

[笔记][Learning Python][2. 类型和运算]

腾蛇起陆

Python

[笔记][Learning Python][2. 类型和运算]

4. 介绍 Python 的对象类型

4.1 Python 概念层级

4.2 为什么要使用内置类型？

4.3 Python 的核心数据类型

4.4 数字

4.5 字符串

4.6 列表

4.7 字典

4.8 元组

4.9 文件

4.10 其它核心类型

4.11 章节总结

4.12 检验你的知识

5. 数字类型

5.1 数字类型基础

5.2 实际应用中的数字

5.3 其它数字类型

5.4 数字扩展

5.5 章节总结

5.6 检验你的知识

- 6. 动态类型简介
 - 6.1 缺少声明语句的情况
 - 6.2 共享引用
 - 6.3 随处可见的动态类型
 - 6.4 章节总结
 - 6.5 检验你的知识
 - 7. 字符串基础
 - 7.1 本章范围
 - 7.2 字符串基础
 - 7.3 字符串写法
-

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 a
nd 3.1)
6.283000000000000004
>>> 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
'd'
'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 # S is unchanged
'SpamSpam'
>>> 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 expression
s to make new objects
>>> S
'zspam'
```

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 another
'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,cccc,dd'
>>> line.split(',') # Split on a delimiter into
a list of substrings
['aaa', 'bbb', 'cccc', 'dd']
>>> S = 'spam'
>>> S.upper() # Upper- and lowercase conversion
S
'SPAM'
>>> S.isalpha() # Content tests: isalpha, isdig
it, etc.
True
>>> line = 'aaa,bbb,cccc,dd\n'
>>> line.rstrip() # Remove whitespace character
s on the right side
'aaa,bbb,cccc,dd'
>>> line.rstrip().split(',') # Combine two oper
ations
['aaa', 'bbb', 'cccc', '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!') # Fo
rmatting 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, p
adding, 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__\n__', '__iter__', '__le__',\n '__len__', '__lt__', '__mod__', '__mul__', '__n\n e__', '__new__', '__reduce__',\n '__reduce_ex__', '__repr__', '__rmod__', '__rmu\n l__', '__setattr__', '__sizeof__',\n '__str__', '__subclasshook__', 'capitalize', 'c\n asefold', 'center', 'count',\n 'encode', 'endswith', 'expandtabs', 'find', 'fo\n rmat', 'format_map', 'index',\n 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'i\n sidentifier', 'islower',\n 'isnumeric', 'isprintable', 'isspace', 'istitle\n ', 'isupper', 'join', 'ljust',\n 'lower', 'lstrip', 'maketrans', 'partition', 'r\n eplace', 'rfind', 'rindex',\n 'rjust', 'rpartition', 'rsplit', 'rstrip', 'spl\n it', 'splitlines', 'startswith',\n 'strip', 'swapcase', 'title', 'translate', 'upp\n er', '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!'\n'spamNI!'\n>>> S.__add__('NI!')\n'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 ta
```



```

b
>>> 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'bbbbbb'bbbb
cccccccccccccc
"""
>>> msg
'\naaaaaaaaaaaa\nbbb'\ ' ' 'bbbbbbbbbb'bbbbbb\nbbb\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\xfe\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 by
tes to str: 'xy'
'x'.encode() + b'y' # Works in 3.X if encode st
r 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', 'H
ello 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/append 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 i
tem 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 sums
{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 k
    eys are unique
{'p': 112, 'a': 97, 's': 115, 'm': 109}
>>> (ord(x) for x in 'spaam') # Generator of va
    lues
<generator object <genexpr> at 0x000000000254DA
    B0>
```

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])) # Zipping
>>> 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 dict
ionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}
>>> D['f'] # Referencing a nonexistent key is a
n 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
    squares.append(x ** 2) # Both run the i
    teration protocol internally
>>> 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 of
fset 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 write)
>>> f.write('Hello\n') # Write strings of chara
cters to it
6
>>> f.write('world\n') # Return number of items
written in Python 3.X
6
>>> f.close() # Close to flush output buffers t
o 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 de
fault processing mode
>>> text = f.read() # Read entire file into a s
tring
>>> text
'Hello\nworld\n'
>>> print(text) # print interprets control char
acters
Hello
world
>>> text.split() # File content is always a str
```

```
ing  
['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使用方法，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', 'flush',  
'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable',  
'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable',  
'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 data
>>> 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 da
ta
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\xfe\x00p\x00\xc4\x00m\x00'
>>> len(text.encode('latin-1')), len(text.encode('utf-16'))
(4, 10)
>>> b'\xff\xfe\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: rea
d raw bytes
'sp\xc3\x84m'
>>> open('unidata.txt').read() # 2.X: raw/undec
oded 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 reordered)
[1, 2, 3]
>>> set('spam') - set('ham') # Finding differences in collections
```

```
{'p', 's'}
>>> set('spam') == set('asmp') # Order-neutral
equality tests (== is False)
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 Nones...]
```

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
```

e 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, 9999999999999999	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
<code>yield x</code>	Generator function send protocol
<code>lambda args: expression</code>	Anonymous function generation
<code>x if y else z</code>	Ternary selection (x is evaluated only if y is true)
<code>x or y</code>	Logical OR (y is evaluated only if x is false)
<code>x and y</code>	Logical AND (y is evaluated only if x is true)
<code>not x</code>	Logical negation
<code>x in y, x not in y</code>	Membership (iterables, sets)
<code>x is y, x is not y</code>	Object identity tests
<code>x < y, x <= y, x > y, x >= y</code>	Magnitude comparison, set subset and superset;
<code>x == y, x != y</code>	Value equality operators
<code>x y</code>	Bitwise OR, set union
<code>x ^ y</code>	Bitwise XOR, set symmetric difference
<code>x & y</code>	Bitwise AND, set intersection
<code>x << y, x >> y</code>	Shift x left or right by y bits
<code>x + y</code>	Addition, concatenation;
<code>x - y</code>	Subtraction, set difference
<code>x * y</code>	Multiplication, repetition;
<code>x % y</code>	Remainder, format;
<code>x / y, x // y</code>	Division: true and floor

<code>-x, +x</code>	Negation, identity
<code>~x</code>	Bitwise NOT (inversion)
<code>x ** y</code>	Power (exponentiation)
<code>x[i]</code>	Indexing (sequence, mapping, others)
<code>x[i:j:k]</code>	Slicing
<code>x(...)</code>	Call (function, method, class, other callable)
<code>x.attr</code>	Attribute reference
<code>(...)</code>	Tuple, expression, generator expression
<code>[...]</code>	List, list comprehension
<code>{...}</code>	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/re  
sult  
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)) [us
e 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, and 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 test
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, trunc 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：

3.X 的整数支持无限长度:

2.x 对长整数有一个单独的类型，但是你别手动加上 **L**，一切都是自动的：

无限精度的整数是非常方便的内置工具。

```
>>> 2 ** 200
16069380442589902755419620923411626025222029937
```

```
82792835301376
>>> 2 ** 200
16069380442589902755419620923411626025222029937
82792835301376L
```

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, d
igits 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 decimal
(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) #
```

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 digits)
(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 version)

```

```
(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 `__builtin__`。

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**。

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('2.775557561565156540423631668E-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('0.1428571428571428571428571429')
>>> 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('0.333333333333333333333333333333')
>>>
>>> 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('0.333333333333333333333333333333')
```

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 object`。

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, denom
inator
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 f
or 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 f
or 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)                                # Conve
rt fraction -> float
0.3333333333333333
>>> float(z)
2.5
>>> float(x + z)
2.8333333333333335
>>> 17 / 6
2.8333333333333335

>>> Fraction.from_float(1.75)                # Conver
t 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 -> Fr
action
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，比如 `set([1, 2, 3])` 就会报错：

```
>>> {[1, 2, 3]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

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 (XOR)
>>> 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                                   # M
membership (sets)
True
>>> 'e' in 'Camelot', 22 in [11, 22, 33]      # B
but works on other types too
(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 it
ems in an iterable
{'s', 'a', 'p', 'm'}

>>> {1, 2, 3, 4}                     # Set litera
ls: 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 wo
rk 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 di
                             fferently
set()
>>> type({})              # Because {} is an em
                             pty 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: se
t('spam')
{'m', 's', 'p', 'a'}

>>> {c * 4 for c in 'spam'}                # Set of coll
ected expression results
{'pppp', 'aaaa', 'ssss', 'mmm'}
>>> {c * 4 for c in 'spamham'}
{'pppp', 'aaaa', 'hhhh', 'ssss', 'mmm'}

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

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 change
['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') == sorted('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 i
n either category
{'bob', 'tom', 'sue', 'vic', 'ann'}

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

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

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

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

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

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

>>> (managers | engineers) - (managers ^ engine
ers)    # 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
True
# Same value
>>> True is 1
False
# But a different object: see the next chapter
>>> True or False
True
# Same as: 1 or 0
>>> True + 4
5
# (Hmmm)
```

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。

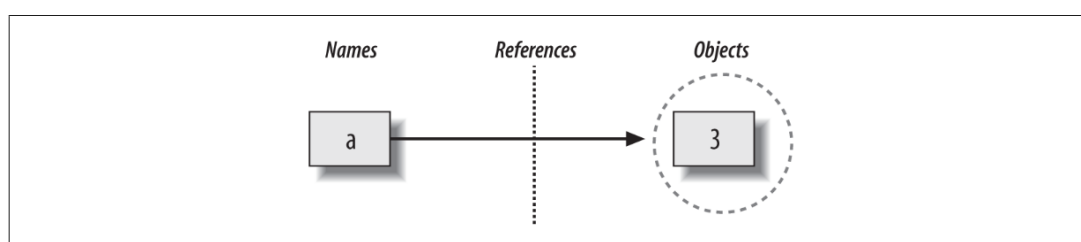
```
>>> a = 3          # Assign a name to an object
```

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 (un
less 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`。

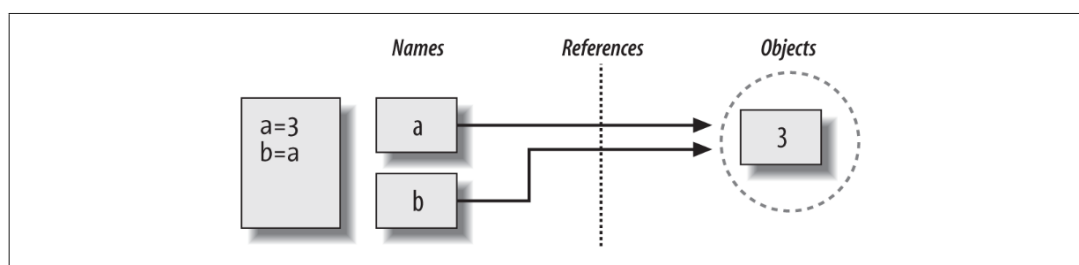
该组件可以被关闭，但是默认是打开的。

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。

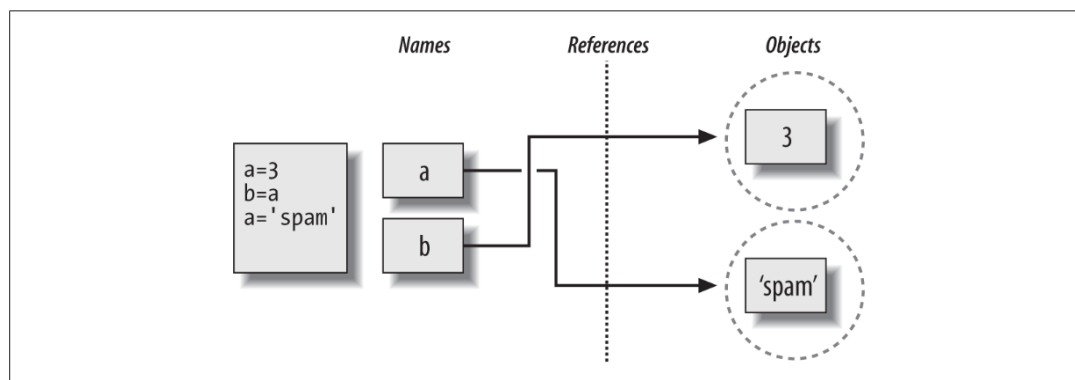
多个名字引用同一对象被称作**共享引用** `shared reference`，有时也叫作**共享对象** `shared object`。

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 t
his 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
<code>S = ''</code>	Empty string
<code>S = "spam's"</code>	Double quotes, same as single
<code>S = 's\np\ta\x00m'</code>	Escape sequences
<code>S = """...multiline..."""</code>	Triple-quoted block strings
<code>S = r'\temp\spam'</code>	Raw strings (no escapes)
<code>B = b'sp\xc4m'</code>	Byte strings in 2.6, 2.7, and 3.X (Chapter 4 , Chapter 37)
<code>U = u'sp\u00c4m'</code>	Unicode strings in 2.X and 3.3+ (Chapter 4 , Chapter 37)
<code>S1 + S2</code>	Concatenate, repeat
<code>S * 3</code>	
<code>S[i]</code>	Index, slice, length
<code>S[i:j]</code>	
<code>len(S)</code>	
<code>"a %s parrot" % kind</code>	String formatting expression
<code>"a {0} parrot".format(kind)</code>	String formatting method in 2.6, 2.7, and 3.X
<code>S.find('pa')</code>	String methods (see ahead for all 43): search,
<code>S.rstrip()</code>	remove whitespace,
<code>S.replace('pa', 'xx')</code>	replacement,
<code>S.split(',')</code>	split on delimiter,

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

```
>>> 'shrubby', "shrubby"
('shrubby', 'shrubby')
```

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"      # Imple
cit 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
<code>\newline</code>	Ignored (continuation line)
<code>\\</code>	Backslash (stores one <code>\</code>)
<code>\'</code>	Single quote (stores <code>'</code>)
<code>\"</code>	Double quote (stores <code>"</code>)
<code>\a</code>	Bell
<code>\b</code>	Backspace
<code>\f</code>	Formfeed
<code>\n</code>	Newline (linefeed)
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\xhh</code>	Character with hex value <i>hh</i> (exactly 2 digits)
<code>\ooo</code>	Character with octal value <i>ooo</i> (up to 3 digits)
<code>\0</code>	Null: binary 0 character (doesn't end string)
<code>\N{ id }</code>	Unicode database ID
<code>\uhhhh</code>	Unicode character with 16-bit hex value
<code>\Uhhhhhhhh</code>	Unicode character with 32-bit hex value ^a
<code>\other</code>	Not an escape (keeps both <code>\</code> and <i>other</i>)

^a The `\Uhhhhh...` 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](#).