[笔记][Learning Python][2. 类型和运算]

腾蛇起陆 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 po
sition
'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 str
ing 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,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 r
ight 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 e
xpression (all)
'spam, eggs, and SPAM!'
>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!') # Format
ting 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 d
igits
'296,999.26'
>>> '%.2f | %+05d' % (3.14159, -42) # Digits, padding, si
gns
'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__', '__it
er__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__n
ew__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__s
etattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'fo
rmat_map', 'index',
'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifi
er', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isuppe
r', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'spli
tlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfil
l']
```

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 term
inate 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\nccc
ccccccccccc\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 i
n files
b'spam'
>>> 'spam'.encode('utf16') # But encoded to 10 bytes in U
TF-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/ho
me: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-t
ype 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 c
olumn 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 col
umn 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 diago
nal 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 value
s, "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 r
ow 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 ordinal
s
[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 un
ique
{'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])) # 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 desc
ription 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 ind
ented
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 expressio
n 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 compr
ehension does
        squares.append(x ** 2) # Both run the iteration p
rotocol 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 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 assignmen
t
>>> 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 characters to i
t
6
>>> f.write('world\n') # Return number of items written i
n 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 proc
essing 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', 'error
s', 'fileno', 'flush',
'isatty', 'line_buffering', 'mode', 'name', 'newlines',
'read', 'readable',
'readline', 'readlines', 'seek', 'seekable', 'tell', 'tru
ncate', '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 fi
le
>>> 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 bina
ry 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 ag
ain
(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') # W
rite/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 enc
oded 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 byte
s
'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 i
n 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 col
lections
{'p', 's'}
>>> set('spam') == set('asmp') # Order-neutral equality t
ests (== 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, No
ne, 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 type
s
<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 mus
t...
        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