[笔记][Effective Python]

Python

---

0. 前言等

设定只允许通过关键字形式来指定的参数。

> jpch89：这里应该说的是 `keyword-only arguments`，强制关键字参数。

用 `zip` 函数来同时迭代两个列表。

> jpch89：这个怎么用不太会。

元类 `metaclass`
动态属性 `dynamic attribute`
最小惊讶原则 `rule of least surprise`

范例代码：https://github.com/bslatkin/effectivepython
原书勘误表：https://github.com/bslatkin/effectivepython/issues
官网：http://www.effectivepython.com/

---

1. 用 Pythonic 方式来思考

`import this`

第 1 条：确认自己所用的 Python 版本

- `python --version`
- `python3 --version`
- 或者：

```
import sys
print(sys.version_info)
print(sys.version)
```

结果为：

```
sys.version_info(major=3, minor=6, micro=6, releaselevel='final',
serial=0)
3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD6
4)]
```

> 【注】
> 还有一种方法是在控制台输入 `python -V`

第 2 条：遵循 PEP8 风格指南

`8` 号 `Python` 增强提案，又叫 `PEP 8`
`Python Enhancement Proposal #8`
http://www.python.org/dev/peps/pep-0008

空白 `whitespace` 会影响代码的含义。

- 使用**四个空格** `space` 缩进，不要用制表符 `tab`
- 每行字符数小于等于 `79`
- 对于占据多行的表达式，首行之后的其余各行都要再缩进 `4` 个空格
- 文件中的函数与类之间要用**两个空行**隔开
- 同一个类中，各方法用**一个空行**隔开
- 使用下标来获取列表元素、调用函数或者给关键字参数赋值时，**不要**在两旁添加空格
- 变量赋值的时候等号两边要各有一个空格

命名：不同的命名风格体现不同角色。

- 函数、变量、属性用小写字母，以下划线连接
- 受保护的实例属性，以单下划线开头
- 私有的实例属性，以双下划线开头
- 类与异常，大驼峰式命名
- 模块级别的常量，全用大写字母，以下划线连接
- 类中的实例方法 `instance method`，首个参数应该叫做 `self`，以**表示该对象本身**
- 类方法 `class method`，首个参数应该叫做 `cls`，以**表示该类本身**

表达式和语句

- 不要把否定词放在前面，要写 `if a is not b` 而不是 `if not a is b`
- 不要用长度检测来**判定为空**（`if len(somelist) == 0`），而是用 `if not somelist`（空值为 `False`）
- **判定非空**用 `if somelist`
- 不要写单行的复合的 `if`、`for`、`while`、`except` 语句，要拆开来写
- `import` 总是放在文件开头
- 导入模块的时候，**使用完整的绝对名称**，而不应该根据当前模块的路径来使用相对名称。例如，导入 `bar` 包中的 `foo` 模块，应该写 `from bar import foo`，而不是 `import foo`。
- 如果一定要用相对名称，就采用明确的写法 `from . import foo`

  > 举个例子：对于 `Python 3`，在包内导入时，假如 `a.py` 导入 `b.py` 写成 `import b`，那么在包外 `import package.a` 或者 `from package import a` 时一定会报错。此时就要用 `from package import b` 或者 `from . import b` 才行。

- `import` 按顺序划分为**标准库模块**、**第三方模块**以及**自用模块**。在每一部分，各 `import` 语句**按字母顺序**排列。

> `Pylint` http://www.pylint.org/
> 源码检测工具

第 3 条：了解 bytes、str 与 unicode 的区别

`Python 3` 中有两种字符串：

- `str` 类型：它的实例包含 `Unicode` 字符
- `bytes` 类型：它的实例包含原始的字节

`Python 2` 中也有两种字符串：

- `unicode` 类型：它的实例包含 `Unicode` 字符
- `str` 类型：它的实例包含原始的字节

二进制数据 –> `Unicode` 字符：`encode` 方法编码
`Unicode` 字符 –> 二进制数据：`decode` 解码

编码和解码的工作要放在界面最外围，核心部分用 `Unicode` 字符串。
并且让程序可以接受多种文本编码，而保证输出一种编码形式（最好是 `UTF-8`）。

要针对 `Python 3` 和 `Python 2` 编写两个辅助 `helper` 函数
`Python 3`

- 总是返回 `str`
- 总是返回 `bytes`

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instance of str


def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instance of bytes
```

`Python 2`

- 总是返回 `unicode`
- 总是返回 `str`

```
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value  # Instance of unicode


def to_str(unicode_or_str):
    if isinstance(unicode_or_str, unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    return value  # Instance of str
```

`ASCII` 码相关知识补充

> ASCII第一次以规范标准的型态发表是在1967年，最后一次更新则是在1986年，至今为止共定义了128个字符，其中33个字符无法显示（这是以现今操作系统为依归，但在DOS模式下可显示出一些诸如笑脸、扑克牌花式等8-bit符号），且这33个字符多数都已是陈废的控制字符，控制字符的用途主要是用来操控已经处理过的文字，在33个字符之外的是95个可显示的字符，包含用键盘敲下空白键所产生的空白字符也算1个可显示字符（显示为空白）。

注意点：

- 在 `Python 2` 中，如果 `str` 只包含 `7` 位 `ASCII` 字符，那么 `unicode` 和 `str` 就成了同一种类型。但是在 `Python 3` 中这两者绝对不等价，空字符串也不行。
  - 可以用 `+` 连接 `str` 与 `unicode`
  - 可用等价于不等价操作符进行比较
  - 格式字符串中可以用 `'%s'` 等形式代表 `unicode` 实例
  - 可以用 `str` 与 `unicode` 给函数传参
- 在 `Python 3` 中 `open` 函数默认使用 ~~`UTF-8` 编码格式~~ **系统本地的编码格式**来操作文件，不能直接往里面写二进制数据 `f.write(os.urandom(10))`，会报错。在 `Python 2` 中文件操作的默认编码格式是二进制形式。同时适配 `Python 2` 和 `Python 3` 的写法是：

```
with open('/tmp/random.bin', 'wb') as f:
    f.write(os.urandom(10))
```

读二进制文件也是一样，要用 `rb`，可以适配 `Python 3` 和 `Python 2`。

> 【注】
> 原书这句话**有误**，在 `Python 3` 中，`open` 函数以文本模式操作文件时，如果不指定编码，该编码是依赖于平台的。
> 以下内容来自官网：https://docs.python.org/3/library/functions.html#open
> encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any text encoding supported by Python can be used. See the codecs module for the list of supported encodings.

`os.urandom(size)` 随机生成 `size` 个字节的二进制数据。

文件句柄 `file handle` 其实是一种标识符或指针。

第 4 条：用辅助函数来取代复杂的表达式

> jpch89：本条体现了两点 `Python` 之禅。
>
> - `Beautiful is better than ugly.`
> - `Simple is better than complex.`

```
from urllib.parse import parse_qs

# 解析查询字符串 query string
my_values = parse_qs('red=5&blue=0&green=',
                     keep_blank_values=True)
# print(repr(my_values))  # 原书写法
print(my_values)  # 返回的是字典，直接这样写就行了
# >>>
# {'red': ['5'], 'blue': ['0'], 'green': ['']}

# 查询字符串中的参数可能有：多个值和空白 blank 值。
# 有些参数则没有出现。
# 使用 get 方法可以不报错的从字典中取值。
print('Red:     ', my_values.get('red'))
print('Green:   ', my_values.get('green'))
print('Opacity: ', my_values.get('opacity'))

print('-' * 50)
# 需求：当查询的参数没有出现在查询字符串中
# 或者参数的值为空白的时候
# 可以返回 0
# 思路：空值和零值都是 False
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
```

```
print('Red:     %r' % red)
print('Green:   %r' % green)
print('Opacity: %r' % opacity)

print('-' * 50)
# 需求：最后要用到的是整数类型
# 思路：类型转换
red = int(my_values.get('red', [''])[0] or 0)
# 这种长表达式的写法看上去很乱！

# 改进1：使用 Python 2.5 添加的三元表达式
red = my_values.get('red', [''])
red = int(red[0]) if red[0] else 0

# 改进2：使用跨行的 if/else 语句
green = my_values.get('green', [''])
if green[0]:
    green = int(green[0])
else:
    green = 0


# 改进3：频繁使用的逻辑，需要封装成辅助函数
def get_first_value(values, key, default=0):
    found = values.get(key, [''])
    if found[0]:
        found = int(found[0])
    else:
        found = default
    return found
```

第 5 条：了解切割序列的办法

切片 `slice` 操作，可以对内置的 `list`、`str`、`bytes` 进行切割。

> jpch89：还有元组 `tuple` 也支持切片操作。

对于实现了 `__getitem__` 和 `__setitem__` 这两个特殊方法的 `Python` 类，也支持切片操作。

基本写法

`somelist[start:end]`

- `start` 起始索引（包括）
- `end` 结束索引（不包括）

如果从列表开始切片，`start` 要留空，不用写 `0`。
`assert a[:5] == a[0:5]`
如果一直要取到末尾，`end` 要留空，写了多余。
`assert a[5:] == a[5:len(a)]`

负索引表示从末尾往前数。

切片时偏移量可以越界，但是索引取值时，偏移量不能越界，会抛出 `IndexError` 异常。

使用负变量作为 `start` 切片，比如 `somelist[-n:]`，当 `n >= 1` 的时候结果是取末尾 `n` 个元素，但是当 `n = 0` 的时候，则会得到原列表的拷贝。

切片返回的是一个新的列表，**对新列表进行修改，不会影响原列表。**

```
a = [1, 2, 3, [4, 5, 6], 7, 8]
b = a[2:]

print('更改前')
print('a为：', a)
print('b为：', b)


b[1] = [111, 222]
print('-' * 50)
print('更改后：')
print('a为：', a)
print('b为：', b)

"""
更改前
a为： [1, 2, 3, [4, 5, 6], 7, 8]
b为： [3, [4, 5, 6], 7, 8]
--------------------------------------------------
更改后：
a为： [1, 2, 3, [4, 5, 6], 7, 8]
b为： [3, [111, 222], 7, 8]
"""
```

赋值语句左侧切片出来的内容会自动替换成右侧的内容。
左右两侧内容的长度不需要相等。

```
a = [1, 2, 3, [4, 5, 6], 7, 8]
print('更改前')
print('a为：', a)
print('-' * 50)
a[:3] = [0]
print('更改后')
print('a为：', a)
```

```
"""
更改前
a为： [1, 2, 3, [4, 5, 6], 7, 8]
--------------------------------------------------
更改后
a为： [0, [4, 5, 6], 7, 8]
"""
```

`b = a[:]` 得到的是对原列表的拷贝

```
a = [1, 2, 3, [4, 5, 6], 7, 8]

# b = a[:] 得到的是对原列表的拷贝
b = a[:]

# 不会抛出 AssertionError
# 因为 a 和 b 值相等
# 但是位于内存的不同位置
assert b == a and b is not a
```

`b = a` 不会在内存中新建对象。 `b` 与 `a` 的内存地址是一样的。
左侧使用留空形式的切片操作，会使用右侧的值替换原列表的全部内容。
`a[:] = ['new', 'content']`

```
a = [1, 2, 3, [4, 5, 6], 7, 8]

# b = a 不会在内存中新建一个对象
b = a
# 所以 a 就是 b
assert a is b
print('更改前')
print('a为：', a)
# 左侧使用留空形式的切片操作，会使用右侧的值替换原列表的全部内容。
a[:] = ['a', 'b', 'c']
print('-' * 50)
print('更改后')
print('a为：', a)
"""
更改前
a为： [0, [4, 5, 6], 7, 8]
--------------------------------------------------
更改后
a为： ['a', 'b', 'c']
"""
```

第 6 条：在单次切片操作内，不要同时指定 start、end 和 stride

`somelist[start:end:stride]`

`stride` 为步进值，假设为 `n`，就是每 `n` 个元素里面取 `1` 个。

- 提取索引值为偶数的元素
  `evens = a[1::2]`
- 提取索引值为奇数的元素
  `odds = a[::2]`

使用 `-1` 作为步进值，其他留空，可以对序列进行逆序操作。
但是对于以 `utf-8` 编码的字节串来说，逆序之后再转码会报错。

```
w = '谢谢你'
x = w.encode('utf-8')
print(x)
y = x[::-1]
# 报错
# UnicodeDecodeError
z = y.decode('utf-8')
```

负步进值从末尾向前选取。

为了容易理解和美观：

- 不要一起写 `start`、`end` 和 `stride`
- 就算非要用 `stride`，尽量用正值，同时省略 `start` 和 `end`
- 如果三个都要用，可以**先步进切片，再范围切片**。或者**先范围切片，再步进切片**。
  两段式切片操作会多产生一份数据的浅拷贝，对执行时间和内存用量有一定影响。
- 可以考虑 `itertools` 模块的 `islide` 方法。该方法不允许对 `start`、`end` 和 `stride` 指定负值。

第 7 条：用列表推导来取代 map 和 filter

`list comprehension` 列表推导：用一份列表来制作另一份。

```
a = [1, 2, 3]
squares = [x ** 2 for x in a]
```

对于简单情况，列表推导要比内置的 `map` 函数更清晰，不用创建 `lambda` 函数：

```
a = [1, 2, 3]
squares = map(lambda x: x ** 2, a)
```

列表推导可以过滤原列表中的元素

```
a = [1, 2, 3]
even_squares = [x ** 2 for x in a if x % 2 == 0]
```

`filter` 结合 `map` 也可以有同样效果，不过更复杂。

```
a = [1, 2, 3]
alt = map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

也有字典 `dict` 推导和集合 `set` 推导。

第 8 条：不要使用含有两个以上表达式的列表推导

- 列表推导支持多级循环，每一级循环也支持多项条件。
- 超过两个表达式的列表推导是很难理解的，应该尽量避免。

```
# 使用两级列表推导展开矩阵
# 推导顺序是从左到右
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)
print('-' * 50)

# 对矩阵每个元素求平方，组成新矩阵
squared = [[x ** 2 for x in row] for row in matrix]
print(squared)
print('-' * 50)

# 假如是三维列表，列表推导要拆成几行才好看
# 不推荐这种写法，因为没有比嵌套循环更清晰方便
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]]
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

```
print(flat)
print('-' * 50)

# 此时改成嵌套循环还更清晰些
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)

# 列表推导支持多个 if 条件
# 多个 if 条件默认以 and 逻辑连接
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]  # 推荐!
c = [x for x in a if x > 4 and x % 2 == 0]

# 每一级 for 循环都可以指定 if 条件
# 需求：找出矩阵中可以被 3 整除，并且所在行各元素之和大于 10 的元素
# 列表推导省空间，但是不利于二次阅读!
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)
```

第 9 条：用生成器表达式来改写数据量较大的列表推导

假如要读取一份文件并返回每行的字符数，如果文件比较大的话，采用列表推导需要把文件的每一行的长度都保存在内存中，会消耗大量内存，甚至导致程序崩溃。

```
value = [len(x) for x in open('ep001_version.py')]
print(value)
"""
[11, 24, 19, 1, 4, 76, 77, 4]
"""
```

可以使用生成器表达式 `generator expression` 来代替列表推导，它会返回一个~~迭代器~~生成器。

> 【注】
> 原书表述有偏差，**生成式返回的当然是生成器**，虽然所有的生成器都是迭代器，但说生成式返回迭代器不太好。
> 下同

~~迭代器~~生成器每次根据生成器表达式产生一项数据，不会占用过多内存。
对~~迭代器~~生成器调用 `next` 来按照生成器表达式输出下一个值。

```
it = (len(x) for x in open('ep001_version.py'))
print(it)
print(next(it))
print(next(it))
"""
<generator object <genexpr> at 0x0000023D58EAAA40>
11
24
"""
```

生成器表达式可以组合使用，即一个生成器表达式返回的~~迭代器~~生成器可以作为另一个生成器表达式的输入值。
外围的~~迭代器~~生成器前进时会推动内部~~迭代器~~生成器前进。
串在一起的生成器表达式执行速度很快。

```
roots = ((x, x ** 0.5) for x in it)
print(next(roots))
"""
(19, 4.358898943540674)
"""
```

连锁生成器表达式适合需要用多种手法操作大批量的数据。
注意：生成器表达式返回的~~迭代器~~生成器是有状态的，不能反复使用。

第 10 条：尽量用 enumerate 取代 range

`range` 在一系列整数上迭代很有用。

```
# range 在一系列整数上迭代很有用
# 比如下面的代码可以生成 64 位二进制随机数
from random import randint

random_bits = 0
for i in range(64):
    if randint(0, 1):
        random_bits |= 1 << i

print(bin(random_bits))
"""
0b1110011100110110000000111101100111110001000110101011110110110 1
"""
```

字符串组成的列表属于序列式数据结构，也可以迭代。

> `pecan`：美洲山核桃

```
# 迭代一个列表
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print('%s is delicious' % flavor)
"""
vanilla is delicious
chocolate is delicious
pecan is delicious
strawberry is delicious
"""
```

迭代列表时通常还想知道当前元素在列表中的索引。

```
# 迭代的同时获取索引
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print('%d: %s' % (i + 1, flavor))
"""
1: vanilla
2: chocolate
3: pecan
4: strawberry
"""
```

> 数组：这里是一种泛称，凡是可以用递增的非负整数做下标来访问其元素的那些数据结构都不妨视为数组。

上面的代码需要：

- 获取列表长度
- 通过下标来访问数组

可以用 `enumerate` 函数代替，它把各种迭代器、序列、支持迭代的对象包装为**生成器**，以便稍后产生输出值。
该生成器每次输出一对值，前者表示下标，后者表示从 `enumerate` 参数中获取到的元素。
所以可以**使用 `enumerate` 代替 `range` 与下标相结合的遍历代码。**

```
# 使用 enumerate 代替 range 与下标相结合的遍历代码
for i, flavor in enumerate(flavor_list):
    print('%d: %s' % (i + 1, flavor))
```

```
"""
1: vanilla
2: chocolate
3: pecan
4: strawberry
"""
```

`enumerate` 第二个参数可以指定初始值，比如从 `1` 开始，这样代码更短：

```
# enumerate 第二个参数指定初始值
for i, flavor in enumerate(flavor_list, 1):
    print('%d: %s' % (i, flavor))
"""
1: vanilla
2: chocolate
3: pecan
4: strawberry
"""
```

第 11 条：用 zip 函数同时遍历两个迭代器

列表推导可以根据一个列表生成另一个列表：

```
# 使用列表推导从一个列表生成另一个
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names]
```

对于这两个列表，如果相同索引处的元素有关联，可以通过循环平行迭代列表：

```
# 相同索引处的两个元素有关联的话
# 可以用循环平行迭代列表
# 下面的代码用来找出最长的名字
longest_name = None
max_letters = 0

for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count

print(longest_name)
"""
Cecilia
"""
```

上面代码的问题在于索引操作出现两次，显得很乱。
可以用 `enumerate` 来改善代码。

```
# 使用 enumerate 代替 range 和索引操作相结合的遍历
longest_name = None
max_letters = 0
for i, name in enumerate(names):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count

print(longest_name)
"""
Cecilia
"""
```

用 `zip` 函数，可以把两个或者以上的迭代器封装为~~生成器~~迭代器，以便稍后求值。

> 【注】
> 我发现原书对于生成器、迭代器这两个概念混淆不清，不知道是原书错误还是翻译有误。
> `zip` 函数返回的 `zip` 对象是一个**由元组组成的迭代器**，而**不是生成器**！
> 官方文档和下述代码皆可验证。
> https://docs.python.org/3/library/functions.html#zip

```
# Python 3.6.6
>>> from collections import Iterator, Generator
>>> z = zip()
>>> isinstance(z, Iterator)
True
>>> isinstance(z, Generator)
False
```

它从每个参数中取一个元素，汇聚成元组 `tuple`。
`zip` 用来**替换通过索引访问多个列表的写法。**

```
# 用 zip 替换通过索引访问多个列表的写法
longest_name = None
max_letters = 0
for name, count in zip(names, letters):
    if count > max_letters:
        longest_name = name
        max_letters = count

print(longest_name)
```

```
"""
Cecilia
"""
```

在 `Python 2` 中，`zip` 返回一个由元组组成的列表，数据较多时会占用大量内存，要使用 `itertools` 中的 `izip` 函数。（参见本书第 `46` 条）

如果 `zip` 中的参数长度不同，只要有一个参数耗尽，`zip` 就停止生成元组。

```
# 只要有一个参数耗尽，zip 就停止生成元组
names.append('Rosalind')
for name, count in zip(names, letters):
    print(name)
"""
Cecilia
Lise
Marie
"""
```

使用 `itertools` 模块中的 `zip_longest` 可以不用管长度是否相等。
`Python 2` 中该函数叫做 `izip_longest`。

第 12 条：不要在 for 和 while 循环后面写 else 块

`Python` 当中的 `for/else` 和 `while/else` 的逻辑有点奇怪。

- `if/else`：如果不执行 `if` 块，那就执行 `else` 块。
- `try/except/else/finally`：如果 `try` 块失败，则执行 `except`；如果 `try` 没有失败，则执行 `else`；无论如何最后都要执行 `finally`。
- 而一开始接触 `Python` 的程序员可能会认为 `for/else` 和 `while/else` 是：如果循环没有正常执行完，那就执行 `else` 块。
- 实际上恰恰相反，它的意思是**如果循环正常执行完毕，则执行 `else`**。如果提前 `break` 跳出，不会执行 `else`。

```
# 正常循环完毕执行 else 块
for i in range(3):
    print('循环 %d' % i)
else:
    print('else 块! ')
"""
循环 0
循环 1
循环 2
else 块!
"""
```

```
# 中断循环则不会执行 else 块
for i in range(3):
    print('循环 %d' % i)
    if i == 1:
        break
else:
    print('else 块! ')
"""
循环 0
循环
```

如果被遍历的对象是空的，会立即执行 `else` 块：

```
# 如果被遍历的对象是空的，会立即执行 else 块
for x in []:
    print('永远不会执行! ')
else:
    print('我是 else 块! ')
"""
我是 else 块!
"""
```

循环初始条件为 `False`，同样也会立即执行 `else` 块：

```
# 循环初始条件为 False，也会立即执行 else 块
while False:
    print('永远不会执行! ')
else:
    print('我是 else 块! ')
"""
我是 else 块!
"""
```

循环配合 `else` 块可以应用在**搜索某个事物的时候。**

```
# 判断两个数是否互质 coprime
# 即除了 1 以外，没有其他公约数
a = 4
b = 9
for i in range(2, min(a, b) + 1):
    print('尝试', i)
    if a % i == 0 and b % i == 0:
        print('不互质! ')
        break
else:
    print('互质! ')
"""
互质!
```

```
    """
```

实际上我们会用其他写法。
比如使用**辅助函数**：

- 只要满足条件，立即返回
- 如果整个循环都执行完毕，说明不满足条件，在最后返回默认值

```
# 使用辅助函数判断是否互质（写法1）
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True
```

或者**用变量记录受测参数是否满足条件**，满足则立刻跳出。

```
# 使用辅助函数判断是否互质（写法2）
def coprime2(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
```

作者不建议循环后面使用 `else`。

> 【注】
> 虽然不建议，但是看到别人这么写你还是要懂！

第 13 条：合理利用 try/except/else/finally 结构中的每个代码块

- `finally` 块：既要将异常向上传播，又要在异常发生时执行清理工作，可以使用 `try/finally`

```
# 确保文件能够可靠的关闭文件句柄
handle = open('ep001_version.py')  # May raise IOError
try:
    data = handle.read()  # May raise UnicodeDecodeError
finally:
```

```
        handle.close()  # Always run after try:
```

第 `1` 句代码放在 `try` 外面是因为，如果发生了异常，那么不应该执行 `finally` 里面的 `close()` 语句。

- `else` 块：`try/except/else` 中，如果 `try` 没有异常，则执行 `else`。这样可以**缩减 `try` 中的代码量**，增加易读性。

```
# 从字符串中加载 JSON 字典数据
def load_json_key(data, key):
    try:
        result_dict = json.loads(data)  # May raise ValueError
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key]  # May raise KeyError
```

- 混合使用 `try/except/else/finally`
  顺利运行 `try` 块后，如果想在 `finally` 块中的清理代码执行之前做一些操作，可以放在 `else` 块中。这种写法必须有 `except`。

---

2. 函数

首先接触的代码组织工具就是函数。
它可以分割大段程序，让代码更易读，易用。
也为复用和重构提供了契机。

第 14 条：尽量用异常来表示特殊情况，而不要返回 None

- 编写工具函数 `utility function` 的时候，有人喜欢给 `None` 返回值附加特殊意义。这样会让调用者容易犯错，因为 `None` 和 `0` 以及空字符串之类的值都等价于 `False`。
- 在遇到特殊情况的时候，应该抛出异常，而不是返回 `None`，并编写相应的文档。这样调用者就会处理该异常。

```
# 辅助函数，计算两数的商
# 如果出现除零错误，返回 None
def divide(a, b):
    try:
```

```
        return a / b
    except ZeroDivisionError:
        return None

# 判断结果是否为 None
x, y = 4, 0
result = divide(x, y)
if result is None:
    print('Invalid inputs')
"""
Invalid inputs
"""

# 但是我们可能不会专门判断结果是否为 None
# 而是假定如果 result 为 False，那么说明函数出错了
x, y = 0, 4
result = divide(x, y)
if not result:
    print('Invalid inputs')  # This is wrong!
"""
Invalid inputs
"""

# 解决方案1：
# 返回一个二元组 two-tuple
# 二元组的第一个元素表示操作是否成功
# 第二个元素表示结果
def divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None

x, y = 4, 0
success, result = divide(x, y)
if not success:
    print('Invalid inputs')
"""
Invalid inputs
"""

# 但是调用者可以用下划线 _ 忽略不想要的第一部分
# 这样还是会出错
x, y = 0, 4
_, result = divide(x, y)
if not result:
    print('Invalid inputs')
"""
Invalid inputs
"""

# 解决方案2（最好的方案）：
```

```python
# 向上一级抛出异常
# 把 ZeroDivisionError 转化成 ValueError
# 表示调用者给的输入值无效
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs') from e

x, y = 4, 0
result = divide(x, y)
print(result)
"""
Traceback (most recent call last):
  File "ep014_notreturnnone.py", line 62, in divide
    return a / b
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "ep014_notreturnnone.py", line 67, in <module>
    result = divide(x, y)
  File "ep014_notreturnnone.py", line 64, in divide
    raise ValueError('Invalid inputs') from e
ValueError: Invalid inputs
"""

# 现在调用者必须处理因输入值无效而引发的异常
# 不用判断返回结果，因为如果没有异常，结果就是正确的
x, y = 5, 2
try:
    result = divide(x, y)
except ValueError:
    print('Invalid inputs')
else:
    print('Result is %.1f' % result)
"""
Result is 2.5
"""
```

第 15 条：了解如何在闭包里使用外围作用域中的变量

需求：把列表中出现在某一个群组中的数字放在其它数字前面。
可以用一个辅助函数 `helper` 判断，并返回排序关键字 `sort key`。

```python
# 把 numbers 中出现在 group 里面的数字放在前面
def sort_priority(values, group):
    def helper(x):
        if x in group:
            return (0, x)
        return (1, x)
    values.sort(key=helper)

numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)
"""
[2, 3, 5, 7, 1, 4, 6, 8]
"""
```

`sort_priority` 能够正常运作，原因在于：

- `Python` 支持闭包 `closure`：**闭包是一种定义在某个作用域中的函数，这种函数引用了作用域里面的变量。** `helper` 访问 `group` 就是因为 `helper` 是闭包。
- `Python` 的函数是一级对象 `first-class object`。我们可以直接引用函数，把函数赋值给变量，把函数当做参数传给其他函数等等。
- `Python` 使用特殊的规则比较元组（对列表也适用）。先比较下标为 `0` 的元素，再依次往后比较。

> https://stackoverflow.com/questions/245192/what-are-first-class-objects
> The rights and privileges of first-class citizens:
>
> - To be named by variables.
> - To be passed as arguments to procedures.
> - To be returned as values of procedures.
> - To be incorporated into data structures

增加需求：
让函数返回一个值，表示 `numbers` 是否出现了 `group` 中的数。

```python
# 新需求：判断 numbers 的数字是否出现在了 group 里面
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True  # 看起来很简单
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
found = sort_priority2(numbers, group)
print('Found:', found)
print(numbers)
# 明明找到了，但是却显示没有找到！
"""
Found: False
[2, 3, 5, 7, 1, 4, 6, 8]
"""
```

为什么 `found` 结果不对呢?
先来熟悉一下 `Python` 解释器**引用变量**时的查找顺序：`LEGB`。

- 当前函数的作用域 `local`
- 任何外围作用域 `external`
- 包含当前代码的那个模块的作用域，也叫全局作用域 `global scope`
- 内置作用域 `builtins`
  如果找不到，抛出 `NameError` 异常。

给**变量赋值**时规则不同。

- 如果当前作用域已经定义了这个变量，会重新赋值。
- 如果当前作用域没有这个变量，把这次赋值视为变量的定义。
  新定义的变量的作用域是包含赋值操作的这个函数。

```
def sort_priority2(numbers, group):
    found = False         # Scope: 'sort_priority2'
    def helper(x):
        if x in group:
            found = True  # Scope: 'helper' -- Bad!
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

这种问题有时称作**作用域 bug** `scoping bug`，也叫作范围 `bug`。
这种设计**可以防止函数中的局部变量污染函数外面的模块。**

修改外部作用域的变量，使用 `nonlocal` 关键字。
限制：不能延伸到模块级别（全局变量），可以防止它污染全局作用域。

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found
        if x in group:
            found = True
            return (0, x)
```

```
            return (1, x)
        numbers.sort(key=helper)
        return found

    numbers = [8, 3, 1, 2, 5, 4, 7, 6]
    group = {2, 3, 5, 7}
    found = sort_priority3(numbers, group)
    print('Found:', found)
    print(numbers)
    """
    Found: True
    [2, 3, 5, 7, 1, 4, 6, 8]
```

`nonlocal` 表明：如果修改闭包内部的变量，实际上是给闭包外面的作用域的变量赋值。
`global` 表示：对该变量的赋值操作会修改模块作用域的那个变量。

注意：仅仅在最简单的函数中使用 `nonlocal`，不要滥用。

如果使用 `nonlocal` 的代码太复杂，要使用辅助类 `helper class` 封装起来。

```
    print('-' * 30)
    class Sorter(object):
        def __init__(self, group):
            self.group = group
            self.found = False

        def __call__(self, x):
            if x in self.group:
                self.found = True
                return (0, x)
            return (1, x)

    sorter = Sorter(group)
    numbers.sort(key=sorter)
    print(numbers)
    assert sorter.found is True

    """
    ------------------------------
    [2, 3, 5, 7, 1, 4, 6, 8]
    """
```

在 `Python 2` 中没有 `nonlocal` 关键字，需要使用可变类型的数据来实现类似的功能。

```
    # Python 2
    def sort_priority(numbers, group):
        found = [False]
        def helper(x):
            if x in group:
```

```
                return (0, x)
            return (1, x)
        numbers.sort(key=helper)
        return found[0]
```

第 16 条：考虑用生成器来改写直接返回列表的函数

函数返回多个结果最简单的方法就是返回一个列表。

```
# 添加词的首字母索引到列表中
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result

address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:3])
"""
[0, 5, 11]
"""
```

上面这段代码的问题：

- 比较拥挤，`result.append` 不是重点，重点是 `index + 1`
- 还要创建和返回列表
- 对于大数据量，会消耗很多内存，导致程序崩溃

用**生成器** `generator` 改写。
包含 `yield` 表达式的函数是**生成器函数**，调用它返回一个生成器。
对生成器调用 `next()`，会推动它到下一个 `yield` 表达式，返回结果给调用者。

> 【注】
> 原书表述不准确，这里已改过来。
> 作者可能对生成器和迭代器有点误解，每次提到这两个东西都会出错。
> 后面的改动不再注明。

用生成器函数改写使用列表的代码：

```
# 用生成器函数改写使用列表的代码
```

```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
# 使用 list 函数可以把生成器转换为列表
```

从文件中读取数据：

```
# 从文件中一行一行的读取
print('-' * 20)

def index_file(handle):
    offset = 0
    for line in handle:
        # 返回每一行首个单词的首字母的偏移量
        if line:
            yield offset
        for letter in line:
            offset += 1
            if letter == ' ':
                yield offset

with open('my_address.txt') as f:
    it = index_file(f)
    from itertools import islice
    results = islice(it, 0, 3)
    print(list(results))
"""
--------------------
[0, 5, 11]
"""
```

上面这个函数的内存消耗，由单行最大字符数决定。

注意：生成器是有状态的，不能重复使用。

总结：使用生成器比把收集到的结果放入列表里返回给调用者更加清晰。

第 17 条：在参数上面迭代时，要多加小心

如果函数接收的参数是个列表，那么有可能需要在这个列表上多次迭代。
下面是一个**标准化函数** `normalization function`（也叫作正规化函数或者归一函数）。
用来求每个城市游客占总游客数量的百分比。

```
# 返回列表的标准化函数
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result

visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
"""
[11.538461538461538, 26.923076923076923, 61.53846153846154]
"""
```

假如每个城市的游客数量在一个文件里面，需要从文件中读取数据。
为了增强可扩展性 `scalability`，使用生成器来重写这个函数。

```
# 使用生成器从文件读取数据
def read_visits(data_path):
    with open(data_path) as f:
        for line in f:
            yield int(line)

# 没有返回结果的原因是：
# normalize 中的 sum 函数已经把 生成器 it 耗尽
it = read_visits('my_numbers.txt')
percentages = normalize(it)
print(percentages)
"""
[]
"""
```

结果是个空列表，因为 `sum` 函数已经把生成器 `it` 使用过一遍了。
迭代器（包括生成器）只能产生一轮结果，在抛出过 `StopIteration` 异常的迭代器或生成器上面继续迭代，不会返回任何结果。
尝试如下：

```
print('-' * 20)
it = read_visits('my_numbers.txt')
print(list(it))
print(list(it))
"""
--------------------
[15, 35, 80]
[]
"""
```

解决方案 `1`：把传来的迭代器做成一份列表。
缺点：如果输入大量数据，仍然会消耗过多内存导致程序崩溃。

```
# 解决方案 1：把输入数据的生成器复制成一份列表
# 缺点：大量输入数据会消耗大量内存
def normalize_copy(numbers):
    numbers = list(numbers)  # 复制生成器
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result

it = read_visits('my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)
"""
[11.538461538461538, 26.923076923076923, 61.53846153846154]
"""
```

解决方案 `2`：不传递生成器，而是传递一个每次调用都可以返回一个新的生成器的函数。每当要用到生成器时都新建一个。
缺点：调用函数时写个 `lambda` 函数不好看。

```
# 解决方案 2：每次迭代都用一个函数产生新的生成器
# 缺点：使用 lambda 函数显得生硬
def normalize_func(get_iter):
    total = sum(get_iter())  # 新生成器
    result = []
    for value in get_iter():  # 又是一个新生成器
        percent = 100 * value / total
        result.append(percent)
    return result

path = 'my_numbers.txt'
percentages = normalize_func(lambda: read_visits(path))
print(percentages)
"""
[11.538461538461538, 26.923076923076923, 61.53846153846154]
"""
```

解决方案 `3`：编写实现了迭代器协议 `iterator protocol` 的类。
缺点：打开两次文件，读取两次数据。

```
# 解决方案 3：编写实现迭代器协议的类
# 缺点：需要打开两次文件，读取两次输入数据
```

```
class ReadVisits(object):
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)

path = 'my_numbers.txt'
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)
"""
[11.538461538461538, 26.923076923076923, 61.53846153846154]
"""
```

> 【注】
> 这里的写法比较特殊。
> 实现的自定义类有 `__iter__` 方法，说明这是可迭代的类。
> 而在 `__iter__` 方法又是一个生成器函数（因为它包含了 `yield`）。
> 所以严格来讲，`for ... in ...` 作用于这个类的对象上面，得到的是一个**生成器**。

这里 `normalize` 函数中的 `sum` 和 `for ... in ...` 都调用了 `visits.__iter__`，获得了新的迭代器对象。

利用对迭代器对象调用内建的 `iter` 函数，会返回迭代器本身这一特性，可以进行防御性编程，让 `normalize` 函数不接收迭代器对象作为参数。
（参考文档：https://docs.python.org/3/library/stdtypes.html#typeiter）

```
# 改进解决方案 3:
# 让 nomralize 函数不接收迭代器作为参数
def normalize_defensive(numbers):
    if iter(numbers) is iter(numbers):  # 迭代器 -- 不要!
        raise TypeError('请提供可迭代对象! ')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result

visits = [15, 35, 80]
normalize_defensive(visits)  # 不报错
visits = ReadVisits(path)    # 不报错

# 如果传递迭代器作为参数，会抛出异常
it = iter(ReadVisits(path))
normalize_defensive(it)
```

```
"""
Traceback (most recent call last):
  File "ep017_exhausteditterator.py", line 114, in <module>
    normalize_defensive(it)
  File "ep017_exhausteditterator.py", line 100, in normalize_defensive
    raise TypeError('请提供可迭代对象！')
TypeError: 请提供可迭代对象！
"""
```

总结

- 函数多次迭代生成器参数要小心，只能迭代一次。
- 了解 `Python` 的迭代器协议
- 判断一个对象是不是迭代器，使用 `iter(it) is iter(it)`，如果成立就是迭代器。`next(it)` 可以推动迭代器。

第 18 条：用数量可变的未知参数减少视觉杂讯

> 减少视觉杂讯 `visual noise`，意思是代码不要太杂乱，突出重要内容。

使用**变长参数**，可以让代码清晰，减少视觉杂讯。
下面是日志函数 `log`，第一个参数是信息，第二个参数是待打印的列表。

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', [1, 2])
log('Hi there', [])
"""
My numbers are: 1, 2
Hi there
"""
```

即使不想打印后面的值，也需要传入空列表，既麻烦，又杂乱。
可以把第二个参数写成**变长参数**。

```
def log(message, *values):  # 只改动了这里
    if not values:
        print(message)
    else:
```

```
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', 1, 2)
log('Hi there')  # 这样写好多了
"""
My numbers are: 1, 2
Hi there
"""
```

调用函数的时候，使用 `*` 来把列表中的元素展开成位置参数。

```
favorites = [7, 33, 99]
log('Favorite colors', *favorites)
"""
Favorite colors: 7, 33, 99
"""
```

使用变长参数 `*args` 的**问题 1**：如果参数为 `*生成器`，则会遍历生成器，存储为元组，有可能消耗大量内存，导致程序崩溃。

```
# 变长参数问题 1:
# 传入生成器会存储为元组
# 数据量较大消耗内存也会变大
def my_generator():
    for i in range(10):
        yield i


def my_func(*args):
    print(args)


it = my_generator()
my_func(*it)
"""
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
"""
```

解决：只有我们确定输入的参数个数较少时，才使用变长参数 `*args`。

使用变长参数 `*args` 的问题 `2`：以后给函数添加新的位置参数，需要修改两个地方，一个是函数的**定义方**，另外一个是函数的**调用方**。
如果忘了修改调用方，会出现难以调试的错误。

```
# 变长参数问题 2:
# 添加位置参数需要修改函数定义和函数调用
# 忘记修改调用会发生难以调试的错误
```

```
def log(sequence, message, *values):
    if not values:
        print('%s: %s' % (sequence, message))
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s: %s' % (sequence, message, values_str))


log(1, 'Favorites', 7, 33)          # 新用法没问题
log('Favorite numbers', 7, 33)      # 旧用法会出错
"""
1: Favorites: 7, 33
Favorite numbers: 7: 33
"""
```

解决：可以使用只能以关键字形式指定的参数来避免这种情况。
`keyword-only argument`，不如叫**强制关键字参数**。

第 19 条：用关键字参数来表达可选的行为

调用函数时，可以按位置传递参数。
`Python` 中所有的位置参数，都可以按关键字传递。
还可以混合使用关键字参数和位置参数。

```
# 调用函数时，可以按照位置传递参数
def remainder(number, divisor):
    return number % divisor

assert remainder(20, 7) == 6

# 所有的位置参数，都可以按照关键字传递
# 还可以混合使用关键字参数和位置参数
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

位置参数必须出现在关键字参数之前。
每个参数只能指定一次。

```
# 位置参数必须出现在关键字参数之前
remainder(number=20, 7)
"""
  File "ep019_kwargs.py", line 15
    remainder(number=20, 7)
                         ^
```

```
SyntaxError: positional argument follows keyword argument
"""

# 每个参数只能指定一次
remainder(20, number=7)
"""
Traceback (most recent call last):
  File "ep019_kwargs.py", line 24, in <module>
    remainder(20, number=7)
TypeError: remainder() got multiple values for argument 'number'
"""
```

使用关键字参数的好处：

- 好处 `1`：**以关键字参数来调用函数，能增强代码易读性**
  比如 `remainder(number=20, divisor=7)` 就可以让读者知道哪个是被除数，哪个是除数。
- 好处 `2`：**关键字参数可以在函数定义中提供默认值。**大部分情况下函数的调用者只需要使用默认值，只有在需要附加功能时才指定关键字参数，这样可以**消除重复代码，让代码变的整洁。**

比如：计算液体流速的函数。

```
# 每秒钟液体的流速
def flow_rate(weight_diff, time_diff):
    return weight_diff / time_diff


weight_diff = 0.5
time_diff = 3
flow = flow_rate(weight_diff, time_diff)
print('%.3f kg per second' % flow)
"""
0.167 kg per second
"""
```

上面是每秒钟流过的千克数，假如要求每小时或者每天流过的千克数，可以增加一个参数，表示两种时间间隔的**比例因子。**

> `scaling factor` 也叫做换算系数或换算因数。

```
# 增加时间换算的参数
def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period
```

这样写的**缺点**是，每次调用函数都要指定 `period` 参数，即便想计算最常见的每秒流率，也要传个 `1`。

```
# 即便计算最简单的每秒流率，也要给 period 传入 1
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

此时就可以给 `period` 参数定义默认值。

> 【注】
> 默认参数，可选参数，缺省参数。
> 我认为这些都是一回事。

```
# 给 period 参数定义默认值
def flow_rate(weight_diff, time_diff, period=1):
    return (weight_diff / time_diff) * period

flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

- 好处 `3`：**提供了扩充函数参数的方式，扩充后的函数与原代码兼容，不用修改调用代码。**

例如：让上面的代码根据别的重量单位来计算流率。

```
# 根据其它重量单位来计算流率
def flow_rate(weight_diff, time_diff,
              period=1, units_per_kg=1):
    return ((weight_diff * units_per_kg) / time_diff) * period

# 原代码可以保持不变
# 新代码可以通过指定关键字参数来使用新功能
pounds_per_hour = flow_rate(weight_diff, time_diff,
                            period=3600, 2.2)
```

缺点：仍然可以通过位置参数的形式指定可选的关键字参数。
比如：

```
# 缺点：仍然可以通过位置参数的形式指定可选的关键字参数。
# 这样容易让人困惑
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

所以可选的关键字参数，总是应该以关键字形式来指定，而不应该以位置参数的形式来指定。
对于接收 `*args` 作为参数的函数，为了保持兼容，新加入的参数需要定义为可选的关键字参数。
更好的办法是用强制关键字参数。

第 20 条：用 None 和文档字符串来描述具有动态默认值的参数

函数的参数非静态，比如打印日志时，想把日志时间作为参数，以该函数调用的时候为准。

```
from datetime import datetime
from time import sleep

# 动态参数
def log(message, when=datetime.now()):
    print('%s: %s' % (when, message))

log('初次见面，请多关照')
sleep(0.1)
log('这么巧，又见面了')
"""
2018-11-29 10:58:31.444798: 初次见面，请多关照
2018-11-29 10:58:31.444798: 这么巧，又见面了
"""
```

两条消息的时间戳 `timestamp` 居然一样。
原因是：**函数参数的默认值，会在该函数所在的模块被加载的时候求出。**
所以模块只加载一次，`datetime.now()` 只运行了一次。
所以时间戳不会改变。

正确的动态默认值参数，习惯上把参数的默认值设置为 `None`，在文档字符串 `docstring` 中描述如何设置该动态参数。编写函数时，对参数进行检测，如果是 `None`，那么就将其设置为实际默认值。

```
# 正确版本的动态默认值参数
def log(message, when=None):
    """Log a message with a timestamp.a

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
            Defaults to the present time.
    """
    when = datetime.now() if when is None else when
    print('%s: %s' % (when, message))

log('初次见面，请多关照')
sleep(0.1)
log('这么巧，又见面了')
"""
2018-11-29 11:09:02.975954: 初次见面，请多关照
2018-11-29 11:09:03.076682: 这么巧，又见面了
```

```
    """
```

如果参数的默认值是可变类型 `mutable`，一定要用 `None` 作为形式上的默认值。
例如：读取 `JSON` 数据，如果读取失败，返回空字典。

```
import json
# 读取 JSON 数据，如果失败，返回空字典
def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default
```

这种写法错在 `default` 参数的默认值会在模块加载时进行一次求值，所以所有获取到的空字典，都是同一份字典。

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
"""
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
"""

# 不报错！
assert foo is bar
```

解决方法同上：

```
# 解决方法：使用 None 作为动态参数的默认值
# 在文档字符串中描述动态参数的设置规则
# 在函数定义时判断参数是否为 None
# 如果为 None，进行相应设置
def decode(data, default=None):
    """Load JSON data from a string.

    Args:
        data: JSON data to decode.
        default: Value to return if decoding fails.
            Defaults to an empty dictionary.
    """
    if default is None:
        default = {}
    try:
        return json.loads(data)
```

```
    except ValueError:
        return default

# 现在就正确了
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

"""
Foo: {'stuff': 5}
Bar: {'meep': 1}
"""
```

小结

- 参数的默认值，只会在程序加载模块并读到本函数的定义时评估一次。对于 `{}` 或者 `[]` 可能发生奇怪的行为。
- 以动态值作为实际默认值的关键字参数来说，应该把形式上的默认值写为 `None`，并在函数的文档字符串里描述该默认值所对应的实际行为。

第 21 条：用只能以关键字形式指定的参数来确保代码明晰

例子：做除法时，有时想忽略 `ZeroDivisionError` 返回无穷，有时想忽略 `OverflowError` 返回 `0`。

```
def safe_division(number, divisor,
    ignore_overflow, ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise

# 忽略 float 溢出错误，并返回 0
result = safe_division(1.0, 10 ** 500, True, False)
```

```
print(result)
"""
0
"""

# 忽略除零错误，并返回无穷
result = safe_division(1, 0, False, True)
print(result)
"""
inf
"""
```

这样写的缺点是，调用的时候不知道哪个值对应哪个参数。
改进 `1`：把后面两个参数定义为关键字参数。

```
# 改进：把后面两个参数定义为关键字参数
def safe_division_b(number, divisor,
                    ignore_overflow=False,
                    ignore_zero_division=True):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise

# 调用者按需覆盖默认参数
safe_division_b(1, 10 ** 500, ignore_overflow=True)
safe_division_b(1, 0, ignore_zero_division=True)
```

缺点：后面两个参数还是可以以位置参数的形式指定，这样读代码的时候搞不懂哪个参数对应哪个值。

```
# 缺点：仍然可以用位置参数的形式来调用
safe_division_b(1, 10 ** 500, True, False)
```

改进：定义强制关键字参数
在 `Python 3` 中，参数列表中的 `*` 星号标志着位置参数就此终结，之后的参数只能以关键字形式指定。

```
# 改进：使用强制关键字参数
def safe_division_c(number, divisor, *,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise

# 再使用位置参数指定就会出错
safe_division_c(1, 10 ** 500, True, False)
"""
Traceback (most recent call last):
  File "ep021_kwonlyargs.py", line 72, in <module>
    safe_division_c(1, 10 ** 500, True, False)
TypeError: safe_division_c() takes 2 positional arguments but 4 were give
n
"""
```

如果不指定关键字参数，默认值会生效：

```
# 如果不指定关键字参数，默认值会生效
safe_division_c(1, 0, ignore_zero_division=True)
# 不报错！

try:
    safe_division_c(1, 0)
except ZeroDivisionError:
    print('捕捉到了除零异常！')
"""
捕捉到了除零异常！
"""
```

在 `Python 2` 中使用 `**kwargs` 模拟强制关键字参数。
`**kwargs` 在函数定义时可以接受数量可变的关键字参数。

```
# Python 2
def print_args(*args, **kwargs):
    print 'Positional:', args
    print 'Keyword:', kwargs
```

```
print_args(1, 2, foo='bar', stuff='meep')
"""
Positional: (1, 2)
Keyword: {'foo': 'bar', 'stuff': 'meep'}
"""
```

使用 `**kwargs` 和 `pop` 方法模拟强制关键字参数：

```
# 模拟强制关键字参数
# Python2
def safe_division_d(number, divisor, **kwargs):
    ignore_overflow = kwargs.pop('ignore_overflow', False)
    ignore_zero_division = kwargs.pop('ignore_zero_division', False)
    if kwargs:
        raise TypeError('Unexpected **kwargs: %r' % kwargs)

# 现在可以正常使用
safe_division_d(1, 10)
safe_division_d(1, 0, ignore_zero_division=True)
safe_division_d(1, 10 ** 500, ignore_overflow=True)

# 不可以通过位置参数指定关键字参数的值
safe_division_d(1, 0, False, True)  # 会报错

# 也不可以传入不符合预期的关键字参数
safe_division_d(0, 0, unexpected=True)
```

> 【注】
> `Python 2` 这里我没有验证，没有安装 `Python 2`

补充
`dict.pop()` 方法：

- `dict.pop(k[, d])`
- 根据键 `k` 删除键值对，并返回值
- 如果键不存在，抛出 `KeyError` 异常
- 如果给出了 `d` 这个默认值参数，如果键不存在不会抛出异常，而是会返回 `d` 的值

---

3. 类与继承

第 22 条：尽量用辅助类来维护程序的状态，而不要用字典和元组

可以通过字典来保存动态信息。
动态 `dynamic` 信息：待保存的信息，其标识符无法提前获知。
例如，记录学生成绩：

```
# 记录学生成绩
class SimpleGradeBook(object):
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)

    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)


# 使用这个类很简单
book = SimpleGradeBook()
book.add_student('牛顿')
book.report_grade('牛顿', 90)
print(book.average_grade('牛顿'))
"""
90.0
"""
```

新需求：
现在想要分科目保存成绩，而不是把所有成绩记录都存在一个列表里。
这样可以把成绩记录定义成一个字典，科目为键，成绩为值。

```
# 分科目记录成绩，需要把成绩记录定义成字典
class BySubjectGradebook(object):
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = {}

    def report_grade(self, name, subject, grade):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
```

```
            grade_list.append(grade)

        def average_grade(self, name):
            by_subject = self._grades[name]
            total, count = 0, 0
            for grades in by_subject.values():
                total += sum(grades)
                count += len(grades)
            return total / count


    # _grades 的键是学生名字，值是分科目的成绩字典 by_subject
    # by_subject 字典的键是科目名字 subject，值是成绩列表
    # 虽然嵌套了一层字典，使用起来还是比较方便
    book = BySubjectGradebook()
    book.add_student('爱因斯坦')
    book.report_grade('爱因斯坦', '数学', 75)
    book.report_grade('爱因斯坦', '数学', 65)
    book.report_grade('爱因斯坦', '体育', 90)
    book.report_grade('爱因斯坦', '体育', 95)
    print(book.average_grade('爱因斯坦'))
    """
    81.25
    """
```

新需求：要记录每次考试成绩的权重。比如随堂测验权重没有期末考试高。
可以把科目作为键，`（分数，权重）`作为值。
这样非常复杂，我们应该从字典和元组迁移到类体系。
当前套多于一层的时候（比如字典包含字典），我们就应该写一个辅助类。

我们可以**从依赖关系树的最底层开始重构。**
用元组记录某科目历次考试成绩就足够了，`（成绩，权重）`。

> 【注】
> 注意单下划线用法：
> `total_weight = sum(weight for _, weight in grades)`
> `_` 代表 `score`，因为是不要的数据，直接写 `_` 更简洁。

但是每次考试要加上老师的评语，容易搞错顺序，此时可以使用 `collections` 模块中的**具名元组 `namedtuple`**。

> 【注】
> 具名元组：我觉得这个翻译的好！

具名元组适合定义精简而又不可变的数据类。
但是缺点有二：

- 无法指定各参数的默认值，对于可选属性比较多的数据用起来不方便（**注**：`Python 3.7` 中可以设置 `defaults` 了）
- 具名元组示例的各项属性可以通过下标和迭代来访问，对于需要公布的 `API`，要考虑其他人以不符合设计者的意图的方式来访问它，可以用自定义类代替。

> 【注】补充 `namedtuple`

`collections` 模块下的 `namdedtuple` 函数返回一个元组的子类，它带有字段名。

具名元组定义：
`namedtuple(typename, field_names, *, rename=False, defaults=None, module=None)`

- `typename` 是类型名，接收一个字符串
- `field_names` 是字段名，接收用空格分开的字符串，或者由字符串组成的可迭代对象

几种生成具名元组类的方式：

- `Point = namedtuple('Point', 'x y')`
- `Point = namedtuple('Point', 'x, y')`
- `Point = namedtuple('Point', ('x', 'y'))`
- `Point = namedtuple('Point', ['x', 'y'])`

查看类的文档字符串：

- `print(Point.__doc__)`

定义好具名元组类之后，就可以实例化一个具名元组了。
几种实例化具名元组的方式：

- 可以通过位置参数或关键字参数实例化：`p = Point(11, y=22)`
- 可以用字典来实例化具名元组 `p = Point({'x': 11, 'y': 22})`
- 具名元组可以通过下标进行索引操作
  `p[0] + p[1]`
- 可以通过 `.` 点号访问字段
  `p.x`

【重要】
重要的官方资料：https://docs.python.org/3/library/collections.html?highlight=namedtuple#collections.namedtuple

---

```
import collections

Grade = collections.namedtuple('Grade', ('score', 'weight'))
```

```python
# 编写科目类，该类包含一系列考试成绩
class Subject(object):
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight


# 编写学生类，包括学生学习的所有课程
class Student(object):
    def __init__(self):
        self._subjects = {}

    def subject(self, name):
        # if name not in self._subjects:
        #     self._subjects[name] = Subject()
        # return self._subjects[name]
        # 我是这么写的，一句搞定
        return self._subjects.setdefault(name, Subject())

    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count


# 最后编写包含所有学生考试成绩的容器类
class Gradebook(object):
    def __init__(self):
        self._students = {}

    def student(self, name):
        # if name not in self._students:
        #     self._students[name] = Student()
        # return self._students[name]
        # 我是这么写的，一句搞定
        return self._students.setdefault(name, Student())


book = Gradebook()
albert = book.student('爱因斯坦')
math = albert.subject('数学')
```

```
math.report_grade(80, 0.1)
math.report_grade(90, 0.1)
print(albert.average_grade())
"""
85.0
"""
```

总结

- 不要使用嵌套字典和过长的元组
- 简单不可变的数据可以用 `namedtuple` 来表示，如有需要，可以修改成类
- 使用多个辅助类来拆分过于复杂的字典

第 23 条：简单的接口应该接收函数，而不是类的实例

很多 `Python` 内置的 `API` 允许调用者传入函数，以定制其行为。
`API` 在执行的时候可以回调这些挂钩 `hook` 函数。

```
# 挂钩函数
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=lambda x: len(x))
print(names)
"""
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
"""
```

其它编程语言可能会用抽象类来定义挂钩。
在 `Python` 中，很多挂钩知识无状态的函数，有明确的参数及返回值。

用函数做挂钩的好处：

- 容易描述出这个挂钩的功能
- 比定义一个类简单

`Python` 中的函数之所以可以充当挂钩，是因为它是一级对象 `first-class object`。
也就是说，函数和方法可以像语言中的其他值那样传递和引用。

`defaultdict` 类允许使用者提供一个函数，当查询不存在的键的时候，不会抛出 `KeyError`，而是会根据函数为该键生成一个新值。
我们可以自定义 `defaultdict` 的挂钩函数。

```
from collections import defaultdict
```

```python
# 自定义 defaultdict 的挂钩函数
def log_missing():
    print('Key added')
    return 0

# 使用挂钩函数
current = {'green': 12, 'blue': 3}
increments = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9)
]
# defaultdict([工厂函数[, 可迭代对象/关键字参数]])
result = defaultdict(log_missing, current)
print('Before:', dict(result))
for key, amount in increments:
    result[key] += amount
print('After:', dict(result))
"""
Before: {'green': 12, 'blue': 3}
Key added
Key added
After: {'green': 12, 'blue': 20, 'red': 5, 'orange': 9}
"""
```

> 【注】补充
>
> - `defaultdict()` 第一个参数是 `default_factory`，默认为 `None`，还可以传一个工厂函数。当访问默认字典 `defaultdict` 中不存在的键的时候，`defaultdict` 会自动调用这个工厂函数，给该键生成一个默认值。
> - `defaultdict()` 的其余参数相当于传给 `dict()` 构造函数。一般为一个可迭代对象，或者多个关键字参数。
> - `defaultdict()` 比 `dict.setdefault()` 使用起来更简单，而且更快。
> - defaultdict objects

需求：给 `defaultdict` 传一个工厂函数，需要统计一共遇到多少次缺失的键。

```python
# 使用带状态的闭包
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count  # Stateful closure
        added_count += 1
        return 0

    result = defaultdict(missing, current)
    for key, amount in increments:
```

```
            result[key] += amount

        return result, added_count

    result, count = increment_with_report(current, increments)
    assert count == 2
```

带状态的闭包函数用作挂钩的**缺点**：读起来比无状态的函数难懂。
改进：**定义辅助类**，封装需要追踪的状态。

```
    # 自定义辅助类
    class CountMissing(object):
        def __init__(self):
            self.added = 0

        def missing(self):
            self.added += 1
            return 0

    counter = CountMissing()
    result = defaultdict(counter.missing, current)

    for key, amount in increments:
        result[key] += amount
    assert counter.added == 2
```

辅助类比带状态的闭包函数更容易理解。
等到看过 `defaultdict` 的用法之后才会明白 `CountMissing` 类的意图。
改进：**定义可调用的对象**，即带有 `__call__()` 方法的类。

```
    # 定义可调用的对象
    class BetterCountMissing(object):
        def __init__(self):
            self.added = 0

        def __call__(self):
            self.added += 1

    # 测试 BetterCountMissing 的实例是否可调用
    counter = BetterCountMissing()
    counter()
    assert callable(counter)

    # 把可调用的对象作为挂钩函数
    counter = BetterCountMissing()
    result = defaultdict(counter, current)
    for key, amount in increments:
        result[key] += amount
    assert counter.added == 2
```

`__call__()` 方法**强烈地暗示了这个类的用途。**
它告诉我们，这个类的功能就相当于一个带有状态的闭包。

所以，如果要用函数来保存状态，那就应该定义新的类，实现 `__call__()` 方法，而不要定义带状态的闭包。

第 24 条：以 @classmethod 形式的多态取通用地构建对象

多态，使得集成体系中的多个类都能以各自所独有的方式来实现某个方法。这些类，都满足相同的接口或继承自相同的抽象类，但却有着各自不同的功能。

为了实现 `MapReduce` 流程（映射-化简、映射-推导流程），定义公共基类来表示输入的数据，它的 `read` 方法必须由子类来实现：

```
class InputData(object):
    def read(self):
        raise NotImplementedError
```

现在编写 `InputData` 的具体子类，以便从磁盘文件里读取数据：

```
class PathInputData(InputData):
    """来自磁盘文件的输入数据类"""
    def __init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        return open(self.path).read()
```

我们可能需要很多像 `PathInputData` 这样的类来充当 `InputData` 的子类，每个子类都需要实现标准接口中的 `read` 方法，并以字节形式返回待处理的数据。其他的 `InputData` 子类可能会通过网络读取并解压缩数据。

此外，我们还需要为 `MapReduce` 共同工作线程定义一套类似的抽象接口，以便用标准的方式来处理输入的数据。

```
class Worker(object):
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None
```

```
    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

下面定义具体的 `Worker` 子类，以实现我们想要的 `MapReduce` 功能。本例所实现的功能是一个简单的**换行符计数器**。

```
class LineCountWorker(Worker):
    def map(self):
        # 调用了输入数据类实例的 read 方法
        data = self.input_data.read()
        self.result = data.count('\n')

    def reduce(self, other):
        self.result += other.result
```

如何把这些组件拼接起来？现在，由谁来负责构建对象并协调 `MapReduce` 流程呢？
最简单的办法是手工构建相关对象，并通过某些辅助函数将这些对象联系起来。

下面这段代码可以列出某个目录的内容，并为该目录下的每个文件创建一个 `PathInputData` 实例：

```
def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))
```

> 团子注：`data_dir` 里面不能有目录，要不然会出错。`os.listdir(path)` 能够输出 `path` 目录下的所有名称，包括带后缀的文件名和目录名，但是不包括 `.` 和 `..` 这两个名称。所以如果带有目录，而目录不支持 `PathInputData` 中的 `read` 操作，这样会报错。

然后，用 `generate_inputs` 方法所返回的 `InputData` 实例来创建 `LineCountWorker` 实例。

```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

现在执行这些 `Worker` 实例，以便将 `MapReduce` 流程中的 `map` 步骤派发到多个线程之中。接下来，反复调用 `reduce` 方法，将 `map` 步骤的结果合并成一个最终值。

```
def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
    # join 的作用是线程同步，可以让主线程等待所有子线程的完成再终止
    for thread in threads: thread.join()

    first, rest = workers[0], workers[1:]
    for worker in rest:
        first.reduce(worker)
    return first.result
```

最后，把上面这些代码片段都拼装到函数里面，以便执行 `MapReduce` 的每个步骤。

```
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

用一些列输入文件来测试 `mapreduce` 函数，可以得到正常的结果。

```
def write_test_files(tempdir):
    # 团子注：这个是我自己随便写的内容
    # 答案一共是 7 个换行
    with open('tmp1.txt', 'w') as f:
        f.write('四\n个\n换\n行\n')
    with open('tmp2.txt', 'w') as f:
        f.write("""为了看看阳光
        我来到这世上
        莱蒙托夫
        """)


from tempfile import TemporaryDirectory

with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    result = mapreduce(tmpdir)

print('There are', result, 'lines')
"""
There are 7 lines
"""
```

但是这种写法有个大问题，就是 `MapReduce` 函数不够通用。如果要编写其他的 `InputData` 或 `Worker` 子类，那就得重写 `generate_inputs`、`create_workers` 和 `mapreduce` 函数，以便与之匹配。

要解决这个问题，就需要以一种通用的方式来构建对象。在其他编程语言中，可以通过构造器多态来解决，也就是令每个 `InputData` 子类都提供特殊的构造器，使得协调 `MapReduce` 流程的那个辅助方法可以用它来通用地构造 `InputData` 对象。但是，`Python` 只允许名为 `__init__` 的构造器方法，所以我们不能要求每个 `InputData` 子类都提供兼容的构造器。

> 团子注：这个不是很明白，其他编程语言中又是什么样的呢？

解决这个问题的最佳方案，是使用 `@classmethod` 形式的多态。这种多态形式，其实与 `InputData.read` 那样的实例方法多态非常相似，只不过它针对的是整个类，而不是从该类构建出来的对象。

现在我们用这套思路来实现与 `MapReduce` 流程有关的类。首先，修改 `InputData` 类，为它添加通用的 `generate_inputs` 类方法，该方法会根据通用的接口来创建新的 `InputData` 实例。

```
class GenericInputData(object):
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

新添加的 `generate_inputs` 方法，接受一份含有配置参数的字典，而具体的 `GenericInputData` 子类则可以解读这些参数。下面这段代码通过 `config` 字典来查询输入文件所在的目录：

```
class PathInputData(GenericInputData):
    # 团子注：这里省略了 __init__

    def read(self):
        return open(self.path).read()

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

接下来，按照类似的方式实现 `GenericWorker` 类的 `create_workers` 辅助方法。为了生成必要的输入数据，调用者必须把 `GenericInputData` 的子类传给该方法的 `input_class` 参数。该方法用 `cls()` 形式的通用构造器，来构造具体的 `GenericWorker` 子类实例。

```
class GenericWorker(object):
    # ...

    def map(self):
```

```
            raise NotImplementedError

        def reduce(self, other):
            raise NotImplementedError

        @classmethod
        def create_workers(cls, input_class, config):
            workers = []
            for input_data in input_class.generate_inputs(config):
                workers.append(cls(input_data))
            return workers
```

上面代码重点是 `input_class.generate_inputs`，它是个类级别的多态方法。
此外，`create_workers` 用另外一种方式构造了 `GenericWorker` 对象，它是通过 `cls` 形式来构造的，而不是像以前那样，直接使用 `__init__` 方法。

至于具体的 `GenericWorker` 子类，则只需修改它所继承的父类即可：

```
class LineCountWorker(GenericWorker):
    # 团子注：这里需要定义 map 和 reduce 方法
    pass
```

最后重写 `mapreduce` 函数，令其变得完全通用。

```
def mapreduce(worker_class, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)
```

这种方式跟前面的区别在于，现在的 `mapreduce` 函数需要更多的参数，以便用更加通用的方式来操作相关对象。

```
with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    config = {'data_dir': tmpdir}
    result = mapreduce(LineCountWorker, PathInputData, config)
```

我们可以继续编写其他的 `GenericInputData` 和 `GenericWorker` 子类，而且不用再修改刚才写好的那些代码了。

要点

- 在 `Python` 程序中，每个类只能有一个构造器，也就是 `__init__` 方法。
- 通过 `@classmethod` 机制，可以用一种与构造器相仿的方式来构造类的对象。
- 通过类方法的多态机制，我们能够以更加通用的方式来构造并拼接具体的子类。

```

```

> 团子注：The most important takeaway from this article is using @classmethod as constructor.

第 25 条：用 super 初始化父类

第 28 条：继承 collections.abc 以实现自定义的容器类型

大部分的 `Python` 编程工作，其实都是在**定义类**。
`Python` 中的类，可以说都是**容器**，因为它们都封装了属性与功能。
内置容器类型有：列表、元组、集合、字典等。

需求：创建一个列表类型，并提供统计各元素出现频率的方法。
解决：可以继承 `list`。

```
class FrequencyList(list):
    def __init__(self, members):
        super().__init__(members)

    def frequency(self):
        counts = {}
        for item in self:
            counts.setdefault(item, 0)
            counts[item] += 1
        return counts

# 继承于 list 的类拥有 list 提供的全部标准功能
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd'])
print('长度为：%s' % len(foo))
foo.pop()
print('弹出之后为：%s' % repr(foo))
print('频率为：%s' % foo.frequency())
"""
长度为：7
弹出之后为：['a', 'b', 'a', 'c', 'b', 'a']
频率为：{'a': 3, 'b': 2, 'c': 1}
"""
```

需求：定义一个二叉树的节点类，它本身不属于 `list` 的子类，但用起来跟序列类型一样，可以通过下标访问其元素。
解决：可以定义 `__getitem__()` 方法。
例子：

```
bar = [1, 2, 3]
bar[0]
```

`Python` 会转译成 `bar.__getitem__(0)`。

---

6. 内置模块

第 42 条：用 functools.wraps 定义函数修饰器

使用装饰器 `decorator` 可以在装饰器里访问并修改原函数的参数及返回值，以实现语义约束 `enforce semantics`，调试程序，注册函数等目标。

打印函数接收的参数和返回值，这种调试功能对于一些递归函数非常有用。

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('%s(%r, %r) -> %r' %
              (func.__name__, args, kwargs, result))
        return result
    return wrapper


@trace  # 等价于：fibonacci = trace(fibonacci)
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)
```

使用 `@` 符号来装饰一个函数，其效果等于先以该函数为参数，调用装饰器。然后把装饰器的返回结果，赋值给同一个作用域中与原函数同名的哪个变量。

但是装饰器的应用会带来副作用，原函数的名称发生了改变。

```
# 副作用：装饰器返回的那个值，名称和原函数不同
print(fibonacci)
"""
<function trace.<locals>.wrapper at 0x0000022B20B9A6A8>
"""
```

对于需要使用内省 `introspection` 机制的那些工具来说，这会干扰它们的正常工作，比如调试器（第 57 条）和对象序列化器（第 44 条）。

```
# fibonacci 的 help 失效
print(help(fibonacci))
"""
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)

None
"""
```

`functools` 模块中的 `wraps` 函数，它本身也是装饰器。
将 `wraps` 装饰器运用到 `wrapper` 函数之后，它会把内部函数相关的重要元数据全部复制到外围函数，比如 `__name__` 和 `__module__` 等。

第 43 条：考虑以 contextlib 和 with 语句来改写可服用的 try/finally 代码

有些代码，需要运行在特殊的情境之下，可以使用 `with` 来表达这些代码的运行时机。
比如互斥锁放在 `with` 中，表示：只有当程序持有该锁的时候，`with` 语句中的代码才会执行。

```
from threading import Lock

# Lock 类支持 with 语句
lock = Lock()
with lock:
    print('Lock is held 拿到锁了')

# 等价的 try/finally 写法
lock.acquire()
try:
    print('Lock is held 拿到锁了')
finally:
    lock.release()
```

支持 `with` 语句有两种方式：

- 自定义类，实现 `__enter__` 和 `__exit__` 方法。
- 使用 `contextlib` 中的 `contextmanager` 装饰器

`contextmanager` 叫做上下文管理器，或者环境管理器。

补充：`logging` 模块只会打印出严重程度 `severity level` 大于或等于自身级别的消息。