**NLP\_chapter4**

**第四章 编写结构化程序**

**4.1 回到基础**

**赋值**

要理解通过一个对象引用修改一个对象 与通过覆盖一个对象引用之间的区别。

要从链表 foo 复制项目到一个新的链表 bar，可以写 bar = foo [:]。这会复制链表中的对象引用。复制结构，而不复制任何对象引用使用 copy .deepcopy()

**等式**

is 操作符测试对象的**标识符**。满足“==”的不一定满足is

可以用position = random.choice(range(size))来生成一个随机的数，用函数id()来检测链表中哪个字符的标识符发生了变化

**条件语句**

在 if 语句的条件部分，一个非空字符串或链表被判定为真，而一个空字符串或链表的 被判定为假。

使用 if...elif 而不是在一行中使用两个 if 语句的区别：

elif 子句比单独的 if 子句 潜在的给我们更多信息;当它被判定为真时，告诉我们不仅条件满足而且前面的 if 子句的条件不满足。

all()函数和 any()函数可以应用到一个链表(或其他序列)，来检查是否全部或任一项目满足一些条件:

sent = ['No','good', 'fish', 'goes', 'anywhere', 'without', 'a', 'porpoise', '.']

print(all(len(w) > 4 for w in sent))

print(any(len(w) > 4 for w in sent))

**4.2 序列**

除了字符串和链表还有另一种序列被称为元组。**元组由逗号操作符构造**，而且通常使用括号括起来。在前面的章节中看到过的元组有时也被称为“配对”，因为总是有两名成员。

元组可以有任何数目的成员。与链表和字符串一样，元组可以被索引和切片，并有长度。

元组使用逗号操作符来构造。括号是一个 Python 语法的一般功能，设计用于分组。定义一个包含单个元素'snark'的元组是通过添加一个尾随的逗号，像这样:'snark',。空元组是一个特殊的情况下，使用空括号()定义。

**序列类型上的操作**

Python 表达式 评论

for item in s 遍历 s 中的元素

for item in sorted(s) 按顺序遍历 s 中的元素

for item in set(s) 遍历 s 中的无重复的元素

for item in reversed(s) 按逆序遍历 s 中的元素

for item in set(s).difference(t) 遍历在集合 s 中不在集合 t 的元素

序列类型之间可以相互转换：tupl e(s)将任何种类的序列转换成一个元组，list(s)将任何种类的序列转换成一个链表。使用 join()函数将一个字符串链表转换成单独的字符串。

可以使用元组重新安排链表中的内容(可以省略括号，因为逗号比赋值的优先级更高)：

words = ['I','tumed','off','the','spectrotoute']

words[2],words[3],words[4]=words[3],words[4],words[2]

print(words)

zip()取两个或两个以上的序列中的项目，将它们“压缩”打包成单个的配对链表。给定一个序列 s，enumerate(s)返回一个包含索引和索引处项目的配对:

words = ['I','tumed','off','the','spectrotoute']

tags = ['noun','verb','prep','det','noun']

print(zip(words,tags))

print(list(zip(words,tags)))

print(list(enumerate(words)))

python3中常常只在被需要的展示计算结果，当在想要看到一个序列的时候看到类似<zip object at 0x10ff95dc8>这样的表达式，将它放到一个需要序列的上下文中来强制对这个对象进行计算，比如：list(x), or for item in x.（所以在print函数中不能省略方括号）

也可以利用元组对文本进行分割，指定想要分割数据的位置，然后在这个位置分割序列

**合并不同类型的序列**

综合关于这三种类型的序列的知识，一起使用链表推导处理一个字符串中的词，按它们的长度排序:

一个简单的字符串实际上是一个其上定义了方法的对象，如 split():

words = 'I turned off the spectroroute'.split()

使用链表推导建立一个元组的链表, 其中每个元组由一 个数字(词长)和这个词组成，例如:(3, 'the'):

wordlens = [(len(word),word) for word in words]

使用 sort()方法就地排序链表:

wordlens.sort()

丢弃长度信息，并将这些词连接回一个字符串(下划线只是一个普通的 Python 变量，约定可以用下划线表示我们不会使用其值的变量。):

print(' '.join(w for (\_,w) in wordlens))

序列类型的重要的区别和共性：

首先，字符串出现在开头和结尾；链表和元组在中间，但使用的目的不同：一个链表是一个典型的具有相同类型的对象的序列，它的长度是任意的。我们经常使用链表保存词序列。相反，一个元组通常是不同类型的对象的集合，长度固定。我们经常使用一个元组来保存一个**纪录**:与一些实体相关的不同**字段**的集合。

用一个链表表示词典，因为它是一个单一类型的对象的集合——词汇条目——没有预定的长度。个别条目被表示为一个元组，因为它是一个有不同的解释的对象的集合，发音都是用链表存储的：

lexicon = [

('the','det',['Di:','D@']),

('off','prep',['Qf','O:f'])

]

决定何时使用元组还是链表的一个好办法是看一个项目的内容是否取决于它的位置。位置重要用元组，不重要（平行关系）用链表。

列表是可变的，而元组是不可变的

**产生器表达式**

可以将文本处理的表达式插入到一些其他函数的调用中，Python 允许我们省略方括号（print函数不可以）：

print(max([w.lower() for w in nltk.word\_tokenize(text)]))

print(max(w.lower() for w in nltk.word\_tokenize(text)))

第二个式子使用了**产生器表达式**。在第一个式子中，链表对象的存储空间必须在 max()的值被计 算之前分配。如果文本非常大的，这将会很慢。在第二个式子中，数据流向调用它的函数。由于调用的函数只是简单的要找最大值——按字典顺序排在最后的词（大写字母在小写字母的前面）——它可以处理数据流，而无需存储迄今为止的最大值以外的任何值。

**4.3 风格的问题**

**Python 代码风格**

可以在圆括号、方括号或花括号内换行，因为 Pyth on 能够探测到该行与下一行是连续的

如果需要在圆括号、方括号或大括号中换行，通常可以添加额外的括号，也可以在行尾需要换行的地方添加一个反斜杠

对于每行输出一个计数值，一个循环计数器似乎是必要的。然而，我们可以使用 enumerate()处理序列 s，为 s 中每个项目产生一个(i, s[i])形式的元组，以(0, s [0])开始。下面我们枚举频率分布的值，捕获变量 rank 和 word 中的整数-字符串对。按照产生排序项列表时的需要，输出 rank+1 使计数从 1 开始:

fd = nltk.FreqDist(nltk.corpus.brown.words())

cumulative = 0.0

most\_common\_words = [word for (word,count) in fd.most\_common()]

for rank,word in enumerate(most\_common\_words):

cumulative += fd.freq(word)

print("{:3} {:6.2%} {}".format(rank+1,cumulative,word))

if cumulative>0.25:

break

**计数器的一些合理用途**

在有些情况下，我们仍然要在链表推导中使用循环变量。例如:我们需要使用一个循环 变量中提取链表中连续重叠的 n-grams:

sent = ['The','dog','gave','John','the','newspaper']

n = 3

print([sent[i:i+n] for i in range(len(sent)-n+1)])

确保循环变量范围的正确相当棘手的。因为这是 NLP中的常见操作，NLTK提供了支 持函数bigrams(text)、trigrams(text)和一个更通用的ngrams(text, n)。

使用循环变量构建多维结构，建立一个 m行n列的数组，其中每个元素是一个集合，使用一个嵌套的链表推导:

m,n = 3,7

array = [[set() for i in range(n)] for j in range(m)]

array[2][5].add('Alice')

pprint.pprint(array)

循环变量 i 和 j 在产生对象过程中没有用到;它们只是需要一个语法正确的 for 语句。

由于前面所讨论的有关对象复制的原因，使用乘法做这项工作是不正确的：

array = [[set()]\*n]\*m

array[2][5].add(7)

pprint.pprint(array)

**4.4 函数:结构化编程的基础**

**函数的输入和输出**

一般情况下，函数应该修改参数的内容(my\_sort1())或返回一个值(my\_sort2())，而不是两个都做(my\_sort3()):

def my\_sort1(mylist):

# good:modifies its argument, no return value

mylist.sort()

def my\_sort2(mylist):

# good: doesn't touch its argument, returns value

return sorted(mylist)

def my\_sort3(mylist):

# bad: modifies its argument and also returns it

mylist.sort()

return mylist

**参数传递**

关键还是要理解前面提到的是引用修改还是覆盖饮用

**变量的作用域**

当在一个函数体内部使用一个现有的名字时， Python 解释器先尝试按照函数本地的名字来解释。如果没有发现，解释器检查它是否是一个模块内的全局名称。如果没有成功， 最后，解释器会检查是否是 Python 内置的名字。这就是所谓的名称解析的 **LGB 规则**:本地 (local)，全局(global)，然后内置(built-in)。

应尽量避免使用全局变量以致上下文依赖性而限制函数的便携性(或重用性)。一般来说，应该使用参数作为函数的输入，返回值作为函数的输出。

**参数类型检查**

使用 assert 语句和 Python 的 basestring 的类型一起，它是 unicode 和 str 的产生类型:

def tag(word):

assert isinstance(word,str),"argument to tag() must be a string"

if word in ['a','the','all']:

return 'det'

else:

return 'noun'

如果 assert 语句失败，它会产生一个不可忽视的错误而停止程序执行。程序中添加断言能帮助你找到逻辑错误，是一种**防御性编程**。

**功能分解**

结构良好的程序通常都广泛使用函数。当一个程序代码块增长到超过 10-20 行，如果将代码分成一个或多个函数，每一个有明确的目的，这将对可读性有很大的帮助。这类似于好文章被划分成段，每段话表示一个主要思想。

适当使用函数使程序更具可读性和可维护性。另外，重新实现一个函数已成为可能——使用更高效的代码替换函数体——不需要关心程序的其余部分。

**文档说明函数**

使用通俗易懂的语言描述每个函数的目的，并且在函数的定义顶部的 docstring 中提供这些描述。

docstring 中可以包括一个 **doctest 块**，说明使用的函数和预期的输出。可以使用 Python 的 docutils 模块自动测试。docstring 中应当记录函数的每个参数的类型和返回类型。 至少可以用纯文本来做这些。NLTK 中使用“epytext”标记语言来记录参数。这种格式可以自动转换成富结构化的 API 文档，并包含某些“字段”的特殊处理。

一个完整的docstring的演示，包括一行总结，一个更详细的解释，一个doctest例子以及特定参数、类型、返回值和异常的 epytext 标记。

**4.5 更多关于函数**

**作为参数的函数**

Python 允许我们传递一个函数作为另一个函数的参数:

sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',

'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']

def extract\_property(prop):

return [prop(word) for word in sent]

print(extract\_property(len))

def last\_letter(word):

return word[-1]

print(extract\_property(last\_letter))

注意，只有在调用一个函数时，才在函数名后使用括号;当只是将函数作为一个对象，括号被省略。

Python 提供了更多的方式来定义函数作为其他函数的参数，即所谓的 **lambda 表达式**：

print(extract\_property(lambda w:w[-1]))

**累计函数**

这些函数以初始化一些存储开始，迭代和处理输入的数据，最后返回一些最终的对象(一个大的结构或汇总的结果)。做到这一点的一个标准的方式是初始化一个空链表，累计材料， 然后返回这个链表:

def search1(substring,words):

result = []

for word in words:

if substring in word:

result.append(word)

return result

def search2(substring,words):

for word in words:

if substring in word:

yield word

函数 search2()是一个产生器。第一次调用此函数，它运行到 yield 语句然后停下来。 调用程序获得第一个词，没有任何必要的处理。一旦调用程序对另一个词做好准备，函数会从停下来的地方继续执行，直到再次遇到 yield 语句。这种方法通常更有效，因为函数只产生调用程序需要的数据，并不需要分配额外的内存来存储输出。

**高阶函数**

filter()函数对作为它的第二个参数的序列中的每个项目运用作为第一个参数的函数，只保留该函数返回 True 的项目。

map()函数，将一个函数运用到一个序列中的每一项。

记住，使用list()将函数结果变为链表才可以进行一些操作

**参数的命名**

分配了默认值的参数被称为**关键字参数**，未命名的参数必须在命名的参数前面，因为未命名参数是根据位置来定义的。可以定义一个函数，接受任意数量的未命名和命名参数，并通过一个就地的参数链表 \*args 和一个就地的关键字参数字典\*\*kwargs 来访问它们。

def generic(\*args,\*\*kwargs):

print(args)

print(kwargs)

generic(1,'African swallow',monty='Python')

当\*args 作为函数参数时，它实际上对应函数所有的未命名参数。

思考处理可变数目的参数的函数 zip():

song = [['four', 'calling', 'birds'],

['three', 'French', 'hens'],

['two', 'turtle', 'doves']]

print(list(zip(song[0],song[1],song[2])))

print(list(zip(\*song)))

输入\*song 仅仅是一个方便的记号，相当于输入了 song[0]，s ong[1]，song[2]。

已命名的参数的另一个作用是它们允许选择性使用参数。

可选参数的另一个常见用途是作为标志使用。

注意不要使用可变对象作为参数的默认值。这个函数的一系列调用将使用同一个对象，有时会出现离奇的结果

**4.6 程序开发**

**Python 模块的结构**

程序模块的目的是把逻辑上相关的定义和函数结合在一起，以方便重用和更高层次的抽象。

模块的一些变量和函数仅用于模块内部。它们的名字应该以下划线开头，如\_ helper()，因为这将隐藏名称。如果另一个模块使用习惯用法:from module import \* 导入这个模块，这些名称将不会被导入。可以选择性的列出一个模块的外部可访问的名称，使用像这样的一个特殊的内置变量:\_\_all\_\_ =['edit\_distance', 'jaccard\_distance']。

**多模块程序**

通过将工作分成几个模块和使用 import 语句访问别处定义的函数，可以保持各个模块简单，易于维护。这种做法也将导致越来越多的模块的集合，使之有可能建立复杂的涉及模块间层次结构的系统。

**误差源头**

即bugs

首先，输入的数据可能包含一些意想不到的字符。比如：对ph. d..n.01进行分解：

rsplit('.', 2) 利用最右边的逗号，最多分割两次，留下字符串 ph.d.不变。

第二，提供的函数可能不会像预期的那样运作。源自于对程序的错误理解。

第三，我们对 Python 语义的理解可能出错。比如对于操作符优先级的错误理解。

**调试技术**

由于大多数代码错误是因为程序员的不正确的假设，检测 bug 要做的第一件事是**检查你的假设**。通过给程序**添加 print 语句定位问题**，显示重要的变量的值，并显示程序的进展程度。

如果程序产生一个“异常”——运行时错误——解释器会输出一个**堆栈跟踪**，精确定位错误发生时程序执行的位置。如果程序取决于输入数据，尽量将它减少到能产生错误的最小尺寸。

使用交互式命令行重现错误发生时的情况往往是有益的。定义一些变量，然后复制粘贴可能 出错的代码行到会话中，看看会发生什么。

Python 提供了一个**调试器**，它允许你监视程序的执行，指定程序暂停运行的行号(即**断点**)，逐步调试代码段和检查变量的值:

>>> import pdb

>>> import mymodule

>>> pdb.run('mymodule.myfunction()')

输入 step(或只输入 s)将执行当前行然后停止。如果当前行调用一个函数，它将进入这个函数并停止在第一行。输入 next(或只输入 n)是类似的，但它会在当前函数中的下一行停止执行。break(或 b)命令可用于创建或列出断点。输入 continue(或 c)会继续执行直到遇到下一个断点。输入任何变量的名称可以检查它的值。

**防御性编程**

不要写 20 行程序然后测试它，而是自下而上的打造一些明确可以运作的小的程序片。每次你将这些程序片组合成更大的单位都要仔细的看它是否能如预期的运作。考虑在你的代码中添加 assert 语句，指定变量的属性，例如:assert(isinstance(text, list))。如果text的值在你的代码被用在一些较大的环境中时变为了一个字符串，将产生一个 AssertionError，于是你会立即得到问题的通知。

一旦你觉得你发现了错误，作为一个假设查看你的解决方案。在重新运行该程序之前尝试预测你修正错误的影响。如果 bug 不能被修正，不要陷入盲目修改代码希望它会奇迹般地重新开始运作的陷阱。相反，每一次修改都要尝试阐明错误是什么和为什么这样修改会解决 这个问题的假设。如果这个问题没有解决就撤消这次修改。

**4.7 算法设计**

解决算法问题的一个重要部分是为手头的问题选择或改造一个合适的算法。有时会有几种选择，能否选择最好的一个取决于对每个选择随数据增长如何执行的知识。

最有名的策略被称为**分而治之**。我们解决一个大小为 n 的问题通过将其分成两个大小为 n/2 的问题，解决这些问题，组合它们的结果成为原问题的结果。

另一个例子是在词典中查找一个词的过程。二分查找，每一步都将问题分裂成一半。

算法设计的另一种方法，我们解决问题通过将它转化为一个我们已经知道如何解决的问题的一个实例。

**递归**

解决一个大小为 n 的问题，可以 将其分成两半，然后处理一个或多个大小为 n/2 的问题。一种一般的方式来实现这种方法是使用**递归**。定义一个函数 f，从而简化了问题，并调用自身来解决一个或多个同样问题的更简单的实例。然后组合它们的结果成为原问题的解答:

def factoriall2(n):

if n == 1:

return 1

else:

return n\*factoriall2(n-1)

递归时需要**基础案例**，即n=1时的解

使用 import 语句的一个新的形式，允许我们缩写名称 wordnet 为 wn:

from nltk.corpus import wordnet as wn

递归时，每次函数调用时，一些状态信息需要推入堆栈，这样一旦函数执行完成可以从离开的地方继续执行。出于这个原因，迭代的解决方案往往比递归解决方案的更高效。

**权衡空间与时间**

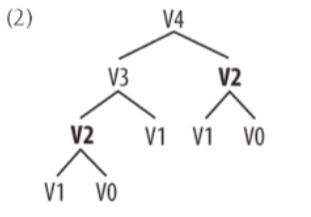
我们有时可以显著的加快程序的执行，通过建设一个辅助的数据结构，例如:索引。

使用整数标识符替换一个语料库的标识符。

维护一个词汇表。如果你需要处理一段输入文本检查所有的词是否在现有的词汇表中，词汇表应存储为一个集合，而不是一个链表。集合中的元素会自动索引，所以测试一个大的集合的成员将远远快于测试相应的链表的成员。使用Timer模块进行测试。

**动态规划**

动态规划用于解决包含多个重叠的子问题的问题。不是反复计算这些子问题，而是简单的将它们的计算结果存储在一个查找表中。



它的工作原理是使用问题的所有较小的实例的计算结果填充一个表格(叫做 lookup )，一旦我们得到了我们感兴趣的值就立即停止。此时，我们读出值，并返回它。最重要的是，每个子问题只计算了一次。

解决较大问题前先解决较小的问题，这被称为**自下而上**的方法进行动态规划。不幸的是，对于某些应用它还是相当浪费资源的，因为它计算的一些子问题在解决主问题时可能并不需要。采用**自上而下**的方法进行动态规划可避免这种计算的浪费。不同于自下而上的方法，这种方法是递归的。通过检查是否先前已存储了结果，它避免了单纯使用递归算法的巨大浪费。如果没有存储，就递归的计算结果，并将结果存储在表中。最后一步返回存储的结果。