6. Advanced Programming in Python

6.1 Introduction

This chapter introduces concepts in algorithms, data structures, program design, and advanced Python programming. It contains many working program fragments which you should try yourself.

6.2 Functions

6.2.1 Defining Functions

It often happens that part of a program needs to be used several times over. For example, suppose we were writing a program that needed to be able to form the plural of a singular noun, and that this needed to be done at various places during the program. Rather than repeating the same code several times over, it is more efficient (and reliable) to localize this work inside a *function*. A function is a programming construct which takes one or more inputs, and produces an output. In this case, we will take the singular noun as input, and generate a plural form as output:

```
>>> def plural(word):
...     if word[-1] == 'y':
...         return word[:-1] + 'ies'
...     elif word[-1] in 'sx':
...         return word + 'es'
...     elif word[-2:] in ['sh', 'ch']:
...         return word + 'es'
...     elif word[-2:] == 'an':
...         return word[:-2] + 'en'
...     return word + 's'
>>> plural('fairy')
'fairies'
>>> plural('woman')
'women'
```

Well-structured programs often make extensive use of functions. Often when a block of program code grows longer than a screenful, it is a great help to readability if it is decomposed into one or more functions.

6.2.2 Predicting the Next Word (Revisited)

```
>>> from nltk_lite.corpora import genesis
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
```

We then examine each token in the corpus, and increment the appropriate sample's count. We use the variable **prev** to record the previous word.

```
>>> prev = None
>>> for word in genesis.raw():
...     cfdist[prev].inc(word)
...     prev = word
```

Note

Sometimes the context for an experiment is unavailable, or does not exist. For example, the first token in a text does not follow any word. In these cases, we must decide what context to use. For this example, we use `None` as the context for the first token. Another option would be to discard the first token.

Once we have constructed a conditional frequency distribution for the training corpus, we can use it to find the most likely word for any given context. For example, taking the word *living* as our context, we can inspect all the words that occurred in that context.

```
>>> word = 'living'
>>> cfdist[word].samples()
['creature,', 'substance', 'soul.', 'thing', 'thing,', 'creature']
```

We can set up a simple loop to generate text: we set an initial context, picking the most likely token in that context as our next word, and then using that word as our new context:

```
>>> word = 'living'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
living creature that he said, I will not be a wife of the land
of the land of the land
```

NOW: doing this with functions...

6.3 Trees

6.4 Recursion

1. recurse over tree to display in some useful way (e.g. whitespace formatting)
2. recurse over tree to look for coordinate constructions (cf 4th example in chapter 1.1)
3. generate new dependency tree from a phrase-structure tree

(possible extension: callback function for Tree.subtrees())

6.5 Writing Complete Programs

6.5.1 Classifying Words Automatically

A tagged corpus can be used to *train* a simple classifier, which can then be used to guess the tag for untagged words. For each word, we can count the number of times it is tagged with each tag. For instance, the word **deal** is tagged 89 times as **nn** and 41 times as **vb**. On this evidence, if we were asked to guess the tag for **deal** we would choose **nn**, and we would be right over two-thirds of the time. The following program performs this tagging task, when trained on the "g" section of the Brown Corpus (so-called *belles lettres*, creative writing valued for its aesthetic content).

```
>>> from nltk_lite.corpora import brown
>>> cfdist = ConditionalFreqDist()
>>> for sentence in brown.tagged('g'):
...     for token in sentence:
...         word = token[0]
...         tag = token[1]
...         cfdist[word].inc(tag)
>>> for word in "John saw 3 polar bears".split():
...     print word, cfdist[word].max()
John np
saw vbd
3 cd-tl
polar jj
bears vbz
```

Note that **bears** was incorrectly tagged as the 3rd person singular form of a verb, since this word appears more frequently as a verb than a noun in esthetic writing.

A problem with this approach is that it creates a huge model, with an entry for every possible combination of word and tag. For certain tasks it is possible to construct reasonably good models which are tiny in comparison. For instance, let's try to guess whether a verb is a noun or adjective from the last letter of the word alone. We can do this as follows:

```
>>> tokens = []
>>> for sent in brown.tagged('g'):
...     for (word,tag) in sent:
...         if tag in ['nn', 'jj'] and len(word) > 3:
...             char = word[-1]
...             tokens.append((char,tag))
>>> split = len(tokens)*9/10
>>> train, test = tokens[:split], tokens[split:]
>>> cfdist = ConditionalFreqDist()
>>> for (char,tag) in train:
...     cfdist[char].inc(tag)
>>> correct = total = 0
>>> for (char,tag) in test:
...     if tag == cfdist[char].max():
...         correct += 1
...     total += 1
>>> print correct*100/total
71
```

This result of 71% is marginally better than the result of 65% that we get if we assign the **nn** tag to every word. We can inspect the model to see which tag is assigned to a word given its final letter. Here we learn that words which end in **c** or **l** are more likely to be adjectives than nouns:

```
>>> print [(c, cfdist[c].max()) for c in cfdist.conditions()]
[('%', 'nn'), ("'", None), ('-', 'jj'), ('2', 'nn'), ('5', 'nn'), ('A', 'nn'), ('D'
```

6.5.2 Exploring text genres

Now that we can load a significant quantity of tagged text, we can process it and extract items of interest. The following code iterates over the fifteen genres of the Brown Corpus (accessed using **brown.groups()**). The material for each genre lives in a set of files (accessed using **brown.items**). Each of these is tokenized in turn. The next step is to check if the token has the **md** tag. For each of these words we increment a count. This uses the conditional frequency distribution, where the condition is the current genre, and the event is the modal.

```
>>> cfdist = ConditionalFreqDist()
>>> for genre in brown.items:                    # each genre
...     for sent in brown.tagged(genre):         # each sentence
...         for (word,tag) in sent:              # each tagged token
...             if tag == 'md':                  # found a modal
...                 cfdist[genre].inc(word.lower())
```

The conditional frequency distribution is nothing more than a mapping from each genre to the distribution of modals in that genre. The following code fragment identifies a small set of modals of interest, and processes the data structure to output the required counts.

```
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> print "%-40s" % 'Genre', ' '.join([("%6s" % m) for m in modals])
Genre                                      can  could    may  might   must   will
>>> for genre in cfdist.conditions():    # generate rows
...     print "%-40s" % brown.item_name[genre],
...     for modal in modals:
...         print "%6d" % cfdist[genre].count(modal),
...     print
press: reportage                            94     86     66     36     50    387
press: reviews                              44     40     45     26     18     56
press: editorial                           122     56     74     37     53    225
skill and hobbies                          273     59    130     22     83    259
religion                                    84     59     79     12     54     64
belles-lettres                             249    216    213    113    169    222
popular lore                               168    142    165     45     95    163
miscellaneous: government & house organs   115     37    152     13     99    237
fiction: general                            39    168      8     42     55     50
learned                                    366    159    325    126    202    330
fiction: science                            16     49      4     12      8     16
fiction: mystery                            44    145     13     57     31     17
fiction: adventure                          48    154      6     58     27     48
fiction: romance                            79    195     11     51     46     43
humor                                       17     33      8      8      9     13
```

There are some interesting patterns in this table. For instance, compare the rows for government literature and adventure literature; the former is dominated by the use of **can, may, must, will** while the latter is characterised by the use of **could** and **might**. With some further work it might be possible to guess the genre of a new text automatically, according to its distribution of modals.

6.5.3 Exercises

1. **Classifying words automatically:** The program for classifying words as nouns or adjectives scored 71%. Try to come up with better conditions, to get the system to score 80% or better.

   a) Revise the condition to use a longer suffix of the word, such as the last two characters, or the last three characters. What happens to the performance? Which suffixes are diagnostic for adjectives?

   b) Explore other conditions, such as variable length prefixes of a word, or the length of a word, or the number of vowels in a word.

   c) Finally, combine multiple conditions into a tuple, and explore which combination of conditions gives the best result.

2. **Exploring text genres:** Investigate the table of modal distributions and look for other patterns. Try to explain them in terms of your own impressionistic understanding of the different genres. Can you find other closed classes of words that exhibit significant differences across different genres?

6.6 Program Development

Programming is a skill which is acquired over several years of experience with a variety of programming languages and tasks. Key high-level abilities are *algorithm design* and its manifestation in *structured programming*. Key low-level abilities include familiarity with the syntactic constructs of the language, and knowledge of a variety of diagnostic methods for trouble-shooting a program which does not exhibit the expected behaviour.

6.6.1 List Comprehensions

List comprehensions are another important Python construct. Many language processing tasks involve applying the same operation to every item in a list. Here we lowercase each word:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> [word.lower() for word in sent]
['the', 'dog', 'gave', 'john', 'the', 'newspaper']
```

As another example, we could remove all determiners from a list of words:

```
>>> def is_lexical(word):
...     return word.lower() not in ('a', 'an', 'the', 'that', 'to')
>>> [word for word in sent if is_lexical(word)]
['dog', 'gave', 'John', 'newspaper']
```

Or equivalently:

```
>>> filter(is_lexical, sent)
['dog', 'gave', 'John', 'newspaper']
```

Combining the transformations...

```
>>> [word.lower() for word in sent if is_lexical(word)]
['dog', 'gave', 'john', 'newspaper']
```

The following code builds a list of tuples, where each tuple consists of a word and its length.

```
>>> [(x, len(x)) for x in sent]
[('The', 3), ('dog', 3), ('gave', 4), ('John', 4), ('the', 3), ('newspaper', 9)]
```

6.6.2 Programming Style

We have just seen how the same task can be performed in different ways, with implications for efficiency. Another factor influencing program development is *programming style*. Consider the following program to compute the average length of words in the Brown Corpus:

```
>>> from nltk_lite.corpora import brown
>>> count = 0
>>> total = 0
>>> for sent in brown.raw('a'):
...     for token in sent:
...         count += 1
...         total += len(token)
>>> print float(total) / count
4.2765382469
```

In this program we use the variable **count** to keep track of the number of tokens seen, and **total** to store the combined length of all words. This is a low-level style, not far removed from machine code, the primitive operations performed by the computer's CPU. The two variables are just like a CPU's registers, accumulating values at many intermediate stages, values which are almost meaningless. We say that this program is written in a *procedural* style, dictating the machine operations step by step. Now consider the following program which computes the same thing:

```
>>> tokens = [token for sent in brown.raw('a') for token in sent]
>>> total = sum(map(len, tokens))
>>> print float(total)/len(tokens)
4.2765382469
```

The first line uses a list comprehension to construct the sequence of tokens. The second line *maps* the **len** function to this sequence, to create a list of length values, which are summed. The third line computes the average as before. Notice here that each line of code performs a complete, meaningful action. Moreover, they do not dictate how the computer will perform the computations; we state high level relationships like "**total** is the sum of the lengths of the tokens" and leave the details to the Python interpreter. Accordingly, we say that this program is written in a *declarative* style.

Here is another example to illustrate the procedural/declarative distinction. Notice again that the procedural version involves low-level steps and a variable having meaningless intermediate values:

```
>>> word_list = []
>>> for sent in brown.raw('a'):
...     for token in sent:
...         if token not in word_list:
...             word_list.append(token)
>>> word_list.sort()
```

The declarative version (given second) makes use of higher-level built-in functions:

```
>>> tokens = [word for sent in brown.raw('a') for word in sent]
>>> word_list = list(set(tokens))
>>> word_list.sort()
```

What do these programs compute? Which version was easier to interpret?

Consider one further example, which sorts three-letter words by their final letters. The words come from the widely-used Unix word-list, made available as an NLTK corpus called **words**. Two words ending with the same letter will be sorted according to their second-last letters. The result of this sort method is that rhyming words will be contiguous. Two programs are given; Which one is more declarative, and which is more procedural?

As an aside, for readability we define a function for reversing strings that will be used by both programs:

```
>>> def reverse(word):
...     return word[::-1]
```

Here's the first program. We define a helper function **reverse_cmp** which calls the built-in **cmp** comparison function on reversed strings. The **cmp** function returns **-1**, **0**, or **1**, depending on whether its first argument is less than, equal to, or greater than its second argument. We tell the list sort function to use **reverse_cmp** instead of **cmp** (the default).

```
>>> from nltk_lite.corpora import words
>>> def reverse_cmp(x,y):
...     return cmp(reverse(x), reverse(y))
>>> word_list = [word for word in words.raw('en') if len(word) == 3]
>>> word_list.sort(reverse_cmp)
>>> print word_list[-12:]
['toy', 'spy', 'cry', 'dry', 'fry', 'pry', 'try', 'buy', 'guy', 'ivy', 'Paz', 'Liz'
```

Here's the second program. In the first loop it collects up all the three-letter words in reversed form. Next, it sorts the list of reversed words. Then, in the second loop, it iterates over each position in the list using the variable **i**, and replaces each item with its reverse. We have now re-reversed the words, and can print them out.

```
>>> word_list = []
>>> for word in words.raw('en'):
...     if len(word) == 3:
...         word_list.append(reverse(word))
>>> word_list.sort()
>>> for i in range(len(word_list)):
...     word_list[i] = reverse(word_list[i])
>>> print word_list[-12:]
['toy', 'spy', 'cry', 'dry', 'fry', 'pry', 'try', 'buy', 'guy', 'ivy', 'Paz', 'Liz'
```

Choosing between procedural and declarative styles is just that, a question of style. There are no hard boundaries, and it is possible to mix the two. Readers new to programming are encouraged to experiment with both styles, and to make the extra effort required to master higher-level constructs, such as list comprehensions, and built-in functions like `map`.

6.6.3 Exercises

1. Write a program to sort words by length. Define a helper function `cmp_len` which uses the `cmp` comparison function on word lengths.

2. Consider the tokenized sentence `['The', 'dog', 'gave', 'John', 'the', 'newspaper']`. Using the `map()` and `len()` functions, write a single line program to convert this list of tokens into a list of token lengths: `[3, 3, 4, 4, 3, 9]`

6.7 Further Reading