在命令式编程中，你需要给计算机一系列任务，然后计算机会一一执行。在执行过程中，计算机可以改变其状态。举个例子，假设你将 A 的初始值设为 5，接下来你还可以改变 A 的值。在变量内部值变化的层面来讲，你可以掌控这些变量。

在函数式编程中，你无需告诉计算机去做什么，而是为它提供一些必要的信息。如什么是一个数字的最大公约数，1 到 n 的乘积是多少等等。

由于这样，变量就无法改变了。一旦你设置了一个变量，它就会永远保持初始状态（注意：在纯函数式语言中，它们不叫作变量）。因此在函数式编程中，函数不会产生“副作用”。“副作用”是指函数可能会修改外部变量的值。让我们通过一个典型的 Python 例子来看一下：

```
a = 3

def some_func():

    global a

    a = 5



some_func()

print(a)
```

这段代码的输出结果是5。在函数式编程中，改变变量是大忌，而且让函数改变外部变量也是绝对禁止的。函数唯一能做的事是执行计算然后返回结果。

现在你可能在想：没有变量，就没有副作用吗？为什么这么做很管用？好问题，下面我们简单讲一下这个问题。

如果一个函数伴随着相同参数被调用两次，它一定会返回一样的结果。如果你对数学上的函数有所了解，你就会理解这里的意义，这被称作引用透明性。因为函数没有副作用，如果你创建了一个可以执行计算的程序，你就可以使该程序提升性能。如果程序知道 func(2) 等于 3，我们可以把这一信息存入表中。这么做可

以防止在我们已经知道答案的情况下，程序依然反复运行同一函数。

一般来说，在函数式编程中，我们不使用循环。而是用递归。递归是一个数学概念，我们通常将其理解为“自己喂自己”。在一个递归函数中，函数将自己作为子函数反复调用。这里有一个易于理解的递归函数的 Python 例子：

```
def factorial_recursive(n):

    # Base case: 1! = 1

    if n == 1:

        return 1



    # Recursive case: n! = n * (n-1)!

    else:

        return n * factorial_recursive(n-1)
```

还有一些编程语言也是很“懒”的，也就是说它们直到最后一刻才会进行计算。如果你写一段想要计算 2+2 的代码，函数式程序只会在你要使用其结果时才会执行计算命令。我们接下来继续探索 Python 都“懒”在哪些方面。

Map

若要理解 map，我们要先看看 iterable 是什么。iterable 指一类可以进行迭代的对象。通常来看，它们是列表或数组，但 Python 有许多不同类型的 iterable。你甚至可以创建自己的 iterable 对象，来执行各种魔术方法 (magic method)。魔术方法可以是一个 API，来使你的对象更加 Pythonic。你需要用两个魔术方法来使对象成为 iterable：

```
class Counter:

    def __init__(self, low, high):
```

```
4
5           # set class attributes inside the magic method
    __init__
6
7           # for "inistalise"
8
9           self.current = low
10
11          self.high = high
12
13
14
15      def __iter__(self):
16
17          # first magic method to make this object
    iterable
18
19          return self
20
21
22
23      def __next__(self):
24
25          # second magic method
26
27          if self.current > self.high:
28
29              raise StopIteration
30
31          else:
32
33              self.current += 1
34
35              return self.current - 1
```

第一个魔术方法 **iter**，或者说 "dunder"（指以双下划线 “__” 作为名字开头和结尾的方法），返回了迭代对象，这常常也被当做循环的开端。dunder 方法接下来会返回下一个对象。

让我们快速查看一下终端会话的结果：

```
for c in Counter(3, 8):

    print(c)
```

输出结果为：

```
3
4
5
6
7
8
```

在 Python 中，迭代器指只包含一个魔术方法 **iter** 的对象。这意味着你可以访问该对象的任何部分，但不能对其循环访问。有些对象包含魔术方法 **next**，以及除了 **iter** 以外的魔术方法，如 sets（下文会进行详细讨论）。在本文中，假定我们涉及的所有东西都是可迭代的对象。

那么现在我们知道了什么是可迭代对象，再回头看一下 map 函数。map 函数可以让我们在同一个 iterable 对象中，把函数作用在每一个元素上。我们通常将函数作用于列表中的每个元素，但这对大多数 iterable 对象也是可行的。Map 需要两个输入，分别是要执行的函数和 iterable 对象。

```
map(function, iterable)
```

假设我们有一个如下的数字列表：

```
[1, 2, 3, 4, 5]
```

然后计算每个数字的平方，我们可以写下面一段代码：

```
x = [1, 2, 3, 4, 5]

def square(num):

    return num*num



print(list(map(square, x)))
```

Python 中的函数式函数也有“懒”的特性。如果我们不引入 "list()"，那函数就会存取 iterable 对象，而不是存取列表本身。我们需要明确告诉 Python 程序“将其转换成列表”，从而供我们使用。

听起来可能有点奇怪，我们对 Python 的评价从 “一点也不懒” 突然转变到 “懒”。如果你对函数式编程的感悟胜过指令式编程，最终你会习惯这种转变的。

现在我们可以很容易写出一个像 "square(num)" 这样的函数了，但看起来不太合适。我们有必要定义一个函数仅仅为了在 map 中调用它一次吗？好吧，我们可以基于 lambda 在 map 中定义一个函数。

Lambda表达式

lambda 表达式是一个单行的函数。以一个计算数字平方的 lambda 表达式为例：

```
square = lambda x: x * x
```

现在执行这行代码：

```
>>> square(3)
9
```

我已经听见你在问了，参数在哪？这到底是怎么回事？它看起来并不像个函数？

这可能有点让人困扰，但可以解释得清楚。首先我们给变量 "square"赋一个值，例如：

```
lambda x:
```

我们告诉 Python 这是一个 lambda 函数，且输入值为 x。冒号后面的部分代表对输入要做的事情，然后它就会返回得到的结果。

我们可以将该计算平方值的程序简化成一行：

```
x = [1, 2, 3, 4, 5]

print(list(map(lambda num: num * num, x)))
```

由此可见，在一个 lambda 表达式中，所有参数都在左边，你想要对其执行的指令都在右边。不得不承认这样看起来有点杂乱。实际上这是一种很受欢迎的编程方式，只有其他的函数式程序员可以读懂代码。同时，把一个函数转化成单行表达式真的很酷。

Reduce

Reduce 是将 iterable 转换成一个结果的函数。通常用作来对一个列表进行计算，将其缩减为一个数字。如下：

```
reduce(function, list)
```

我们可以将 lambda 表达式用作函数，事实上我们通常也是这么做的。

一个列表的乘积为每个单独的数字相乘在一起的结果。你可以通过如下程序实现：

```
product = 1

x = [1, 2, 3, 4]

for num in x:

    product = product * num

```

但基于 reduce 你可以将上面程序写作：

```
from functools import reduce


product = reduce((lambda x, y: x * y),[1, 2, 3, 4])

```

得到的乘积结果是一样的。基于对函数式编程的理解，代码量减小了，代码也变得更简洁。

Filter

filter 函数用于传入一个 iterable，并过滤掉这个 iterable 中所有你不想要的序列。

通常，filter 函数传入一个函数和一个列表。将该函数作用在列表中的任意一个元素上，如果该函数返回 True，不做任何事情。如果返回 False，将该元素从列表中删除。

语法如下：

```
filter(function, list)••
```

案例：(不使用 filter)

```
x = range(-5, 5)

new_list = []



for num in x:

    if num < 0:

        new_list.append(num)
```

(使用 filter)

```
x = range(-5, 5)

all_less_than_zero = list(filter(lambda num: num < 0, x))
```

高阶函数

高阶函数可以将函数作为参数传入并返回，如下：

```
def summation(nums):

    return sum(nums)



def action(func, numbers):
```

```
8
9       return func(numbers)
10
11
12
13  print(action(summation, [1, 2, 3]))
14
15
16
17  # Output is 6
```

再比如：

```
1   def rtnBrandon():
2
3       return "brandon"
4
5   def rtnJohn():
6
7       return "john"
8
9
10
11  def rtnPerson():
12
13      age = int(input("What's your age?"))
14
15
16
17      if age == 21:
18
19          return rtnBrandon()
20
21      else:
22
23          return rtnJohn()••
```

你之前知道我提到的纯函数式编程语言没有变量是怎么说的吗？高阶序列函数能让这件事变得更为简单。你不需要储存一个变量，如果你就是为了将数据通过函数的管道进行传递。

Python 中的所有函数都是一等对象。一等对象具有以下一种或多种特征：

- 运行时创建
- 将变量或元素赋值在一个数据结构中
- 作为一个参数传递给一个函数
- 作为函数结果返回

因此，Python 中的所有函数都是第一类且可以作为高阶函数使用。

偏函数应用

偏函数应用（又叫闭包）有点难理解，但超级酷。你可以调用一个函数而无需提供它所需要的全部参数。先来看看这个例子：我们想创建一个函数，传入的两个参数分别是 base 和 exponet，然后返回 base 的 exponent 次方。如下：

```
1 def power(base, exponent):
2
3   return base ** exponent
```

现在，我们想要一个专用的平方函数，使用 power 函数得到一个数字的平方。如下：

```
1 def square(base):
2
3   return power(base, 2)
```

这个方式可以，但如果我们想要一个三次方函数呢？或者是四次方？我们能一直这样写吗？当然，你是可以的。但程序员可没那么勤快。如果你一遍又一遍地重复做一件事，那么你就需要用一种更高效率的方式做事，无需重复。因此，我们采用了闭包的方法。以下是一个采用闭包的平方函数的例子：

```
from functools import partial



square = partial(power, exponent=2)

print(square(2))



# output is 4
```

这样是不是很酷！通过告诉 Python 第二个参数是什么，我们只用一个参数就能调用需要两个参数的函数。

我们还能用一个 loop，产生一个乘方函数以实现从三次方到 1000 次方的计算:

```
from functools import partial



powers = []

for x in range(2, 1001):

  powers.append(partial(power, exponent = x))



print(powers[0](3))

# output is 9
```

▌有时，函数式编程无法与Python相匹配

你可能已经注意到了，我们想要在函数式编程中完成的事情都会列表相关。除了 reduce 函数和偏函数应用外，所有你看到的函数都会产生列表。Python 之父 Guido 不喜欢 Python 当中的函数式编程部分，因为 Python 已经产生自己列表的方式。

如果你在 Python 的命令执行环境中输入「import this」，你会得到如下提示：

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one — and preferably only one — obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea — let's do more of those!
```

这就是 Python 的精妙所在。在 Python 环境中，map&filter 可以实现列表解析式同样的事情。这个打破了 Python 的一条规则，于是这部分的函数式编程看起来不那么 Pythonic了。

另一个需要讨论的是 lambda。在 Python 中，一个 lambda 函数是一个常函数。Lambda 实际上是一个语法糖，因此二者是等价的：

```
foo = lambda a: 2



def foo(a):

  return 2
```

理论上，一个常函数可以实现一个 lambda 函数能实现的任何事情，但反过来却不行——一个 lambda 函数无法实现一个常函数所能做的所有事情。这也是为什么大家会有争论函数式编程不能很好地与整个 Python 生态系统匹配。

列表解析式

列表解析式是 Python 产生列表的一种方式，语法如下：

```
[function for item in iterable]
```

然后将列表中的所有数字进行平方：

```
print([x * x for x in [1, 2, 3, 4]])
```

这样，可以看到如何将一个函数作用于列表中的每一个元素。如果利用 filter 呢？请先看下此前出现过的代码：

```
x = range(-5, 5)



all_less_than_zero = list(filter(lambda num: num < 0, x))

print(all_less_than_zero)••
```

然后将其转换成列表解析式：

```
x = range(-5, 5)



all_less_than_zero = [num for num in x if num < 0]••
```

列表解析式支持这样的 if 表达式。你不需要通过上百万个函数最终得到你想要的。

那么如果想要将列表中的所有数字进行平方呢？采用 lambda、map 、 filter 你会这么写：

```
x = range(-5, 5)



all_less_than_zero = list(map(lambda num: num * num,
list(filter(lambda num: num < 0, x))))
```

这样看起来相当冗长且复杂。如果用列表解析式只需要这样：

```
x = range(-5, 5)



all_less_than_zero = [num * num for num in x if num < 0]
```

其他解析式

你可以创建任何一个 iterable 的解析式。

通过解析式可产生任何一个 iterable。从 Python 2.7 开始，你甚至可以创作出一本字典了。

```
# Taken from page 70 chapter 3 of Fluent Python by Luciano Ramalho



DIAL_CODES = [

    (86, 'China'),

    (91, 'India'),

    (1, 'United States'),

    (62, 'Indonesia'),

    (55, 'Brazil'),

    (92, 'Pakistan'),

    (880, 'Bangladesh'),

    (234, 'Nigeria'),

```

```
    (7, 'Russia'),

    (81, 'Japan'),

    ]



>>> country_code = {country: code for code, country in
DIAL_CODES}

>>> country_code

{'Brazil': 55, 'Indonesia': 62, 'Pakistan': 92,
'Russia': 7, 'China': 86, 'United States': 1, 'Japan':
81, 'India': 91, 'Nigeria': 234, 'Bangladesh': 880}

>>> {code: country.upper() for country, code in
country_code.items() if code < 66}

{1: 'UNITED STATES', 7: 'RUSSIA', 62: 'INDONESIA', 55:
'BRAZIL'}
```

如果这是一个 iterable，那么就能实现。我们最后来看一个关于集合（sets）的例子。

- 集合是元素列表，且没有重复出现两次的元素。
- 集合的排序无关紧要。

```
# taken from page 87, chapter 3 of Fluent Python by Luciano Ramalho



>>> from unicodedata import name

>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')}

{'×', '¥', '°', '£', '©', '#', '¬', '%', 'µ', '>', '¤', '±', '¶', '§', '<', '=', '®', '$', '÷', '¢', '+'}
```

你可能会注意到集合具有和字典中一样的花括号。这就在于 Python 的智能性，它会根据你是否提供了额外的值以判断你写的是 dictionary comprehension 还是 set comprehension。

总结

函数式编程是优雅而简洁的。函数式代码可以非常简洁，但也可以非常凌乱。一些 Python 程序员不喜欢用 Python 函数式解析。因此，你应该用你想用的，用最好的工具完成任务。