函数式编程思维

回溯编程语言的历史就会发现，早期编程语言主要用来描述机器运行行为以及为数学家服务。在1950到1960年期间被发明的重要语言都是用于数值计算和数学函数式编程。随着计算机的通用化，面向对象、结构化、并发、逻辑等等基础范式慢慢演进确立。这些语言主要是用来解决具体的工程问题，充分利用机器的性能。

在数字化时代，数据量越来越多，对数据计算、处理和分析的诉求比以往更多。数值计算和函数式编程重新回归编程范式的主流，并且扩展到各领域广泛应用。

函数式编程，指输入数据经一组函数处理，每个函数都在其输入上运行并产生一些输出。函数式编程不赞成使用有“副作用”的函数，避免函数修改内部状态，避免函数进行其他在函数返回值中不可见的改变。没有任何“副作用”的函数被称为纯粹的功能。避免副作用意味着在程序运行时不能改变使用的数据结构，每个函数的输出只能依赖于它的输入。

一些编程语言对纯函数的要求非常严格，甚至不允许出现赋值语句。但在真实的编程中完全避免“副作用”非常困难，因为打印到屏幕终端或者是写入文件都是副作用。Python的print()和time.sleep()函数都没有任何结果返回，但它们一个发送文本到屏幕，一个暂停程序执行1秒钟，都产生了副作用。

用函数式编写的Python程序通常不会避免所有的I/O操作或任务执行。程序会提供一些函数式的功能接口，但在内部会使用非函数式的一些特性。 通常会实现一个函数仍然会使用赋值给局部变量，但它不会修改全局变量或有其他副作用。

函数式编程可以被认为是面向对象编程的反面。 对象是包含一些内部状态的小胶囊以及方法调用的集合，可以让你修改这些状态，程序定义了如何正确的修改数据状态。 函数式编程希望尽可能地避免状态变化，并且处理函数之间的数据流。 在Python中，可以通过编写函数来结合这两种方法，这些函数接收和返回应用程序中对象的实例（电子邮件消息，事务处理等）。

一个简单的例子

我们从一个简单的例子开始。下面是一个海鸥程序，鸟群合并则变成了一个更大的鸟群，繁殖则增加了鸟群的数量，增加的数量就是它们繁殖出来的海鸥的数量。注意这个程序并不是面向对象的良好实践，它只是强调当前这种变量赋值方式的一些弊端。

```python
class Flock(object):
    def __init__(self, n):
        self.seagulls = n

    def conjoin(self, other):
        self.seagulls += other.seagulls
        return self

    def breed(self, other):
        self.seagulls = self.seagulls * other.seagulls
        return self

flock_a = Flock(4)
flock_b = Flock(2)
flock_c = Flock(0)
result = flock_a.conjoin(flock_c).breed(flock_b).conjoin(flock_a.breed(flock_b))
print(result)

>>> 32
```

上面的示例代码内部可变状态非常难以追踪，而且最终的答案还是错的！正确答案是 16。 但是因为 flock_a 在运算过程中永久地改变了，所以得出了错误的结果。 如果你看不懂这个程序，没关系。示例的重点是状态和可变值非常难以追踪，即便是在这么小的一个程序中也不例外。

我们试试另一种更函数式的写法：

```python
def conjoin(flock_x, flock_y):
    return flock_x + flock_y


def breed(flock_x, flock_y):
    return flock_x * flock_y

flock_a = 4
flock_b = 2
flock_c = 0
result = conjoin(breed(flock_b, conjoin(flock_a, flock_c)), breed(flock_a, flock_
print(result)

>>> 16
```

这次我们得到了正确的答案，而且少写了很多代码。不过函数嵌套有点让人费解（之后会解决这个问题）。这种写法也更优雅，不过代码肯定是越直白越好，所以如果我们再深入挖掘，看看这段代码究竟做了什么事，我们会发现，它不过是在进行简单的加（conjoin） 和乘（breed）运算而已。 代码中的两个函数除了函数名有些特殊，其他没有任何难以理解的地方。我们把它们重命名一下，看看它们

的真面目。

```python
def add(flock_x, flock_y):
    return flock_x + flock_y


def multiply(flock_x, flock_y):
    return flock_x * flock_y

flock_a = 4
flock_b = 2
flock_c = 0
result = add(multiply(flock_b, add(flock_a, flock_c)), multiply(flock_a, flock_b)
print(result)

>>> 16
```

这么一来，你会发现我们不过是在运用高中已获得的知识：

```python
# 结合律 (assosiative)
add(add(x, y), z) == add(x, add(y, z))

# 交换律 (commutative)
add(x, y) == add(y, x)

# 恒等式 (identity)
add(x, 0) == x

# 分配律 (distributive)
multiply(x, add(y,z)) == add(multiply(x, y), multiply(x, z))
```

我们来看看能否运用这些定律简化这个海鸥小程序。

```python
# python内置计算
from operator import add, mul

# 原有代码
add(multiply(flock_b, add(flock_a, flock_c)), multiply(flock_a, flock_b))

# 应用同一律，去掉多余的加法操作 (add(flock_a, flock_c) == flock_a)
add(mul(flock_b, flock_a), mul(flock_a, flock_b))

# 再应用分配律
mul(flock_b, add(flock_a, flock_a))
```

除了调用的函数，一点多余的代码都不需要写。当然这里我们定义 add 和 multiply 是为了代码完整性，实际上并不必要在调用之前它们在某个类库里定义好了。

你可能在想“你也太偷换概念了吧，居然举一个这么数学的例子”，或者“真实世界的应用程序比这复杂太多，不能这么简单地推理”。我之所以选择这样一个例子，是因为大多数人都知道加法和乘法，所以很容易就能理解数学可以如何为我们所用。

不要感到绝望，本书后面还会穿插一些范畴学（category theory）、集合论（set theory）以及 lambda 运算的知识，教你写更加复杂的代码，而且一点也不输本章这个海鸥程序的简洁性和准确性。你也不需要成为一个数学家，本书要教给你的编程范式实践起来就像是使用一个普通的框架或者 API 一样。

我们可以像上例那样遵循函数式的范式去书写完整的、日常的应用程序，有着优异性能的程序，简洁且易推理的程序，以及不用每次都重新造轮子的程序。我们希望能够承认并遵守数学之法。

我们希望去践行每一部分都能完美接合的理论，希望能以一种通用的、可组合的组件来表示我们的特定问题，然后利用这些组件的特性来解决这些问题。相比命令式编程的那种“某某去做某事”的方式，函数式编程将会有更多的约束，不过你会震惊于这种强约束、数学性的“框架”所带来的回报。 我们已经看到函数式的点点星光了，但在真正开始我们的旅程之前，我们要先掌握一些具体的概念。

函数式编程特性

函数式编程使用表达式和评估来定义计算——通常封装在函数定义中。它不再强调或者说避免状态变化和可变对象的复杂性，去创建更简洁和更有表达的程序。在教程中，我们将介绍一些表明函数式编程特征的技术点，通过实例来描述这些特征。最后将讨论在使用这些设计模式构建Python应用程序时，函数式编程带来的好处。 Python有许多函数式编程特征，但它不是一个纯粹的函数式编程语言。Python提供了足够多也足够有效的函数式编程特性，语言本身赋予函数式编程的好处，也保留了命令式编程语言提供的所有优化功能。

另外在本教程中，也会密切关注探索性数据分析（EDA），因为它的算法通常是函数式编程的最佳例子。

头等函数

头等函数（first-class function）是指在程序设计语言中，函数被当作头等公民。这意味着，函数可以作为别的函数的参数、函数的返回值，赋值给变量或存储在数据结构中。有人主张应包括支持匿名函数（函数字面量，function literals）。在这样的语言中，函数的名字没有特殊含义，它们被当作具有函数类型的普通的变量对待。1960年代中期，克里斯托弗·斯特雷奇在“functions as first-class citizens”中提出这一概念。可以简单总结为作为头等公民，其他类型具备的特征，函数都具备。

头等函数是函数式程序设计所必须的，通常还要使用高阶函数。把函数作为参数和返回值会遇到一些困难，特别是存在非局部变量、嵌套函数和匿名函数。历史上，这被称作函数参数问题。早期的命令式编程语言，或者不支持函数作为结果类型（如ALGOL 60, Pascal），或者忽略嵌套函数与非局部变

量（如C语言）。早期的函数式语言Lisp采取了动态作用域方法，把非局部变量绑定到函数执行点最近的变量定义。Scheme语言支持词法作用域的头等函数，把对函数的引用绑定到闭包（closure）而不是函数指针，这使得垃圾收集成为必须。

在Python里面函数编译为bytecode对象，可以作为参数和返回值传递，也可以在嵌套或作为匿名函数执行。利用这种属性生成的高阶函数如map等也非常易用。

```python
def example(a, b, **kw):
    return a*b

type(example)
# <class 'function'>
example.__code__.co_varnames
('a', 'b', 'kw')
example.__code__.co_argcount
# 2
```

Language		高阶函数		非局部变量			局部应用	注释
		实参	返回结果	嵌套函数	匿名函数	闭包		
Algol 家族	ALGOL 60	是	否	是	否	否	否	有函数类型
	ALGOL 68	是	是[8]	是	是	否	否	
	Pascal	是	否	是	否	否	否	
	Oberon	是	Non-nested only	是	否	否	否	
	Delphi	是	是	是	2009	2009	否	
C 家族	C	是	是	否	否	否	否	有函数指针
	C++	是	是	C++11 closures[9]	C++11[10]	C++11[10]	C++11	有函数指针、函数对象。使用 std::bind 可显式局部应用。
	C#	是	是	部分	2.0 / 3.0	2.0	3.0	有delegate(2.0)及lambda表达式(3.0)
	Go	是	是	是	是	是	否	
	Objective-C	是	是	否	2.0 + Blocks[11]	2.0 + Blocks	否	有函数指针
	Java	部分	部分	否	Java 8	Java 8	否	有匿名内部类.
	Limbo	是	是	是	是	是	否	
	Newsqueak	是	是	是	是	是	否	
	Rust	是	是	是	是	是	否	
函数式语言	Lisp	Syntax	Syntax	是	是	Common Lisp	否	(见下)
	Scheme	是	是	是	是	是	SRFI 26[12]	
	Clojure	是	是	是	是	是	是	
	ML	是	是	是	是	是	是	
	Haskell	是	是	是	是	是	是	
	Scala	是	是	是	是	是	是	
脚本语言	JavaScript	是	是	是	是	是	ECMAScript 5	用户代码在ES3上有局部应用 [13]
	PHP	是	是	5.3 closures	5.3 closures	5.3 closures	否	户代码可有局部应用
	Perl	是	是	anonymous, 6	是	是	6[14]	(见下)
	Python	是	是	是	部分	是	2.5[15]	(见下)
	Ruby	Syntax	Syntax	Unscoped	是	是	1.9	(见下)
其他语言	Io	是	是	是	是	是	否	
	Maple	是	是	是	是	是	否	
	Mathematica	是	是	是	是	是	否	
	MATLAB	是	是	是	是[16]	是	是	新的函数可自动产生局部应用[17]
	Smalltalk	是	是	是	是	是	部分	通过库可有局部应用
	Fortran	是	是	是	否	否	否	
	Swift	是	是	是	是	是	是	
	Ada	是	是	是	否	Downward Closure	否	

嵌套函数与闭包

python是允许创建嵌套函数的，也就是说我们可以在函数内部定义一个函数，这些函数都遵循各自的作用域和生命周期规则。

```python
def outer():
    x = 1
    def inner():
        print(x) # 1
    inner() # 2

outer()
outer. __closure__
# None
```

上面这个函数是嵌套函数，在outer中声明了一个inner，然后执行函数inner。

1. **1的地方，python寻找名为x的local变量，在inner作用域内的locals中寻找不到，python就在外层作用域中寻找，其外层是outer函数。x是定义在outer作用域范围内的local变量。**
2. #2的地方，调用了inner函数。这里需要特别注意：inner只是一个变量名，是遵循python的变量查找规则。

我们把程序改进一步变成下面的函数：

```python
def outer():
    x = 1
    def inner():
        y = 2
        print(x) # 1
        print(y) # 2
    return inner # 3

foo = outer()
foo. __closure__
# output:  (<cell at 0x108c76a38: int object at 0x106198830>,)
```

1. **1的地方，python寻找名为x的local变量，在inner作用域内的locals中寻找不到，python就在外层作用域中寻找，其外层是outer函数。x是**

定义在outer作用域范围内的local变量。

2. **2的地方，y是inner的local变量。提问：假设把y=2改为x=2，打印出来的会是?**
3. **3的地方，返回了inner函数。这里需要特别注意：inner也只是一个变量名，是遵循python的变量查找规则的（Python先在outer函数的作用域中寻找名为inner的local变量）**

> 上面的示例也是一个高阶函数，高阶函数的定义和使用会在接下来的章节说明，这里请大家关注嵌套函数的使用。

两个函数对比之后，我们可以发现，如果你的嵌套函数没有：

- 访问本地封闭范围之外的变量，
- 不会在这个范围之外执行时，

那么它不是闭包的。

闭包的定义

在一些语言中，函数中可以（嵌套）定义另一个函数。如果内部的函数引用了外部的函数的变量，则可能产生闭包(closure)。运行时，一旦外部的函数被执行，一个闭包就形成了。 闭包中包含了内部函数的代码，以及所需外部函数中的变量的引用。其中所引用的变量称作上值(upvalue)。

那么闭合有什么好处呢?

闭包可以避免使用全局值，并提供某种形式的数据隐藏。它也可以为这个问题提供一个面向对象的解决方案。

当一个类中只有很少的方法（大多数情况下是一种方法）时，闭包可以提供一个备用和更优雅的解决方案。 但是，当属性和方法的数量变大时，应该去实现一个类。

下面是一个简单的例子，闭包可能比定义类和设计对象更可取。

```python
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier

# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)

# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

匿名函数

python 允许用 lambda 关键字创造匿名函数。匿名是因为不需要以标准的方式来声明，比如说， 使用 def 语句。(除非赋值给一个局部变量，这样的对象也不会在任何的名字空间内创建名字.)然而， 作为函数，它们也能有参数。一个完整的 lambda“语句”代表了一个表达式，这个表达式的定义体 必须和声明放在同一行。我们现在来演示下匿名函数的语法:

> lambda [arg1[, arg2, ... argN]]: expression

参数是可选的，如果使用的参数话，参数通常也是表达式的一部分。

```python
def true(): return True

# 等价于
true = lambda :True
```

用合适的表达式调用一个lambda生成一个可以像其他函数一样使用的函数对象。 它们可被传入给其他函数，用额外的引用别名化，作为容器对象以及作为可调用的对象被调用(如果需要的话， 可以带参数)。当被调用的时候，如过给定相同的参数的话，这些对象会生成一个和相同表达式等价 的结果。它们和那些返回等价表达式计算值相同的函数是不能区分的。

高阶函数

在数学和计算机科学中，高阶函数是至少满足下列一个条件的函数：

- 接受一个或多个函数作为输入
- 输出一个函数

在数学中，高阶函数也被称为运算符或函数，其他函数都是一阶函数。微积分中的微分算子是一个常见的例子，因为它将一个函数映射到它的导数，也是一个函数。 高阶函数不应与整个数学中“函子”这个词的其他用法混淆。

在无类型的lambda演算中，所有函数都是高阶的; 在一个类型化的lambda演算中，大多数函数式编程语言都是从这个演算中派生出来的，以一个函数作为参数的高阶函数是具有表单类型的值
$(T1->T2)->T3$

在很多函数式编程语言中都能找到高阶函数map，它接受一个函数f作为参数，并返回接受一个列表并应用f到它的每个元素的一个函数。

用Python的内置函数 Max() 作为示例，我们先来看看Max函数是如何来工作的。

```Python
>>> year_cheese = [(2000, 29.87), (2001, 30.12), (2002, 30.6), (2003,
30.66),(2004, 31.33), (2005, 32.62), (2006, 32.73), (2007, 33.5),
(2008, 32.84), (2009, 33.02), (2010, 32.92)]

>>> max(year_cheese)
(2010, 32.92)
```

上面的代码可以看出，max默认对比元组中的第一个元素，返回第一个元素最大的元组数据。因为max函数是高阶函数，支持接收一个函数作为key参数，定制对比规则。

```Python

>>> max(year_cheese, key=lambda yc: yc[1])

(2007, 33.5)
```

max函数是一个相对简单的高阶函数，Python提供了丰富的高阶函数集合，在后续会陆续介绍说明。

数据不可变性

函数式编程避免使用变量来跟踪计算的状态，焦点需要停留在不可变的对象上。所以在函数式编程中广泛使用tuples和namedtuples来提供更复杂的、不可变的数据结构。 不可变对象的思想对Python来说并不陌生，使用不可变元组代替更复杂的可变对象会有性能优势。在某些情况下，好处在于重新考虑算法，以避免对象变异的代价。

在函数式编程中一般会完全避免类定义，也不需要有状态的对象，但在面向对象编程（OOP）语言中

避免使用类就很奇怪。我们将在本教程中看到这一点。 定义可调用对象是有原因的，为紧密相关的功能提供命名空间是一种整洁的方式，它支持令人愉快的可用性级别。

现在介绍一个常用的设计模式，它可以很好地处理不可变的对象：wrapper()函数。我们经常用以下两种方法之一来处理不可变对象元组列表：

- 使用高阶函数：如前所述，我们提供了lambda作为max()函数的参数：max(year_cheese, key=lambda yc：yc[1])
- 使用Wrap-Process-Unwrap模式：在一个功能上下文中，我们称之为unwrap(process(wrap(structure))) 模式

```Python
>>> max(map(lambda yc:(yc[1],yc), year_cheese))
(33.5, (2007, 33.5))

>>> _[1]
(2007, 33.5)
```

上面这段代码使用了三种模式，虽然看起来并不明显。 - 首先，我们使用 wrap模式 map(lambda yc: (yc[1]，yc)，year_cheese) 进行换行。把每个数据项转换成(key, item)的元组。在这个例子中，比较键仅是yc[1]。 - 其次，使用高阶max()函数进行处理。由于每个数据片段都被简化为一个位置为零的二元组，所以我们并不需要max()函数的高阶函数特征。 max()函数的默认行为就是我们需要的。 - 最后，我们用unwrap模式，即下标[1]。它将取出max()函数返回的两个元组的第二个元素。

这种wrap和unwrap非常常见，以至于一些语言如haskell和ocaml作为内置的函数功能，函数名为fst()和snd()。 在Python中也可以将它们用作为函数，而不是[0]或[1]的语法结构。按这个想法来修改我们的wrap-process-unwrap例子，如下所示：

```Python
snd= lambda x: x[1]

snd(max(map(lambda yc: (yc[1],yc), year_cheese)))
```

定义了一个snd()函数来从元组中选取第二个元素。这为我们提供了一个更容易阅读的unwrap(process(wrap())) 版本。 我们使用map(lambda ...，year_cheese) 来wrap我们的原始数据项目。 我们使用max()函数来处理过程，最后snd()函数从元组中提取第二项。

纯函数/无副作用

为了更具表达性，函数式编程设计中使用的函数要保证没有“副作用(Side-Effects)”造成的混淆。 使用纯函数也可以通过改变执行顺序来进行程序优化，但纯函数最大的优势是在概念上更简单、更容易测试。

> 纯函数是这样一种函数，即相同的输入，永远会得到相同的输出，而且没有任何可观察的副作

> 用。

要在Python中编写纯函数，我们必须只使用本地变量，我们必须避免全局声明。 需要密切关注任何非本地变量的使用，这些可能另一个范围产生副作用，尽量定义为一个嵌套的函数。 在Python中这是一个很容易达到的标准，纯函数也是Python编程的一个常见功能。

在纯函数定义中提到的万分邪恶的副作用到底是什么？“作用”我们可以理解为一切除结果计算之外发生的事情。 “作用”本身并没什么坏处，而且在本书后面的章节你随处可见它的身影。“副作用”的关键部分在于“副”。就像一潭死水中的“水”本身并不是幼虫的培养器，“死”才是生成虫群的原因。同理，副作用中的“副”是滋生bug的温床。

> 副作用是在计算结果的过程中，系统状态的一种变化，或者与外部世界进行的可观察的交互。

副作用可能包含，但不限于： - 更改文件系统 - 往数据库插入记录 - 发送一个 http 请求 - 可变数据 - 打印/log - 获取用户输入 - 访问系统状态

这个列表还可以继续写下去。概括来讲，只要是跟函数外部环境发生的交互就都是副作用——这一点可能会让你怀疑无副作用编程的可行性。函数式编程的哲学就是假定副作用是造成不正当行为的主要原因。确保Python函数没有副作用的影响并不是一件简单的事情。

实际编程中很容易不小心就破坏纯函数的规则。如果担心遵循这个规则的能力，我们可以编写一个函数，使用dis模块来扫描给定函数的**code**.co*code变量，这个编译后的值有全局引用说明。它可以报告内部闭包的使用情况，以及code.co*freevars元组方法。这是一个相当复杂的解决罕见问题的办法，具体的方法暂时不会进一步深究。 Python lambda是一个纯函数。虽然这不是强烈推荐的风格，但通过lambda值创建纯函数当然可以的。

这并不是说，要禁止使用一切副作用，而是说，要让它们在可控的范围内发生。后面讲到 functor 和 monad 的时候我们会学习如何控制它们，目前还是尽量远离这些阴险的函数为好。 副作用让一个函数变得不纯是有道理的：从定义上来说，纯函数必须要能够根据相同的输入返回相同的输出；如果函数需要跟外部事物打交道，那么就无法保证这一点了。

```Python
>>> mersenne = lambda x: 2**x-1
>>> mersenne(17)
131071
```

我们使用lambda创建了一个纯函数，并将其分配给变量mersenne。 这是具有单个参数值的可调用对象，它返回单个值。 因为lambda不能有赋值语句，所以它们总是纯粹的函数，并且适用于函数式编程。

追求“纯”的理由

- 可缓存性（Cacheable）
- 可移植性 / 自文档化
- 可测试性（Testable）

- 并行代码

如果手头没有一些工具，那么纯函数程序写起来就有点费力。我们不得不玩杂耍似的通过到处传递参数来操作数据，而且还被禁止使用状态，更别说“作用”了。没有人愿意这样自虐。所以让我们来学习一个叫 curry 的新工具。

柯里化和局部函数

curry 的概念很简单：只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。 你可以一次性地调用 curry 函数，也可以每次只传一个参数分多次调用。

```Python
def add(x):
    return lambda y: x+y

increment = add(1)
addTen = add(10)

increment(2)
>>> 3

addTen(2);
>>> 12
```

这里我们定义了一个 add 函数，它接受一个参数并返回一个新的函数。调用 add 之后，返回的函数就通过闭包的方式记住了 add 的第一个参数。一次性地调用它实在是有点繁琐，好在我们可以使用一个特殊的 curry 帮助函数（helper function）使这类函数的定义和调用更加容易。

在Python中，函数的局部执行是通过functools的高阶函数**partial**来处理的，柯里化提供了语法糖。

```Python
from operator import mul
from functools import partial

double = partial(mul, 2)  # Partial evaluation
doubled = double(2)       # Currying
```

这些语法糖配合高阶函数非常有价值。

```python
from toolz import compose, frequencies
# compose 组合函数会在下一部分具体说明

def stem(word):
    """ Stem word to primitive form """
    return word.lower().rstrip(",.!:;'-\"").lstrip("'\"")

wordcount = compose(frequencies, partial(map, stem), str.split)

temp_str = "A friend in need is a friend indeed."

wordcount(temp_str)
# output: {'a': 2, 'friend': 2, 'in': 1, 'indeed': 1, 'is': 1, 'need': 1}
```

上面的代码，我们要将词干函数映射到由str.split生成的每个单词。我们需要一个stem_many函数来获取单词列表，然后返回一个列表。 完整的形式如下所示：

```python
def stem_many(words):
    return map(stem, words)
```

局部函数让我们更自然地创造这个功能。

```python
stem_many = partial(map, stem)

# 通俗的局部函数实现如下所示
def f(x, y, z):
    # Do stuff with x, y, and z

# partially evaluate f with known values a and b
def g(z):
    return f(a, b, z)

# partially evaluate f with known values a and b
g = partial(f, a, b)
```

在多数情况下，柯里化只是局部函数执行的语法糖。如果它没有收到足够的参数来计算结果，curried函数会局部执行程序。

```python
from toolz import curry

# 柯里化作为装饰器
@curry
def mul(x, y):
    return x * y

double = mul(2)
# mul没有收到足够的参数来执行
# 所以它保留2的值并等待，返回一个函部函数double

>>> double(5)
10
```

上面的单词统计函数中，如果对map进行柯里化： ``` python map = curry(map)

wordcount = compose(frequencies, partial(map, stem), str.split)

wordcount = compose(frequencies, map(stem), str.split) ``` 在这个特定的例子中可以更简单使用柯里化后的map函数代替局部函数的使用。而toolz内置了标准Python高阶函数的curried版本，比如map，filter，reduce，所以可以直接使用它们。

函数组合

Compose函数指以串联操作一系列函数，函数会以右向左的顺序组合执行，所以compose(f, g, h)(x, y) 和 f(g(h(x, y))) 是同一个意思。 f、g、h 都是函数，x和y 是在它们之间通过“管道”传输的参数。

Compose(组合)看起来像是在饲养函数。选择两个有特点且你喜欢的函数，让它们结合，产下一个崭新的函数。组合的用法如下：

```python
from toolz import compose

def toUpperCase(x):
    return x.upper()

def exclaim(x):
    return x + "!"

shout = compose(exclaim, toUpperCase)

shout("send in the clowns")
# output: 'SEND IN THE CLOWNS!'
```

在 compose 的定义中，g 将先于 f 执行，因此就创建了一个从右到左的数据流。这样做的可读性远远高于嵌套一大堆的函数调用，如果不用组合 shout 函数将会是这样的：

```python
def shout(x):
    return exclaim(toUpperCase(x))
```

让代码从右向左运行，而不是由内而外运行，我觉得可以称之为“左倾”。

尽管我们可以定义一个从左向右的版本，但是从右向左执行更加能够反映数学上的含义——是的，组合的概念直接来自于数学课本。

```python
# 结合律 (associativity)
associative = compose(f, compose(g, h)) == compose(compose(f, g), h)
```

实际上，现在是时候去看看所有的组合都有的一个特性：结合律。符合结合律意味着不管你是把 g 和 h 分到一组，还是把 f 和 g 分到一组都不重要。所以，如果我们想把字符串变为大写，可以这么写：

```python
compose(toUpperCase, compose(head, reverse));

// 或者
compose(compose(toUpperCase, head), reverse);
```

组合像一系列管道那样把不同的函数联系在一起，数据就可以也必须在其中流动——毕竟纯函数就是输入对输出，所以打破这个链条就是不尊重输出，就会让我们的应用一无是处。 我们认为组合是高于其他所有原则的设计原则，这是因为组合让我们的代码简单而富有可读性。另外范畴学将在应用架构、模拟副作用和保证正确性方面扮演重要角色。

引用透明度

引用透明是函数式编程中常用的一个术语，意思是给定一个函数和一个输入值，你将总是得到相同的输出。也就是说在函数中没有使用外部状态。

这是一个引用透明函数的例子：

```python
def plusOne(x):
    return x+1
```

使用引用透明函数给定一个输入和一个函数，可以用一个值代替它，而不是调用该函数。 所以，不要用调用plusOne传入参数5，我们可以用6替换它。

另一个很好的例子是一般的数学，在给定一个函数和一个输入值的数学中，它总是映射到相同的输出值。 f(x)=x+1 因此，数学中的函数是引用透明的。

这个概念对于研究人员来说很重要，因为这意味着当你有一个引用透明的功能时，它就可以简化自动并行和缓存。

相反，存在指称不透明的概念。 这意味着相反。 调用函数可能并不总是产生相同的输出。

```python
G = 10

def plusG(x):
    # 如果G变量被外部函数修改，就会返回不同的结果
    return x + G

plusG(5)
G = 12  # 其他程序修复了全局变量，置为12
plusG(5)
```

另一个例子是面向对象编程语言中的成员函数。成员函数通常对其成员变量进行操作，因此将是不透明的。成员函数当然可以是透明的。

还有一个例子是从文本文件中读取并打印输出的函数，这个外部文本文件可能会随时改变，所以这个函数将是不透明的。

一言蔽之： 引用透明的功能是仅依赖于其输入的功能。

其他特性

Language	Evaluation	Typing	Type Inference	Pattern Matching	Syntax Sugars	GIL	TCO	OO	AGDT
Haskell	Lazy	Static	Yes	Yes	Yes	No	Yes	No	Yes
Ocaml	Strict	Static	Yes	Yes	Yes	Yes	Yes	Yes	Yes
F# (F sharp)	Strict	Static	Yes	Yes	Yes	No	Yes	Yes	Yes
Scheme	Strict	Dynamic	No	No	Yes/ Macros	*	Yes	No	No
Clojure	Strict	Dynamic	No	Yes. Destructuring and macros	Yes/ Macros	No	No	No	No
Scala	Strict	Static	Yes	Yes	Yes	No	Yes	Yes	Yes
Erlang	Strict	Dynamic	?	Yes	Yes	No	Yes	?	?
Elixir	Strict	Dynamic	?	Yes	Yes	No	Yes	?	?
Python	Strict	Dynamic	No	No	No	Yes	No	Yes	No
JavaScript	Strict	Dynamic	No	No	No	?*	No	Yes	No
R	Strict	Dynamic	No	No	No	?	No	Yes	-
Mathematica	Strict	Dynamic	Yes	?	?	?	??	?	?

备注:

> AGDT - 代数数据类型
>
> GIL - 全局线程锁。 使用GIL的语言线程不能利用多核处理器。
>
> TCO - 尾调用优化。 没有TCO的语言不能安全地执行递归，它可能导致大量迭代的堆栈溢出。
>
> OO - 面向对象

迭代器和生成器

在编程语言理论中，懒惰执行或按需执行是一种执行策略，它将表达式的执行延迟到需要它的值（非严格执行），并且也避免重复执行（共享）。共享可以减少某些功能的运行时间的指数因素比其他非严格的执行策略，如名称。

懒惰执行的好处包括：

- 将控制流（结构）定义为抽象而不是基元的能力。
- 能够定义潜在的无限数据结构。 这允许更直接地实施一些算法。
- 通过避免不必要的计算来提高性能，以及执行复合表达式时的错误条件。

这些好处在函数式编程中，特别是在数据处理中特别有用。

我们来看下Python中的懒执行实例。Python默认使用的立即执行，为了在Python中获得懒惰的执行方式，程序员必须使用迭代器或者生成器。

Python

```
def lazy_list():
    """ Infinite list """
    x = 0
    while True:
        x += 2
        yield x  # 生成器

>>> gen = lazy_list()
>>> next(gen)
2
>>> next(gen)
4
>>> next(gen)
6

def take(n, iterable):
    return [next(iterable) for i in range(n)]

def mapi(func, iterable):
    while True:
        yield func(next(iterable))

f = lambda x: x**5

>>> take(5, lazy_list())
[2, 4, 6, 8, 10]
>>> take(10, lazy_list())
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>>

>>> take(5, mapi(f, lazy_list()))
[32, 1024, 7776, 32768, 100000]
>>>
>>> take(6, mapi(f, lazy_list()))
[32, 1024, 7776, 32768, 100000, 248832]
```

迭代器

迭代器是在版本 2.2 被加入 Python 的, 它为类序列对象提供了一个类序列的接口. 我们在 前边的第 6 章已经正式地介绍过序列. 它们是一组数据结构，你可以利用它们的索引从 0 开始一直 "迭代" 到序列的最后一个条目. 用"计数"的方法迭代序列是很简单的. Python 的迭代无缝地支持 序列对象, 而且它还允许程序员迭代非序列类型, 包括用户定义的对象。

迭代器用起来很灵巧, 你可以迭代不是序列但表现出序列行为的对象, 例如字典的 key，一个文件的行等等。当你使用循环迭代一个对象条目时, 你几乎不可能分辨出它是迭代器还是序列. 你不必去关注这些, 因为 Python 让它象一个序列那样操作。

引用 PEP(234) 中对迭代器的定义: - 提供了可扩展的迭代器接口. - 对列表迭代带来了性能上的增强. - 在字典迭代中性能提升. - 创建真正的迭代接口, 而不是原来的随机对象访问. - 与所有已经存在的用户定义的类以及扩展的模拟序列和映射的对象向后兼容 - 迭代非序列集合(例如映射和文件)时, 可以创建更简洁可读的代码.

根本上说, 迭代器就是有一个 next() 方法的对象, 而不是通过索引来计数. 当你或是一个循 环机制(例如 for 语句)需要下一个项时, 调用迭代器的 next() 方法就可以获得它. 条目全部取 出后, 会引发一个 StopIteration 异常, 这并不表示错误发生, 只是告诉外部调用者, 迭代完成。 不过, 迭代器也有一些限制。例如你不能向后移动, 不能回到开始, 也不能复制一个迭代器. 如果你要再次(或者是同时)迭代同个对象, 你只能去创建另一个迭代器对象。

不过还有其他的工具来帮助你使用迭代器，reversed() 内建函数将返回一个反序访问的迭代器。enumerate() 内建函数同样也返回迭代器。 另外两个内建函数 any() 和 all(), 如果迭代器中某个/所有条目的值都为布尔真时，则它们返回值为真.

创建迭代器

对一个对象调用 iter() 就可以得到它的迭代器. 它的语法如下：

```python
iter(obj)
iter(func, sentinel)
```

如果你传递一个参数给 iter(), 它会检查你传递的是不是一个序列。 如果是那么很简单: 根据索引从 0 一直迭代到序列结束。 如果是传递两个参数给 iter() , 它会重复地调用 func , 直到迭代器的下个值等于 sentinel。

另一个创建迭代器的方法是使用类, 一个实现了 **iter**() 和 next() 方法的类可以作为迭代器使用。

```python
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a, b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def next(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration();
        return self.a # 返回下一个值
```

使用迭代器

你之前可能见过迭代器的使用：

```python
>>> myTuple = (123, 'xyz', 45.67)
>>> i = iter(myTuple)
>>> i.next()
123
>>> i.next()
'xyz'
>>> i.next()
45.67
>>> i.next()
Traceback (most recent call last):
File "", line 1, in ?
StopIteration
```

如果这是一个实际应用程序, 那么我们需要把代码放在一个 try-except 块中。 序列现在会自动地产生它们自己的迭代器, 所以一个 for 循环:

```python
for i in seq:
    do_something_to(i)

# 实际上是这样工作的

fetch = iter(seq)
while True:
    try:
        i = fetch.next()
    except StopIteration:
        break
    do_something_to(i)
```

不过你不需要改动你的代码, 因为 for 循环会自动调用迭代器的 next() 方法(以及监视 StopIteration 异常)。

列表解析

列表解析(List comprehensions, 或缩略为 list comps) 来自函数式编程语言 Haskell。它是一个简单有用而且灵活的工具，可以用来动态地创建列表。它在 Python 2.0 中被加入。Python支持的函数式编程特性, 例如 lambda、map() 以及 filter() 等, 通过列表解析它们可以被简化为一个列表解析式子。

让我们看看列表解析的语法:

> [expr for iter_var in iterable]

这个语句的核心是 for 循环, 它迭代 iterable 对象的所有条目. 前边的 expr 应用于序列 的每个成员, 最后的结果值是该表达式产生的列表. 迭代变量并不需要是表达式的一部分。

它有一个计算序列成员的平方的 lambda 函数表达式:

```python
>>> map(lambda x: x ** 2, range(6))
[0, 1, 4, 9, 16, 25]
```

我们可以使用下面这样的列表解析来替换它:

```python
>>> [x ** 2 for x in range(6)]
[0, 1, 4, 9, 16, 25]
```

列表解析的表达式可以取代内建的 map() 函数以及 lambda , 而且效率更高. 结合 if 语句，列表解析还提供了一个扩展版本的语法:

> [expr for iter*var in iterable if cond*expr]

这个语法在迭代时会过滤/捕获满足条件表达式 cond_expr 的序列成员. 回想下 odd() 函数, 它用于判断一个数值对象是奇数还是偶数(奇数返回 1 , 偶数返回 0):

```python
def odd(n):
    return n % 2
```

我们可以借用这个函数的核心操作, 使用 filter() 和 lambda 挑选出序列中的奇数:

```python
>>> seq = [11, 10, 9, 9, 10, 10, 9, 8, 23, 9, 7, 18, 12, 11, 12]
>>> filter(lambda x: x % 2, seq)
[11, 9, 9, 9, 23, 9, 7, 11]
```

和先前的例子一样, 即使不用 filter() 和 lambda，我们同样可以使用列表解析来完成操作， 获得想要的数字:

```python
>>> [x for x in seq if x % 2]
[11, 9, 9, 9, 23, 9, 7, 11]
```

在 Python 2.0 中我们加入了列表解析, 使语言有了一次革命化的发展, 提供给用户了一个强大的工具, 只用一行代码就可以创建包含特定内容的列表。你可以去问一个有多年 Python 经验的程序员是什么改变了他们编写 Python 程序的方式, 那么列表解析一定会是最多的答案。

生成器

迭代器仅仅只有一个方法，用于调用获得下个元素的 next()。除非你实现了一个迭代器的类，迭代器并没有那么“聪明”！ 被调用函数还没有强大到，在迭代中以某种方式生成下一个值并且返回和next()调用一样简单的东西。这个是生成器产生的动机之一。

生成器的另外更强的方面是协同程序的概念。协同程序是可以运行的独立函数调用，可以暂停或者挂起，并从程序离开的地方继续或者重新开始。 在有调用者和(被调用的)协同程序也有通信。

举例来说，当协同程序暂停的时候，我们能从其中获得一个中间的返回值，当调用回到程序中时，能够传入额外或者改变了的参数，但仍能够从我们上次离开的地方继续，并且所有状态完整。 挂起返回出中间值并多次继续的协同程序被称为生成器，那就是 python 的生成器真正在做的事。

什么是 python 式的生成器?从句法上讲，生成器是一个带yield语句的函数。一个函数或者子程序只返回一次，但一个生成器能暂停执行并返回一个中间的结果----那就是 yield 语句的功能，返回一个值给调用者并暂停执行。当生成器的next()方法被调用的时候，它会准确地从离开地方继续。

一个简单的函数说明生成器的行为:

```python
def gen_123():  # 1
    yield 1  # 2
    yield 2
    yield 3

>>> gen_123 # doctest: +ELLIPSIS
<function gen_123 at 0x...>  # 3

>>> gen_123()
<generator object gen_123 at 0x...>  # 4

for i in gen_123():  # 5
    print(i)
# output 1 2 3

>>> g = gen_123()  # 6
>>> next(g) # 7
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)  # 8
Traceback (most recent call last):
StopIteration
```

1 只要 Python 函数中包含关键字 yield，该函数就是生成器函数。 2生成器函数的定义体中通常都有循环，不过这不是必要条件;这里我重复使用 3 次 yield。 3 仔细看，gen*123 是函数对象。 4 但是调用时，gen*123() 返回一个生成器对象。 5 生成器是迭代器，会生成传给 yield 关键字的表达式的值。 6 为了仔细检查，我们把生成器对象赋值给 g。 7 因为 g 是迭代器，所以调用 next(g) 会获取 yield 生成的下一个元素。 8 生成器函数的定义体执行完毕后，生成器对象会抛出 StopIteration 异常。

生成器函数会创建一个生成器对象，包装生成器函数的定义体。把生成器传给 next(...) 函数时，生成器函数会向前，执行函数定义体中的下一个 yield 语句，返回产出的值，并 在函数定义体的当前位置暂停。最终，函数的定义体返回时，外层的生成器对象会抛出 StopIteration 异常——这一点与迭代器协议一致。

yield from

生成器函数需要产出另一个生成器生成的值，传统的解决方法是使用嵌套的 for 循环。 但在python3.3之后引入了 yield from语句，下面的程序对使用做一个对比。

```python
def cities():
    for city in ["Berlin", "Hamburg", "Munich", "Freiburg"]:
        yield city

def squares():
    for number in range(10):
        yield number ** 2

def generator_all_in_one():
    for city in cities():
        yield city
    for number in squares():
        yield number

def generator_splitted():
    yield from cities()
    yield from squares()

lst1 = [el for el in generator_all_in_one()]
lst2 = [el for el in generator_splitted()]
print(lst1 == lst2)
```

前面的代码返回True，因为生成器generator*all*in*one*和*generator*splitted产生相同的元素。 这意味着如果yield from中的是另一个生成器，效果与子生成器的主体在yield from语句处内联相同。 此外，允许子生成器执行带有值的返回语句，并且该值成为来自表达式的的值。

生成器表达式

列表解析的一个不足就是必要生成所有的数据, 用以创建整个列表。 这可能对有大量数据的迭代器有负面效应，生成器表达式通过结合列表解析和生成器解决了这个问题。

生成器表达式在 Python 2.4 被引入, 它与列表解析非常相似，而且它们的基本语法基本相同。 不过它并不真正创建数字列表, 而是返回一个生成器，这个生成器在每次计算出一个条目后，把这 个条目“产生”(yield)出来。 生成器表达式使用了"延迟计算"(lazy evaluation), 所以它在使用 内存上更有效. 我们来看看它和列表解析到底有多相似:

列表解析:

> [expr for iter*var in iterable if cond*expr]

生成器表达式:

> (expr for iter*var in iterable if cond*expr)

生成器并不会让列表解析废弃, 它只是一个内存使用更友好的结构, 基于此, 有很多使用生 成器地方. 下面我们提供了一些使用生成器表达式的例子, 最后例举一个冗长的样例, 从它你可 以感觉到 Python 代

码在这些年来的变化。

生成器表达式就好像是懒惰的列表解析(这反而成了它主要的优势). 它还可以用来处理其他列 表或生成器, 例如这里的 rows 和 cols :

```python
rows = [1, 2, 3, 17]
def cols(): # example of simple generator
    yield 56
    yield 2
    yield 1
```

不需要创建新的列表, 直接就可以创建配对。 我们可以使用下面的生成器表达式:

```python
x_product_pairs = ((i, j) for i in rows for j in cols())
```

现在我们可以循环 x*product*pairs , 它会懒惰地循环 rows 和 cols

```python
>>> for pair in x_product_pairs: ...
    print(pair)

(1, 56)
(1, 2)
(1, 1)
(2, 56)
(2, 2)
(2, 1)
(3, 56)
(3, 2)
(3, 1)
(17, 56)
(17, 2)
(17, 1)
```

生成器表达式 在 Python 2.4 中被加入, 你可以在 PEP 289 中找到更多相关内容。

Python函数式编程模块

itertools

这个模块实现了许多基于APL，Haskell和SML构造的迭代器构建模块。 每一个都已经以适合Python的形式进行了重构。

该模块标准化了一套快速，高效的内存工具，这些工具本身或其组合是有用的。 它们一起构成了一

个“迭代器代数”，可以在纯Python中简洁高效地构建专门的工具。

例如，SML提供了一个列表工具：tabulate(f)，它产生一个序列f(0)，f(1)，...。通过结合map()和count() map(f，count())。

这些工具及其内置的对应物也可以很好地适用于操作员模块中的高速功能。 例如可以将乘法运算符映射到两个向量上以形成有效的点乘积：sum(map(operator.mul, vector1, vector2))。

无限循环迭代 (Infinite iterators)

- count(start=0, step=1)

count迭代器将返回均匀间隔的值，从传入的数字开始，作为其起始参数。 Count还接受一个步骤参数。 我们来看一个简单的例子：

```python
from itertools import count

for i in count(10):
    if i > 15:
        break
    else:
        print(i)
...
10
11
12
13
14
15
```

这里我们从itertools中导入count，并且创建一个for循环。 我们添加一个条件检查，如果迭代器超过15，将跳出循环，否则打印出我们在迭代器中的位置。 你会注意到，输出开始于10，因为这是我们通过计数作为我们的起始价值。

- cycle(iterator)

itertools的cycle迭代器方法允许你创建一个循环遍历一系列值的迭代器。 让我们传递一个3字母的字符串，看看会发生什么：

```Python
from itertools import cycle
count = 0
for item in cycle('XYZ'):
    if count > 7:
        break
    print(item)
    count += 1
...
X
Y
Z
X
Y
Z
X
Y
```

这里我们创建一个for循环来循环三个字母的无限循环：XYZ。 当然，我们不希望真正的循环，所以我们添加一个简单的计数器来打破循环。

在下面的代码中，我们创建一个简单的多边形列表并传递它们以循环。 我们将新的迭代器保存到一个变量中，然后将该变量传递给下一个函数。 每次我们下一次调用，它都会返回迭代器中的下一个值。 由于这个迭代器是无限的，所以我们可以永远调用，永远不会耗尽序列。

```Python
>>> polys = ['triangle', 'square', 'pentagon', 'rectangle']
>>> iterator = cycle(polys)
>>> next(iterator)
'triangle'
>>> next(iterator)
'square'
>>> next(iterator)
'pentagon'
>>> next(iterator)
'rectangle'
>>> next(iterator)
'triangle'
>>> next(iterator)
'square'
```

- repeat(object[, times]) 重复迭代器将永远返回同一个对象一次又一次，除非你设置它的时间参数。它与循环非常相似，只是它不重复循环一组值。 我们来看一个简单的例子：

```Python
>>> from itertools import repeat
>>> repeat(5, 3)
repeat(5, 3)
>>> iterator = repeat(5, 3)
>>> next(iterator)
5
>>> next(iterator)
5
>>> next(iterator)
5
>>> next(iterator)
Traceback (most recent call last)
<ipython-input-154-3733e97f93d6> in <module>()
----> 1 next(iterator)
StopIteration:
```

在这里我们导入重复，并告诉它重复三次。 然后我们再次调用我们的新的迭代器四次，看看它是否正常工作。 当你运行这个代码的时候，你会看到StopIteration被提出，因为我们的迭代器中的值已经用完了。

按输入终止迭代 (Iterators that terminate)

迭代器	参数	结果	示例
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, …	accumulate([1,2,3,4,5]) --> 1 3 6 10 15
chain()	p, q, …	p0, p1, … plast, q0, q1, …	chain('ABC', 'DEF') --> A B C D E F
chain.from_iterable()	iterable	p0, p1, … plast, q0, q1, …	chain.from_iterable(['ABC', 'DEF']) --> A B CD E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
dropwhile()	pred, seq	seq[n], seq[n+1], starting when pred fails	dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 41
filterfalse()	pred, seq	elements of seq where pred(elem) is false	filterfalse(lambda x: x%2, range(10)) --> 0 24 6 8
groupby()	iterable[, key]	sub-iterators grouped by value of key(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	islice('ABCDEFG', 2, None) --> C D E F G
starmap()	func, seq	func(*seq[0]), func(*seq[1]), …	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 91000
takewhile()	pred, seq	seq[0], seq[1], until pred fails	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
tee()	it, n	it1, it2, … itn splits one iterator into n	
zip_longest()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	zip_longest('ABCD', 'xy', fillvalue='-') -->Ax By C- D-

接下来我们拿几个常见的作为使用示例讲解下。

- accumulate 累积迭代器将返回累积的总和或两个参数函数的累积结果，您可以将其累加。累加的默认值是加法，所以让我们快速尝试一下：

```Python
>> from itertools import accumulate
>>> list(accumulate(range(10)))
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

在这里，我们导入累积并将其传递给0-9的10个数字。 依次添加它们，所以第一个是0，第二个是0+1，第三个是1+2等。 现在我们导入运算符模块并将其添加到混合中：

```Python
>>> import operator
>>> list(accumulate(range(1, 5), operator.mul))
[1, 2, 6, 24]
```

在这里，我们将数字1-4传递给我们的累积迭代器。 我们也传递一个函数：operator.mul，这个函数接受要相乘的参数。 所以对于每次迭代，它使用相乘而不是相加（1×1=1，1×2=2，2×3=6等）。

- groupby(iterable, key=None) groupby迭代器将从您的iterable中返回连续的键和组。 如果没有看到一个具体例子，确实很难让你明白它的意义。 所以让我们来看一个示例！把下面的代码放到你的解释器中，或者把它保存在一个文件中：

```Python
from itertools import groupby

vehicles = [('Ford', 'Taurus'), ('Dodge', 'Durango'),
            ('Chevrolet', 'Cobalt'), ('Ford', 'F150'),
            ('Dodge', 'Charger'), ('Ford', 'GT')]

sorted_vehicles = sorted(vehicles)

for key, group in groupby(sorted_vehicles, lambda make: make[0]):
    print(key, group)
    for make, model in group:
        print('{model} is made by {make}'.format(model=model,
                                                 make=make))
    print ("**** END OF GROUP ***\n")
```

这里我们导入groupby，然后创建一个元组列表。 然后我们对数据进行排序，这样在输出数据时就更有意义了，而且它也可以让groupby实际上正确地分组项目。 接下来我们实际遍历由groupby返回的迭代器，它给了我们键和组。 然后我们遍历该组并打印出其中的内容。 如果你运行这个代码，你应该看到这样的东西：

```
Cobalt is made by Chevrolet
**** END OF GROUP ***

Charger is made by Dodge
Durango is made by Dodge
**** END OF GROUP ***

F150 is made by Ford
GT is made by Ford
Taurus is made by Ford
**** END OF GROUP ***
```

尝试改变代码，直接groupby(vehicles)。你会很快知道为什么你应该在通过groupby运行之前对数据进行排序。

- tee(iterable, n=2) tee工具将从单个迭代中创建**n**个迭代器。 这意味着你可以从一个迭代中创建多个迭代器。 让我们看看它的工作原理的一些解释性代码：

Python

```
>>> from itertools import tee
>>> data = 'ABCDE'
>>> iter1, iter2 = tee(data)
>>> for item in iter1:
...     print(item)
...
A
B
C
D
E
>>> for item in iter2:
...     print(item)
...
A
B
C
D
E
```

在这里，我们创建一个5个字母的字符串，并将其传递给三通。 由于tee缺省为2，我们使用多个赋值来获取从tee返回的两个迭代器。 最后我们遍历每个迭代器并打印出它们的内容。 正如你所看到的，他们的内容是一样的。

组合生成器 (Combinatoric generators)

itertools库包含四个组合迭代器，可用于创建数据的组合和排列。

- combinations(iterable, r)

如果你有创建组合的需要，Python已经覆盖了itertools.combinations。 什么组合可以让你做的是从一个长度很长的迭代器中创建一个迭代器。 让我们来看看：

```Python
>>>from itertools import combinations
>>>list(combinations('WXYZ', 2))
[('W', 'X'), ('W', 'Y'), ('W', 'Z'), ('X', 'Y'), ('X', 'Z'), ('Y', 'Z')]
```

请注意，组合函数按字典顺序进行组合，所以如果您对迭代进行了排序，那么您的组合元组也将被排序。 另外值得注意的是，如果所有输入元素都是唯一的，组合将不会在组合中产生重复值。

- combinations*with*replacement(iterable, r)

带有迭代器的 **combinations*with*replacement** 与**combinations**非常相似。 唯一的区别是，它实际上会创建元素重复的组合。 我们来试一下上一节的例子来说明：

```Python
>>> from itertools import combinations_with_replacement
>>> for item in combinations_with_replacement('WXYZ', 2):
...     print(''.join(item))

...
WW
WX
WY
WZ
XX
XY
XZ
YY
YZ
ZZ
```

- product(*iterables, repeat=1)

itertools包有一个简洁的函数，用于从一系列输入迭代中创建笛卡尔积，该功能是**product**。 让我们看看它是如何工作的！

```Python
>>> from itertools import product
>>> arrays = [(-1,1), (-3,3), (-5,5)]
>>> cp = list(product(*arrays))
>>> cp

[(-1, -3, -5),
 (-1, -3, 5),
 (-1, 3, -5),
 (-1, 3, 5),
 (1, -3, -5),
 (1, -3, 5),
 (1, 3, -5),
 (1, 3, 5)]
```

在这里，我们导入产品，然后设置我们分配给变量**array**的元组列表。 接下来我们将这些数组称为产品。 你会注意到我们使用 *array 来调用它。 这将导致列表“分解”或依次传递参数于product函数，这意味着你传递了三个参数而不是一个。 如果你愿意，可以尝试去掉星号前缀来调用它，看看会发生什么。

- permutations itertools的permutations子模块将从您赋予的迭代中返回连续的r元素长度排列。 就像组合函数一样，排列按字典顺序排列。 让我们来看看：

```Python
>>> from itertools import permutations
>>> for item in permutations('WXYZ', 2):
...     print(''.join(item))

WX
WY
WZ
XW
XY
XZ
YW
YX
YZ
ZW
ZX
ZY
```

你会注意到它的输出比combinations的输出长得多。 当你使用permutations的时候，它会遍历字符串的所有排列，但是如果输入元素是唯一的，它将不会重复值。

functools

functools模块用于高阶函数：作用于或返回其他函数的函数。 一般而言，任何可调用的对象都可以被视为一个函数，用于本模块的需要。

高阶函数将一个或多个函数作为输入并返回一个新函数。 这个模块中最有用的工具是functools.partial()函数。

对于用函数样式编写的程序，有时候需要构造一些现有函数的变体，这些函数有一些参数被填充。考虑一个Python函数f(a, b, c); 你可能希望创建一个等价于 f(1，b，c) 的新函数 g(b，c); 你在为f()的某个参数填充一个值。 这样的函数被称为“局部函数程序”。

functools的函数分为三类，装饰类型(Decorators)、对比类型、调度类型。

装饰类型

- partial

```python
import functools

def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print('%s: %s' % (subsystem, message))
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

列表很棒，但是有时候用一个更清晰的方式与函数交互是很好的。 有一个无限的方法来做到这一点，但我喜欢类的点操作。 这里是一个“partial structure”类，它遵循我认为非常方便的模式：

```python
from functools import partial

def power(base, exponent):
    return base ** exponent

class PowerMeta(type):
    def __init__(cls, name, bases, dct):

        # generate 50 partial power functions:
        for x in range(1, 51):

            # Set the partials to the class
            setattr(
                # cls represents the class
                cls,

                # name the partial
                "p{}".format(x),

                # partials created here
                partial(power, exponent=x)
            )
        super(PowerMeta, cls).__init__(name, bases, dct)

class PowerStructure(object, metaclass=PowerMeta):
    pass

# 自动构建幂函数
>>> p = PowerStructure()
>>> p.p2(10)
100
>>> p.p5(2)
32
>>> p.p50(2)
1125899906842624

# 你也可以这样使用
>>> PowerStructure.p2(10)
100
```

- update_wrapper

partial对象默认没有**name**或**doc**属性，没有这些属性，被修饰的函数更难以调试。 使用update_wrapper()，将原始函数的属性复制或添加到局部函数对象。

```python
def another_function(func):
    """
    A function that accepts another function
    """

    def wrapper():
        """
        A wrapping function
        """
        val = "The result of %s is %s" % (func(),
                                          eval(func())
                                          )

        return val
    return wrapper

@another_function
def a_function():
    """A pretty useless function"""
    return "1+1"

>>> a_function.__name__
'wrapper'
>>> a_function.__doc__
'\n        A wrapping function\n        '


>>> functools.update_wrapper(a_function, another_function)
>>> a_function.__name__
'another_function'
```

上面的示例可以看到，函数 a_function 名称和文档被掩盖，通过 updaate_wrapper() 函数，可以定位到被装饰的原函数名与原函数文档。

- wraps

wraps 是一个更方便的函数，用于在定义包装函数时作为函数装饰器调用update_wrapper()。 它相当于partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated))。 例如：

```Python

>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

- lru_cache

函数缓存允许我们根据参数来缓存函数的返回值。 当使用相同的参数定期调用I/O绑定函数时，可以节省时间。 在Python 3.2之前，我们必须编写一个自定义的实现。 在Python3.2+中，有一个lru_cache装饰器，它允许我们快速缓存和取消缓存函数的返回值。

```Python
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> print([fib(n) for n in range(16)])
# Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

对比类型

- cmp_to_key

```python
def reverse_numeric(x, y):
    return y - x

# 在python2中 排序使用 cmp
sorted([5, 2, 4, 1, 3], cmp=reverse_numeric)

# 在python3中 排序使用key指定排序函数
sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric))
```

python2 中sorted函数接收 cmp参数，但在python3中只接收 key参数。

将老式的比较函数（comparison function）转化为关键字函数（key function），与接受key function的工具一同使用（如 sorted(), min(), max(), heapq.nlargest(), itertools.groupby())。 该函数主要用来将程序转成 Python 3 格式的，因为 Python 3 中不支持比较函数。

比较函数是可调用的，接受两个参数，比较这两个参数并根据他们的大小关系返回负值、零或正值中的某一个。 关键字函数也是可调用的，但接受一个参数，同时返回一个可以用作排序关键字的值。

- total_ordering

这是一个类装饰器，给定一个类，这个类定义了一个或多个比较排序方法，这个类装饰器将会补充其余的比较方法，减少了自己定义所有比较方法时的工作量。

被修饰的类必须至少定义 **lt**(), **le**(), **gt**() 或 **ge**() 中的一个，同时，被修饰的类还应该提供 **eq**() 方法。

```python
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

调度类型

- reduce 将两个参数的函数累加到序列的项中，从左到右，以便将序列减少为单个值。 比如，reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). 左参数，x是累加值，右参数，y是来自序列的更新值。如果存在可选的初始化器，则它将放置在计算中序列的项目之前，并在序列为空时用作默认值。如果未提供初始化程序且序列只包含一个项目，则返回第一个项目。

```python
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num
```

等同于

```python
from functools import reduce
product = reduce(lambda x, y: x * y, [1, 2, 3, 4])
```

- singledispatch

singledispatch 目的是实现C++语言中的泛型函数。是Python3.4中的新功能。

```python
from functools import singledispatch

@singledispatch
def add(a, b):
    raise NotImplementedError('Unsupported type')

@add.register(int)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)

@add.register(str)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)

@add.register(float)
@add.register(Decimal)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)

@add.register(list)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)

if __name__ == '__main__':
    add(1, 2)
    add('Python', 'Programming')
    add([1, 2, 3], [5, 6, 7])
    add(None, None)
```

在这里，我们从functools导入singledispatch，并将其应用于一个简单的函数，我们称之为add。 这个函数是我们的catch-all函数，只有在其他装饰函数没有处理传递的类型时才会被调用。 你会注意到我们当前处理整数，字符串和列表作为第一个参数。 如果我们用别的东西（比如一个字典或None）调用我们的add函数，那么会引发一个NotImplementedError异常。

Toolz

一个利用 Toolz 统计文本字数的代码。

```python
from toolz.curried import map
from toolz import frequencies, compose, concat, merge_with

def stem(word):
    """ Stem word to primitive form

    >>> stem("Hello!")
    'hello'
    """
    return word.lower().rstrip(",.!)-*_?:;$'-\"").lstrip("-*'\"(_$'")


wordcount = compose(frequencies, map(stem), concat, map(str.split), open)

if __name__ == '__main__':
    # Filenames for thousands of books from which we'd like to count words
    filenames = ['Book_%d.txt'%i for i in range(10000)]

    # Start with sequential map for development
    # pmap = map

    # Advance to Multiprocessing map for heavy computation on single machine
    # from multiprocessing import Pool
    # p = Pool(8)
    # pmap = p.map

    # Finish with distributed parallel map for big data
    from ipyparallel import Client
    p = Client()[:]
    pmap = p.map_sync

    total = merge_with(sum, pmap(wordcount, filenames))
```

设计实践

一个面向对象的kmeans算法示例

```python
from sklearn.datasets.samples_generator import make_blobs
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from matplotlib import animation
import random
from functools import reduce, partial
from operator import add
```

```

class KMeans:
  """good old class based solution"""
  def __init__(self, k):
    self.k = k
    self.means = [None for _ in range(k)]

  def fit(self, points, num_iters=10):
    assignments = [None for _ in points]
    self.means = random.sample(list(points), self.k)
    for _ in range(num_iters):
      for i, point in enumerate(points):
        assignments[i] = self.predict(point)
      for j in range(self.k):
        cluster = [p for p, c in zip(points, assignments) if c == j]
        self.means[j] = list(map(lambda x: x / len(cluster), reduce(partial(map,

  def predict(self, point):
    d_min = float('inf')
    for j, m in enumerate(self.means):
      d = sum((m_i - p_i)**2 for m_i, p_i in zip(m, point))
      if d < d_min:
        prediction = j
        d_min = d
    return prediction

def run_kmeans(seed=1):
  random.seed(seed)
  points = np.random.random((100,2))

  model = KMeans(5)
  model.fit(points, num_iters=100)
  assignments = [model.predict(point) for point in points]

  for x, y in model.means:
    plt.plot(x, y, marker='*', markersize=10, color='black')

  for j, color in zip(range(5),
                      ['r', 'g', 'b', 'm', 'c']):
    cluster = [p
               for p, c in zip(points, assignments)
               if j == c]
    xs, ys = zip(*cluster)
    plt.scatter(xs, ys, color=color)

  plt.show()

# 执行示例
```

```
run_kmeans()
```

改为函数式编程示例：

```python
def k_meanses(points, k):
  initial_means = random.sample(points, k)
  return iterate(partial(new_means, points),
                 initial_means)

def no_repeat(prev, curr):
  if prev == curr: raise StopIteration
  else: return curr

def until_convergence(it):
  return accumulate(it, no_repeat)

def new_means(points, old_means):
  k = len(old_means)
  assignments = [closest_index(point, old_means)
                 for point in points]
  clusters = [[point
               for point, c in zip(points, assignments)
               if c == j] for j in range(k)]
  return [cluster_mean(cluster) for cluster in clusters]

def closest_index(point, means):
  return min(enumerate(means),
             key=lambda pair: squared_distance(point, pair[1]))[0]

def squared_distance(p, q):
  return sum((p_i - q_i)**2 for p_i, q_i in zip(p, q))

def cluster_mean(points):
  num_points = len(points)
  dim = len(points[0]) if points else 0
  sum_points = [sum(point[j] for point in points)
                for j in range(dim)]
  return [s / num_points for s in sum_points]


def run_kmeans_functional(seed=0):
  random.seed(seed)
  data = [(random.random(), random.random()) for _ in range(500)]
  meanses = [mean for mean in until_convergence(k_meanses(data, 5))]

  x, y = zip(*data)
  plt.scatter(x, y, color='black')
```

```
44
45    colors = ['r', 'g', 'b', 'c', 'm']
46    for i, means in enumerate(meanses):
47      for m, color in zip(means, colors):
48        plt.plot(*m, color=color,
49                 marker='*',
50                 markersize=3*i)
51
52    plt.show()
```

其他机器学习算法示例，待之后更新。