按需获取。
生成器和文件对象都是可迭代对象。
在 `3.X` 中， `map` 和 `range` 也是可迭代对象。

> 【?】这句也不懂
> Both are iterable objects in Python — a category that expands in 3.X to include core tools like range and map.

每个从左到右扫描对象的 `Python` 工具都使用了迭代协议 `iteration protocol` 。
这就是为什么 `sorted` 方法可以直接用于字典，因为字典是可迭代对象，有一个 `next` 方法，返回接下来的键。

这也是列表解析式总可以被改写成 `for` 循环的等价构造法：

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
```

```
>>> squares
[1, 4, 9, 16, 25]
```

等价于：

```
>>> squares = []
>>> for x in [1, 2, 3, 4, 5]: # This is what a list comprehension does
        squares.append(x ** 2) # Both run the iteration protocol internal
ly
>>> squares
[1, 4, 9, 16, 25]
```

两种方法内部都使用了迭代协议，生成了同样的结果。
然而，函数式编程工具，比如列表解析式、 `filter` 、 `map` 函数，通常比 `for` 循环快，有时候快两倍。但是性能管理在 `Python` 中很困难，每个版本都会发生变化。

在 `Python` 中，简洁性和可读性第一，性能第二。
性能可以在程序正常运行之后，证明有必要提升的时候再关心。
如果你想为了提升性能而调整代码， `Python` 有 `time` 和 `timeit` 模块来测量不同选择的速度，还有 `profile` 模块用来隔离瓶颈 `bottleneck` 。

---

# 4.8 元组

元组（读作 `toople` 或者 `tuhple` ）大致就像一个不可变的列表。
元组也是序列，但是不可变。
它们被圆括号包围，支持任意类型，任意嵌套，和一般的序列操作。

```
>>> T = (1, 2, 3, 4) # A 4-item tuple
>>> len(T) # Length
4
>> T + (5, 6) # Concatenation
(1, 2, 3, 4, 5, 6)
>>> T[0] # Indexing, slicing, and more
1
```

元组同样也有类型特定的方法，但是不如列表的多。

```
>>> T.index(4) # Tuple methods: 4 appears at offset 3
3
>>> T.count(4) # 4 appears once
1
```

元组是不可变序列，一旦创建无法更改。
注意单元素元组需要一个逗号作为尾巴。

```
>>> T[0] = 2 # Tuples are immutable
...error text omitted...
TypeError: 'tuple' object does not support item assignment
>>> T = (2,) + T[1:] # Make a new tuple for a new value
>>> T
(2, 2, 3, 4)
```

正如列表和字典，元组支持混合类型和嵌套，但是长度不可以增减，因为不可变：

```
>>> T = 'spam', 3.0, [11, 22, 33]
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

包围元组的括号通常可以被省略。

---

**为什么用元组？**
元组用的不如列表多，但是它的主要优点就在于不可变性。
元组提供了一种完整性的约束，在写大程序的时候很方便。
后面会介绍元组，和基于元组的一种扩展 `named tuple` 。

> `named tuple` 有翻译成命名元组的，有翻译成具名元组的。

---

# 4.9 文件

可以用内置函数 `open` 来创建一个文件对象。

```
>>> f = open('data.txt', 'w') # Make a new file in output mode ('w' is wr
ite)
>>> f.write('Hello\n') # Write strings of characters to it
6
>>> f.write('world\n') # Return number of items written in Python 3.X
6
>>> f.close() # Close to flush output buffers to disk
```

在当前路径下写入文本，如果要在其它路径下写入，需要给出完整路径。
读取刚才写的文件，用 `r` 模式，这是默认模式，如果调用的时候省略模式也不要紧。
不管文件是什么类型，读取出来都会是一个字符串：

```
>>> f = open('data.txt') # 'r' (read) is the default processing mode
>>> text = f.read() # Read entire file into a string
>>> text
'Hello\nworld\n'
>>> print(text) # print interprets control characters
Hello
world
>>> text.split() # File content is always a string
['Hello', 'world']
```

文件对象还有很多其它的方法，比如 `read` ，接收一个最大读取长度的可选参数； `readline` 一次读一行； `seek` 移动到新文件位置。
然而，最好的读取文件的方法是**不去读它**，文件对象提供一个迭代器，在 `for` 循环中或者其他上下文中可以按行读取：

```
>>> for line in open('data.txt'): print(line)
```

想提前知道文件对象的用法，对任何一个打开的文件对象使用 `dir` 函数，然后用 `help` ：

```
>>> dir(f)
[ ...many names omitted...
'buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno', 'f
lush',
'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readabl
e',
'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writabl
e',
'write', 'writelines']
>>>help(f.seek)
...try it and see...
```

---

**二进制字节文件**
`3.X` 中，文本文件 `text file` 在读写数据的时候会自动进行 `Unicode` 的编码和解码，它的内容就跟普通的 `str` 字符串一样。
而二进制文件 `binary file` 以字节串 `bytes string` 展现，允许你读取原始的文件内容。
`2.X` 也是如此，但是并不强制的施行这种分类，并且它的工具也不太一样。
比如 `Python` 的 `struct` 模块可以创建和解包二进制数据。

```
>>> import struct
>>> packed = struct.pack('>i4sh', 7, b'spam', 8) # Create packed binary d
ata
>>> packed # 10 bytes, not objects or text
b'\x00\x00\x00\x07spam\x00\x08'
>>>
```

```
>>> file = open('data.bin', 'wb') # Open binary output file
>>> file.write(packed) # Write packed binary data
10
>>> file.close()
```

在 `2.X` 中同样可以工作，但是不用写 `b` 前缀，并且 `b` 前缀不会显示（即忽略 `b` 前缀）。

读取二进制数据也是一样。

```
>>> data = open('data.bin', 'rb').read() # Open/read binary data file
>>> data # 10 bytes, unaltered
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8] # Slice bytes in the middle
b'spam'
>>> list(data) # A sequence of 8-bit bytes
[0, 0, 0, 7, 115, 112, 97, 109, 0, 8]
>>> struct.unpack('>i4sh', data) # Unpack into objects again
(7, b'spam', 8)
```

---

**`Unicode` 文本文件**

假如提供了编码名称， `Python` 文本文件会在写入时自动编码 `encode` ，读取时自动解码 `decode` 。

```
>>> S = 'sp\xc4m' # Non-ASCII Unicode text
>>> S
'spÄm'
>>> S[2] # Sequence of characters
'Ä'
>>> file = open('unidata.txt', 'w', encoding='utf-8') # Write/encode UTF-
8 text
>>> file.write(S) # 4 characters written
4
>>> file.close()
>>> text = open('unidata.txt', encoding='utf-8').read() # Read/decode UTF
-8 text
>>> text
'spÄm'
>>> len(text) # 4 chars (code points)
4
```

你也可以用二进制模式打开文件，看看文件中到底存了什么：

```
>>> raw = open('unidata.txt', 'rb').read() # Read raw encoded bytes
>>> raw
b'sp\xc3\x84m'
>>> len(raw) # Really 5 bytes in UTF-8
```

```
5
```

对于不是从文件中获取的数据，你也可以手动编码、解码：

```
>>> text.encode('utf-8') # Manual encode to bytes
b'sp\xc3\x84m'
>>> raw.decode('utf-8') # Manual decode to str
'spÄm'
```

使用不同的编码方式对文本文件进行编码，会产生不同的二进制文件，但是在内存中的字符串都是一样的：

```
>>> text.encode('latin-1') # Bytes differ in others
b'sp\xc4m'
>>> text.encode('utf-16')
b'\xff\xfes\x00p\x00\xc4\x00m\x00'
>>> len(text.encode('latin-1')), len(text.encode('utf-16'))
(4, 10)
>>> b'\xff\xfes\x00p\x00\xc4\x00m\x00'.decode('utf-16') # But same string
 decoded
'spÄm'
```

在 `2.X` 中工作方式类似，但是 `Unicode` 字符串前面有 `u` ，字节串不需要也不显示前导 `b` 。 `Unicode` 文本文件必须用 `codecs.open` 打开，如同 `3.X` 的 `open` 函数一样，它也接收一个编码名，在内存中用特殊的 `unicode` 字符串来显示内容。 二进制文件模式在 `2.X` 是可选的，因为普通文件就是基于二进制的数据。但它要求不能改变换行符。

```
>>> import codecs
>>> codecs.open('unidata.txt', encoding='utf8').read() # 2.X: read/decode
 text
u'sp\xc4m'
>>> open('unidata.txt', 'rb').read() # 2.X: read raw bytes
'sp\xc3\x84m'
>>> open('unidata.txt').read() # 2.X: raw/undecoded too
'sp\xc3\x84m'
```

`Python` 同样支持非 `ASCII` 文件名。

---

**其它类文件对象 `file-like object`**

`Python` 还有其它类文件的工具：

- 管道 `pipe`
- 先进先出队列 `FIFO`
- 套接字 `socket`
- 通过键访问文件 `keyed-access file`

- 对象持久 `persistent object shelves`
- 基于描述符的文件 `descriptor-based files`
- 关系型和面向对象的数据库接口

---

# 4.10 其它核心类型

集合 `set` 是唯一且不可变的对象的无序集合。
使用 `set` 函数创建集合，或者使用 `2.7` 和 `3.X` 的集合语法。
集合就像是没有值的字典，所以用 `{}` 是很自然的。

```
>>> X = set('spam') # Make a set out of a sequence in 2.X and 3.X
>>> Y = {'h', 'a', 'm'} # Make a set with set literals in 3.X and 2.7
>>> X, Y # A tuple of two sets without parentheses
({'m', 'a', 'p', 's'}, {'m', 'a', 'h'})
>>> X & Y # Intersection
{'m', 'a'}
>>> X | Y # Union
{'m', 'h', 'a', 'p', 's'}
>>> X - Y # Difference
{'p', 's'}
>>> X > Y # Superset
False
>>> {n ** 2 for n in [1, 2, 3, 4]} # Set comprehensions in 3.X and 2.7
{16, 1, 4, 9}
```

集合可以用来去重，找不同和无序相等性检测：

```
>>> list(set([1, 2, 1, 3, 1])) # Filtering out duplicates (possibly reord
ered)
[1, 2, 3]
>>> set('spam') - set('ham') # Finding differences in collections
{'p', 's'}
>>> set('spam') == set('asmp') # Order-neutral equality tests (== is Fals
e)
True
```

集合也支持成员检测 `membership test` ：

```
>>> 'p' in set('spam'), 'p' in 'spam', 'ham' in ['eggs', 'spam', 'ham']
(True, True, True)
```

`Python` 加入了新的算术类型：十进制数 `decimal number` ，即固定精度的浮点数。
还有分数 `fraction number` ，是有分子 `numerator` 和分母 `denominator` 的有理数。
它们都可以规避浮点数的限制和内在不准确性：

```
>>> 1 / 3 # Floating-point (add a .0 in Python 2.X)
0.3333333333333333
>>> (2/3) + (1/2)
1.1666666666666665
>>> import decimal # Decimals: fixed precision
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal('4.141')
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33')
>>> from fractions import Fraction # Fractions: numerator+denominator
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)
```

还有布尔 `Boolean` 类型，是实现定义的 `True` 和 `False` 对象，实际上就是 `1` 和 `0` 定制了显示方式。
另外 `Python` 一直支持一个占位符（ `placeholder` ） `None` ，通常用来初始化名字和对象。

```
>>> 1 > 2, 1 < 2 # Booleans
(False, True)
>>> bool('spam') # Object's Boolean value
True
>>> X = None # None placeholder
>>> print(X)
None
>>> L = [None] * 100 # Initialize a list of 100 Nones
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, ...a list of 100 None
s...]
```

---

**如何破坏程序的灵活性**
类型对象（ `type object` ），由内置函数 `type` 返回，它是给出其它对象类型的对象。
在 `3.X` 中它的结果稍有不同，因为 `3.X` 中类型和类完全融合。

```
# In Python 2.X:
>>> type(L) # Types: type of L is list type object
<type 'list'>
>>> type(type(L)) # Even types are objects
<type 'type'>
# In Python 3.X:
>>> type(L) # 3.X: types are classes, and vice versa
```

```
<class 'list'>
>>> type(type(L)) # See Chapter 32 for more on class types
<class 'type'>
```

在 `3.X` 中类型就是类，类就是类型。

检查对象的类型至少有三种方法：

```
>>> if type(L) == type([]): # Type testing, if you must...
        print('yes')
yes
>>> if type(L) == list: # Using the type name
        print('yes')
yes
>>> if isinstance(L, list): # Object-oriented tests
        print('yes')
yes
```

但是这样写几乎总是错的，原因在于这样破坏了代码的灵活性，你让它只能对一种类型起作用，没有这样的类型检测，你的代码可能可以对多种类型生效。
这牵涉到多态 `polymorphism` ，它源自于 `Python` 没有类型声明。
在 `Python` 中，我们编写对象接口（即对象所支持的操作），而不是编写类型。
换句话说，**我们关心一个对象能够做什么，而不关心一个对象是什么。**
不关心特定的类型意味着代码能自动适用于很多类型，只要这个对象有兼容的接口，不论它的具体类型是什么。

> 【!】作者认为多态可能是 `Python` 最核心的概念，用好 `Python` 必须理解多态。

---

**用户定义的类**
`Python` 的面向对象编程 `object-oriented programming` 是可选且强大的特性。
用户定义的对象的新类型扩展了核心类型：

```
>>> class Worker:
        def __init__(self, name, pay): # Initialize when created
            self.name = name # self is the new object
            self.pay = pay
        def lastName(self):
            return self.name.split()[-1] # Split string on blanks
        def giveRaise(self, percent):
            self.pay *= (1.0 + percent) # Update pay in place
```

`name` 和 `pay` 是对象的属性 `attribute` （状态信息 `state information` ）
像函数一样调用类，产生新类型的实例，类方法通过 `self` 参数自动接收该实例。

```
>>> bob = Worker('Bob Smith', 50000) # Make two instances
>>> sue = Worker('Sue Jones', 60000) # Each has name and pay attrs
>>> bob.lastName() # Call method: bob is self
'Smith'
>>> sue.lastName() # sue is the self subject
'Jones'
>>> sue.giveRaise(.10) # Updates sue's pay
>>> sue.pay
66000.0
```

类的继承机制支持软件层级，让软件可以被扩展 `extension` 定制。
我们通过写新类来扩展软件，不改变已经工作的部分。

---

**其它**
其它的非核心对象类型，不是由语法实现，它们要么与程序执行有关（比如类、模块、函数和编译后的代码），要么由导入的模块中的函数实现。
这里的对象和面向对象是两回事，面向对象通常涉及到继承和 `Python` 的 `class` 语句。

---

# 4.11 章节总结

本章介绍了 `Python` 的核心对象类型，和可以应用的操作。

---

# 4.12 检验你的知识

1. 说出四种核心数据类型。
   数字，字符串，列表，字典，元组，集合，文件通常被认为核心对象（数据）类型。
   类型、 `None` 和布尔 `Boolean` 有时候也归为此类。
   有多种数字类型：整数，浮点数，复数，分数，十进制数
   有多种字符串类型：
   - `2.X` 的普通字符串和 `Unicode` 字符串
   - `3.X` 的文本字符串 `text string` 和 字节字符串 `byte string` 。
2. 它们为什么被称作**核心**数据类型？
   核心数据类型与 `Python` 的语法紧密相关，而创建其它对象通常要使用导入模块里的函数。
3. 什么是不可变 `immutable` ？哪三种核心数据类型是不可变的？
   不可变指的是一个对象一旦被创建就无法被改变。
   数字，字符串和元组是不可变的。
   虽然你不能原地改变它们，但是你可以新建一个。
   字节数组 `bytearray` 给文本提供了可变性，但是它们不是普通的字符串，而且只适用于简单的 `8-bit` 文本（比如 `ASCII` ）。

4. 序列 `sequence` 是什么意思？哪三种数据类型是序列？
   序列是对象的有序集合。
   字符串，列表和元组是序列。
   它们共享序列操作，比如索引，拼接和切片，但是同样有特定类型的方法调用。
   有一个相关术语叫做**可迭代** `iterable` ，意味着该对象要么是一个物理上的序列，要么是一个可以按需生成元素的虚拟序列。
5. 映射 `mapping` 是什么意思？哪种核心数据类型是映射？
   映射表示一个把键和值关联起来的对象。
   字典是映射。
   映射没有从左到右的顺序，支持按键存取数据，另外还有类型特定的方法调用。
6. 多态 `polymorphism` 是什么？你为什么要关注它？
   多态意味着操作的意义取决于操作的作用对象。
   举个例子，当加法这一操作的作用对象是数字时，就进行数学运算；当加法的作用对象是序列（比如列表）时，就进行序列的拼接。
   这是 `Python` 的关键概念，具有多态性质的代码不被特定类型所限制，它可以自动适应于很多类型。

---

# 5. 数字类型

在 `Python` 中，数据以对象的形式存在。
对象是所有 `Python` 程序的基础。

---

## 5.1 数字类型基础

在 `Python` 中，数字不是单一的对象类型，而是一类相似的类型。
数字类型包括：

- 整数和浮点数对象
- 复数对象
- 十进制：固定精度对象
- 分数：有理数对象
- 集合：带有数学运算的集合
- 布尔：真和假
- 内置函数和模块： `round` ， `math` ， `random` 等等
- 第三方扩展： 向量 `vector` ，库，可视化，绘图等等

集合 `set` 同时具有数字 `numeric` 类型和集合 `collection` 类型的性质，但是更多的被认为是前者。

---

**数字类型的写法**
`integer` 整数可以有无限精度，只要内存允许

`floating-point numbers` 浮点数，有时候简称 `floats` ，有小数部分的数
十六进制数 `hexadecimal` ，八进制数 `octal` ，二进制数 `binary`
复数

*Table 5-1. Numeric literals and constructors*

| Literal | Interpretation |
| --- | --- |
| 1234, -24, 0, 99999999999999 | Integers (unlimited size) |
| 1.23, 1., 3.14e-10, 4E210, 4.0e+210 | Floating-point numbers |
| 0o177, 0x9ff, 0b101010 | Octal, hex, and binary literals in 3.X |
| 0177, 0o177, 0x9ff, 0b101010 | Octal, octal, hex, and binary literals in 2.X |
| 3+4j, 3.0+4.0j, 3J | Complex number literals |
| set('spam'), {1, 2, 3, 4} | Sets: 2.X and 3.X construction forms |
| Decimal('1.0'), Fraction(1, 3) | Decimal and fraction extension types |
| bool(X), True, False | Boolean type and constants |

整数和浮点数：浮点数还有指数形式，用 `e` 或者 `E` 表示。在 `CPython` 中，浮点数作为 `C` 语言的 `double` 双精度浮点数来实现，因此精度与用来编译 `Python` 解释器的 `C` 编译器提供的精度一样。

`2.X` 中的整数：分为普通整数 `normal` 和长整数 `long` 。普通整数通常是 `32` 位。整数可以以 `l` 或者 `L` 结尾，强制转换为长整数。当整数溢出的时候，会自动升级成长整数，所以不用手动添加 `L` 。

`3.X` 中的整数：只有一种类型。普通整数和长整数合二为一了。所以不能在数字末尾加 `L` 。
十六进制，八进制和二进制写法：十六进制前导是 `0x` 或者 `0X` ，后面接十六进制数。大小写都行。八进制前导是 `0o` 或者 `0O` ，在 `2.X` 中，单独加一个零 `0` 也可以，但是 `3.X` 不支持。而 `0o` 表示法在 `2.6` 中可以使用。 `2.6` 和 `3.0` 支持二进制写法，用前导 `0b` 和 `0B` 来表示。
还有内置函数 `hex(I)` 、 `oct(I)` 和 `bin(I)` 把整数转换成相应的进制数的字符串。
还有 `int(str, base)` 可以把字符串转换成相应的进制数。比如 `int('0x64', 16)` 得到 `100` 。

复数：复数写作**实部 + 虚部**，其中虚部以 `j` 或者 `J` 结尾。复数在 `Python` 内部是作为一对浮点数实现的。不过支持复数运算。还可以通过 `complex(real, imag)` 来创建。

其它数字类型：之后介绍。

---

**内置数字工具**
`Python` 提供了处理数字对象的一系列工具：

- 表达式运算符： `+` ， `-` ， `*` ， `/` ， `>>` ， `**` ， `&` 等
- 内置数学函数： `pow` ， `abs` ， `round` ， `int` ， `hex` ， `bin` 等
- 实用模块： `random` ， `math` 等

- 类型特定的方法：比如 `as_integer_ratio` 方法和 `is_integer` 方法还有 `3.1` 新引入的 `bit_length` 方法（表示一个对象需要多少位）。

另外，集合也支持表达式和方法。

---

**`Python` 表达式运算符**
处理数字最基本的工具是表达式。
表达式：数字或者其它对象与运算符的组合，在 `Python` 执行的时候会对其求值。

`is` 检测对象身份（即内存中的地址，更严格的相等性测试）
`lambda` 创建匿名函数

*Table 5-2. Python expression operators and precedence*

| Operators | Description |
|---|---|
| `yield x` | Generator function send protocol |
| `lambda args: expression` | Anonymous function generation |
| `x if y else z` | Ternary selection (x is evaluated only if y is true) |
| `x or y` | Logical OR (y is evaluated only if x is false) |
| `x and y` | Logical AND (y is evaluated only if x is true) |
| `not x` | Logical negation |
| `x in y, x not in y` | Membership (iterables, sets) |
| `x is y, x is not y` | Object identity tests |
| `x < y, x <= y, x > y, x >= y` | Magnitude comparison, set subset and superset; |
| `x == y, x != y` | Value equality operators |
| `x \| y` | Bitwise OR, set union |
| `x ^ y` | Bitwise XOR, set symmetric difference |
| `x & y` | Bitwise AND, set intersection |
| `x << y, x >> y` | Shift x left or right by y bits |
| `x + y` | Addition, concatenation; |
| `x - y` | Subtraction, set difference |
| `x * y` | Multiplication, repetition; |
| `x % y` | Remainder, format; |
| `x / y, x // y` | Division: true and floor |

| | |
|---|---|
| -x, +x | Negation, identity |
| ~x | Bitwise NOT (inversion) |
| x ** y | Power (exponentiation) |
| x[i] | Indexing (sequence, mapping, others) |
| x[i:j:k] | Slicing |
| x(...) | Call (function, method, class, other callable) |
| x.attr | Attribute reference |
| (...) | Tuple, expression, generator expression |
| [...] | List, list comprehension |
| {...} | Dictionary, set, set and dictionary comprehensions |

下面是关于版本的一些说明：

- `2.X` 可以使用 `!=` 和 `<>` 表示不等于，而 `3.X` 只能使用 `!=`
- `2.X` 中，反引号 `X` 等同于 `repr(X)` ，把对象转换为字符串。 `3.X` 中取消了这个用法，使用 `str` 和 `repr` 内置函数代替。
- `X // Y` 在 `2.X` 和 `3.X` 是地板除。而 `X / Y` 在 `3.X` 中是真除法，在 `2.X` 中是经典除法（截断除法）。
- `yield` 和 三元 `if/else` 选择表达式在 `2.5` 以后适用。前者返回生成器中的 `send(...)` 参数。后者是多行 `if` 语句的缩写形式。 `yield` 如果不是单独的位于赋值语句的右边的话，需要加圆括号。
- 比较运算符可以连接使用：比如 `X < Y < Z` 。
- 最近的 `Python` 版本中，切片表达式 `X[I:J:K]` 等价于 `X[slice(I, J, K)]`
- `3.X` 中不支持混合类型的比较，也不支持字典的比较，可以用 `sorted(aDict.items())` 代替。

**运算符优先级 `operator precedence`**
表 `5-2` 按照优先级排序，越往下的运算符优先级越高，同行的运算符按照从左到右结合（除了幂运算是从右到左结合的），比较运算符从左到右连接。

**括号为子表达式分组**
当使用小括号的时候，就覆盖了 `Python` 的优先级规则， `Python` 总是会先对括号内的表达式求值。
在长表达式中使用括号是有好处的：

- 按照你规定的顺序求值
- 提高可读性

**混合类型会被升级**
除了在表达式中混合操作符，你还可以混合数字类型。
在混合类型的数字表达式中， `Python` 会先把操作数 `operand` 升级成最复杂的类型，然后对两个同类型的操作数进行运算。这种行为类似于 `C` 语言中的类型转换。
数字类型的复杂度：整数 < 浮点数 < 复数。

```
>>> 40 + 3.14 # Integer to float, float math/result
43.14
```

在 `2.X` 中，如果整数太长会自动转换成长整数，而 `3.X` 中的整数类型已经包括了长整数类型。

你还可以进行强制类型转换：

```
>>> int(3.1415) # Truncates float to integer
3
>>> float(3) # Converts integer to float
3.0
```

通常你不用这么做， `Python` 会自动完成这些工作。
另外， `Python` 一般不在其它类型之间自动转换，仅在数字类型中转换。
比如说把字符串加到整数类型上会报错。

> `2.X` 中，非数字类型的混合类型可以进行比较：比较对象类型名字的字符串。
> `3.X` 中不允许比较非数字的混合类型，会抛出异常。

---

**预告：运算符重载和多态**
所有 `Python` 的运算符都可以通过 `Python` 类和 `C` 扩展被重载（实现），可以作用于你创建的对象。
比如你会看到可以用加法连接的类，可以使用索引操作的类等等。

`Python` 自己会自动重载一些运算符，这样就可以根据内置对象的类型的不同采取不同的动作。比如 `+` 加号应用于数字就执行加法，而应用于序列对象则执行拼接。

这个特性被称作**多态** `polymorphism` ：一项操作的意义却决于它的作用对象的类型。

---

# 5.2 实际应用中的数字

**变量和基本表达式**
变量就是名字，用来记录 `keep track of` 程序中的信息。
在 `Python` 中：

- 变量在第一次赋值时创建。
- 在表达式中，变量被它们的值所替换。
- 变量必须先赋值，才能使用在表达式中。
- 变量指向对象，不用事先声明。

```
% python
>>> a = 3 # Name created: not declared ahead of time
```

```
>>> b = 4
```

**注释 `comment`** ：在 `#` 后面一直到行末的文本是注释，会被 `Python` 忽略。

```
>>> a + 1, a - 1 # Addition (3 + 1), subtraction (3 - 1)
(4, 2)
>>> b * 3, b / 2 # Multiplication (4 * 3), division (4 / 2)
(12, 2.0)
>>> a % 2, b ** 2 # Modulus (remainder), power (4 ** 2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b # Mixed-type conversions
(6.0, 16.0)
```

这里返回的结果实际上是有两个值的**元组**，因为交互提示模式中的两个表达式被逗号隔开了。
使用没有被赋值的变量会报错：

```
>>> c * 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

在 `Python` 中，不用预先声明变量，但是使用前必须至少被赋值一次。

```
>>> b / 2 + a # Same as ((4 / 2) + 3) [use 2.0 in 2.X]
5.0
>>> b / (2.0 + a) # Same as (4 / (2.0 + 3)) [use print before 2.7]
0.8
```

如果你想要在 `3.X` 中使用整数除法 `integer division` ，那么要用 `//` 。
如果你想要在 `2.X` 中使用真除法，用 `.0` 表示整数，比如上面的 `b / 2.0 + a` 。

---

**数字显示格式**
在 `2.6` ， `3.0` 及以前版本：

```
>>> b / (2.0 + a) # Pythons <= 2.6: echoes give more (or fewer) digits
0.80000000000000004
>>> print(b / (2.0 + a)) # But print rounds off digits
0.8
```

这是因为硬件限制带来的浮点数精度问题，在有限的比特位数之内，无法精确的展现某些数值。
其实你的硬件和 `Python` 都没有出错，这是一个显示的问题。
交互提示模式的自动结果回显和 `print` 语句采用了不同的算法，在内存中都是一样的数字。

使用 `print` 你会得到用户友好的展示。
在 `2.7` 和 `3.1` 中，浮点数通常显示的位数比较小，智能多了。

**`str` 和 `repr` 显示格式**
默认的交互式回显和 `print` 语句的区别对应于内置函数 `repr` 和 `str` 函数的区别。

```
>>> repr('spam') # Used by echoes: as-code form
"'spam'"
>>> str('spam') # Used by print: user-friendly form
'spam'
```

这两个函数能把任意对象转换为它们的字符串形式：
`repr` 以及默认的交互式回显生成类代码的结果；
`str` 以及 `print` 操作返回更加用户友好的格式。
有些对象两者都有， `str` 用作一般用途， `repr` 提供额外细节。

`str` 内置函数同样是字符串数据类型的名字，在 `3.X` 中可以传入编码名来对字节串进行解码，比如 `str(b'xy', 'utf8')` 。
`str` 还等价于 `bytes.decode` 方法。

不是所有数字都有很多位数：

```
>>> 1 / 2.0
0.5
```

除了 `print` 和自动回显，还有更多显示数字的方法：

```
>>> num = 1 / 3.0
>>> num # Auto-echoes
0.3333333333333333
>>> print(num) # Print explicitly
0.3333333333333333
>>> '%e' % num # String formatting expression
'3.333333e-01'
>>> '%4.2f' % num # Alternative floating-point format
'0.33'
>>> '{0:4.2f}'.format(num) # String formatting method: Python 2.6, 3.0, a
nd later
'0.33
```

后面三种方法用到了字符串格式化 `string formatting` ，在第 `7` 章学字符串的时候会学到。

---

**普通比较和链式比较**
比较操作返回一个布尔值：

```
>>> 1 < 2 # Less than
True
>>> 2.0 >= 1 # Greater than or equal: mixed-type 1 converted to 1.0
True
>>> 2.0 == 2.0 # Equal value
True
>>> 2.0 != 2.0 # Not equal value
False
```

注意只有在数字表达式中，混合类型是被允许的。

`Python` 中允许链式比较，做范围测试 `range test` 很方便。
`A < B < C` 等价于 `A < B and B < C` 。

```
>>> X = 2
>>> Y = 4
>>> Z = 6
>>> X < Y < Z # Chained comparisons: range tests
True
>>> X < Y and Y < Z
True
```

链式比较打的字少，而且速度稍微快一点，因为 `Y` 只求值了一次。
链式比较的长度可以是任意多个：

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False
>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

你还可以使用其它的比较符号，但是可能不会很直观：

```
>>> 1 == 2 < 3 # Same as: 1 == 2 and 2 < 3
False # Not same as: False < 3 (which means 0 < 3, which is true!)
```

浮点数的比较有时候可能会出乎你的意料：

```
>>> 1.1 + 2.2 == 3.3 # Shouldn't this be True?...
False
>>> 1.1 + 2.2 # Close to 3.3, but not exactly: limited precision
3.3000000000000003
```

```
>>> int(1.1 + 2.2) == int(3.3) # OK if convert: see also round, floor, tr
unc ahead
True # Decimals and fractions (ahead) may help here too
```

由于比特位数的限制，浮点数无法精确的表示某些值。
这一问题在数值程序中普遍存在，并不是 `Python` 所独有的。
后面我们会学到固定精度的十进制数 `decimal` 和分数 `fraction` ，它们可以处理这样的问题。

---

**除法：经典除法，地板除法和真除法**
`3.X` 和 `2.X` 的除法不太一样。
总共有三种风格的除法（经典除法、真除法、地板除法），两种不同的操作符。

- `X / Y`
  `2.X` 中的经典除法：经典除法对于整数是地板除，比如 `3 / 2 = 1` ， `-3 / 2 = -2` 。对于浮点数是真除法（保留小数部分）。
  `3.X` 中的真除法。
- `X // Y`
  `2.X` 和 `3.X` 中的地板除。

`3.X` 中引入真除法主要是因为经典除法的结果取决于操作数的类型，而 `Python` 又是动态类型的语言，导致结果难以预料。

即 `/` 和 `//` 对应 `3.X` 的真除法和地板除法，对应 `2.X` 的经典除法和地板除法，但是你可以启用真除法。

```
C:\code> C:\Python33\python
>>>
>>> 10 / 4 # Differs in 3.X: keeps remainder
2.5
>>> 10 / 4.0 # Same in 3.X: keeps remainder
2.5
>>> 10 // 4 # Same in 3.X: truncates remainder
2
>>> 10 // 4.0 # Same in 3.X: truncates to floor
2.0

C:\code> C:\Python27\python
>>>
>>> 10 / 4 # This might break on porting to 3.X!
2
>>> 10 / 4.0
2.5
>>> 10 // 4 # Use this in 2.X if truncation needed
2
>>> 10 // 4.0
2.0
```

注意地板除法 `//` 的返回值依赖于操作数的数据类型。

---

**支持两个 `Python` 版本**
如果你的程序需要地板除法，在 `2.X` 和 `3.X` 中都使用 `//` 。
如果你的程序对于整数需要带余数的浮点数结果，使用 `float` 进行强制类型转换，保证一个操作数是浮点数。
比如在 `2.X` 中可以这么做：

```
X = Y // Z # Always truncates, always an int result for ints in 2.X and
 3.X
X = Y / float(Z) # Guarantees float division with remainder in either 2.X
 or 3.X
```

或者，你可以在 `2.X` 版本中启用 `3.X` 的 `/` 除法选项：

```
C:\code> C:\Python27\python
>>> from __future__ import division # Enable 3.X "/" behavior
>>> 10 / 4
2.5
>>> 10 // 4 # Integer // is the same in both
2
```

> 【!】其实 `2.X` 和 `3.X` 就是除法 `/` 发生了改变，地板除法 `//` 的特性根本没变。
> 除法 `/` 在 `2.X` 是经典除法，在 `3.X` 是真除法。

在交互式中键入的这条特殊的 `from` 语句在整个会话期间有效，如果写到脚本中， `from` 语句要放在**第一个可执行**。

---

**地板除法和截断除法**
`//` 被不正式的称作截断除法 `truncating division` ，准确来说它应该是地板除法 `floor division` 。
地板除法：返回小于真实结果的最近整数。

```
>>> import math
>>> math.floor(2.5) # Closest number below value
2
>>> math.floor(-2.5)
-3
>>> math.trunc(2.5) # Truncate fractional part (toward zero)
2
>>> math.trunc(-2.5)
-2
```

`//` 运算符对于正数是截断除法，对于负数是地板除法。

```
C:\code> c:\python33\python
>>> 5 / 2, 5 / -2
(2.5, -2.5)
>>> 5 // 2, 5 // -2 # Truncates to floor: rounds to first lower integer
(2, -3) # 2.5 becomes 2, -2.5 becomes -3
>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)
>>> 5 // 2.0, 5 // -2.0 # Ditto for floats, though result is float too
(2.0, -3.0)
```

在 `2.X` 中类似：

```
C:code> c:\python27\python
>>> 5 / 2, 5 / -2 # Differs in 3.X
(2, -3)
>>> 5 // 2, 5 // -2 # This and the rest are the same in 2.X and 3.X
(2, -3)
>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)
>>> 5 // 2.0, 5 // -2.0
(2.0, -3.0)
```

【!】直接理解成 `/` 和 `//` 这两个运算符，不管在 `2.X` 还是 `3.X` 中，都没有截断除法的意思，截断除法只能使用 `math.trunc` 函数。

```
C:\code> c:\python33\python
>>> import math
>>> 5 / -2 # Keep remainder
-2.5
>>> 5 // -2 # Floor below result
-3
>>> math.trunc(5 / -2) # Truncate instead of floor (same as int())
-2
C:\code> c:\python27\python
>>> import math
>>> 5 / float(-2) # Remainder in 2.X
-2.5
>>> 5 / -2, 5 // -2 # Floor in 2.X
(-3, -3)
>>> math.trunc(5 / float(-2)) # Truncate in 2.X
-2
```

对于 `3.X` ，使用除法时可以参考：

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2) # 3.X true division
```

```
(2.5, 2.5, -2.5, -2.5)
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2) # 3.X floor division
(2, 2.0, -3.0, -3)
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0) # Both
(3.0, 3.0, 3, 3.0)
```

对于 `2.X` ，使用除法时可以参考：

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2) # 2.X classic division (diff
ers)
(2, 2.5, -2.5, -3)
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2) # 2.X floor division (sa
me)
(2, 2.0, -3.0, -3)
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0) # Both
(3, 3.0, 3, 3.0)
```

---

**整数精度**
`3.X` 的整数支持无限长度：

```
>>> 99999999999999999999999999999 + 1 # 3.X
1000000000000000000000000000000
```

`2.X` 对长整数有一个单独的类型，但是你不用手动加上 `L` ，一切都是自动的：

```
>>> 99999999999999999999999999999 + 1 # 2.X
1000000000000000000000000000000L
```

无限精度的整数是非常方便的内置工具。
在 `2.X` 和 `3.X` 中：

```
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

因为 `Python` 需要做额外的工作来支持更多的精度，所以当数字很大的时候，整数运算会显著变慢。但是它带来的便携性超出了性能的降低。

---

**复数**
复数被表示为两个浮点数，即实部和虚部，虚部要加上 `j` 或者 `J` 。

我们同样可以用 `+` 加号连接来构建有非零实部的复数，比如 `2 + -3j` 。

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
```

可以用属性的方式提取实部和虚部，还可以用 `cmath` 模块里的工具来处理。
`cmath` 模块是 `math` 数学模块的复数版本。

---

**十六进制，八进制，二进制：写法和转换**
注意在内存中，整数的值都是一样的，不管我们给它们指定什么样的进制。

```
>>> 0o1, 0o20, 0o377 # Octal literals: base 8, digits 0-7 (3.X, 2.6+)
(1, 16, 255)
>>> 0x01, 0x10, 0xFF # Hex literals: base 16, digits 0-9/A-F (3.X, 2.X)
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111 # Binary literals: base 2, digits 0-1 (3.X,
 2.6+)
(1, 16, 255)
```

八进制值 `0o377` ，十六进制值 `0xFF` 和二进制值 `0b11111111` 都是十进制的 `255` 。

各种进制转换成十进制：

```
>>> 0xFF, (15 * (16 ** 1)) + (15 * (16 ** 0)) # How hex/binary map to dec
imal
(255, 255)
>>> 0x2F, (2 * (16 ** 1)) + (15 * (16 ** 0))
(47, 47)
>>> 0xF, 0b1111, (1*(2**3) + 1*(2**2) + 1*(2**1) + 1*(2**0))
(15, 15, 15)
```

`Python` 的 `print` 语句默认以十进制的形式打印整数值。
但是提供了内置函数，可以让你把整数转换为其它进制的字符串：

```
>>> oct(64), hex(64), bin(64) # Numbers=>digit strings
('0o100', '0x40', '0b1000000')
```

`oct` 把整数转换为八进制字符串， `hex` 把整数转换为十六进制字符串， `bin` 把整数转换为二进制字符串。

反过来， `int` 可以把数字字符串转换为整数，第二个可选参数可以制定按照什么进制转换。

```
>>> 64, 0o100, 0x40, 0b1000000 # Digits=>numbers in scripts and strings
(64, 64, 64, 64)
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)
>>> int('0x40', 16), int('0b1000000', 2) # Literal forms supported too
(64, 64)
```

`eval` 函数，把字符串当成 `Python` 代码运行。
但是这个函数速度比较慢，因为它实际上把字符串当作程序编译并运行，并假定该字符串是安全的。小心一个聪明的用户可以提交一串可以删除你的文件的字符串。

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

你还可以用字符串格式化方法来把整数转换成相应的非十进制数。

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64) # Numbers=>digits, 2.6+
'100, 40, 1000000'
>>> '%o, %x, %x, %X' % (64, 64, 255, 255) # Similar, in all Pythons
'100, 40, ff, FF'
```

`2.X` 的用户记住，可以仅用一个前导零来编写八进制数：

```
>>> 0o1, 0o20, 0o377 # New octal format in 2.6+ (same as 3.X)
(1, 16, 255)
>>> 01, 020, 0377 # Old octal literals in all 2.X (error in 3.X)
(1, 16, 255)
```

但是在 `3.X` 中这样是错误的，为了向后兼容， `2.6` 和 `2.7` 中最好使用 `0o...` 来表示八进制数。

注意这些写法支持任意长的整数。

```
>>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF
>>> X
5192296858534827628530496329220095
>>> oct(X)
'0o1777777777777777777777777777777777777777'
>>> bin(X)
'0b1111111111111111111111111111111111111111111111111111111111111111 ...and so o
n... 11111'
```

以上是在 `3.X` 中的结果，在 `2.X` 中，八进制数没有 `o` ，八进制和十进制数末尾有 `L` 。

**位运算**

```
>>> x = 1 # 1 decimal is 0001 in bits
>>> x << 2 # Shift left 2 bits: 0100
4
>>> x | 2 # Bitwise OR (either bit=1): 0011
3
>>> x & 1 # Bitwise AND (both bits=1): 0001
1
```

这样的位掩码操作 `bitmasking operation` 让我们可以用单独一个整数来编码、提取标志位和其它的值。

在 `3.0` 和 `2.6` 中，我们可以用比特字符串来编写和查看数字：

```
>>> X = 0b0001 # Binary literals
>>> X << 2 # Shift left
4
>>> bin(X << 2) # Binary digits string
'0b100'
>>> bin(X | 0b010) # Bitwise OR: either
'0b11'
>>> bin(X & 0b1) # Bitwise AND: both
'0b1'
```

对于十六进制数，或者经过进制转换的数也同样适用：

```
>>> X = 0xFF # Hex literals
>>> bin(X)
'0b11111111'
>>> X ^ 0b10101010 # Bitwise XOR: either but not both
85
>>> bin(X ^ 0b10101010)
'0b1010101'
>>> int('01010101', 2) # Digits=>number: string to int per base
85
>>> hex(85) # Number=>digits: Hex digit string
'0x55'
```

在 `3.1` 和 `2.7` 中，引入了一个新的整数方法 `bit_length` ，可以查询表示一个数值需要多少位比特位。
你可以通过 `bin` 字符串的长度减去 `2` 获得同样的结果，只不过比较低效率。

```
>>> X = 99
>>> bin(X), X.bit_length(), len(bin(X)) - 2
('0b1100011', 7, 7)
>>> bin(256), (256).bit_length(), len(bin(256)) - 2
```

```
('0b100000000', 9, 9)
```

假如你需要进行很多位操作时，你就要考虑你到底应该用哪种语言编程。

---

**其它内置数字工具**

`Python` 还提供一些内置函数和标准库模块来进行数字处理。
比如内置函数 `pow` 和 `abs` 分别计算乘方和绝对值。
下面是内置模块 `math` 的一些例子：

```
>>> import math
>>> math.pi, math.e # Common constants
(3.141592653589793, 2.718281828459045)
>>> math.sin(2 * math.pi / 180) # Sine, tangent, cosine
0.03489949670250097
>>> math.sqrt(144), math.sqrt(2) # Square root
(12.0, 1.4142135623730951)
>>> pow(2, 4), 2 ** 4, 2.0 ** 4.0 # Exponentiation (power)
(16, 16, 16.0)
>>> abs(-42.0), sum((1, 2, 3, 4)) # Absolute value, summation
(42.0, 10)
>>> min(3, 1, 2, 4), max(3, 1, 2, 4) # Minimum, maximum
(1, 4)
```

`sum` 接收一个序列， `max` 和 `min` 可以接收序列或者单个参数。
还有一些去掉浮点数的小数点部分的函数：

```
>>> math.floor(2.567), math.floor(-2.567) # Floor (next-lower integer)
(2, -3)
>>> math.trunc(2.567), math.trunc(-2.567) # Truncate (drop decimal digit
s)
(2, -2)
>>> int(2.567), int(-2.567) # Truncate (integer conversion)
(2, -2)
>>> round(2.567), round(2.467), round(2.567, 2) # Round (Python 3.X versi
on)
(3, 2, 2.57)
>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567) # Round for display (Chapter
 7)
('2.6', '2.57')
```

【!】在 `3.6` 中 `round(2.57)` 得到的是 `3` ，而在 `2.7.8` 中 `round(2.57)` 得到的是 `3.0` 。

字符串格式化得到的是一个字符串。

```
>>> (1 / 3.0), round(1 / 3.0, 2), ('%.2f' % (1 / 3.0))
(0.3333333333333333, 0.33, '0.33')
```

在 `Python` 中有三种求平方根 `square root` 的方法：

- 模块函数
- 表达式
- 内置函数

```
>>> import math
>>> math.sqrt(144) # Module
12.0
>>> 144 ** .5 # Expression
12.0
>>> pow(144, .5) # Built-in
12.0
>>> math.sqrt(1234567890) # Larger numbers
35136.41828644462
>>> 1234567890 ** .5
35136.41828644462
>>> pow(1234567890, .5)
35136.41828644462
```

注意到模块需要导入，而内置函数不需要。
模块是外部组件，而内置函数存在于隐含的命名空间中， `Python` 会在其中自动查找程序中用到的名字。
在 `3.X` 中，这个命名空间对应于 `builtins` 标准库模块，而在 `2.X` 中对应 `__builtins__` 。

还有 `random` 模块：

```
>>> import random
>>> random.random()
0.5566014960423105
>>> random.random() # Random floats, integers, choices, shuffles
0.051308506597373515
>>> random.randint(1, 10)
1
>>> random.randint(1, 10)
10
```

【!】注意 `random.random()` 产生 `[0, 1)` 的随机数，而 `random.randint(a, b)` 产生 `[a, b]` 的随机数

这个模块同样可以随机从一个序列中选取一项，和随机打乱一个列表（改变原列表）。

```
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Holy Grail'
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Life of Brian'
>>> suits = ['hearts', 'clubs', 'diamonds', 'spades']
```

```
>>> random.shuffle(suits)
>>> suits
['spades', 'hearts', 'diamonds', 'clubs']
>>> random.shuffle(suits)
>>> suits
['clubs', 'diamonds', 'hearts', 'spades']
```

---

# 5.3 其它数字类型

我们目前学了整数，浮点数和复数。

**十进制类型 `Decimal Type`**
【?】不知道怎么翻译好，中文版的第四版翻译成**小数对象**。中文的第五版翻译成**小数类型**。
`2.4` 引入了 `Decimal` ，通过调用导入模块的函数来计算，它就像浮点数一样，不过有固定的精度。
虽然它的性能有时不如普通的浮点数类型，但是表示固定精度的数方便又精确。

**基础知识**
之前我们在学习比较操作的时候发现，浮点数没那么精确。

```
>>> 0.1 + 0.1 + 0.1 - 0.3 # Python 3.3
5.551115123125783e-17
```

在 `3.1` 和 `2.7` 之前，使用 `print` 也没什么用。
【!】上面那个式子在 `3.6.6` 中也没什么用

```
>>> print(0.1 + 0.1 + 0.1 - 0.3) # Pythons < 2.7, 3.1
5.55111512313e-17
```

但是使用小数类型就很准确了。

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

可以调用 `decimal` 模块中的 `Decimal` 构造函数，传入小数字符串，来创建小数对象。
当多种精度发生混合计算时， `Python` 自动升级为精度最高的那个。

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal('0.00')
```

在 `2.7` 和 `3.1` 之后，还可以通过浮点数来新建小数对象：

---

```
decimal.Decimal.from_float(1.25)
```

最近的版本允许直接使用浮点数：

```
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)
Decimal('2.77555756156515654042363166 8E-17')
```

浮点数到小数对象的转换是精确的，但有时候会出现很大的默认精度。
`3.3` 及以后，小数的性能急剧提升。

**设置全局精度**
全局精度适用于调用线程中的所有小数：

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7) # Default: 28 digits
Decimal('0.14285714285714285714285714 29')
>>> decimal.getcontext().prec = 4 # Fixed precision
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3) # Closer to
 0
Decimal('1.110E-17')
```

对于与货币相关的应用很有用，因为货币精确到小数点后两位：

```
>>> 1999 + 1.33 # This has more digits in memory than displayed in 3.3
2000.33
>>>
>>> decimal.getcontext().prec = 2
>>> pay = decimal.Decimal(str(1999 + 1.33))
>>> pay
Decimal('2000.33')
```

**小数上下文管理器**
在 `2.6` 和 `3.0` 及以后，可以使用 `with` 上下文管理器语句暂时性的重置精度。
退出上下文管理语句之后，精度自动恢复原来的值。

```
C:\code> C:\Python33\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33333333333333333333333333 33')
>>>
>>> with decimal.localcontext() as ctx:
... ctx.prec = 2
... decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
```

```
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33333333333333333333333333333')
```

---

**分数类型 `Fraction Type`**
`2.6` 和 `3.0` 引入了分数类型，实现了有理数对象 `rational number objcet` 。
性能不会很好，但是使用方便。
它显式的保有一个分子 `numerator` 和一个分母 `denominator` 。

**基础知识**
分数和小数一样，都可以解决浮点数的数值精度问题。
通过导入模块，并给分数的构造器传入一个分子和一个分母，就新建了分数对象。

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3) # Numerator, denominator
>>> y = Fraction(4, 6) # Simplified to 2, 3 by gcd
>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

分数也可以进行运算：

```
>>> x + y
Fraction(1, 1)
>>> x - y # Results are exact: numerator, denominator
Fraction(-1, 3)
>>> x * y
Fraction(2, 9)
```

分数也可以由浮点数字符串创建：

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)
>>>
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

**分数和小数数字精度**
浮点数的精度受限于硬件，体现在由于内存位数限制无法精确表示的数。

```
>>> a = 1 / 3.0
>>> b = 4 / 6.0
>>> a
0.3333333333333333
>>> b
0.6666666666666666
# Only as accurate as floating-point hardware
# Can lose precision over many calculations
>>> a + b
1.0
>>> a - b
-0.3333333333333333
>>> a * b
0.2222222222222222
```

分数 `Fraction` 和小数 `Decimal` 都能获得精确结果，尽管牺牲了一些速度和代码简洁性。

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
# This should be zero (close, but not exact)
>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

而且分数和小数的结果更加直观，分别用有理数和精度限制来实现。

```
>>> 1 / 3
0.3333333333333333 # Use a ".0" in Python 2.X for true "/"
>>> Fraction(1, 3)
Fraction(1, 3) # Numeric accuracy, two ways
>>> import decimal
>>> decimal.getcontext().prec = 2
>>> Decimal(1) / Decimal(3)
Decimal('0.33')
```

分数还能自动化简结果。

```
>>> (1 / 3) + (6 / 12)
0.8333333333333333 # Use a ".0" in Python 2.X for true "/"
>>> Fraction(6, 12)
Fraction(1, 2) # Automatically simplified
>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)
>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
```

```
Decimal('0.83')
>>> 1000.0 / 1234567890
8.100000073710001e-07
>>> Fraction(1000, 1234567890) # Substantially simpler!
Fraction(100, 123456789)
```

**分数转换和混合类型**
浮点数转换为分数：

- `Fraction(*浮点数.as_integer_ratio())`
- `Fraction.from_float(浮点数)`

分数转换为浮点数：

- `float(分数)`

**注意**：星号 `*` 展开元组为单个参数，第 `18` 章会学到。

```
>>> (2.5).as_integer_ratio()            # float object method
(5, 2)
>>> f = 2.5
>>> z = Fraction(*f.as_integer_ratio()) # Convert float -> fraction: two
 args
>>> z                                   # Same as Fraction(5, 2)
Fraction(5, 2)

>>> x                                   # x from prior interaction
Fraction(1, 3)
>>> x + z
Fraction(17, 6)                         # 5/2 + 1/3 = 15/6 + 2/6

>>> float(x)                            # Convert fraction -> float
0.3333333333333333
>>> float(z)
2.5
>>> float(x + z)
2.8333333333333335
>>> 17 / 6
2.8333333333333335

>>> Fraction.from_float(1.75)          # Convert float -> fraction: other
 way
Fraction(7, 4)

>>> Fraction(*(1.75).as_integer_ratio())
Fraction(7, 4)
```

允许一定的类型转换：

```
>>> x
Fraction(1, 3)
>>> x + 2                 # Fraction + int -> Fraction
Fraction(7, 3)
>>> x + 2.0               # Fraction + float -> float
2.3333333333333335
>>> x + (1./3)            # Fraction + float -> float
0.6666666666666666
>>> x + (4./3)
1.6666666666666665
>>> x + Fraction(4, 3)    # Fraction + Fraction -> Fraction
Fraction(5, 3)
```

**警告**虽然你可以把浮点数转换为分数，但是可能会出现精度丢失，因为最初的浮点数形式就已经不精确了。
你可以通过限制最大分母值来化简结果：`分数.limit_denominator(10)` （分母小于等于 `10` ）

```
>>> 4.0 / 3
1.3333333333333333
>>> (4.0 / 3).as_integer_ratio()           # Precision loss from float
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)
>>> 22517998136852479 / 13510798882111488  # 5 / 3 (or close to it!)
1.6666666666666667
>>> a.limit_denominator(10)                # Simplify to closest fraction
Fraction(5, 3)
```

---

**集合 `set`**
属于集合类型 `collection type` 。
**集合 `set`** 是唯一且不可变对象的无序集合 `collection` ，支持数学上的集合运算。
一个元素在集合中只会出现一次，不管它被添加多少次。

【!】集合的元素必须是不可变的对象

集合可迭代，可以按需扩大和缩小，可以包含不同的对象类型。
集合的行为就像只有键，没有值的字典，并且它支持额外的运算。
集合是无序的，所以它不是序列；集合没有把键映射到值，所以它也不是字典；集合自成一类。
集合本质上与数学密切相关。

**`Python 2.6` 及以前版本的集合基础**
往内置函数 `set` 中传入一个序列或者其他可迭代对象。

---

```
>>> x = set('abcde')
>>> y = set('bdxyz')
```

注意返回的集合对象是无序的。

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])        # Pythons <= 2.6 display format
```

数学上的集合运算可以用表达式运算符 `expression operators` ：

```
>>> x - y                                  # Difference
set(['a', 'c', 'e'])
>>> x | y
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z']) # Union
>>> x & y
set(['b', 'd'])                            # Intersection
>>> x ^ y
set(['a', 'c', 'e', 'y', 'x', 'z'])        # Symmetric difference (XO
R)
>>> x > y, x < y
(False, False)                             # Superset, subset
```

`in` 集合成员测试，除了适用于集合，还适用于字符串、列表等。

```
>>> 'e' in x                                # Membership (sets)
True
>>> 'e' in 'Camelot', 22 in [11, 22, 33]    # But works on other types to
o
(True, True)
```

除了表达式，还有相应的方法。
`add` 插入一项元素
`update` 即时 `in-place` 并集
`delete` 根据值删除键

```
>>> z = x.intersection(y)             # Same as x & y
>>> z
set(['b', 'd'])
>>> z.add('SPAM')                     # Insert one item
>>> z
set(['b', 'd', 'SPAM'])
>>> z.update(set(['X', 'Y']))         # Merge: in-place union
>>> z
set(['Y', 'X', 'b', 'd', 'SPAM'])
>>> z.remove('b')                     # Delete one item
>>> z
```

```
set(['Y', 'X', 'd', 'SPAM'])
```

作为可迭代容器，集合可以使用 `len` ， `for` 循环和列表解析等操作。
但是由于集合是无序的，所以不支持索引和切片。

```
>>> for item in set('abc'): print(item * 3)
aaa
ccc
bbb
```

尽管**集合表达式**需要两个集合，但是与之对应的方法通常适用于任意可迭代类型。

```
>>> S = set([1, 2, 3])
>>> S | set([3, 4])             # Expressions require both to be sets
set([1, 2, 3, 4])
>>> S | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> S.union([3, 4])             # But their methods allow any iterable
set([1, 2, 3, 4])
>>> S.intersection((1, 3, 5))
set([1, 3])
>>> S.issubset(range(-5, 5))
True
```

**`Python 3.X` 和 `2.7` 的集合写法**
除了可以用内置函数 `set` 新建集合，还可以用花括号写法来新建集合。

```
set([1, 2, 3, 4])    # Built-in call (all)
{1, 2, 3, 4}         # Newer set literals (2.7, 3.X)
```

这种写法是容易理解的，因为集合就像是无值字典，因为集合的元素是无序、唯一和不可变的，行为非常像字典的键。
集合和字典键相似之处是惊人的：字典键列表在 `3.X` 中是视图对象 `view object` ，支持集合运算，比如交集和并集。（详见第 `8` 章）

`Python 3.X` 以新写法（花括号）来展示集合， `2.7` 接受新写法，但是按照 `2.6` 的方式展示集合。

在所有版本的 `Python` 中，只能用内置函数 `set` 来构建空集合和从现有的可迭代对象生成集合。而新写法对于初始化已知结构的集合很有用。

```
C:\code> c:\python33\python
>>> set([1, 2, 3, 4])              # Built-in: same as in 2.6
{1, 2, 3, 4}
>>> set('spam')                    # Add all items in an iterable
```

```
{'s', 'a', 'p', 'm'}

>>> {1, 2, 3, 4}                    # Set literals: new in 3.X (and 2.7)
{1, 2, 3, 4}
>>> S = {'s', 'p', 'a', 'm'}
>>> S
{'s', 'a', 'p', 'm'}

>>> S.add('alot')                   # Methods work as before
>>> S
{'s', 'a', 'p', 'alot', 'm'}
```

所有的操作在 `3.X` 中都一样，只不过结果显示不太一样：

```
>>> S1 = {1, 2, 3, 4}
>>> S1 & {1, 3}             # Intersection
{1, 3}
>>> {1, 5, 3, 6} | S1       # Union
{1, 2, 3, 4, 5, 6}
>>> S1 - {1, 3, 4}          # Difference
{2}
>>> S1 > {1, 3}             # Superset
True
```

在所有的 `Python` 版本中， `{}` 仍然代表一个空字典，空集合需要用 `set()` 来创建，显示结果也一样。

```
>>> S1 - {1, 2, 3, 4}      # Empty sets print differently
set()
>>> type({})               # Because {} is an empty dictionary
<class 'dict'>

>>> S = set()              # Initialize an empty set
>>> S.add(1.23)
>>> S
{1.23}
```

与 `2.6` 类似， `3.X` 或 `2.7` 也支持同样的方法，这些方法可以接收可迭代对象作为参数。

```
>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> {1, 2, 3}.union([3, 4])
{1, 2, 3, 4}
>>> {1, 2, 3}.union({3, 4})
{1, 2, 3, 4}
```

```
>>> {1, 2, 3}.union(set([3, 4]))
{1, 2, 3, 4}

>>> {1, 2, 3}.intersection((1, 3, 5))
{1, 3}
>>> {1, 2, 3}.issubset(range(-5, 5))
True
```

**不可变的限制和冻结集合**
集合只能包含不可变的（即可散列的 `hashable` ）对象类型。
因此列表和字典不能嵌入到集合中，但是元组可以。

```
>>> S
{1.23}
>>> S.add([1, 2, 3])                # Only immutable objects work in a set
TypeError: unhashable type: 'list'
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'
>>> S.add((1, 2, 3))
>>> S                               # No list or dict, but tuple OK
{1.23, (1, 2, 3)}

>>> S | {(4, 5, 6), (1, 2, 3)}      # Union: same as S.union(...)
{1.23, (4, 5, 6), (1, 2, 3)}
>>> (1, 2, 3) in S                  # Membership: by complete values
True
>>> (1, 4, 3) in S
False
```

集合还可以包含模块，类型对象。
集合本身是可变的，所以集合不能直接嵌套；可以使用 `frozenset` 内置函数，生成一个不可变的集合，然后嵌套在别的集合中。

**`Python 3.X` 和 `2.7` 的集合解析**

```
>>> {x ** 2 for x in [1, 2, 3, 4]}       # 3.X/2.7 set comprehension
{16, 1, 4, 9}
```

解析式也可以迭代其他类型的对象，比如字符串。

```
>>> {x for x in 'spam'}           # Same as: set('spam')
{'m', 's', 'p', 'a'}

>>> {c * 4 for c in 'spam'}       # Set of collected expression results
{'pppp', 'aaaa', 'ssss', 'mmmm'}
>>> {c * 4 for c in 'spamham'}
{'pppp', 'aaaa', 'hhhh', 'ssss', 'mmmm'}
```

```
>>> S = {c * 4 for c in 'spam'}
>>> S | {'mmmm', 'xxxx'}
{'pppp', 'xxxx', 'mmmm', 'aaaa', 'ssss'}
>>> S & {'mmmm', 'xxxx'}
{'mmmm'}
```

**为什么要使用集合？**

- 去重 `filter duplicates` ，但是顺序可能会被打乱。

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                                # Remove duplicates
>>> L
[1, 2, 3, 4, 5]
>>> list(set(['yy', 'cc', 'aa', 'xx', 'dd', 'aa']))   # But order may cha
nge
['cc', 'xx', 'yy', 'dd', 'aa']
```

- 找不同 `isolate differences`
  注：最后两项比较的结果是 `3.X` 的。

```
>>> set([1, 3, 5, 7]) - set([1, 2, 4, 5, 6]) # Find list differences
{3, 7}
>>> set('abcdefg') - set('abdghij')          # Find string differences
{'c', 'e', 'f'}
>>> set('spam') - set(['h', 'a', 'm'])       # Find differences, mixed
{'p', 's'}

>>> set(dir(bytes)) - set(dir(bytearray))    # In bytes but not bytearray
{'__getnewargs__'}
>>> set(dir(bytearray)) - set(dir(bytes))
{'append', 'copy', '__alloc__', '__imul__', 'remove', 'pop', 'insert',
 ...more...]
```

- 顺序无关的相等性测试 `order-neutral equality`
  两个集合相等：每个集合的每个元素都在另一个集合中，即各自是对方的子集。
  也可以先排序再进行相等性测试。但是集合不依赖于开销大的排序。

```
>>> L1, L2 = [1, 3, 5, 2, 4], [2, 5, 3, 4, 1]
>>> L1 == L2                            # Order matters in sequences
False
>>> set(L1) == set(L2)                  # Order-neutral equality
True
>>> sorted(L1) == sorted(L2)            # Similar but results ordered
True
```

```
>>> 'spam' == 'asmp', set('spam') == set('asmp'), sorted('spam') == sorte
d('asmp')
(False, True, True)
```

更现实的例子：

```
>>> engineers = {'bob', 'sue', 'ann', 'vic'}
>>> managers = {'tom', 'sue'}

>>> 'bob' in engineers           # Is bob an engineer?
True

>>> engineers & managers         # Who is both engineer and manager?
{'sue'}

>>> engineers | managers         # All people in either category
{'bob', 'tom', 'sue', 'vic', 'ann'}

>>> engineers - managers         # Engineers who are not managers
{'vic', 'ann', 'bob'}

>>> managers - engineers         # Managers who are not engineers
{'tom'}

>>> engineers > managers         # Are all managers engineers? (superset)
False

>>> {'bob', 'sue'} < engineers   # Are both engineers? (subset)
True

>>> (managers | engineers) > managers      # All people is a superset of
 managers
True

>>> managers ^ engineers         # Who is in one but not both?
{'tom', 'vic', 'ann', 'bob'}

>>> (managers | engineers) - (managers ^ engineers)   # Intersection!
{'sue'}
```

注意：`^` 和 `symmetric_difference()` 是取对称差集，取只在两个集合其中之一的元素。

---

**布尔 `Boolean`**

`Python` 的布尔类型只有两个值， `True` 和 `False` ，它们本质上是整数， 是 `1` 和 `0` 的定制版本，只不过是打印输出不同。

布尔数据类型被称作 `bool` ， 名字 `True` 和 `False` 是 `bool` 的实例， `bool` 类是内置数据类型

`int` 的子类。 `True` 和 `False` 的行为与 `1` 和 `0` 完全一样，除了有定制化的打印逻辑。 `bool` 类通过重定义 `str` 和 `repr` 字符串的格式来实现这一点。

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> True == 1               # Same value
True
>>> True is 1               # But a different object: see the next chapter
False
>>> True or False           # Same as: 1 or 0
True
>>> True + 4                # (Hmmm)
5
```

---

# 5.4 数字扩展

- `NumPy` （ `Numeric Python` ）提供了高级数字编程工具，比如矩阵、向量等等。 `Python` 加上 `NumPy` 被认为是免费的、更灵活的 `Matlab` 。
- `SciPy`

具体信息可以上网搜索。

---

# 5.5 章节总结

本章浏览了 `Python` 的数字对象类型和可以应用的操作。

- 整数
- 浮点数
- 复数
- 小数
- 分数
- 集合
- 布尔

还探索了表达式语法、类型转换、位运算和编写数字的不同写法。

---

# 5.6 检验你的知识

1. 在 `Python` 中，表达式 `2 * (3 + 4)` 的值是多少？
   `14`
2. 在 `Python` 中，表达式 `2 * 3 + 4` 的值是多少？
   `10`
3. 在 `Python` 中，表达式 `2 + 3 * 4` 的值是多少？
   `14`
4. 如何求平方根、平方？
   求平方根：
   `import math` 然后 `math.sqrt(N)`
   或者 `X ** .5`
   求平方：
   `pow(X, 2)`
   或者 `X ** 2`
5. 表达式 `1 + 2.0 + 3` 结果的类型是什么？
   浮点数 `float`
6. 如何截断、四舍五入一个浮点数？
   `int(N)` 和 `math.trunc(N)` 截断一个浮点数。
   `round(N, digits)` 、字符串格式化操作四舍五入一个数。
   `math.floor(N)` 取不大于 `N` 的最大整数。
7. 如何把整数转换为浮点数？
   `float(I)`
   在表达式中混合整数和浮点数
   `Python 3.X` 中的除法 `/` 也会转化结果为浮点数
8. 如何用八进制、十六进制、二进制来显示一个十进制数？
   使用 `oct(I)` 、 `hex(I)` 、 `bin(I)`
   `%` 字符串格式化表达式
   字符串的 `format` 方法
9. 如何把八进制、十六进制、二进制的字符串转换为十进制数？
   使用 `int(S, base)` ，比如 `int(S, 8)` 、 `int(S, 16)` 、 `int(S, 2)` 。
   `eval(S)` 也可以，但是开销大且不安全。
   整数在内存中总是以二进制存储，这些只是字符串显示方式的转换。

---

# 6. 动态类型简介

`Python` 没有声明对象类型，大多数程序也根本不关心具体的类型。

---

## 6.1 缺少声明语句的情况

静态类型语言： `C` 、 `C++` 、 `Java`
动态类型模型 `dynamic typing model`
在 `Python` 中，类型在运行时自动确定，不用事先声明变量。

**变量，对象和引用**

变量创建 `variable creation` ：当给变量赋初值的时候变量被创建。未来的赋值改变已经创建的名字的值。技术上 `Python` 在运行代码前先检测一些名字，但是可以理解为**赋初值创建变量**。
变量类型 `variable type` ：变量没有与之相关的类型信息或者约束。类型的概念与对象关联，而不是与变量。
变量使用 `variable use` ：当变量在表达式中出现的时候，自动替换为它当前引用的对象。而且所有变量使用之前必须被显式地赋值；引用未被赋值的变量会报错。

**当给变量赋值的时候就创建了变量，变量可以引用任意类型的对象，使用变量之前必须赋值。**

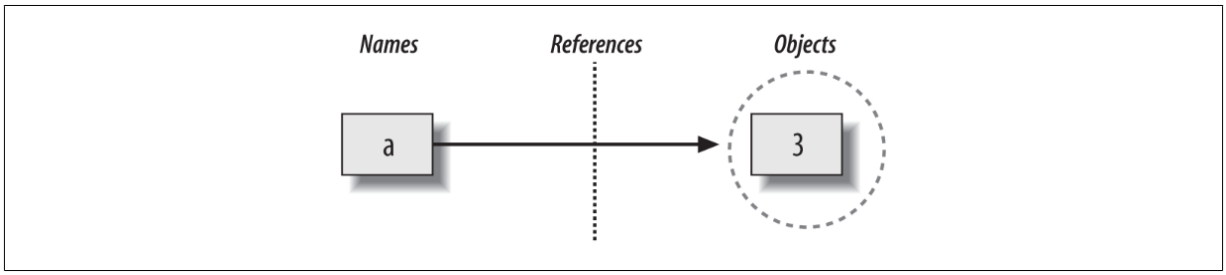
```
>>> a = 3         # Assign a name to an object
```

1. 创建一个代表值 `3` 的对象
2. 如果 `a` 不存在的话，创建变量 `a`
3. 把新对象 `3` 和变量 `a` 连接起来

变量总是与对象连接，而不会与其他变量连接。但是对象可以连接到其他对象，比如列表对象可以连接到它包含的对象。

在 `Python` 中，从变量到对象的连接被称为**引用** `reference` 。
引用是一种关联，在内存中作为**指针** `pointer` 实现。

- 变量是系统表的一条条记录，有指向对象的连接的空间。
- 对象是被分配好的内存片段，有足够的空间来表示它们所代表的值。
- 引用是被 `Python` 自动跟踪的，从变量到对象的指针。




作为优化， `Python` 在内部缓存并复用一些不可变的对象，比如 `0` 。但是逻辑上每个表达式的值是独立的对象，每个对象占用独立的内存。

技术上来说，对象除了拥有表示值的空间，还有两个标准头字段 `standard header field` 。

- 类型标识符 `type designator` 用来标记对象类型
- 引用计数器 `reference counter` 用来确认什么时候可以回收对象

> 引用类似于 `C` 语言中的指针，实际上它就是这样实现的

**类型属于对象，而不是变量**

```
>>> a = 3             # It's an integer
>>> a = 'spam'        # Now it's a string
```

```
>>> a = 1.23        # Now it's a floating point
```

名字没有类型，类型属于对象。
对象包含头部字段，其中类型标识符会告诉 `Python` 这个对象是整数，其实它是一个指向 `int` 对象的指针。字符串对象的类型标识符指向字符串类型 `str` 。

**对象被垃圾回收**
当没有名字或者对象引用一个对象的空间时，它会被回收，这叫做垃圾回收机制 `garbage collection` 。

```
>>> x = 42
>>> x = 'shrubbery'        # Reclaim 42 now (unless referenced elsewhere)
>>> x = 3.1415             # Reclaim 'shrubbery' now
>>> x = [1, 2, 3]          # Reclaim 3.1415 now
```

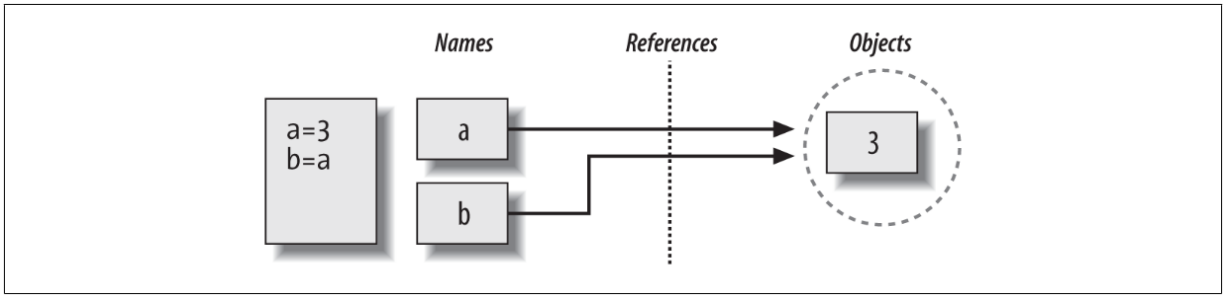
类型属于对象，不属于名字，名字是指向对象的引用。
每当 `x` 分配给新的对象，之前的对象被自动回收，它占用的空间被自动扔给自由空间池 `free space pool` ，以便未来的对象使用。
`Python` 通过追踪对象的当前引用个数来实现垃圾回收机制。
当引用计数器的值变为 `0` 时，该对象占用的内存空间被自动回收。

`Python` 的垃圾回收机制大部分依赖于引用计数器 `reference counter` ，同时它还有一个组件专门检测和回收循环引用 `cyclic reference` 。
该组件可以被关闭，但是默认是打开的。
更多关于循环检测器 `cycle detector` 的内容详见 `gc` 模块的文档。
本节描述的垃圾回收机制只适用于 `CPython` ，但是其他实现也有类似效果，不用手动管理内存。

---

# 6.2 共享引用

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



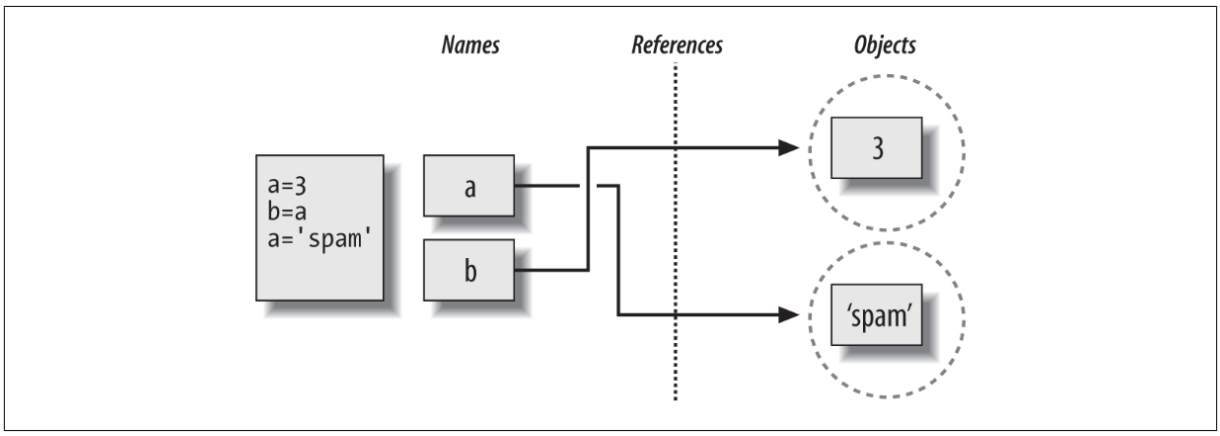

`a` 和 `b` 指向同一个对象，即指向同一块内存空间。

多个名字引用同一对象被称作**共享引用** `shared reference` ，有时也叫作**共享对象** `shared object` 。

注意变量 `a` 和 `b` 没有相连，在 `Python` 里面永远无法连接两个变量。而两个变量通过各自的引用指向了同一对象。

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

最后一个赋值语句新建了一个字符串对象 `'spam'` ，并让 `a` 引用这个新对象。
因为最后的赋值语句没有在原处修改 `3` ，而是新建了一个对象，所以它没有改变 `b` 的值， `b` 仍然引用最初的对象，整数 `3` 。




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

最后的赋值语句让 `a` 引用了一个完全不同的对象 `5` ，所以并没有改变 `b` 。

给变量重新赋值并不改变原先的对象，但是会让变量引用一个完全不同的对象。

**共享引用和原地修改**
`Python` 的可变类型，包括列表、字典和集合会在原处修改对象。
比如对列表的某一偏移量赋值，会在原地修改列表对象，而不会生成全新的列表对象。
对于支持原地修改的对象，一定要小心使用共享引用，因为修改一个名字的值会影响其他名字。

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

假如执行这条语句：

```
>>> L1 = 24
```

`L1` 是完全不同的对象， `L2` 仍然引用原始的列表。

但是如果这样结果会完全不同：

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

>>> L1                      # L1 is different
[24, 3, 4]
>>> L2                      # But so is L2!
[24, 3, 4]
```

这种行为只出现在支持原地修改 `in-place change` 的可变对象上。
如果你不想要这种行为，可以让 `Python` 拷贝 `copy` 对象而不是建立引用。
有很多方法可以拷贝列表，比如内置函数 `list` 和标准库 `copy` 模块，最简单的是从头到尾切片操作。

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

>>> L1
[24, 3, 4]
>>> L2              # L2 is not changed
[2, 3, 4]
```

但是切片操作不适用于其它可变类型，比如字典和集合，因为它们不是序列。所以需要使用 `X.copy()` 方法（列表在 `3.3` 中也有）。
另外 `copy` 模块也可以拷贝对象，还可以拷贝嵌套的对象结构，比如包含列表的字典。

```
import copy
X = copy.copy(Y)         # Make top-level "shallow" copy of any object Y
X = copy.deepcopy(Y)     # Make deep copy of any object Y: copy all nested
 parts
```

**共享引用和相等**

```
>>> x = 42
>>> x = 'shrubbery'       # Reclaim 42 now?
```

因为 `Python` 缓存并复用小整数和小字符串，所以 `42` 很可能不会被立即回收。

由于 `Python` 的引用模型，有两种方法测试相等性。

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

```
True
```

`==` 测试两个对象是否有相同的值。
`is` 测试对象身份，两个名字如果指向同一个对象，则返回 `True` 。
`is` 比较实现引用的指针是否相等，可以检测代码中是否出现共享引用。

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]              # M and L reference different objects
>>> L == M                     # Same values
True
>>> L is M                     # Different objects
False
```

看看测试小数会发生什么：

```
>>> X = 42
>>> Y = 42                 # Should be two different objects
>>> X == Y
True
>>> X is Y                  # Same object anyhow: caching at work!
True
```

`X` 和 `Y` 都引用了同一个 `42` 。

`sys` 模块下的 `getrefcount` 函数可以返回对象的引用数（实际上应该 `-1` ，因为这个函数也引用了对象一次）

```
>>> import sys
>>> sys.getrefcount(1)      # 647 pointers to this shared piece of memory
647
```

对象缓存和复用与你的代码无关，除非你想要进行 `is` 检测。
因为不能在原地修改不可变的数字和字符串，所以同一个对象有多少引用并不重要，每个引用永远只能访问到不变的值。这种行为是 `Python` 优化执行速度的体现。

---

# 6.3 随处可见的动态类型

`Python` 里面一切都是基于赋值和引用的，一旦掌握了这个模型，到哪里都适用。

- 从实际层面出发，动态类型可以减少你的代码量。
- 同样重要的是，动态类型是 `Python` 多态 `polymorphism` 的根源。

因为我们不约束类型，所以代码会简洁且灵活。

**“弱”引用 `"weak" reference`**
弱引用由 `weakref` 标准库模块实现，弱引用本身不会防止对象被垃圾回收。
如果一个对象的引用全是弱引用，这个对象会被回收，弱引用会被自动删除（或者被通知）。
它可以用在大型对象的基于字典的缓存中，否则这些缓存引用本身就会让对象在内存中永驻。

---

# 6.4 章节总结

本章介绍了：

- 动态类型模型，即 `Python` 如何自动为我们记录对象的类型
- 变量和对象是如何通过引用关联
- 垃圾回收机制
- 共享引用如何影响到多个变量值的
- 引用如何影响相等性的概念

---

# 6.5 检验你的知识

1. 下面三条赋值语句会改变 `A` 的打印值吗？

```
A = "spam"
B = A
B = "shrubbery"
```

不会

2. 下面三条赋值语句会改变 `A` 的打印值吗？

```
A = ["spam"]
B = A
B[0] = "shrubbery"
```

会。我们没有改变 `A` 和 `B` ，只是原地修改了它们引用 / 指向的对象。

3. `A` 被改变了吗？

```
A = ["spam"]
B = A[:]
B[0] = "shrubbery"
```

没有。
第二句在赋值之前先对列表进行了拷贝，所以 `A` 和 `B` 指向不同的对象。

# 7. 字符串基础

本章学习字符串，它是有序的字符集合，用来表示文本和基于字节的信息。

---

## 7.1 本章范围

不会过多的讨论 `unicode` ，在第 `37` 章会有详细论述，而是会讲基础 `str` 字符串类型。

**`Unicode` 简介**
`ASCII` 是 `Unicode` 的简单形式。

`Python` 区别对待文本和二进制数据，通过文本对象类型和文件接口分别处理它们。

- `3.X` 中有三种字符串类型： `str` 字符串用来处理 `Unicode` 文本（包括 `ASCII` ）； `bytes` 字节串用来处理二进制数据（包括编码后的文本）； `bytearray` 字节数组是字节串 `bytes` 的可变版本。文件有两种工作模式：文本 `text` 模式，以字符串 `str` 的形式展现其内容，采用 `Unicode` 编码；二进制 `binary` 模式，处理原始字节 `bytes` ，不做数据转换。
- `2.X` 中 `unicode` 字符串代表 `Unicode` 文本， `str` 字符串处理 `8` 位文本（比如 `ASCII` 文本）和二进制数据，字节数组 `bytearray` 在 `2.6` 中可用。普通文件的内容作为 `str` 展示， `codecs` 模块打开 `Unicode` 文本文件，处理编码，把内容展示为 `unicode` 对象。

实际上 `Unicode` 的区别主要在于翻译（编码 `encoding` ）的步骤，即如何把它转换成文件或者从文件转换回来。

---

## 7.2 字符串基础

`Python` 中的字符串可以当做是 `C` 里面的字符数组来使用，但是它自带一系列的处理工具；另外与 `C` 不同的是，它没有单一的字符类型，只有一个字符的字符串。

`Python` 的字符串是不可变序列 `immutable sequence` ，意味着它包含的字符有从左到右的相对位置，并且不能原地修改。
本章介绍的序列操作同样适用于其它序列类型，比如列表、元组。

*Table 7-1. Common string literals and operations*

| Operation | Interpretation |
|---|---|
| `S = ''` | Empty string |
| `S = "spam's"` | Double quotes, same as single |
| `S = 's\np\ta\x00m'` | Escape sequences |
| `S = """...multiline..."""` | Triple-quoted block strings |
| `S = r'\temp\spam'` | Raw strings (no escapes) |
| `B = b'sp\xc4m'` | Byte strings in 2.6, 2.7, and 3.X (Chapter 4, Chapter 37) |
| `U = u'sp\u00c4m'` | Unicode strings in 2.X and 3.3+ (Chapter 4, Chapter 37) |
| `S1 + S2` | Concatenate, repeat |
| `S * 3` | |
| `S[i]` | Index, slice, length |
| `S[i:j]` | |
| `len(S)` | |
| `"a %s parrot" % kind` | String formatting expression |
| `"a {0} parrot".format(kind)` | String formatting method in 2.6, 2.7, and 3.X |
| `S.find('pa')` | String methods (see ahead for all 43): search, |
| `S.rstrip()` | remove whitespace, |
| `S.replace('pa', 'xx')` | replacement, |
| `S.split(',')` | split on delimiter, |

| Operation | Interpretation |
| --- | --- |
| S.isdigit() | content test, |
| S.lower() | case conversion, |
| S.endswith('spam') | end test, |
| 'spam'.join(strlist) | delimiter join, |
| S.encode('latin-1') | Unicode encoding, |
| B.decode('utf8') | Unicode decoding, etc. (see Table 7-3) |
| for x in S: print(x) | Iteration, membership |
| 'spam' in S | |
| [c * 2 for c in S] | |
| map(ord, S) | |
| re.match('sp(.*)am', line) | Pattern matching: library module |

除了基础工具，`Python` 还有 `re` 正则表达式模块和 `XML` 解析器。

---

# 7.3 字符串写法

- 单引号：`'spa"m'`
- 双引号：`"spa'm"`
- 三引号：`'''... spam ...'''` ，`"""... spam ..."""`
- 转义序列：`"s\tp\na\0m"`
- 原始字符串：`r"C:\new\test.spm"`
- 字节写法（`3.X` 和 `2.6+`）：`b'sp\x01am'`
- `Unicode` 写法（`2.X` 和 `3.3+`）：`u'eggs\u0020spam'`

**单引号和双引号一样**

```
>>> 'shrubbery', "shrubbery"
('shrubbery', 'shrubbery')
```

之所以支持两种形式的引号是为了可以方便的嵌入另外一种引号而不用反斜杠转义。

```
>>> 'knight"s', "knight's"
('knight"s', "knight's")
```

本书偏好使用单引号，因为更容易阅读。大多数程序员都偏好使用单引号。

**隐式拼接和显式拼接**

```
>>> title = "Meaning " 'of' " Life"      # Implicit concatenation
>>> title
'Meaning of Life'
```

使用加号 `+` 显式拼接。

如果在字符串之间加逗号 `,` ，会得到一个元组。
注意 `Python` 的打印输出都是单引号，除非内嵌了一个单引号。

```
>>> 'knight\'s', "knight\"s"
("knight's", 'knight"s')
```

**转义序列代表特殊字符**
反斜杠 `backslash` 用来引入特殊字符，被称作转义序列 `escape sequence` 。
转义字符可以让我们在字符串中嵌入不容易在键盘上打印出来的字符。
`\` 反斜杠和它后面的一个或者多个字符，在字符串对象中会被转换成一个单一字符，它的二进制由转义序列指定。

```
>>> s = 'a\nb\tc'
```

`\n` 代表一个单一字符，换行符的二进制值。
`\t` 被替换成制表符。
这种字符串在交互式回显中把特殊字符显示为转义序列，但是在 `print` 语句中会翻译特殊字符。

```
>>> s
'a\nb\tc'
>>> print(s)
a
b       c
```

用 `len` 来查看长度：

```
>>> len(s)
5
```

> 这里的长度 `5` 未必是 `5` 字节。
> 在 `Unicode` 中，单一字符并不对应于单一字节。
> `3.X` 把 `str` 定义成 `Unicode` 码点的序列，而不是字节。
> 把字符串里的东西理解为字符 `characters` 而不是字节 `bytes` ，这是 `C` 语言使用者需要适应的地方。

注意反斜杠 `\` 本身并没有跟随字符串存储在内存中，它们只被用来描述特殊的字符值。
表 `7.2` 转义字符

| Escape | Meaning |
|---|---|
| *newline* | Ignored (continuation line) |
| \\ | Backslash (stores one \) |
| \' | Single quote (stores ') |
| \" | Double quote (stores ") |
| \a | Bell |
| \b | Backspace |
| \f | Formfeed |
| \n | Newline (linefeed) |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \x*hh* | Character with hex value *hh* (exactly 2 digits) |
| *ooo* | Character with octal value *ooo* (up to 3 digits) |
| \0 | Null: binary 0 character (doesn't end string) |
| \N{ id } | Unicode database ID |
| \u*hhhh* | Unicode character with 16-bit hex value |
| \U*hhhhhhhh* | Unicode character with 32-bit hex value[a] |
| *other* | Not an escape (keeps both \ and *other*) |

[a] The \U*hhhh...* escape sequence takes exactly eight hexadecimal digits (*h*); both \u and \U are recognized only in Unicode string literals in 2.X, but can be used in normal strings (which *are* Unicode) in 3.X. In a 3.X *bytes* literal, hexadecimal and octal escapes denote the byte with the given value; in a *string* literal, these escapes denote a Unicode character with the given code-point value. There is more on Unicode escapes in Chapter 37.

可以在字符串中嵌入二进制值，这里是以八进制的形式：

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

> jpch89：这里的二进制值是以八进制的一位数形式表示的。
> `\ooo` 八进制数最多三位数

在 `Python` 中零字符 `\0`（空 `Null` 字符）不会终止一个字符串，这与 `C` 语言中不一样。
在 `Python` 中没有任何一个字符可以终止字符串。

```
>>> s = '\001\002\x03'
>>> s
```

```
'\x01\x02\x03'
>>> len(s)
3
```

上面是用八进制表示的二进制值 `1` 和 `2`，跟着一个十六进制值表示的二进制 `3`。
**注意**：`Python` 用十六进制来表示无法打印的字符，不管它们是如何指定的。
你可以随意组合这些转义字符：

```
>>> S = "s\tp\na\x00m"
>>> S
's\tp\na\x00m'
>>> len(S)
7
>>> print(S)
s       p
a m
```

如果 `Python` 不认识 `\` 后面的转义字符，它会保留反斜杠 `\`，并显示为 `\\`

```
>>> x = "C:\py\code" # Keeps \ literally (and displays it as \\)
>>> x
'C:\\py\\code'
>>> len(x)
10
```

建议**不要**依赖这种行为，如果需要表示反斜杠，就要明确的写出来 `\\`，或者使用原始字符串。

---

### 原始字符串取消转义

> `less-than-stellar` 不良的，不完美的，不怎么好的

转义字符有时候会带来麻烦，比如下面的代码会有些问题
`myfile = open('C:\new\text.dat', 'w')`

这时就要用到原始字符串，在字符串的引号前面加一个 `r` 或者 `R`，它能**关闭转义机制**。
所以上面的代码要改成：

- `myfile = open(r'C:\new\text.dat', 'w')`
- 或者：`myfile = open('C:\\new\\text.dat', 'w')`

```
>>> path = r'C:\new\text.dat'
>>> path # Show as Python code
'C:\\new\\text.dat'
>>> print(path) # User-friendly format
C:\new\text.dat
```

```
>>> len(path) # String length
15
```

原始字符串的两个**应用场景**：

- 表示 `Windows` 系统上的路径
- 正则表达式

**注意** `Python` 通常用 `/` 斜杠来表示路径，它也适用于 `Windows`。

> 原始字符串无法以奇数个反斜杠结尾。
> 因为它会把结尾的引号也转义，这样就缺少一个引号。
> 有三个方法可以解决这个问题：
>
> - 写两个反斜杠然后切片 `r'abc\\'[:-1]`
> - 使用字符串拼接添加反斜杠 `r'abc' + '\\'` 或者 `r'abc''\\'`
> - 直接使用普通字符串 `'abc\\'`

---

**三引号多行块字符串**
多行字符串用三引号（单双皆可）表示，也叫作块字符串 `block string`
所键即所得。
`Python` 会把多行字符串收集到一个单行字符串中
想要解析换行符，需要 `print` 打印它而不是让它回显。

```
>>> mantra = """Always look
...   on the bright
... side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
>>> print(mantra)
Always look
  on the bright
side of life.
```

三引号对会把其中的所有字符当做字符串。
所以注释不要写在三引号对里面，要写在它的上面或者下面。
或者利用邻近字符串的自动拼接，配合换行符，同时用括号包裹以便续行。

```
>>> menu = """spam     # comments here added to string!
... eggs               # ditto
... """
>>> menu
'spam     # comments here added to string!\neggs               # ditto\n'
>>> menu = (
```

```
... "spam\n"           # comments here ignored
... "eggs\n"           # but newlines not automatic
... )
>>> menu
'spam\neggs\n'
```

多行字符串通常用作文档字符串 `documentation strings`
文档字符串不一定非要是三引号括起来的字符串，但是这样它就可以进行多行注释了。

小技巧：多行字符串还可以用来注释掉多行代码。

```
X = 1
"""
import os              # Disable this code temporarily
print(os.getcwd())
"""
Y = 2
```

注意这样会让 `Python` 在运行时创建一个这样的多行字符串，多少会对性能有影响。

---

# 7.4 字符串应用

**基本操作**
下面解释表 `7-1` 的字符串操作。
使用 `+` 拼接字符串，使用 `*` 重复字符串。

```
% python
>>> len('abc')            # Length: number of items
3
>>> 'abc' + 'def'         # Concatenation: a new string
'abcdef'
>>> 'Ni!' * 4             # Repetition: like "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

使用 `len` 内置函数 `built-in function` 会返回字符串长度。
在 `Python` 中不用事先声明数据类型和内存大小，你只管创建对象即可，剩下的工作都被自动接管。

当需要输入很多一样的符号的时候，就会体会到重复操作的妙用：

```
>>> print('------- ...more... ---')     # 80 dashes, the hard way
>>> print('-' * 80)                     # 80 dashes, the easy way
```

注意到这里出现了运算符重载 `operator overloading`
因为知道被相加和相乘的对象类型，所以 `Python` 进行了正确的运算。
注意 `Python` 不允许在相加操作中混用字符串和数字。

可以用 `for` 遍历字符串。
可以用 `in` 进行成员检测，它有点类似 `str.find()` 方法，只不过 `in` 返回布尔值，而 `str.find()` 方法返回子字符串的位置。

```
>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')   # Step through items, print each
 (3.X form)
...
h a c k e r
>>> "k" in myjob                        # Found
True
>>> "z" in myjob                        # Not found
False
>>> 'spam' in 'abcspamdef'              # Substring search, no position r
eturned
True
```

`for` 循环给序列中的元素分配了一个变量，并为每一项执行操作。

---

**索引和切片**
`Python` 的偏移量跟 `C` 一样，从 `0` 到 `字符串的长度减 1`。
和 `C` 不一样的是，`Python` 可以使用负偏移量索引。
负偏移量加上字符串长度可以推导出正偏移量。
也可以想象成是从末尾开始数。

```
>>> S = 'spam'
>>> S[0], S[-2]             # Indexing from front or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]   # Slicing: extract a section
('pa', 'pam', 'spa')
```

> 有数学头脑的人发现，正索引从 `0` 开始，而负索引从 `-1` 开始，似乎不对称。但是在 `Python` 中并没有 `-0` 这个值（跟 `0` 是一样的）
> `alas` 哎（语气词）

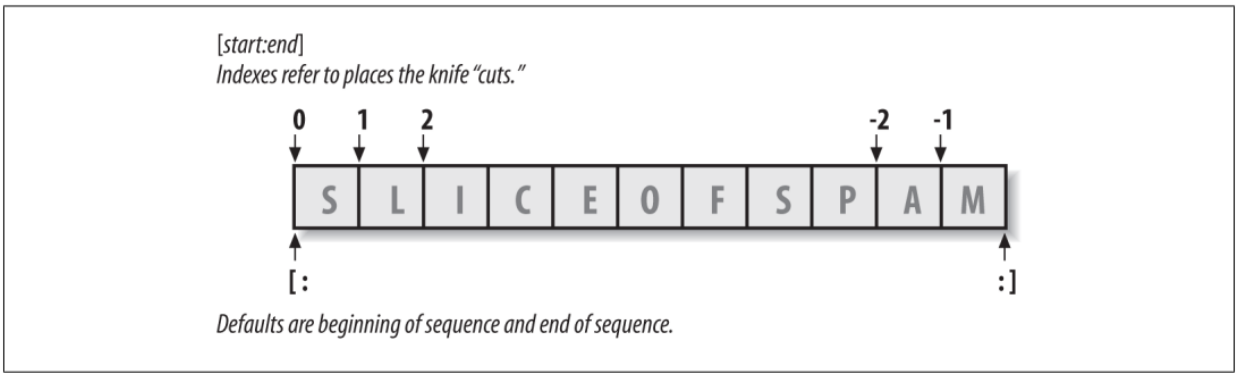



*Figure 7-1. Offsets and slices: positive offsets start from the left end (offset 0 is the first item), and negatives count back from the right end (offset –1 is the last item). Either kind of offset can be used to give positions in indexing and slicing operations.*

切片操作左闭右开，即包括左偏移量，不包括右偏移量。
如果省略，左偏移量默认为 `0` ，右偏移量默认为整个对象的长度。

总结如下
**索引** `S[i]` 按照偏移量取值

- 第一项偏移量为 `0`
- 负索引意味着从右往左数（从末尾开始数）
- `S[0]` 取第一个值
- `S[-2]` 从末尾取第二个值（相当于 `S[len(S) - 2]` ）

**切片** `S[i:j]` 连续取值

- 不含上界
- 如果省略，默认为 `0` 或者序列长度
- `S[1:3]` 取偏移量为 `1` 到 `3` （不含 `3` ）的元素
- `S[1:]` 取偏移量为 `1` 直到末尾（即偏移量为序列长度）的所有元素
- `S[:3]` 取偏移量为 `0` 到 `3` （不含 `3` ）的元素
- `S[:-1]` 取偏移量为 `0` 到 `-1` （不含 `-1` ）的元素
- `S[:]` 取偏移量为 `0` 直到末尾（即偏移量为序列长度）的所有元素，相当对 `S` 做了**顶层拷贝** `top-level copy` 。

**扩展切片 S[i:j:k]** `extended slicing`
接收一个步长 `step/stride` ，默认为 `+1`

`S[:]` 非常常用，它对序列对象进行顶层拷贝，得到的对象值相同，但是所占据的内存空间不同。
这对于字符串来讲没什么用，但是对于那些可以原地改变的对象比如列表很有用。

方括号根据偏移量对列表进行索引，后面还会看到方括号根据键对字典进行索引。

**扩展切片**
`Python 2.3` 以后支持的特性。
切片的完全体是 `X[I:J:K]` ，意思是从 `X` 中取偏移量为 `I` 到 `J-1` 的元素，步长为 `K` 。
第三个参数步长，默认为 `+1`

```
>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]               # Skipping items
```

```
'bdfhj'
>>> S[::2]
'acegikmo'
```

负步长可以把列表逆序，比如 `'hello'[::-1]` 得到 `'olleh'`。
第一个参数默认为 `0`，第二个参数默认为列表长度减一，`-1` 代表从后往前取。

```
>>> S = 'hello'
>>> S[::-1]          # Reversing items
'olleh'
```

另一个例子（这里也是左闭右开）：

```
>>> S = 'abcedfg'
>>> S[5:1:-1]        # Bounds roles differ
'fdec'
```

切片操作等价于以一个切片对象 `slice object` 做索引。
写类的时候最好同时支持这两种操作：

```
>>> 'spam'[1:3]             # Slicing syntax
'pa'
>>> 'spam'[slice(1, 3)]     # Slice objects with index syntax + object
'pa'
>>> 'spam'[::-1]
'maps'
>>> 'spam'[slice(None, None, -1)]
'maps'
```

切片的应用
比如运行 `Python` 程序的时候带的参数可以通过 `sys.argv` 访问到

```
# File echo.py
import sys
print(sys.argv)

% python echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```

通常我们只关心程序名之后的参数，这就要用到切片 `sys.argv[1:]`。
切片还经常用于去掉行尾的换行符 `line[:-1]`。
当然更常用的是 `line.rstrip` 方法，假如行尾没有换行符的话，它可以保持字符串完整性。

---

**字符串转换工具**
`Python` 的设计逻辑之一是杜绝猜疑。
举个例子，不能把字符串和数字相加。

```
# Python 3.X
>>> "42" + 1
TypeError: Can't convert 'int' object to str implicitly

# Python 2.X
>>> "42" + 1
TypeError: cannot concatenate 'str' and 'int' objects
```

你需要做的是进行类型转换

```
>>> int("42"), str(42)      # Convert from/to string
(42, '42')
>>> repr(42)                # Convert to as-code string
'42'
```

`repr` 也能将对象转换成其字符串表示（`3.X` 废弃了之前使用的 `backquote` 反引号）
它返回的**代码形式字符串** `as-code string` 可以重新运行来重新创建这个对象。

注意使用 `print` 打印出来的字符串和交互回显出来的字符串的不同之处：

```
>>> print(str('spam'), repr('spam'))     # 2.X: print str('spam'), repr
('spam')
spam 'spam'
>>> str('spam'), repr('spam')            # Raw interactive echo displays
('spam', "'spam'")
```

还有类似的内置函数可以进行字符串和浮点数之间的转换：

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)
>>> text = "1.234E-10"
>>> float(text)          # Shows more digits before 2.7 and 3.1
1.234e-10
```

`eval` 函数可以把字符串转换成任意 `Python` 对象。
`int` 和 `float` 只能转换到数字，这个限制意味着它更快也更安全。
字符串格式化也可以把数字转换成字符串。

---

**字符码转换**
使用 `ord` 内置函数，把单个字符转换成 `ASCII` 码。

`chr` 相反，把 `ASCII` 码转换成单个字符。

```
>>> ord('s')
115
>>> chr(115)
's'
```

实际上 `ord` 和 `chr` 可以完成字符到 `Unicode` 码点（ `ordinal` 或者 `code point` ）的转换。

这两个函数还可以基于字符串做一些数学运算。

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

对于单个字符，是除了 `int` 类型转换的另一种进行数学计算的方法

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```

这种转换可以跟循环配合使用。
比如转换二进制字符串到整数：

```
>>> B = '1101'         # Convert binary digits to integer with ord
>>> I = 0
>>> while B != '':
... I = I * 2 + (ord(B[0]) - ord('0'))
... B = B[1:]
...
>>> I
13
```

位左移操作与 `* 2` 效果一样：`I << 1`

`Python` 有内置函数进行数的进制转换：

```
>>> int('1101', 2)       # Convert binary to integer: built-in
13
>>> bin(13)              # Convert integer to binary: built-in
'0b1101'
```

**修改字符串**
字符串是不可变序列。
你不能在原地修改它的值，也就是不能通过索引操作给一个元素赋值。

```
>>> S = 'spam'
>>> S[0] = 'x'          # Raises an error!
TypeError: 'str' object does not support item assignment
```

想要修改字符串，新建一个就好了：

```
>>> S = S + 'SPAM!'     # To change a string, make a new one
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[-1]
>>> S
'spamBurger!'
```

`str.replace` 方法可以替换字符串

```
>>> S = 'splot'
>>> S = S.replace('pl', 'pamal')
>>> S
'spamalot'
```

所有字符串方法都会产生一个新的字符串对象。
还可以使用格式化字符串来生成新的字符串：

```
>>> 'That is %d %s bird!' % (1, 'dead') # Format expression: all Pythons
That is 1 dead bird!
>>> 'That is {0} {1} bird!'.format(1, 'dead') # Format method in 2.6, 2.
7, 3.X
'That is 1 dead bird!'
```

> `3.0` 和 `2.6` 引入了一个新的字符串类型 `bytearray` 字节数组，它是可变的，所以可以在原地更改。
> jpch89：略过一些原书内容，关于 `bytearray` 字节数组的使用参见
> https://blog.csdn.net/qq_27297393/article/details/80806868

# 7.5 字符串方法

方法相较于表达式和内置函数更加针对于某种特定的对象类型。

---

**方法调用语法**
方法是属于和作用于特定对象的函数。
方法实际上是对象的属性，这个属性引用了一个可调用函数。
方法调用做了两件事情：

- 属性获取
  `对象.属性`（`object.attribute`）
  获取对象中的属性值
- 函数调用
  `函数(参数)`（`function(arguments)`）
  调用函数中的代码，传入零至多个用逗号分开的参数对象，返回函数的结果值。

方法调用表达式：`对象.方法(参数)`（`object.method(arguments)`）

```
>>> S = 'spam'
>>> result = S.find('pa')  # Call the find method to look for 'pa' in str
ing S
```

为了调用对象方法，必须存在一个对象。

---

**字符串方法**
对字符串对象或者 `str` 类型名使用 `dir` 或者 `help` 函数，查看最新的方法列表。
`Python 2.X` 的字符串方法包括了 `decode`。
`S` 在下表中是字符串对象。

*Table 7-3. String method calls in Python 3.3*

| | |
|---|---|
| S.capitalize() | S.ljust(width [, fill]) |
| S.casefold() | S.lower() |
| S.center(width [, fill]) | S.lstrip([chars]) |
| S.count(sub [, start [, end]]) | S.maketrans(x[, y[, z]]) |
| S.encode([encoding [,errors]]) | S.partition(sep) |
| S.endswith(suffix [, start [, end]]) | S.replace(old, new [, count]) |
| S.expandtabs([tabsize]) | S.rfind(sub [,start [,end]]) |
| S.find(sub [, start [, end]]) | S.rindex(sub [, start [, end]]) |
| S.format(fmtstr, *args, **kwargs) | S.rjust(width [, fill]) |
| S.index(sub [, start [, end]]) | S.rpartition(sep) |
| S.isalnum() | S.rsplit([sep[, maxsplit]]) |
| S.isalpha() | S.rstrip([chars]) |
| S.isdecimal() | S.split([sep [,maxsplit]]) |
| S.isdigit() | S.splitlines([keepends]) |
| S.isidentifier() | S.startswith(prefix [, start [, end]]) |
| S.islower() | S.strip([chars]) |
| S.isnumeric() | S.swapcase() |
| S.isprintable() | S.title() |
| S.isspace() | S.translate(map) |
| S.istitle() | S.upper() |
| S.isupper() | S.zfill(width) |
| S.join(iterable) | |

**字符串方法例子：修改字符串**

我们无法原地修改字符串，因为它是不可变的。

我们可以更改 `bytearray` 字节数组，但是它只能表达简单的 `8` 位类型。

替换

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')      # Replace all mm with xx in S
>>> S
'spaxxy'
```

`replace` 方法是全局搜索替换

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'
```

`find` 返回子字符串第一次出现的偏移量，如果没找到返回 `-1`

`replace` 也可以限制替换次数

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')          # Replace all
'xxxxEGGSxxxxEGGSxxxx'

>>> S.replace('SPAM', 'EGGS', 1)       # Replace one
'xxxxEGGSxxxxSPAMxxxx
```

`replace` 每次都返回新的字符串

如果你需要对一个很长的字符串经常进行改动，你可以通过把该字符串转换成支持原处修改的对象，以便提高性能。

```
>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
>>> L[3] = 'x'                      # Works for lists, not strings
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
```

使用 `join` 重新把它转成字符串：

```
>>> S = ''.join(L)
>>> S
'spaxxy'
```

`join` 是字符串的方法，而不是列表的方法。
前面的字符串是分隔符。

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

通常 `join` 连接子字符串要比拼接速度更快。

---

**字符串方法例子：解析文本**
用切片来提取子字符串

```
>>> line = 'aaa bbb ccc'
```

```
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

使用 `split` 拆分字符串，默认以空白（包括空格、换行、制表符）为分隔符拆分。
返回一个列表。

```
>>> line = 'aaa bbb ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

还可以指定分隔符

```
>>> line = 'bob,hacker,40'
>>> line.split(',')
['bob', 'hacker', '40']
```

除了单个字符，一个字符串也可以作为分隔符

```
>>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")
["i'm", 'a', 'lumberjack']
```

逗号分隔是 `CSV` 文件的一种格式，`Python` 的 `csv` 标准库里面还有更多高级工具。

---

**其他字符串方法**

```
>>> line = "The knights who say Ni!\n"
>>> line.rstrip()
'The knights who say Ni!'
>>> line.upper()
'THE KNIGHTS WHO SAY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
>>> line.startswith('The')
True
```

还有一些等价的用法：

```
>>> line
'The knights who say Ni!\n'
>>> line.find('Ni') != -1      # Search via method call or expression
True
>>> 'Ni' in line
True
>>> sub = 'Ni!\n'
>>> line.endswith(sub)         # End test via method call or slice
True
>>> line[-len(sub):] == sub
True
```

可以查看帮助来学习具体用法，比如 `help(S.method)` 或者 `help(str.method)`。
字符串方法有时候会比 `re` 模块速度更快。

---

**最初的 `string` 模块函数（ `3.X` 已废弃 ）**
之前字符串方法是作为 `string` 模块的函数而存在的。
后来到了 `Python 2.0`，它们才成为了字符串对象的方法，但是为了保持后向兼容，`string` 模块仍然被保留。
今天你应该**只使用字符串方法**。
`string` 模块在 `3.X` 中已经不存在了。
在 `Python 2.X` 中，一下两者等价：

- `X.method(arguments)`
- `string.method(X, arguments)`（需要事先 `import string`）

```
>>> S = 'a+b+c+'
>>> x = S.replace('+', 'spam')
>>> x
'aspambspamcspam'
```

等价于：

```
>>> import string
>>> y = string.replace(S, '+', 'spam')
>>> y
'aspambspamcspam'
```

在 `Python 3.X` 中的 `string` 模块仍然有一些常量和 `Template` 对象系统，它早于字符串的 `format` 方法，不是很好用。

---

# 7.6 字符串格式化表达式

一共有两种格式化字符串的方法（除了用的极少的 `string` 模块的 `Template` ）

- 字符串格式化表达式 `'...%s...' % (values)` 。基于 `C` 语言的 `printf` 模型。
- 字符串格式化方法调用 `'...{}...'.format(values)` 。这是在 `2.6` 和 `3.0` 中引入的新工具，一部分上借鉴了 `C#/.NET` 中的同名工具。

---

**字符串表达式基础**
格式化字符串：

- `%` 的左边放需要格式化的字符串，可以嵌入多个待转换目标，每个目标都以 `%` 开头。
- `%` 的右边提供需要插入的对象（多个对象需要放入一个元组）

```
>>> exclamation = 'Ni'
>>> 'The knights who say %s!' % exclamation      # String substitution
'The knights who say Ni!'
>>> '%d %s %g you' % (1, 'spam', 4.0)           # Type-specific substitu
tions
'1 spam 4 you'
>>> '%s -- %s -- %s' % (42, 3.14159, [1, 2, 3])  # All types match a %s t
arget
'42 -- 3.14159 -- [1, 2, 3]'
```

每种类型的对象都可以转换为字符串，因此每种对象都适用于 `%s` 转换。

---

**高级字符串表达式语法**
转换类型码 `conversion type code`
`C` 程序员会认得许多，因为支持大部分 `printf` 语句的格式码。

*Table 7-4. String formatting type codes*

| Code | Meaning |
|---|---|
| s | String (or any object's `str(X)` string) |
| r | Same as s, but uses `repr`, not `str` |
| c | Character (int or str) |
| d | Decimal (base-10 integer) |
| i | Integer |
| u | Same as d (obsolete: no longer unsigned) |
| o | Octal integer (base 8) |
| x | Hex integer (base 16) |
| X | Same as x, but with uppercase letters |
| e | Floating point with exponent, lowercase |
| E | Same as e, but uses uppercase letters |
| f | Floating-point decimal |
| F | Same as f, but uses uppercase letters |
| g | Floating-point e or f |
| G | Floating-point E or F |
| % | Literal % (coded as %%) |

转换目标的结构如下：
`%[(keyname)][flags][width][.precision]typecode`
在 `%` 和类型码之间你可以：

- 写一个 `(键名)`，然后在表达式的右边放一个字典
- 标志位：`-` 代表左对齐，`+` 可以给数字添加正负号，空格可以让正负数对齐（即正数前面是空格，负数前面是 `-`），`0` 可以填充零。
- 总体的最小字段宽度（jpch89：理解为**显示位宽**吧！）
- 精度
- 显示位宽和精度都可以写成 `*`，可以从表达式右边取值，它们的值在运行时决定。

**高级格式化表达式例子**
`g` 根据数字内容选择格式：

- 如果指数小于 `-4` 或者不小于精度，则使用 `e` 格式
- 其他情况使用小数格式 `f`
- 默认的总位数精度为 `6`

```
>>> x = 1.23456789
>>> x
1.23456789
>>> '%e | %f | %g' % (x, x, x)
'1.234568e+00 | 1.234568 | 1.23457'
>>> '%E' % x
'1.234568E+00'
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23   | 01.23 | +001.2'
>>> '%s' % x, str(x)
('1.23456789', '1.23456789')
```

字符串格式化表达式 `*` 的用法：

```
>>> '%f, %.2f, %*.*f' % (1/3.0, 1/3.0, 6, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

**基于字典的格式化表达式**
生成类似 `HTML` 或 `XML` 文本的程序往往利用这一技术。

```
>>> '%(qty)d more %(food)s' % {'qty': 1, 'food': 'spam'}
'1 more spam'
>>> reply = """
Greetings...
Hello %(name)s!
Your age is %(age)s
"""
>>> values = {'name': 'Bob', 'age': 40} # Build up values to substitute
>>> print(reply % values) # Perform substitutions
Greetings...
Hello Bob!
Your age is 40
```

这一技巧通常与内置函数 `vars()` 配合使用。

```
>>> food = 'spam'
```

```
>>> qty = 10
>>> '%(qty)d more %(food)s' % vars()
'10 more spam'
```

---

# 7.7 字符串格式化方法调用

> jpch89：其实就是 `format` 方法嘛

---

**字符串格式化方法基础**
在 主体字符串 中，花括号通过 *位置*（`{1}`），*关键字*（`{food}`），*相对位置*（`{}`，仅在 `3.1` 和 `2.7` 以及之后有效）来指定替换目标及将要插入的参数。

```
>>> template = '{0}, {1} and {2}' # By position
>>> template.format('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '{motto}, {pork} and {food}' # By keyword
>>> template.format(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'

>>> template = '{motto}, {0} and {food}' # By both
>>> template.format('ham', motto='spam', food='eggs')
'spam, ham and eggs'

>>> template = '{}, {} and {}' # By relative position
>>> template.format('spam', 'ham', 'eggs') # New in 3.1 and 2.7
'spam, ham and eggs'
```

与 字符串格式化表达式 的对比：
字符串格式化表达式使用字典，而字符串格式化 `format` 方法使用关键字参数

```
>>> template = '%(motto)s, %(pork)s and %(food)s'
>>> template % dict(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'
```

**添加键、属性和偏移量**
`.format()` 方法可以用：

- 方括号取得字典的键
- 方括号来执行序列的索引操作
- 点号取得对象属性

---

```
>>> import sys
>>> 'My {1[kind]} runs {0.platform}'.format(sys, {'kind': 'laptop'})
'My laptop runs win32'
>>> 'My {map[kind]} runs {sys.platform}'.format(sys=sys, map={'kind': 'la
ptop'})
'My laptop runs win32'
```

注意：偏移量必须是 单个正偏移量
如果要用到负偏移量或者切片操作，要放在格式化字符串外面完成。

```
>>> somelist = list('SPAM')
>>> somelist
['S', 'P', 'A', 'M']

>>> 'first={0[0]}, third={0[2]}'.format(somelist)
'first=S, third=A'

>>> 'first={0}, last={1}'.format(somelist[0], somelist[-1]) # [-1] fails
 in fmt
'first=S, last=M'

>>> parts = somelist[0], somelist[-1], somelist[1:3] # [1:3] fails in fmt
>>> 'first={0}, last={1}, middle={2}'.format(*parts) # Or '{}' in 2.7/3.1
+
"first=S, last=M, middle=['P', 'A']
```

---

**高级格式化方法语法**
`{fieldname component !conversionflag :formatspec}`
中间不含空格。

- `fieldname` 数字或者字段名。在 `2.7` 和 `3.1` 及其后续版本中，可以省略不写
- `component` 大于等于零个的 `.name` 或者 `[index]` 引用的字符串
- `!conversionflag` 如果出现以 `!` 开始。可以跟 `r`、`s`、`a`。对这个值分别调用 `repr()`，`str()` 或者 `ascii()` 内置函数。
- `:formatspec` 如果出现以冒号 `:` 开始。

`formatspec` 的格式如下：
`[[fill]align][sign][#][0][width][,][.precision][typecode]`

- `fill` 是除了 `{}` 以外的任意填充字符
- `align` 可以是 `<`、`>`、`=` 或者 `^`；分别表示左对齐、右对齐、符号字符 后的填充、居中对齐。
- `sign` 可以是 `+`、`-` 或者空格
- `,` 千分位分隔符
- `width` 和 `precision` 与 `%` 表达式用法一样
- `typecode` 与 `%` 表达式基本一样

> 可以参考我的这篇文章
> 478”>https://blog.csdn.net/jpch89/article/details/84099277#58478

单独的对象也可以用 `format(object, formatspec)` 来格式化。
在自定义类中，还可以使用 `__format__` 重载方法来定制。

---

**高级格式化方法举例**
进制转换（注意区别，没有 `0x` ，`0o` 这种前缀）

```
>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255) # Hex, octal, binary
'FF, 377, 11111111'
>>> bin(255), int('11111111', 2), 0b11111111 # Other to/from binary
('0b11111111', 255, 255)
>>> hex(255), int('FF', 16), 0xFF # Other to/from hex
('0xff', 255, 255)
>>> oct(255), int('377', 8), 0o377 # Other to/from octal, in 3.X
('0o377', 255, 255) # 2.X prints and accepts 0377
```

嵌套格式化语法（与格式化字符串表达式的 `*` 语法对比）

```
>>> '{0:.{1}f}'.format(1 / 3.0, 4) # Take value from arguments
'0.3333'
>>> '%.*f' % (4, 1 / 3.0) # Ditto for expression
'0.3333
```

`Python 2.6` 和 `3.0` 引入了一种新的内置 `format` 函数，它可以用来格式化单独的一项。
语法：`format(obj, '格式说明符')`
实际上，`format` 内置函数会运行主体对象的 `__format__` 方法。

```
# String method 字符串方法
>>> '{0:.2f}'.format(1.2345)
'1.23'

# Built-in function 内置函数
>>> format(1.2345, '.2f')
'1.23'

# Expression 表达式
>>> '%.2f' % 1.2345
'1.23'
```

---

### 与 `%` 格式化表达式比较

实际上，在通常的情况下，格式化表达式可能比格式化方法调用更容易编写。
另一方面，格式化方法提供了一些潜在的优点。
当进行更为复杂的格式化时，这两种方法的复杂性趋向等同。
注意 `%` 表达式没有二进制的表示方法：

```
>>> '%x, %o' % (255, 255)
'ff, 377'
```

注意 `%` 表达式默认右对齐。

```
# Hardcoded references in both
>>> import sys
>>> 'My {1[kind]:<8} runs {0.platform:>8}'.format(sys, {'kind':
'laptop'})
'My laptop runs win32'
>>> 'My %(kind)-8s runs %(plat)8s' % dict(kind='laptop', plat=sys.platfor
m)
'My laptop runs win32'
```

实际上很少直接向上面那样把替换内容硬编码，而是会事先做一个模板。

```
# Building data ahead of time in both
>>> data = dict(platform=sys.platform, kind='laptop')
>>> 'My {kind:<8} runs {platform:>8}'.format(**data)
'My laptop runs win32'
>>> 'My %(kind)-8s runs %(platform)8s' % data
'My laptop runs win32'
```

在 `Python 3.1` 和 `2.7` 中的 `format` 方法可以使用逗号作为分隔符。

```
>>> '{:,d}'.format(999999999999)
'999,999,999,999'
```

在之前的版本可以导入 `formats.py` 中的函数来实现这一点：

```
>>> from formats import commas, money
>>> '%s' % commas(999999999999)
'999,999,999,999'
>>> '%s' % money(296999.2567)
'$296,999.26'
```

在更加高级的环境中应用：

```
>>> [commas(x) for x in (9999999, 8888888)]
['9,999,999', '8,888,888']
>>> '%s %s' % tuple(commas(x) for x in (9999999, 8888888))
'9,999,999 8,888,888'
>>> ''.join(commas(x) for x in (9999999, 8888888))
'9,999,9998,888,888'
```

`format` 方法提供的额外功能，比如二进制表示，和逗号分隔符，有一些替代方式。

```
>>> '{0:b}'.format((2 ** 16) - 1) # Expression (only) binary format code
'1111111111111111'
>>> '%b' % ((2 ** 16) - 1)
ValueError: unsupported format character 'b'...

>>> bin((2 ** 16) - 1) # But other more general options work too
'0b1111111111111111'
>>> '%s' % bin((2 ** 16) - 1) # Usable with both method and % expression
'0b1111111111111111'
>>> '{}'.format(bin((2 ** 16) - 1)) # With 2.7/3.1+ relative numbering
'0b1111111111111111'

>>> '%s' % bin((2 ** 16) - 1)[2:] # Slice off 0b to get exact equivalent
'1111111111111111'
```

`8` 行的可重用函数可以实现与 `format` 方法的逗号分隔符等价的效果。

```
>>> '{:,d}'.format(999999999999) # New str.format method feature in 3.1/
2.7
'999,999,999,999'
>>> '%s' % commas(999999999999) # But % is same with simple 8-line functi
on
'999,999,999,999'
```

对键值的引用对比

```
>>> '{name} {job} {name}'.format(name='Bob', job='dev')
'Bob dev Bob'
>>> '%(name)s %(job)s %(name)s' % dict(name='Bob', job='dev')
'Bob dev Bob'

>>> D = dict(name='Bob', job='dev')
>>> '{0[name]} {0[job]} {0[name]}'.format(D) # Method, key references
'Bob dev Bob'
>>> '{name} {job} {name}'.format(**D) # Method, dict-to-args
'Bob dev Bob'
>>> '%(name)s %(job)s %(name)s' % D # Expression, key references
'Bob dev Bob'
```

使用格式化字符串，要格式化的元组必须嵌套在一个元组中：

```
>>> '%s' % 1.23 # Single value, by itself
'1.23'
>>> '%s' % (1.23,) # Single value, in a tuple
'1.23'
>>> '%s' % ((1.23,),) # Single value that is a tuple
'(1.23,)'
```

但是使用 `format` 方法，就不用做这样的嵌套：

```
>>> '{0:.2f}'.format(1.2345) # Single value
'1.23'
>>> '{0:.2f} {1}'.format(1.2345, 99) # Multiple values
'1.23 99'
>>> '{0}'.format(1.23) # Single value, by itself
'1.23'
>>> '{0}'.format((1.23,)) # Single value that is a tuple
'(1.23,)'
```

评判两种格式化方式孰优孰劣，更多地偏向于理论上的探讨，而不是实际编程时必须做出的选择。

格式化方法的最后一个小优势：它是一个函数，可以在表达式不能出现的地方使用。
简单来说，`str.format` 方法和 `format` 内置函数都可以作为参数传递进其他函数，或存储于其他对象中。
表达式如果要传递的话，要使用一行 `def` 函数定义或者 `lambda` 表达式包装一下。

```
def myformat(fmt, args): return fmt % args # See Part IV

myformat('%s %s', (88, 99)) # Call your function object
str.format('{} {}', 88, 99) # Versus calling the built-in

otherfunction(myformat) # Your function is an object too
```

其实有三种内置的格式化工具

- 字符串格式化表达式
- 字符串的 `foramt` 方法
- `string` 模块的 `Template` 工具

```
>>> import string
>>> t = string.Template('$num = $title')
>>> t.substitute({'num': 7, 'title': 'Strings'})
'7 = Strings'
>>> t.substitute(num=7, title='Strings')
'7 = Strings'
```

```
>>> t.substitute(dict(num=7, title='Strings'))
'7 = Strings'
```

---

# 7.8 通用类型分类

> 这个翻译的比较好：`in-place change` 在原位置修改

`Python` 有三大类型

- 数字（整数、浮点数、小数、分数等）
- 序列（字符串、列表、元组）
- 映射（字典）

集合是自成一派的分类。

同一分类中的类型共享同一个操作集
序列类型都支持序列操作：

- 索引
- 切片
- 重复（乘法）
- 拼接

可变类型能够在原位置修改。不可变的对象需要创建新的对象。

> 这里的位置指的是内存地址。

**不可变类别**：数字，字符串，元组，不可变集合 `frozenset`
**可变类别**：列表，字典，集合，字节数组 `bytearray`

---

# 7.9 本章小结

---

# 7.10 本章习题

1. 字符串的 `find` 方法可以用来搜索列表吗？
   不行。因为 方法是特定于类型的。然而表达式和内置函数是有一般性的，可以用于多种类型上。

2. 字符串切片表达式可以用于列表吗？
   可以。表达式具有一般性，可以用于多种类型。切片属于序列操作，序列操作可以在任何类型的序列对象上使用。
3. 如何将一个字符转换为 `ASCII` 整数码？如何进行反向转换，将一个整数转换为字符？
   `ord(S)` 可以把字符转换成 `ASCII` 码。
   `chr(I)` 可以将 `ASCII` 码转换成字符。
4. 在 `Python` 中，如何修改一个字符串？
   没办法，只能新建。
5. 已知字符串 `S` 的值为 `"s,pa,m"`，说出两种取出中间两个字符的方式。
   `S[2:4]`
   `S.split(',')[1]`
6. 字符串 `"a\nb\x1f\000d"` 中有多少个字符？
   `6` 个。
7. 为什么有时要使用 `string` 模块，而不是字符串方法调用？
   不要使用 `string` 模块，`Python 3.X` 已经废弃。
   但是可以用它的其他工具，比如预定义的常数。

---

# 8. 列表与字典

## 8.1 列表

列表的特点：

- *任意对象的有序集合*
- *通过偏移访问*
- *可变长度、异构以及任意嵌套*

  > 异构的 `heterogeneous`
  > 我觉得翻译成 *多样化的* 比较好

- *属于“可变序列”的分类*
- *对象引用数组*
  `Python` 列表包含了零个或多个其他对象的引用。
  类似其他语言中的 指针（地址）数组。
  在标准 `Python` 解释器内部，列表就是 `C` 数组。

`help(list)` 或者 `dir(list)` 查看 `list` 方法的完整列表清单。

*Table 8-1. Common list literals and operations*

| Operation | Interpretation |
|---|---|
| `L = []` | An empty list |
| `L = [123, 'abc', 1.23, {}]` | Four items: indexes 0..3 |
| `L = ['Bob', 40.0, ['dev', 'mgr']]` | Nested sublists |
| `L = list('spam')` | List of an iterable's items, list of successive integers |
| `L = list(range(-4, 4))` | |
| `L[i]` | Index, index of index, slice, length |
| `L[i][j]` | |
| `L[i:j]` | |
| `len(L)` | |
| `L1 + L2` | Concatenate, repeat |

| | |
|---|---|
| `L * 3` | |
| `for x in L: print(x)` | Iteration, membership |
| `3 in L` | |
| `L.append(4)` | Methods: growing |
| `L.extend([5,6,7])` | |
| `L.insert(i, X)` | |
| `L.index(X)` | Methods: searching |
| `L.count(X)` | |
| `L.sort()` | Methods: sorting, reversing, |
| `L.reverse()` | copying (3.3+), clearing (3.3+) |
| `L.copy()` | |
| `L.clear()` | |
| `L.pop(i)` | Methods, statements: shrinking |
| `L.remove(X)` | |
| `del L[i]` | |
| `del L[i:j]` | |
| `L[i:j] = []` | |
| `L[i] = 3` | Index assignment, slice assignment |
| `L[i:j] = [4,5,6]` | |
| `L = [x**2 for x in range(5)]` | List comprehensions and maps (Chapter 4, Chapt |
| `list(map(ord, 'spam'))` | |

# 8.2 列表的实际应用

`+` 左右两边必须是相同类型的序列，比如不能将列表和字符串拼接在一起。
列表推导的编码更简单，而且在今天的 `Python` 中似乎运行起来更快。
你需要 区分原位置修改一个对以及生成一个新对象的不同。

列表赋值分为：

- **索引赋值** `index assignment`
- **切片赋值** `slice assignment`
  - 删除：删除等号左边指定的切片
  - 插入：在被删除的位置插入等号右边的内容

注意：插入元素的数目不需要与删除元素的数目相匹配

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5] # Replacement/insertion
>>> L
[1, 4, 5, 3]
>>> L[1:1] = [6, 7] # Insertion (replace nothing)
>>> L
[1, 6, 7, 4, 5, 3]
>>> L[1:2] = [] # Deletion (insert nothing)
>>> L
[1, 7, 4, 5, 3]
```

切片赋值功能强大，可以用来替换、插入、删除，但在实践中并不常见。
程序员更喜欢用更加简单直接便于记忆的方式实现替换、插入以及删除。
比如 `insert`，`pop`，和 `remove` 列表方法。

**在头部插入和在尾部追加**

```
>>> L = [1]
>>> L[:0] = [2, 3, 4] # Insert all at :0, an empty slice at front
>>> L
[2, 3, 4, 1]
>>> L[len(L):] = [5, 6, 7] # Insert all at len(L):, an empty slice at end
>>> L
[2, 3, 4, 1, 5, 6, 7]
>>> L.extend([8, 9, 10]) # Insert all at end, named method
>>> L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]
```

方法 就是附属并作用于特定对象的函数。
实际上，方法是指向函数的对象属性。

`L.append(X)` 与 `L + [X]` 的结果类似。
只不过前者会原位置修改 `L`，而后者会生成新的列表。
由于 `append` 并不一定生成新的对象，所以它通常也比 `+` 更快。
切片赋值的写法：`L[len(L):] = [X]`。
还可以用 `L.insert(len(L), X)`

列表的 `sort` 方法默认使用了 *字符串比较*，可以应用于任何类型的对象，默认是升序。
`reverse=True` 参数，可以降序
`key` 指定了一个接收单个参数的函数，它返回在排序中使用的值

注意 `L.append` 和 `L.sort` 返回值为 `None`，它们都是在原位置修改列表，没有返回值，所以不能这么写 `L = L.append(X)`。

`sorted` 内置函数不在原位置修改列表，而是会把排序后的列表返回
`reversed` 内置函数逆转列表，不修改原列表，返回一个新的迭代器对象。

```
>>> L
[1, 2, 3, 4]
>>> L.reverse() # In-place reversal method
>>> L
[4, 3, 2, 1]
>>> list(reversed(L)) # Reversal built-in with a result (iterator)
[1, 2, 3, 4]
```

`extend` 与 `append` 的技术上的区别：

- `extend` 会循环访问传入的可迭代对象，并逐个把产生的元素添加到列表尾部
- `append` 会直接把传入的可迭代对象添加到尾部而不会遍历它

> 【注】：`append` 之类写传入的对象会不会好一点？

后进先出 `last-in-first-out, LIFO` 的栈结构可以通过 `pop` 方法和 `append` 方法联用实现。列表的末端作为栈的顶端。

`pop` 默认是删除并返回最后一个元素，即索引为 `-1`，也可以指定索引

切片删除：

```
>>> L = ['spam', 'eggs', 'ham', 'toast']
>>> del L[0] # Delete one item
>>> L
['eggs', 'ham', 'toast']
>>> del L[1:] # Delete an entire section
>>> L # Same as L[1:] = []
['eggs']
```

也可以把空列表赋值给切片进行切片删除。
然而如果将空列表赋值给一个索引，则会把该位置替换成一个空列表，而不是删除。

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

---

# 8.3 字典

字典当中的元素是通过键来存取的，而不是通过偏移来存取的。

字典的主要特性：

- 通过键而不是偏移量来读取
  有时候字典又叫做 *关联数组* `associative array` 或 *散列表* `hash`（尤其被其他脚本语言的用户这么称呼）
- 任意对象的无序集合
- 长度可变、异构、任意嵌套
- 属于 “可变映射” 类型
- 对象引用表（散列表）
  从本质上来讲，字典是作为散列表（支持快速检索的数据结构）来实现的。字典存储的是对象的引用，而不是对象的副本（除非你要求它这么做）。

一个空字典就是一对空的大括号。

*Table 8-2. Common dictionary literals and operations*

| Operation | Interpretation |
|---|---|
| `D = {}` | Empty dictionary |
| `D = {'name': 'Bob', 'age': 40}` | Two-item dictionary |
| `E = {'cto': {'name': 'Bob', 'age': 40}}` | Nesting |
| `D = dict(name='Bob', age=40)` | Alternative construction techniques: |
| `D = dict([('name', 'Bob'), ('age', 40)])` | keywords, key/value pairs, zipped key/value pairs, key lists |
| `D = dict(zip(keyslist, valueslist))` | |
| `D = dict.fromkeys(['name', 'age'])` | |
| `D['name']` | Indexing by key |
| `E['cto']['age']` | |
| `'age' in D` | Membership: key present test |
| `D.keys()` | Methods: all keys, |
| `D.values()` | all values, |
| `D.items()` | all key+value tuples, |
| `D.copy()` | copy (top-level), |
| `D.clear()` | clear (remove all items), |
| `D.update(D2)` | merge by keys, |
| `D.get(key, default?)` | fetch by key, if absent default (or None), |
| `D.pop(key, default?)` | remove by key, if absent default (or error) |
| `D.setdefault(key, default?)` | fetch by key, if absent set default (or None), |

| | |
|---|---|
| `D.popitem()` | remove/return any (key, value) pair; etc. |
| `len(D)` | Length: number of stored entries |
| `D[key] = 42` | Adding/changing keys |
| `del D[key]` | Deleting entries by key |
| `list(D.keys())` | Dictionary views (Python 3.X) |
| `D1.keys() & D2.keys()` | |
| `D.viewkeys(), D.viewvalues()` | Dictionary views (Python 2.7) |
| `D = {x: x*2 for x in range(10)}` | Dictionary comprehensions (Python 3.X, 2.7) |

`D.popitem()` 可以删除并返回键值对组成的元组。如果字典为空，抛出 `KeyError`。

`update()` 用法：

```
update(...) method of builtins.dict instance
    D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
    If E is present and has a .keys() method, then does:  for k in E: D
[k] = E[k]
    If E is present and lacks a .keys() method, then does:  for k, v in
E: D[k] = v
    In either case, this is followed by: for k in F:  D[k] = F[k]
```

`D.update([E, ]**F)`

- `E` 是可选的，它是一个可迭代对象。
- 如果 `E` 实现了 `.keys()` 方法，那么：`for k in E: D[k] = E[k]`
- 如果 `E` 没有 `.keys()` 方法，那么：`for k, v in E: D[k] = v`
- `**F` 是必

举例：

1. 使用 `**`

```
>>> D = {}
>>> d = {'a': 7, 'b': 4}
>>> D.update(**d)
>>> D
{'a': 7, 'b': 4}
```

2. 使用关键字参数

```
>>> D = {}
>>> D.update(a=7, b=4)
>>> D
{'a': 7, 'b': 4}
```

3. 使用字典

```
>>> D = {}
>>> d = {'a': 7, 'b': 4}
>>> D.update(d)
>>> D
{'a': 7, 'b': 4}
```

4. 混合

```
>>> d = {'a': 7, 'b': 4}
>>> D = {}
>>> D.update([('你', '好'), ('我', '好')] , **d)
>>> D
{'你': '好', '我': '好', 'a': 7, 'b': 4}
```

---

# 8.4 字典的实际应用

字典内部键由左至右的次序几乎总是和原先输入的顺序不同。
这样设计的目的是为了快速执行键查找（也就是散列查找），键会在内存中重新排序。
该顺序是伪随机的。

> 【注】最新版的 `Python` 字典总是按照输入顺序排列

用于字符串和列表的 `in` 成员关系测试同样适用于字典——它能够检查某个键是否存储在字典内。
技术上，这样行得通是因为 *字典定义了自动遍历键值列表的迭代器*。

> 【注】字典是一个可迭代对象。
> 书上翻译错了：
> Technically, this works because dictionaries define iterators
> that step through their keys lists automatically.
> 遍历的是 键列表，而不是键值列表！

在 `Python` 中，所有集合数据类型 `collection data types` 都可以彼此任意嵌套。

> 【注】这里翻译成 **容器数据类型** 会更好一些。`Python` 的集合是 `set`。

在 `3.X` 中，`D.items()`、`D.keys()`、`D.values()` 返回的都是可迭代对象，而不是列表。

获取不存在的键的值会报错，可以使用 `get` 方法，返回 `None` 或者自定义的默认值。

```
>>> D = {'spam': 2}
>>> D.get('spam') # A key that is there
2
>>> print(D.get('toast')) # A key that is missing
None
>>> D.get('toast', 88)
88
```

`update` 类似于拼接，如果键相同则覆盖原始值。

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

`pop` 必须接收一个键作为参数。会删除键值对，并返回值。

```
# pop a dictionary by key
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
>>> D.pop('muffin')
5
>>> D.pop('toast') # Delete and return from a key
4
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
```

`Python` 标准库中有可以维护键插入顺序的扩展类型，参见 `collections` 容器模块中的 `OrderedDict` 有序字典。它使用了额外的空间并降低了速度，以实现其功能。
在 `OrderedDict` 中，键被保存在一个冗余的链表中用来支持序列操作。

`for key in D` 和 `for key in D.keys()` 效果一样。

从值访问键：

```
>>> K = 'Holy Grail'
>>> table[K] # Key=>Value (normal usage)
'1975'
>>> V = '1975'
>>> [key for (key, value) in table.items() if value == V] # Value=>Key
['Holy Grail']
>>> [key for key in table.keys() if table[key] == V] # Ditto
['Holy Grail']
```

注意：字典不能使用相加来拼接，要使用 `update`

任何不可变对象都可以作为字典的键，不一定非要是字符串。
可以用整数作为键，这让字典在取值的时候很像列表的索引操作。
自定义类的实例对象也可以用作键，只要存在一个合理的协议方法，告诉 `Python` 其值是可散列的同时是不变的就行。

**用字典模拟灵活的列表：整数键**
整数键可以让字典变成一个更加灵活的列表，不需要事先分配空间。
比如不用 `[0] * 100`，可以直接按下面的做：

```
>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

**对稀疏数据结构使用字典：用元组做键**

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4 # ; separates statements: see Chapter 10
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

> 这一句有意思：使用分号来分隔语句
> `X = 2; Y = 3; Z = 4`

**避免键不存在错误**
读取不存在的键的错误在稀疏矩阵中很常见。
三种解决办法：

- 事先用 `if` 进行成员测试
- `try` 捕获异常
- `get` 方法为不存在的键提供默认值

```
>>> if (2, 3, 6) in Matrix: # Check for key before fetch
... print(Matrix[(2, 3, 6)]) # See Chapters 10 and 12 for if/else
... else:
... print(0)
...
0
>>> try:
```

```
... print(Matrix[(2, 3, 6)]) # Try to index
... except KeyError: # Catch and recover
... print(0) # See Chapters 10 and 34 for try/except
...
0
>>> Matrix.get((2, 3, 4), 0) # Exists: fetch and return
88
>>> Matrix.get((2, 3, 6), 0) # Doesn't exist: use default arg
0
```

**字典的嵌套**
一般来说，字典可以取代搜索数据结构，因为用键进行索引是一种搜索操作。
它可以扮演其他编程语言中的“记录”和“结构体”一类的角色。

`shelve` 工具，会自动地在对象和外部文件之间进行映射，从而使对象持久化。

**创建字典的其他方式**

```
# 如果可以事先一次性写出字典，这种方法不错
{'name': 'Bob', 'age': 40} # Traditional literal expression

# 如果要动态建立字典的字段，这种方法合适
D = {} # Assign by keys dynamically
D['name'] = 'Bob'
D['age'] = 40

# 所需代码量比字面量少，但是键全部都是字符串
dict(name='Bob', age=40) # dict keyword argument form

# 在程序运行时通过序列构建字典
dict([('name', 'Bob'), ('age', 40)]) # dict key/value tuples form
```

第三种方式通常与 `zip` 函数一起使用：

```
dict(zip(keyslist, valueslist)) # Zipped key/value tuples form (ahead)
```

这在解析数据文件的列时比较有用。（ `zip` 来连接键值 ）

`fromkeys` 方法，如果一开始初始值相同，可以用这个方法。
传入键列表，以及所有这些键的初始值。

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

`Python 3.X` 与 `2.7` 中可以用字典推导表达式。

**字典 vs 列表**
列表将元素赋值给位置，而字典却将元素赋值给更加便于记忆的键。
因此，字典数据通常为人类读者提供更多的信息。

集合，像是没有值的字典。

```
>>> D = {}
>>> D['state1'] = True # A visited-state dictionary
>>> 'state1' in D
True
>>> S = set()
>>> S.add('state1') # Same, but with sets
>>> 'state1' in S
True
```

**`Python 3.X` 和 `2.7` 中的字典变化**
字典推导在 `Python 3.X` 和 `2.7` 可用

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3])) # Zip together keys and values
[('a', 1), ('b', 2), ('c', 3)]
>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3])) # Make a dict from zip resu
lt
>>> D
{'b': 2, 'c': 3, 'a': 1}
```

字典推导的写法：

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

`fromkeys` 初始化与字典推导的对比：

```
>>> 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'}
>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'eggs': 'EGGS!', 'spam': 'SPAM!', 'ham': 'HAM!'}
```

`3.X` 中，`keys`，`values` 和 `items` 都返回字典视图对象，而在 `2.7` 中它们返回的是列表，要在 `2.7` 中也返回字典视图 `dictionary view` 则要用新的方法。

- 视图对象是可迭代对象。

- 字典视图还保持了字典成分的最初顺序
- 字典视图支持集合操作
- 但是不能索引、直接打印和使用列表的方法

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> K = D.keys() # Makes a view object in 3.X, not a list
>>> K
dict_keys(['b', 'c', 'a'])
>>> list(K) # Force a real list in 3.X if needed
['b', 'c', 'a']
>>> V = D.values() # Ditto for values and items views
>>> V
dict_values([2, 3, 1])
>>> list(V)
[2, 3, 1]
>>> D.items()
dict_items([('b', 2), ('c', 3), ('a', 1)])
>>> list(D.items())
[('b', 2), ('c', 3), ('a', 1)]
>>> K[0] # List operations fail unless converted
TypeError: 'dict_keys' object does not support indexing
>>> list(K)[0]
'b'
```

往往没有必要用 `dict.keys()` 因为字典本身就有迭代器，返回键。

在 `3.X` 中，字典的修改可以动态的反映到视图对象上，但是在 `2.X` 中，列表一旦创建，与字典就是分离开的。

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

>>> K = D.keys()
>>> V = D.values()
>>> list(K) # Views maintain same order as dictionary
['b', 'c', 'a']
>>> list(V)
[2, 3, 1]

>>> del D['b'] # Change the dictionary in place
>>> D
{'c': 3, 'a': 1}

>>> list(K) # Reflected in any current view objects
['c', 'a']
>>> list(V) # Not true in 2.X! - lists detached from dict
[3, 1]
```

`3.X` 的 `keys` 方法返回的字典视图对象类似集合，支持交集和并集等常见集合操作。但是 `values` 视图对象则不像集合，因为字典值是可以重复的，也可以是可变的。
`items` 的结果也像是集合，如果键值对是唯一且可散列的话。

```
>>> K, V
(dict_keys(['c', 'a']), dict_values([3, 1]))
>>> K | {'x': 4} # Keys (and some items) views are set-like
{'c', 'x', 'a'}
>>> V & {'x': 4}
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'
>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_val
ues'
```

在集合操作中，视图可以与其他的视图、集合和字典混合。其中字典被当做视图 `字典.keys()` 来处理

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D.keys() & D.keys() # Intersect keys views
{'b', 'c', 'a'}
>>> D.keys() & {'b'} # Intersect keys and set
{'b'}
>>> D.keys() & {'b': 1} # Intersect keys and dict
{'b'}
>>> D.keys() | {'b', 'c', 'd'} # Union keys and set
{'b', 'c', 'a', 'd'}
```

如果字典项视图是可散列的话（即只包含不可变对象），它们类似于集合。

> 字典键视图、字典值视图、字典项视图

```
>>> D = {'a': 1}
>>> list(D.items()) # Items set-like if hashable
[('a', 1)]
>>> D.items() | D.keys() # Union view and view
{('a', 1), 'a'}
>>> D.items() | D # dict treated same as its keys
{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)} # Set of key/value pairs
{('d', 4), ('a', 1), ('c', 3)}

>>> dict(D.items() | {('c', 3), ('d', 4)}) # dict accepts iterable sets t
oo
{'c': 3, 'a': 1, 'd': 4}
```

注意最后，`dict` 也接收可迭代的集合。

**字典键排序**

字典键视图对象不能直接 `.sort()`，要使用内置函数 `sorted`，它接收任何可迭代对象。

```
>>> Ks = list(Ks) # Force it to be a list and then sort
>>> Ks.sort()
>>> for k in Ks: print(k, D[k]) # 2.X: omit outer parens in prints
...
a 1
b 2
c 3

>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> Ks = D.keys() # Or you can use sorted() on the keys
>>> for k in sorted(Ks): print(k, D[k]) # sorted() accepts any iterable
... # sorted() returns its result
a 1
b 2
c 3

# 推荐直接使用字典的键迭代器!
# 2 3 版本通用!
>>> D
{'b': 2, 'c': 3, 'a': 1} # Better yet, sort the dict directly
>>> for k in sorted(D): print(k, D[k]) # dict iterators return keys
...
a 1
b 2
c 3
```

**`3.X` 中字典大小比较不再有效。**

模拟实现：`sorted(D1.items()) < sorted(D2.items()) # Like 2.X D1 < D2`

`3.X` 相等性测试仍然有效。`D1 == D2`

**在 `3.X` 中 `has_key` 方法已死，`in` 方法万岁！**

使用 `in` 成员测试或者带有默认值判断的 `get` 方法。通常建议使用 `in`。

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

>>> D.has_key('c') # 2.X only: True/False
AttributeError: 'dict' object has no attribute 'has_key'

>>> 'c' in D # Required in 3.X
True
>>> 'x' in D # Preferred in 2.X today
False
>>> if 'c' in D: print('present', D['c']) # Branch on result
```

```
...
present 3

>>> print(D.get('c')) # Fetch with default
3
>>> print(D.get('x'))
None
>>> if D.get('c') != None: print('present', D['c']) # Another option
...
present 3
```

**总结字典在 `3.X` 的变化**

- 字典推导只能在 `3.X` 和 `2.7` 中写
- 字典视图只能在 `3.X` 中写，在 `2.7` 则需要通过其他的方法名
- 排序、手动比较和 `has_key` 可以在 `2.X` 中编写，未来很容易的可以迁移到 `3.X`

---

# 8.5 本章小结

---

# 8.6 本章习题

1. 举出两种方式来创建内含 `5` 个整数零的列表。
   - `[0, 0, 0, 0, 0]` 字面量
   - `[0] * 5` 列表重复
   - `L.append(0)` 配合循环
   - `L = [0 for i in range(5)]`
2. 举出两种方式来创建一个字典，有两个键 `'a'` 和 `'b'`，而每个键相关联的值都是 `0`。
   - `{'a': 0, 'b': 0}` 字面量表达式
   - `D = {}; D['a'] = 0; D['b'] = 0`
   - `dict(a=0, b=0)` 关键字参数
   - `dict([('a', 0), ('b', 0)])` 键值对序列
   - `dict.fromkeys('ab', 0)`
   - `{k: 0 for k in 'ab'}` 字典推导
3. 举出四种在原位置修改列表对象的运算。
   - `append`，`extend`
   - `sort`，`reverse`
   - `insert`
   - `remove`，`pop`
   - `del`
   - 索引赋值和切片赋值
4. 举出四种在原位置修改字典对象的运算。

- 赋值
- `del`
- `update`
- `D.pop(key)`
- `setdefault`

5. 为什么在一些情况下你要使用字典而不是列表？
   如果数据带有标签的话，字典通常是更好的选择。
   列表最适合于无标签项目的集合。
   字典查找通常比列表搜索更快。

---

# 9. 元组、文件与其他核心类型

## 9.1 元组

元组的特性：

- 任意对象的有序集合
- 通过偏移量存取
- 属于 “不可变序列”
- 固定长度、多样性、任意嵌套
- 对象引用的数组

*Table 9-1. Common tuple literals and operations*

| Operation | Interpretation |
| --- | --- |
| `()` | An empty tuple |
| `T = (0,)` | A one-item tuple (not an expression) |
| `T = (0, 'Ni', 1.2, 3)` | A four-item tuple |
| `T = 0, 'Ni', 1.2, 3` | Another four-item tuple (same as prior line) |

| Operation | Interpretation |
| --- | --- |
| `T = ('Bob', ('dev', 'mgr'))` | Nested tuples |
| `T = tuple('spam')` | Tuple of items in an iterable |
| `T[i]` | Index, index of index, slice, length |
| `T[i][j]` | |
| `T[i:j]` | |
| `len(T)` | |
| `T1 + T2` | Concatenate, repeat |
| `T * 3` | |
| `for x in T: print(x)` | Iteration, membership |
| `'spam' in T` | |
| `[x ** 2 for x in T]` | |
| `T.index('Ni')` | Methods in 2.6, 2.7, and 3.X: search, count |
| `T.count('Ni')` | |
| `namedtuple('Emp', ['name', 'jobs'])` | Named tuple extension type |

**元组的特殊语法：逗号和圆括号**

```
>>> x = (40) # An integer!
>>> x
40
>>> y = (40,) # A tuple containing an integer
>>> y
(40,)
```

不引起歧义的情况下，可以忽略圆括号。
在两种情况下，不能省略圆括号

- 圆括号是必要的：比如在函数调用中，或者嵌套在更大的表达式中
- 逗号是必要的：比如嵌套在更大的列表、字典中，或者用于 `2.X` 的 `print` 语句中

对元组排序：

- 转换成列表，使用列表的 `sort` 方法，然后转换成元组
- 使用 `sorted(元组)`，注意它返回的是 *列表*，还要转换成元组

列表推导也可以把元组转成列表。列表推导可以被视作一种 *迭代工具*。

在 `2.6` 和 `3.0` 以前，元组根本没有方法。

```
>>> T = (1, 2, 3, 2, 4, 2) # Tuple methods in 2.6, 3.0, and later
```

```
>>> T.index(2) # Offset of first appearance of 2
1
>>> T.index(2, 2) # Offset of appearance after offset 2
3
>>> T.count(2) # How many 2s are there?
3
```

注意：元组的不可变性只适用于元组本身顶层，并非其内容。
即元组具有单层深度的不可变性。

**为什么有了列表还需要元组**

`Python` 的设计者是个数学家。
元组借用自数学领域，同时也用来指代关系数据表中的一行。
因为元组的不可变性提供了某种一致性。

**有名元组**

注意有名元组按属性访问，不是按键索引，跟字典不一样。
`collections.namedtuple`

```
>>> from collections import namedtuple # Import extension type
>>> Rec = namedtuple('Rec', ['name', 'age', 'jobs']) # Make a generated c
lass
>>> bob = Rec('Bob', age=40.5, jobs=['dev', 'mgr']) # A named-tuple recor
d
>>> bob
Rec(name='Bob', age=40.5, jobs=['dev', 'mgr'])

>>> bob[0], bob[2] # Access by position
('Bob', ['dev', 'mgr'])
>>> bob.name, bob.jobs # Access by attribute
('Bob', ['dev', 'mgr'])
```

还可以转换成一个字典 `OrderedDict`：

```
>>> O = bob._asdict() # Dictionary-like form
>>> O['name'], O['jobs'] # Access by key too
('Bob', ['dev', 'mgr'])
>>> O
OrderedDict([('name', 'Bob'), ('age', 40.5), ('jobs', ['dev', 'mgr'])])
```

有名元组继承扩展了元组类，对每个有名字段插入了 `property` 访问器（`property accessor`）方法，将他们的名字映射到对应的序号。

元组与有名元组都支持解包元组赋值。

# 9.2 文件

*Table 9-2. Common file operations*

| Operation | Interpretation |
| --- | --- |
| `output = open(r'C:\spam', 'w')` | Create output file (`'w'` means write) |
| `input = open('data', 'r')` | Create input file (`'r'` means read) |
| `input = open('data')` | Same as prior line (`'r'` is the default) |
| `aString = input.read()` | Read entire file into a single string |
| `aString = input.read(N)` | Read up to next N characters (or bytes) into a string |
| `aString = input.readline()` | Read next line (including `\n` newline) into a string |
| `aList = input.readlines()` | Read entire file into list of line strings (with `\n`) |
| `output.write(aString)` | Write a string of characters (or bytes) into file |
| `output.writelines(aList)` | Write all line strings in a list into file |
| `output.close()` | Manual close (done for you when file is collected) |
| `output.flush()` | Flush output buffer to disk without closing |
| `anyFile.seek(N)` | Change file position to offset N for next operation |
| `for line in open('data'): use line` | File iterators read line by line |
| `open('f.txt', encoding='latin-1')` | Python 3.X Unicode text files (`str` strings) |
| `open('f.bin', 'rb')` | Python 3.X bytes files (`bytes` strings) |
| `codecs.open('f.txt', encoding='utf8')` | Python 2.X Unicode text files (`unicode` strings) |
| `open('f.bin', 'rb')` | Python 2.X bytes files (`str` strings) |

```
afile = open(filename, mode)
afile.method()
```

第三个参数是可选参数，用来控制输出缓冲。

- `0` 输出无缓冲。调用写入方法的时候立即传给外部文件。

**使用文件**

- 文件迭代器最适合逐行读取
  *从文本文件逐行读取的最佳方式就是根本不要读取该文件*
- 内容是字符串，不是对象
  读出来的是字符串，写入的时候也必须是字符串
- 文件是被缓冲的以及可定位的
  默认输出文件总是被缓冲，不会立即自动从内存转入硬盘。
  除非关闭或者 `flush`。
  `seek` 可以按字节跳转。
- `close` 通常是可选的：回收时自动关闭
  `close` 之后：终止与外部文件的连接、释放操作系统的资源，将内存中的缓冲内容输出到磁

盘并清空缓冲区。

> 【注】但是最好还是手动关闭，或者用上下文管理器吧！

**文件的实际应用**
空字符串就是文件的结尾（文件的空行是 `\n` 而不是空字符串）
`write` 在 `3.X` 中会返回写入的字符数，而 `2.X` 不会有返回。

```
>>> myfile = open('myfile.txt', 'w') # Open for text output: create/empty
>>> myfile.write('hello text file\n') # Write a line of text: string
16
>>> myfile.write('goodbye text file\n')
18

>>> myfile.close() # Flush output buffers to disk
>>> myfile = open('myfile.txt') # Open for text input: 'r' is default
>>> myfile.readline() # Read the lines back
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline() # Empty string: end-of-file
''
```

**文本和二进制文件**
在 `3.X` 中：

- 文本文件把文件内容当成 `str` 字符串，自动执行 `Unicode` 编码和解码，并且默认执行末行转换。`perform end-of-line translation by default.`

  > 【注】啥叫末行转换？
  > 末行转换就是将换行符 `\n` 转义。

- 二进制文件把内容当做特殊的字节串类型 `bytes`，允许程序不修改地访问文件内容

  > 【？】不修改的访问文件内容？

在 `3.X` 中，如果以文本模式打开了二进制文件，会出错，因为没法将其内容解码成 `Unicode` 文本。

交互模式下打开文件，回显会直接显示 `\n`。使用 `print` 会把 `\n` 解析成换行符，更加用户友好。
文件最后一行末尾有时候没有换行符。

**如何去掉 `\n`？**

- `line.rstrip()`
- `line[:-1]`
- `int(line)`：`int` 转换直接忽略字符串左右的空白

```
>>> num = '\t 444\n'
>>> int(num)
444
```

还可以使用 `eval` 从文本还原对象：

```
>>> line = F.readline()
>>> line
"[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> parts = line.split('$') # Split (parse) on $
>>> parts
['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]
>>> eval(parts[0]) # Convert to any object type
[1, 2, 3]
>>> objects = [eval(P) for P in parts] # Do same for all in list
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

**使用 `pickle` 存储 `Python` 原生对象**

`eval` 在无法信赖文件来源的时候，不能随意使用。
`pickle` 模块可以将原生对象序列化 `object serialization` 成一种串形式。
`shelve` 模块用 `pickle` 把 `Python` 对象存放到一种按键访问的文件系统中。

在 `3.X` 中，必须以二进制打开用 `pickle` 操作的文件。
`Python 2.X` 有一个 `cPickle` 模块，是 `pickle` 的优化版，可以直接快速导入。
`3.X` 中这个模块叫做 `_pickle`，`pickle` 模块会自动使用它。

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F) # Pickle any object to file
>>> F.close()

>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F) # Load any object from file
>>> E
{'a': 1, 'b': 2}
```

**用 `JSON` 格式存储 `Python` 对象**

`MongoDB` 将数据存储在一个 `JSON` 文件的数据库中（它使用二进制 `JSON` 格式）。

```
>>> rec
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age':
 40.5}

# json.dumps(obj) 把 Python 对象 dumps 成 json 字符串
>>> import json
```

```
>>> json.dumps(rec)
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "ag
e": 40.5}'
>>> S = json.dumps(rec)
>>> S
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "ag
e": 40.5}'

# json.loads(json字符串) 把 json 字符串 loads 成 Python 对象
>>> O = json.loads(S)
>>> O
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age':
 40.5}
>>> O == rec
True
```

从文件中读取和写入 `JSON` 格式数据。

```
>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)
>>> print(open('testjson.txt').read())
{
    "job": [
        "dev",
        "mgr"
    ],
    "name": {
        "last": "Smith",
        "first": "Bob"
    },
    "age": 40.5
}
>>> P = json.load(open('testjson.txt'))
>>> P
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age':
 40.5}
```

`JSON` 文件中的所有字符串都是 `Unicode` 编码的。
`2.X` 中转换过来的字符串前面带一个 `u`

`CSV` 是 `comma-separated value` 逗号分隔值

```
>>> import csv
>>> rdr = csv.reader(open('csvdata.txt'))
>>> for row in rdr: print(row)
...
['a', 'bbb', 'cc', 'dddd']
['11', '22', '33', '44']
```

**存储打包的二进制数据 `struct`**

`struct` 模块可以把文件中的字符串转换为二进制数据。
它可以构造并解析打包二进制数据。

```
>>> F = open('data.bin', 'wb') # Open binary output file
>>> import struct
>>> data = struct.pack('>i4sh', 7, b'spam', 8) # Make packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(data) # Write byte string
>>> F.close()

>>> F = open('data.bin', 'rb')
>>> data = F.read() # Get packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> values = struct.unpack('>i4sh', data) # Convert to Python objects
>>> values
(7, b'spam', 8)
```

其中 `>i4sh` 是格式化字符串，指定了存储内容的格式。

**文件上下文管理器**

```
with open(r'C:\code\data.txt') as myfile: # See Chapter 34 for details
    for line in myfile:
        ...use line here...
```

多 `3` 行代码的 `try/finally` 语句（如果不 `close` 那就多 `2` 行代码）

```
myfile = open(r'C:\code\data.txt')
try:
    for line in myfile:
        ...use line here...
    finally:
        myfile.close()
```

**其他文件工具**

`2.X` 可以 `dir(file)` 或者 `help(file)`，但是 `3.X` 不可以。
**注意**：`2.X` 中 `open` 和 `file` 通用。

其他类似的文件工具：

- 标准流（`sys.stdout`）
- `os` 模块中的描述文件
  `x` 模式，新建，如果已有则抛出异常
- 套接字、管道和 `FIFO` 文件
  用于同步进程或者通过网络进行通信的类文件对象

- 通过键来存取的文件，如 `shelve` 模块
- `Shell` 命令流
  `os.open` 或者 `subprocess.Popen`

---

# 9.3 核心类型复习与总结

数字包括：整数、（2.X 的长整数）、浮点数、复数、小数和分数

`bytearray` 可变（3.X，2.6及以上）

*Table 9-3. Object classifications*

| Object type | Category | Mutable? |
|---|---|---|
| Numbers (all) | Numeric | No |
| Strings (all) | Sequence | No |
| Lists | Sequence | Yes |
| Dictionaries | Mapping | Yes |
| Tuples | Sequence | No |
| Files | Extension | N/A |
| Sets | Set | Yes |
| Frozenset | Set | No |
| bytearray | Sequence | Yes |

**引用 vs 复制**
你可以在程序范围内人和也地方传递大型对象而不必在途中进行开销巨大的复制操作。
如果需要复制：

- 切片 `[:]`
- 字典、集合或列表的 `copy` 方法（`3.3` 新增了列表的 `copy` 方法）
- 内置函数 `list(L)`、`dict(D)`、`set(S)`
- `copy` 标准库模块的 `copy`

上述方法只能进行顶层复制，如果要进行深层复制：
`copy.deepcopy(X)` 它可以递归地遍历对象来复制它们所有的组件

**比较、等价性和真值**
嵌套对象存在时，`Python` 会自动遍历数据结构，进行比较。

有时候叫做 *递归比较*。

- `==` 测试值的等价性
- `is` 测试对象的同一性（是否有相同地址）

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
>>> S1 == S2, S1 is S2
(True, False)
```

> 【！】我这里是 `True` ！

比较流程：

- 数字转换成必要的公共最高类型，比较大小
- 字符串按照字母顺序比较（`ord` 返回的数字比较）
- 列表和元组从左到右递归比较
- 集合只有在含有相同元素的情况下才是相等的（互为子集）
- 字典比较排序之后的 `(key, value)` 列表（`3.X` 不支持）
- 非数字与数字比较，会出错（`2.X` 允许）

**混合比较和排序**

```
# 2.X
c:\code> c:\python27\python
>>> 11 == '11' # Equality does not convert non-numbers
False
>>> 11 >= '11' # 2.X compares by type name string: int, str
False
>>> ['11', '22'].sort() # Ditto for sorts
>>> [11, '11'].sort()

# 3.X
c:\code> c:\python33\python
>>> 11 == '11' # 3.X: equality works but magnitude does not
False
>>> 11 >= '11'
TypeError: unorderable types: int() > str()
>>> ['11', '22'].sort() # Ditto for sorts
>>> [11, '11'].sort()
TypeError: unorderable types: str() < int()

>>> 11 > 9.123 # Mixed numbers convert to highest type
True
>>> str(11) >= '11', 11 >= int('11') # Manual conversions force the issue
(True, True)
```

`3.X` 移除了字典的比较，因为大小比较会导致很大的计算开销。
`3.X` 等价性比较不是直接比较排序后的键值对列表，而是采用了一种优化的方案。
替代方案：

```
>>> list(D1.items())
[('b', 2), ('a', 1)]
>>> sorted(D1.items())
[('a', 1), ('b', 2)]
>>>
>>> sorted(D1.items()) < sorted(D2.items()) # Magnitude test in 3.X
True
>>> sorted(D1.items()) > sorted(D2.items())
False
```

**`Python` 中 `True` 和 `False` 的含义**

- 整数 `0` 代表假，整数 `1` 代表真
- 任何空数据结构视为假，非空数据结构视为真
- 每个对象非真即假

*Table 9-4. Example object truth values*

| Object | Value |
|---|---|
| "spam" | True |
| "" | False |
| [1, 2] | True |
| [] | False |
| {'a': 1} | True |
| {} | False |
| 1 | True |
| 0.0 | False |
| None | False |

建议不要比较空的同类型对象：`if X != '':`
而是要编写 `if X:` 这样的语句，测试对象自身看它是否包含任何内容

`None` 是一种特殊数据类型的唯一值，一般起到占位符的作用，类似 `C` 语言中的 `NULL` 指针。

比如用来初始化列表：

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ... ]
```

`None` 不是 `undefined` 未定义，它是某些内容，而不是没有内容。
它是一个真实的对象，有一块真实的内存。

`bool` 类型
内置的 `True` 和 `False` 关键字只是整数 `1` 和 `0` 的定制版本而已。

- 当用在真值测试时，`True` 和 `False` 关键字变成了 `1` 和 `0`。但它们是程序员的意图更明确。
- 交互式运行布尔测试的结果被打印成单词 `True` 和 `False`，使得程序的结果更明确。

`bool` 可以显式地测试某个对象的布尔值。

**`Python` 的类型层次**
类型本身也是一种类型的对象，它们是 `type` 类型的对象。
`3.X` 中，类型与类合并，所有类型都是类。比如 `dict`，`list` 等等。
调用这些名称，相当于调用类的构造函数。

`3.X` 的 `types` 标准库模块中，包含了非内置类型的名称。

```
# 3.6.6
>>> import types
>>> dir(types)
['AsyncGeneratorType', 'BuiltinFunctionType', 'BuiltinMethodType', 'CodeT
ype', 'CoroutineType', 'DynamicClassAttribute', 'FrameType', 'FunctionTyp
e', 'GeneratorType', 'GetSetDescriptorType', 'LambdaType', 'MappingProxyT
ype', 'MemberDescriptorType', 'MethodType', 'ModuleType', 'SimpleNamespac
e', 'TracebackType', '_GeneratorWrapper', '__all__', '__builtins__', '__c
ached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
 '__spec__', '_ag', '_calculate_meta', '_collections_abc', '_functools',
'coroutine', 'new_class', 'prepare_class']
```

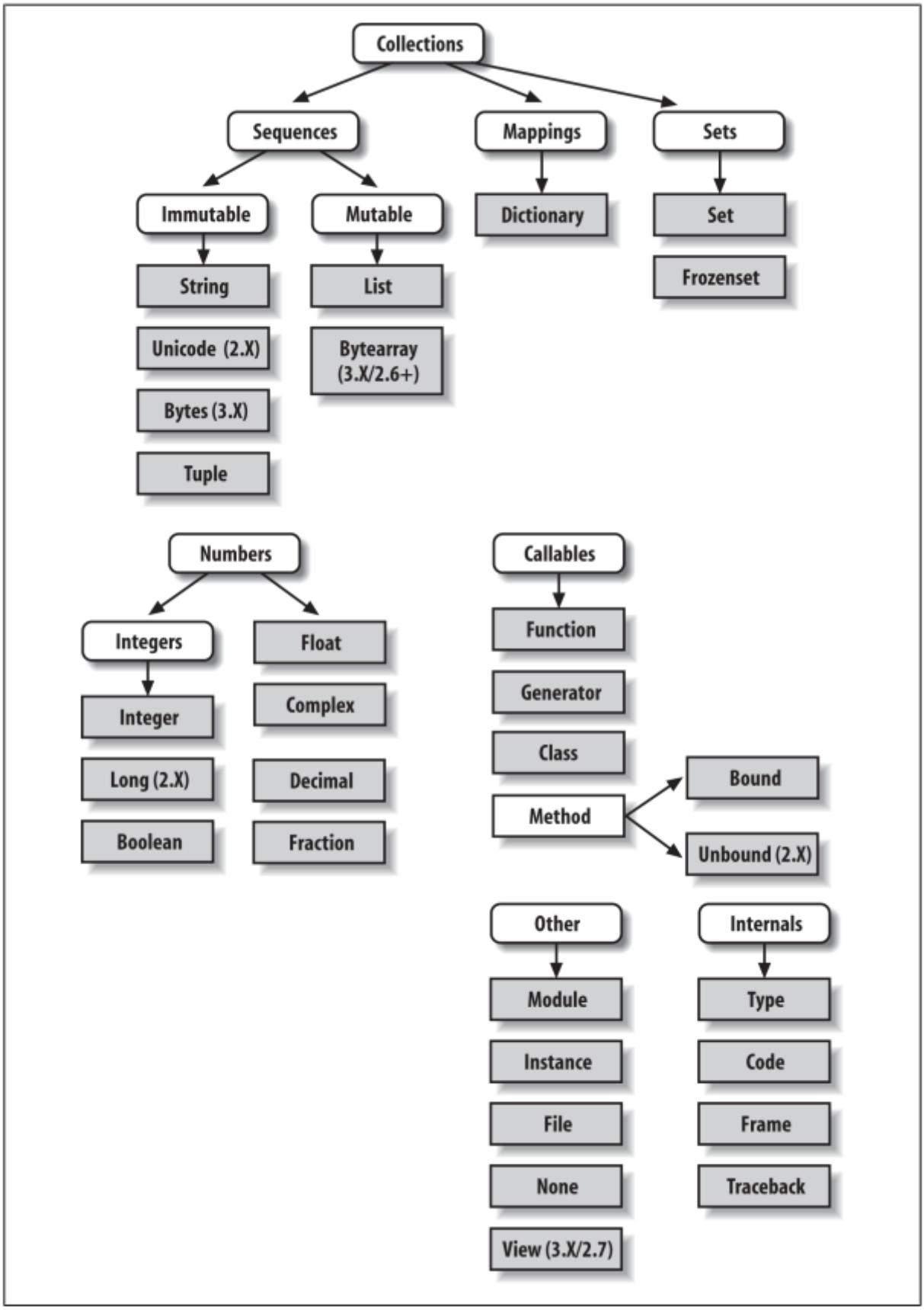
举例：

```
type([1]) == type([])    # Compare to type of another list
type([1]) == list        # Compare to list type name
isinstance([1], list)    # Test if list or customization thereof

import types             # types has names for other types
def f(): pass
type(f) == types.FunctionType
```

对于 `3.X` 和 `2.X` 的新式类来说，一个类实例的类型就是生成该实例的那个类。
对于 `2.X` 的经典类来说，所有的类实例都是 “实例” 类型的对象。

所以如果要比较实例的类型，需要比较它们的 `__class__` 属性。




内置类型与其他数据类型之间的区别主要在于，内置类型有针对它们的对象的特殊创建语法。而其他在标准库中通过 `import` 导入进来的类型，通常不被认为是核心类型。

# 9.4 内置类型陷阱

赋值创建引用，而不是复制。

切片参数默认为 `0` 以及被切片序列的长度；如果两者都省略，切片就会抽取序列中的每一项，并创建一个顶层复制。

```
>>> L = [4, 5, 6]
>>> X = L * 4 # Like [4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4 # [L] + [L] + ... = [L, L,...]

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]

# 列表嵌套在另一个列表中时重复，修改原列表会出现联动修改副作用
>>> L[1] = 0 # Impacts Y but not X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]

# 要把嵌套的列表浅拷贝，才会消除与原列表之间的联动修改副作用
>>> L = [4, 5, 6]
>>> Y = [list(L)] * 4 # Embed a (shared) copy of L
>>> L[1] = 0
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]

# 但是重复的每一个列表，引用都一样，还是会出现联动修改副作用
>>> Y[0][1] = 99 # All four copies are still the same
>>> Y
[[4, 99, 6], [4, 99, 6], [4, 99, 6], [4, 99, 6]]

# 真正消除联动修改的方法，是为每一次重复都新建一次拷贝
>>> L = [4, 5, 6]
>>> Y = [list(L) for i in range(4)]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
>>> Y[0][1] = 99
>>> Y
[[4, 99, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

注意：重复、拼接和切片只是复制操作数对象的顶层。

**循环数据结构**
如果一个复合对象包含指向自身的引用，就称之为循环对象 `cyclic object`。
检测到循环对象会打印成 `[...]`，而不会陷入无限循环（早期版本会）

```
>>> L = ['grail'] # Append reference to same object
>>> L.append(L) # Generates cycle in object: [...]
>>> L
['grail', [...]]
```

除非真的需要，否则不要用循环引用，同时确保你在相应的程序中考虑到了这种情况。

不可变类型不可以在原位置改变。
如果要改变元组需要新建一个。

```
T = (1, 2, 3)
T[2] = 4 # Error!
T = T[:2] + (4,) # OK: (1, 2, 4)
```

---

# 9.5 本章小结

---

# 9.6 本章习题

1. 如何确定元组的大小？为什么这个工具是内置工具？
   `len` 返回任何容器对象的长度。因为它适用于多种对象。

   > 容器是 `container`，那么 `collections` 模块到底怎么翻译比价好？集合体？

2. 编写一个修改元组中第一个元素的表达式。把 `(4, 5, 6)` 变成 `(1, 5, 6)`。
   也可以转换成列表再修改。
   但是记住：如果需要在原位置修改，就用列表。

```
T = (4, 5, 6)
T = (1, ) + T[1:]
```

3. 文件的 `open` 方法，处理模式的参数默认是什么？
   `r`

   > 实际上是 `rt` 吧。

4. 什么模块可以存储 `Python` 对象到文件，而不需要亲自把它们转换成字符串？
   `pickle` 可以把 `Python` 对象存储到文件。
   `struct` 模块类似，但是它嘉定数据在文件中被打包成二进制形式。
   `json` 可以在 `Python` 对象与 `JSON` 文本之间转换。

5. 如何一次性递归地赋值嵌套结构的所有组成部分？
   `copy.deepcopy(X)`
6. `Python` 在什么情况下会判断一个对象为真？
   非零数字、非空对象、非 `None` 就是真。

   > 非零非空非 `None` ！

7. 我们的目标是什么？

   > 成为海贼王？

---

# 9.7 第二部分练习题

1. 重现下列表达式
   注意：分号用作语句分隔符。`X=1;X` 会赋值并打印 `X`

```
>>> ('x', )[0]
'x'
>>> {'a': 1, 'b': 2}['a']
1
```

2. 索引和切片运算
   偏移量不能越界。
   切片可以越界，是因为 `Python` 会调整越界的切片，比如 `[-1000:100]`，切片的上下界别调整为 `0` 和序列的长度。
   左切片比右切片大时，会让右切片等于左切片。`[3:1]` 会调整成 `[3:3]`。

```
>>> L = [1, 2, 3, 4]

>>> L[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]

>>> L[3:1]
[]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. 索引、切片以及 `del`
   切片赋值会删除切片，然后在原位置插入右侧的新值。

切片赋值右边需要一个序列，而不是单个元素，给单个元素会报错。
`del` 也可以删除切片 `del L[1:]`

4. 元组用来交换变量
   左侧的东西并不是元组，而是一组独立的赋值目标。
   右侧才是元组，它在赋值的过程中被解包。

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
>>> X
'eggs'
>>> Y
'spam'
```

5. 字典键
6. 字典索引运算
   变量名称与字典键的工作方式一样。
   在引用之前必须已经被赋值，而在首次赋值时会被创建。
   你可以把变量名当做是字典键。它们在模块命名空间 `module namespace` 或栈帧字典 `stack-frame dictionaries` 中都是可见的。
7. 通用操作
   a. 在混合类型之间使用 `+` 会报错
   b. 当操作数之一是字典时，`+` 运算符不能工作（要用 `update`）
8. 字符串索引运算

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'

# 如果列表的元素不是字符串，就不能这样连续索引
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. 不可变类型
   把 `S = 'spam'` 变成 `'slam'`。
   注意：切片赋值不可以，因为字符串不可变

```
# 索引操作加切片
>>> S = "spam"
>>> S = S[0] + 'l' + S[2:]
>>> S
'slam'

# 索引操作
>>> S = S[0] + 'l' + S[2] + S[3]
>>> S
```

```
'slam'
```

10. 嵌套
11. 文件

```
with open('testfile.txt', 'w') as f:
    print('啦啦啦', file=f)

with open('testfile.txt') as f:
    print(f.read())
```

---

完成于 `2018.12.29`