# Chapter 10

# Advanced Programming in Python

This chapter introduces concepts in algorithms, data structures, program design, and advanced Python programming. It also contains review of the basic mathematical notions of set, relation, and function, and illustrates them in terms of Python data structures. It contains many working program fragments which you should try yourself.

## 10.1 Object-Oriented Programming in Python

Object-Oriented Programming is a programming paradigm in which complex structures and processes are decomposed into modules, each encapsulating a single data type and the legal operations on that type.

### 10.1.1 Variable scope (notes)

- local and global variables
- scope rules
- global variables introduce dependency on context and limits the reusability of a function
- importance of avoiding side-effects
- functions hide implementation details

### 10.1.2 Modules

### 10.1.3 Data Classes: Trees in NLTK

An important data type in language processing is the syntactic tree. Here we will review the parts of the NLTK code which defines the `Tree` class.

The first line of a class definition is the `class` keyword followed by the class name, in this case `Tree`. This class is derived from Python's built-in `list` class, permitting us to use standard list operations to access the children of a tree node.

```
>>> class Tree(list):
```

Next we define the initializer, also known as the *constructor*. It has a special name, starting and ending with double underscores; Python knows to call this function when you asks for a new tree object by writing `t = Tree(node, children)`. The constructor's first argument is special, and is standardly called `self`, giving us a way to refer to the current object from inside the definition. This constructor calls the list initializer (similar to calling `self = list(children)`), then defines the `node` property of a tree.

```
...     def __init__(self, node, children):
...         list.__init__(self, children)
...         self.node = node
```

Next we define another special function that Python knows to call when we index a Tree. The first case is the simplest, when the index is an integer, e.g. `t[2]`, we just ask for the list item in the obvious way. The other cases are for handling slices, like `t[1:2]`, or `t[:]`.

```
...     def __getitem__(self, index):
...         if isinstance(index, int):
...             return list.__getitem__(self, index)
...         else:
...             if len(index) == 0:
...                 return self
...             elif len(index) == 1:
...                 return self[int(index[0])]
...             else:
...                 return self[int(index[0])][index[1:]]
...
```

This method was for accessing a child node. Similar methods are provided for setting and deleting a child (using `__setitem__`) and `__delitem__`).

Two other special member functions are `__repr__()` and `__str__()`. The `__repr__()` function produces a string representation of the object, one which can be executed to re-create the object, and is accessed from the interpreter simply by typing the name of the object and pressing 'enter'. The `__str__()` function produces a human-readable version of the object; here we call a pretty-printing function we have defined called `pp()`.

```
...     def __repr__(self):
...         import string
...         childstr = string.join([repr(c) for c in self])
...         return '(%s: %s)' % (self.node, childstr)
...     def __str__(self):
...         return self.pp()
```

Next we define some member functions that do other standard operations on trees. First, for accessing the leaves:

```
...     def leaves(self):
...         leaves = []
...         for child in self:
...             if isinstance(child, Tree):
...                 leaves.extend(child.leaves())
...             else:
...                 leaves.append(child)
...         return leaves
```

Next, for computing the height:

```
...     def height(self):
...         max_child_height = 0
...         for child in self:
...             if isinstance(child, Tree):
...                 max_child_height = max(max_child_height, child.height())
...             else:
...                 max_child_height = max(max_child_height, 1)
...         return 1 + max_child_height
```

And finally, for enumerating all the subtrees (optionally filtered):

```
...     def subtrees(self, filter=None):
...         if not filter or filter(self):
...             yield self
...         for child in self:
...             if isinstance(child, Tree):
...                 for subtree in child.subtrees(filter):
...                     yield subtree
```

### 10.1.4 Processing Classes: N-gram Taggers in NLTK

This section will discuss the `tag.ngram` module.

## 10.2 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.

### 10.2.1 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 did you find 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 many 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` and `filter`.

### 10.2.2 Debugging

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
```

Commands:
list [first [,last]]: list sourcecode for the current file
next: continue execution until the next line in the current function is reached
cont: continue execution until a breakpoint is reached (or the end of the program)
break: list the breakpoints
break n: insert a breakpoint at this line number in the current file
break file.py:n: insert a breakpoint at this line in the specified file
break function: insert a breakpoint at the first executable line of the function

### 10.2.3 Case Study: T9

### 10.2.4 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]`


Figure 10.1: T9: Text on 9 Keys

## 10.3 XML

1. ◑ Write a recursive function to produce an XML representation for a tree, with non-terminals represented as XML elements, and leaves represented as text content, e.g.:

```
<S>
    <NP type="SBJ">
        <NP>
            <NNP>Pierre</NNP>
            <NNP>Vinken</NNP>
        </NP>
        <COMMA>,</COMMA>
        <ADJP>
            <NP>
                <CD>61</CD>
                <NNS>years</NNS>
            </NP>
            <JJ>old</JJ>
        <COMMA>,</COMMA>
    </NP>
    <VP>
        <MD>will</MD>
        <VP>
            <VB>join</VB>
            <NP>
                <DT>the</DT>
                <NN>board</NN>
            </NP>
            <PP type="CLR">
                <IN>as</IN>
                <NP>
                    <DT>a</DT>
                    <JJ>nonexecutive</JJ>
```

<NN>director</NN>
</NP>
</PP>
<NP type="TMP">
<NNP>Nov.</NNP>
<CD>29</CD>
</NP>
</VP>
</VP>
<PERIOD>.</PERIOD>
</S>

## 10.4 Algorithm Design

An *algorithm* is a "recipe" for solving a problem. For example, to multiply 16 by 12 we might use any of the following methods:

1. Add 16 to itself 12 times over
2. Perform "long multiplication", starting with the least-significant digits of both numbers
3. Look up a multiplication table
4. Repeatedly halve the first number and double the second, 16*12 = 8*24 = 4*48 = 2*96 = 192
5. Do 10*12 to get 120, then add 6*12

Each of these methods is a different algorithm, and requires different amounts of computation time and different amounts of intermediate information to store. A similar situation holds for many other superficially simple tasks, such as sorting a list of words. Now, as we saw above, Python provides a built-in function `sort()` that performs this task efficiently. However, NLTK-Lite also provides several algorithms for sorting lists, to illustrate the variety of possible methods. To illustrate the difference in efficiency, we will create a list of 1000 numbers, randomize the list, then sort it, counting the number of list manipulations required.

```
>>> from random import shuffle
>>> a = range(1000)                    # [0,1,2,...999]
>>> shuffle(a)                         # randomize
```

Now we can try a simple sort method called *bubble sort*, which scans through the list many times, exchanging adjacent items if they are out of order. It sorts the list `a` in-place, and returns the number of times it modified the list:

```
>>> from nltk_lite.misc import sort
>>> sort.bubble(a)
250918
```

We can try the same task using various sorting algorithms. Evidently *merge sort* is much better than bubble sort, and *quicksort* is better still.

```
>>> shuffle(a); sort.merge(a)
6175
>>> shuffle(a); sort.quick(a)
2378
```

Readers are encouraged to look at `nltk_lite.misc.sort` to see how these different methods work. The collection of NLTK-Lite modules exemplify a variety of algorithm design techniques, including brute-force, divide-and-conquer, dynamic programming, and greedy search. Readers who would like a systematic introduction to algorithm design should consult the resources mentioned at the end of this tutorial.

### 10.4.1 Decorate-Sort-Undecorate

In Chapter 6 we saw how to sort a list of items according to some property of the list.

```
>>> words = 'I turned off the spectroroute'.split()
>>> words.sort(cmp)
>>> words
['I', 'off', 'spectroroute', 'the', 'turned']
>>> words.sort(lambda x, y: cmp(len(y), len(x)))
>>> words
['spectroroute', 'turned', 'off', 'the', 'I']
```

This is inefficient when the list of items gets long, as we compute `len()` twice for every comparison (about 2nlog(n) times). The following is more efficient:

```
>>> [pair[1] for pair in sorted((len(w), w) for w in words)[::-1]]
['spectroroute', 'turned', 'the', 'off', 'I']
```

This technique is called **decorate-sort-undecorate**. We can compare its performance by timing how long it takes to execute it a million times.

```
>>> from timeit import Timer
>>> Timer("sorted(words, lambda x, y: cmp(len(y), len(x)))",
...       "words='I turned off the spectroroute'.split()").timeit()
8.3548779487609863
>>> Timer("[pair[1] for pair in sorted((len(w), w) for w in words)]",
...      "words='I turned off the spectroroute'.split()").timeit()
9.9698889255523682
```

MORE: consider what happens as the lists get longer...

### 10.4.2 Problem Transformation (aka Transform-and-Conquer)

Find words which, when reversed, make legal words. Extremely wasteful solution:

```
>>> from nltk_lite.corpora import words
>>> for word1 in words.raw():
...     for word2 in words.raw():
...         if word1 == word2[::-1]:
...             print word1
```

More efficient:

```
>>> from nltk_lite.corpora import words
>>> wordlist = set(words.raw())
>>> rev_wordlist = set(w[::-1] for w in wordlist)
>>> sorted(wordlist.intersection(rev_wordlist))
['ah', 'are', 'bag', 'ban', 'bard', 'bat', 'bats', 'bib', 'bob', 'boob', 'brag',
'bud', 'buns', 'bus', 'but', 'civic', 'dad', 'dam', 'decal', 'deed', 'deeps', 'deer',
'deliver', 'denier', 'desserts', 'deus', 'devil', 'dial', 'diaper', 'did', 'dim',
'dog', 'don', 'doom', 'drab', 'draw', 'drawer', 'dub', 'dud', 'edit', 'eel', 'eke',
'em', 'emit', 'era', 'ere', 'evil', 'ewe', 'eye', 'fires', 'flog', 'flow', 'gab',
'gag', 'garb', 'gas', 'gel', 'gig', 'gnat', 'god', 'golf', 'gulp', 'gum', 'gums',
'guns', 'gut', 'ha', 'huh', 'keel', 'keels', 'keep', 'knits', 'laced', 'lager',
'laid', 'lap', 'lee', 'leek', 'leer', 'leg', 'leper', 'level', 'lever', 'liar',
'live', 'lived', 'loop', 'loops', 'loot', 'loots', 'mad', 'madam', 'me', 'meet',
'mets', 'mid', 'mood', 'mug', 'nab', 'nap', 'naps', 'net', 'nip', 'nips', 'no',
'nod', 'non', 'noon', 'not', 'now', 'nun', 'nuts', 'on', 'pal', 'pals', 'pan',
'pans', 'par', 'part', 'parts', 'pat', 'paws', 'peek', 'peels', 'peep', 'pep',
'pets', 'pin', 'pins', 'pip', 'pit', 'plug', 'pool', 'pools', 'pop', 'pot', 'pots',
'pup', 'radar', 'rail', 'rap', 'rat', 'rats', 'raw', 'redder', 'redraw', 'reed',
'reel', 'refer', 'regal', 'reined', 'remit', 'repaid', 'repel', 'revel', 'reviled',
'reviver', 'reward', 'rotator', 'rotor', 'sag', 'saw', 'sees', 'serif', 'sexes',
'slap', 'sleek', 'sleep', 'sloop', 'smug', 'snap', 'snaps', 'snip', 'snoops',
'snub', 'snug', 'solos', 'span', 'spans', 'spat', 'speed', 'spin', 'spit', 'spool',
'spoons', 'spot', 'spots', 'stab', 'star', 'stem', 'step', 'stew', 'stink', 'stool',
'stop', 'stops', 'strap', 'straw', 'stressed', 'stun', 'sub', 'sued', 'swap', 'tab',
'tang', 'tap', 'taps', 'tar', 'teem', 'ten', 'tide', 'time', 'timer', 'tip', 'tips',
'tit', 'ton', 'tool', 'top', 'tops', 'trap', 'tub', 'tug', 'war', 'ward', 'warder',
'warts', 'was', 'wets', 'wolf', 'won']
```

Observe that this output contains redundant information; each word and its reverse is included. How could we remove this redundancy?

Presorting, sets:

Find words which have at least (or exactly) one instance of all vowels. Instead of writing extremely complex regular expressions, some simple preprocessing does the trick:

```
>>> words = ["sequoia", "abacadabra", "yiieeaouuu!"]
>>> vowels = "aeiou"
>>> [w for w in words if set(w).issuperset(vowels)]
['sequoia', 'yiieeaouuu!']
>>> [w for w in words if sorted(c for c in w if c in vowels) == list(vowels)]
['sequoia']
```

### 10.4.3 Exercises

1. ◑ Consider again the problem of hyphenation across linebreaks. Suppose that you have successfully written a tokenizer that returns a list of strings, where some strings may contain a hyphen followed by a newline character, e.g. `long-\nterm`. Write a function which iterates over the tokens in a list, removing the newline character from each, in each of the following ways:

   a) Use doubly-nested for loops. The outer loop will iterate over each token in the list, while the inner loop will iterate over each character of a string.

   b) Replace the inner loop with a call to `re.sub()`

c) Finally, replace the outer loop with call to the `map()` function, to apply this substitution to each token.

d) Discuss the clarity (or otherwise) of each of these approaches.

2. ★ Develop a simple extractive summarization tool, which prints the sentences of a document which contain the highest total word frequency. Use `FreqDist` to count word frequencies, and use `sum` to sum the frequencies of the words in each sentence. Rank the sentences according to their score. Finally, print the *n* highest-scoring sentences in document order. Carefully review the design of your program, especially your approach to this double sorting. Make sure the program is written as clearly as possible.

## 10.5 Search

Many NLP tasks can be construed as search problems. For example, the task of a parser is to identify one or more parse trees for a given sentence. As we saw in Part II, there are several algorithms for parsing. A *recursive descent parser* performs **backtracking search**, applying grammar productions in turn until a match with the next input word is found, and backtracking when there is no match. We saw in Chapter 8 that the space of possible parse trees is very large; a parser can be thought of as providing a relatively efficient way to find the right solution(s) within a very large space of candidates.

As another example of search, suppose we want to find the most complex sentence in a text corpus. Before we can begin we have to be explicit about how the complexity of a sentence is to be measured: word count, verb count, character count, parse-tree depth, etc. In the context of learning this is known as the **objective function**, the property of candidate solutions we want to optimize.

In this section we will explore some other search methods which are useful in NLP. For concreteness we will apply them to the problem of learning word segmentations in text, following the work of [Brent, 1995]. Put simply, this is the problem faced by a language learner in dividing a continuous speech stream into individual words. We will consider this problem from the perspective of a child hearing utterances from a parent, e.g.

(1a) doyouseethekitty

(1b) seethedoggy

(1c) doyoulikethekitty

(1d) likethedoggy

Our first challenge is simply to represent the problem: we need to find a way to separate the text content from the segmentation. We will borrow an idea from IOB-tagging (Chapter 5), by annotating each character with a boolean value to indicate whether or not a word-break appears after the character. We will assume that the learner is given the utterance breaks, since these often correspond to extended pauses. Here is a possible representation, including the initial and target segmentations:

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "000000000000001000000000010000000000000000100000000000"
>>> seg2 = "010010010010000100100100001010010001001000010001010000"
```

Observe that the segmentation strings consist of zeros and ones. They are one character shorter than the source text, since a text of length $n$ can only be broken up in $n - 1$ places.

Now let's check that our chosen representation is effective. We need to make sure we can read segmented text from the representation. The following function, `segment()`, takes a text string and a segmentation string, and returns a list of strings.

**Listing 1** Program to Reconstruct Segmented Text from String Representation

```
    def segment(text, segs):
        words = []
        last = 0
        for i in range(len(segs)):
            if segs[i] == '1':
                words.append(text[last:i+1])
                last = i+1
        words.append(text[last:])
        return words

>>> segment(text, seg1)
['doyouseethekitty', 'seethedoggy', 'doyoulikethekitty', 'likethedoggy']
>>> segment(text, seg2)
['do', 'you', 'see', 'the', 'kitty', 'see', 'the', 'doggy', 'do', 'you',
 'like', 'the', kitty', 'like', 'the', 'doggy']
```

Now the learning task becomes a search problem: find the bit string that causes the text string to be correctly segmented into words. Our first task is done: we have represented the problem in a way that allows us to reconstruct the data, and to focus on the information to be learned.

Now that we have effectively represented the problem we need to choose the objective function. We assume the learner is acquiring words and storing them in an internal lexicon. Given a suitable lexicon, it is possible to reconstruct the source text as a sequence of lexical items. Following [Brent, 1995], we can use the size of the lexicon and the amount of information needed to reconstruct the source text as the basis for an objective function, as shown in Figure 10.2.

It is a simple matter to implement this objective function, as shown in Listing 10.2.

### 10.5.1 Exhaustive Search

- brute-force approach
- enumerate search space, evaluate at each point
- this example: search space size is $2^{55}$ = 36,028,797,018,963,968

For a computer that can do 100,000 evaluations per second, this would take over 10,000 years!

### 10.5.2 Hill-Climbing Search

Starting from a given location in the search space, evaluate nearby locations and move to a new location only if it is an improvement on the current location.

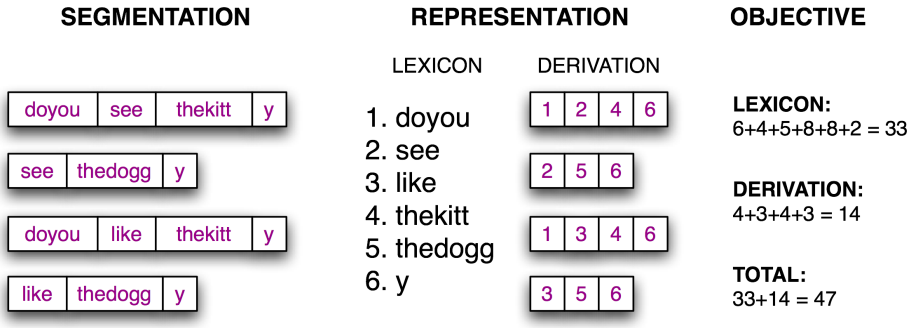


Figure 10.2: Calculation of Objective Function for Given Segmentation

**Listing 2** Computing the Cost of Storing the Lexicon and Reconstructing the Source Text

```
def evaluate(text, segs):
    import string
    words = segment(text, segs)
    text_size = len(words)
    lexicon_size = len(string.join(list(set(words))))
    return text_size + lexicon_size

>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg3 = "000010010000001100100000011000010001000000110001000000l"
>>> segment(text, seg3)
['doyou', 'see', 'thekitt', 'y', 'see', 'thedogg', 'y', 'doyou', 'like',
 'thekitt', 'y', 'like', 'thedogg', 'y']
>>> evaluate(text, seg3)
47
```

**Listing 3** Hill-Climbing Search

```
def flip(segs, pos):
    return segs[:pos] + `1-int(segs[pos])` + segs[pos+1:]
def hill_climb(text, segs, iterations):
        for i in range(iterations):
            pos, best = 0, evaluate(text, segs)
            for i in range(len(segs)):
                score = evaluate(text, flip(segs, i))
                if score < best:
                    pos, best = i, score
            if pos != 0:
                segs = flip(segs, pos)
                print evaluate(text, segs), segment(text, segs)
        return segs

>>> print evaluate(text, seg1), segment(text, seg1)
63 ['doyouseethekitty', 'seethedoggy', 'doyoulikethekitty', 'likethedoggy']
>>> hill_climb(text, segs1, 20)
61 ['doyouseethekittyseethedoggy', 'doyoulikethekitty', 'likethedoggy']
59 ['doyouseethekittyseethedoggydoyoulikethekitty', 'likethedoggy']
57 ['doyouseethekittyseethedoggydoyoulikethekittylikethedoggy']
```

### 10.5.3 Non-Deterministic Search

- Simulated annealing

## 10.6 Miscellany

### 10.6.1 Named Arguments

One of the difficulties in re-using functions is remembering the order of arguments. Consider the following function, which finds the `n` most frequent words that are at least `min_len` characters long:

```
>>> from nltk_lite.probability import FreqDist
>>> from nltk_lite import tokenize
>>> def freq_words(file, min, num):
...     freqdist = FreqDist()
...     text = open(file).read()
...     for word in tokenize.wordpunct(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...     return freqdist.sorted_samples()[:num]
>>> freq_words('programming.txt', 4, 10)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words',
'very', 'using']
```

This function has three arguments. It follows the convention of listing the most basic and substantial argument first (the file). However, it might be hard to remember the order of the second and third

**Listing 4** Non-Deterministic Search Using Simulated Annealing

```
    def flip_n(segs, n):
        for i in range(n):
            segs = flip(segs, randint(0,len(segs)-1))
        return segs
    def anneal(text, segs, iterations, rate):
        distance = float(len(segs))
        while distance > 0.5:
            best_segs, best = segs, evaluate(text, segs)
            for i in range(iterations):
                guess = flip_n(segs, int(round(distance)))
                score = evaluate(text, guess)
                if score < best:
                    best = score
                    best_segs = guess
            segs = best_segs
            score = best
            distance = distance/rate
            print evaluate(text, segs),
        print
        return segs

>>> anneal(text, segs, 5000, 1.2)
60 ['doyouseetheki', 'tty', 'see', 'thedoggy', 'doyouliketh', 'ekittylike', 'thedo
58 ['doy', 'ouseetheki', 'ttysee', 'thedoggy', 'doy', 'o', 'ulikethekittylike', 't
56 ['doyou', 'seetheki', 'ttysee', 'thedoggy', 'doyou', 'liketh', 'ekittylike', 't
54 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'likethekittylike', 'thedo
53 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like'
51 ['doyou', 'seethekittysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 't
42 ['doyou', 'see', 'thekitty', 'see', 'thedoggy', 'doyou', 'like', 'thekitty', 'l
```

arguments on subsequent use. We can make this function more readable by using **keyword arguments**. These appear in the function's argument list with an equals sign and a default value:

```
>>> def freq_words(file, min=1, num=10):
...     freqdist = FreqDist()
...     text = open(file).read()
...     for word in tokenize.wordpunct(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...     return freqdist.sorted_samples()[:num]
```

Now there are several equivalent ways to call this function:

```
>>> freq_words('programming.txt', 4, 10)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
>>> freq_words('programming.txt', min=4, num=10)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
>>> freq_words('programming.txt', num=10, min=4)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
```

When we use an integrated development environment such as IDLE, simply typing the name of a function at the command prompt will list the arguments. Using named arguments helps someone to re-use the code...

A side-effect of having named arguments is that they permit optionality. Thus we can leave out any arguments for which we are happy with the default value.

```
>>> freq_words('programming.txt', min=4)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
>>> freq_words('programming.txt', 4)
['string', 'word', 'that', 'this', 'phrase', 'Python', 'list', 'words', 'very', 'using']
```

Another common use of optional arguments is to permit a flag, e.g.:

```
>>> def freq_words(file, min=1, num=10, trace=False):
...     freqdist = FreqDist()
...     if trace: print "Opening", file
...     text = open(file).read()
...     if trace: print "Read in %d characters" % len(file)
...     for word in tokenize.wordpunct(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...             if trace and freqdist.N() % 100 == 0: print "."
...     if trace: print
...     return freqdist.sorted_samples()[:num]
```

### 10.6.2 Accumulative Functions

These functions start by initializing some storage, and iterate over input to build it up, before returning some final object (a large structure or aggregated result). The standard way to do this is to initialize an empty list, accumulate the material, then return the list:

```
>>> def find_nouns(tagged_text):
...     nouns = []
...     for word, tag in tagged_text:
```

```
...         if tag[:2] == 'NN':
...             nouns.append(word)
...     return nouns
```

We can apply this function to some tagged text to extract the nouns:

```
>>> tagged_text = [('the', 'DT'), ('cat', 'NN'), ('sat', 'VBD'),
...                ('on', 'IN'), ('the', 'DT'), ('mat', 'NN')]
>>> find_nouns(tagged_text)
['cat', 'mat']
```

However, a superior way to do this is to define a **generator**

```
>>> def find_nouns(tagged_text):
...     for word, tag in tagged_text:
...         if tag[:2] == 'NN':
...             yield word
```

The first time this function is called, it gets as far as the `yield` statement and stops. The calling program gets the first word and does any necessary processing. Once the calling program is ready for another word, execution of the function is continued from where it stopped, until the next time it encounters a `yield` statement.

Let's see what happens when we call the function:

```
>>> find_nouns(tagged_text)
<generator object at 0x14b2f30>
```

We cannot call it directly. Instead, we can convert it to a list.

```
>>> list(find_nouns(tagged_text))
['cat', 'mat']
```

We can also iterate over it in the usual way:

```
>>> for noun in find_nouns(tagged_text):
...     print noun,
cat mat
```

[Efficiency]

## 10.7 Sets and Mathematical Functions

### 10.7.1 Sets

Knowing a bit about sets will come in useful when you look at Chapter 11. A set is a collection of entities, called the **members** of the set. Sets can be finite or infinite, or even empty. In Python, we can define a set just by listing its members; the notation is similar to specifying a list:

```
>>> set1 = set(['a', 'b', 1, 2, 3])
>>> set1
set(['a', 1, 2, 'b', 3])
```

In mathematical notation, we would specify this set as:

(2) {'a', 'b', 1, 2, 3}

Set membership is a relation — we can ask whether some entity $x$ belongs to a set $A$ (in mathematical notation, written $x \in A$).

```
>>> 'a' in set1
True
>>> 'c' in set1
False
```

However, sets differ from lists in that they are *unordered* collections. Two sets are equal if and only if they have exactly the same members:

```
>>> set2 = set([3, 2, 1, 'b', 'a'])
>>> set1 == set2
True
```

The **cardinality** of a set $A$ (written $|A|$) is the number of members in $A$. We can get this value using the `len()` function:

```
>>> len(set1)
5
```

The argument to the `set()` constructor can be any sequence, including a string, and just calling the constructor with no argument creates the empty set (written ).

```
>>> set('123')
set(['1', '3', '2'])
>>> a = set()
>>> b = set()
>>> a == b
True
```

We can construct new sets out of old ones. The **union** of two sets $A$ and $B$ (written $A \cup B$) is the set of elements which belong to $A$ or $B$. Union is represented in Python with `|`:

```
>>> odds = set('13579')
>>> evens = set('02468')
>>> numbers = odds | evens
>>> numbers
set(['1', '0', '3', '2', '5', '4', '7', '6', '9', '8'])
```

The **intersection** of two sets $A$ and $B$ (written $A \cap B$) is the set of elements which belong to both $A$ and $B$. Intersection is represented in Python with `&`. If the intersection of two sets is empty, they are said to be **disjoint**.

```
>>> ints
set(['1', '0', '2', '-1', '-2'])
>>> ints & nats
set(['1', '0', '2'])
>>> odds & evens
set([])
```

The **(relative) complement** of two sets $A$ and $B$ (written $A - B$) is the set of elements which belong to $A$ but not $B$. Complement is represented in Python with `-`.

```
>>> nats - ints
set(['3', '5', '4', '7', '6', '9', '8'])
>>> odds == nats - evens
True
>>> odds == odds - set()
True
```

So far, we have described how to define 'basic' sets and how to form new sets out of those basic ones. All the basic sets have been specified by listing all their members. Often we want to specify set membership more succinctly:

(3) the set of positive integers less than 10

(4) the set of people in Melbourne with red hair

We can informally write these sets using the following **predicate notation**:

(5) $\{x \mid x \text{ is a positive integer less than } 10\}$

(6) $\{x \mid x \text{ is a person in Melbourne with red hair}\}$

In axiomatic set theory, the axiom schema of comprehension states that given a one-place predicate $P$, there is set $A$ such that for all $x$, $x$ belongs to $A$ if and only if (written $\equiv$) $P(x)$ is true:

(7) $A \forall x.(x \in A \equiv P(x))$

From a computational point of view, (7) is problematic: we have to treat sets as finite objects in the computer, but there is nothing to stop us defining infinite sets using comprehension. Now, there is a variant of (7), called the axiom of restricted comprehension, which allows us to specify a set $A$ with a predicate $P$ so long as we only consider $x$s which belong to some *already defined set* $B$:

(8) $\forall B\, A \forall x.\ (x \in A \equiv x \in B \wedge P(x))$

(For all sets $B$ there is a set $A$ such that for all $x$, $x$ belongs to $A$ if and only if $x$ belongs to $B$ and $P(x)$ is true.) This is equivalent to the following set in predicate notation:

(9) $\{x \mid x \in B \wedge P(x))$

(8) corresponds pretty much to what we get with list comprehension in Python: if you already have a list, then you can define a new list in terms of the old one, using an `if` condition. In other words, (10) is the Python counterpart of (8).

(10) `set([x for x in B if P(x)])`

To illustrate this further, the following list comprehension relies on the existence of the previously defined set `nats` (`n % 2` is the remainder when `n` is divided by `2`):

```
>>> nats = set(range(10))
>>> evens1 = set([n for n in nats if n % 2 == 0])
>>> evens1
set([0, 8, 2, 4, 6])
```

Now, when we defined `evens` before, what we actually had was a set of *strings*, rather than Python integers. But we can use `int` to coerce the strings to be of the right type:

```
>>> evens2 = set([int(n) for n in evens])
>>> evens1 == evens2
True
```

If every member of $A$ is also a member of $B$, we say that $A$ is a subset of $B$ (written $A \subseteq B$). The subset relation is represented in Python with <=.

```
>>> evens1 <= nats
True
>>> set() <= nats
True
>>> evens1 <= evens1
True
```

As the above examples show, $B$ can contain more members than $A$ for $A \subseteq B$ to hold, but this need not be so. Every set is a subset of itself. To exclude the case where a set is a subset of itself, we use the relation **proper subset** (written $A \subset B$). In Python, this relation is represented as <.

```
>>> evens1 < nats
True
>>> evens1 < evens1
False
```

Sets can contain other sets. For instance, the set $A = \{\{a\}, \{b\}\}$ contains the two singleton sets $\{a\}$ and $\{b\}$. Note that $\{a\} \subseteq A$ does not hold, since $a$ belongs to $\{a\}$ but not to $A$. In Python, it is a bit more awkward to specify sets whose members are also sets; the latter have to be defined as `frozensets`, i.e., immutable objects.

```
>>> a = frozenset('a')
>>> aplus = set([a])
>>> aplus
set([frozenset(['a'])])
```

We also need to be careful to distinguish between the empty set and the set whose only member is the empty set: $\{\}$.

### 10.7.2 Exercises

1. ☼ For each of the following sets, write a specification by hand in predicate notation, and an implementation in Python using list comprehension.

   a. {2, 4, 8, 16, 32, 64}
   b. {2, 3, 5, 7, 11, 13, 17}
   c. {0, 2, -2, 4, -4, 6, -6, 8, -8}

2. ☼ The **powerset** of a set $A$ (written $A$) is the set of all subsets of $A$, including the empty set. List the members of the following sets:

   a. $\{a, b, c\}$:
   b. $\{a\}$
   c. $\{\}$
   d.

3. ◑ Write a Python function to compute the powerset of an arbitrary set. Remember that you will have to use `frozenset` for this.

### 10.7.3 Tuples

We write $x_1, \ldots, x_n$ for the **ordered n-tuple** of objects $x_1, \ldots, x_n$, where $n \geq 0$. These are exactly the same as Python tuples. Two tuples are equal only if they have the same lengths, and the same objects in the same order.

```
>>> tup1 = ('a', 'b', 'c')
>>> tup2 = ('a', 'c', 'b')
>>> tup1 == tup2
False
```

A tuple with just 2 elements is called an **ordered pair**, with just three elements, an **ordered triple**, and so on.

Given two sets $A$ and $B$, we can form a set of ordered pairs by drawing the first member of the pair from $A$ and the second from $B$. The *Cartesian product* of $A$ and $B$, written $A \times B$, is the set of all such pairs. More generally, we have for any sets $S_1, \ldots, S_n$,

(11) $S_1 \times \ldots \times S_n = \{ \quad x_1, \ldots, x_n \quad \mid x_i \in S_i \}$

In Python, we can build Cartesian products using list comprehension. As you can see, the sets in a Cartesian product don't have to be distinct.

```
>>> A = set([1, 2, 3])
>>> B = set('ab')
>>> AxB = set([(a, b) for a in A for b in B])
>>> AxB
set([(1, 'b'), (3, 'b'), (3, 'a'), (2, 'a'), (2, 'b'), (1, 'a')])
>>> AxA = set([(a1, a2) for a1 in A for a2 in A])
>>> AxA
set([(1, 2), (3, 2), (1, 3), (3, 3), (3, 1), (2, 1),
(2, 3), (2, 2), (1, 1)])
```

### 10.7.4 Relations and Functions

In general, a **relation** $R$ is a set of tuples. For example, in set-theoretic terms, the binary relation *kiss* is the set of all ordered pairs $x, y$ such that *x kisses y*. More formally, an **n-ary relation** over sets $S_1$, $\ldots$, $S_n$ is any set $R \subseteq S_1 \times \ldots \times S_n$.

Given a binary relation $R$ over two sets $A$ and $B$, not everything in $A$ need stand in the $R$ relation to something in $B$. As an illustration, consider the set `evens` and the relation `mod` defined as follows:

```
>>> evens = set([2, 4, 6, 8, 10])
>>> mod = set([(m,n) for m in evens for n in evens if n % m == 0 and m < n])
>>> mod
set([(4, 8), (2, 8), (2, 6), (2, 4), (2, 10)])
```

Now, `mod` $\subseteq$ `evens` $\times$ `evens`, but there are elements of `evens`, namely `6`, `8` and `10`, which do not stand in the `mod` relation to anything else in `evens`. In this case, we say that only `2` and `4` are in the **domain** of the `mod` relation. More formally, for a relation $R$ over $A \times B$, we define

(12) $dom(R) = \{x \mid y. \quad x, y \quad \in A \times B\}$

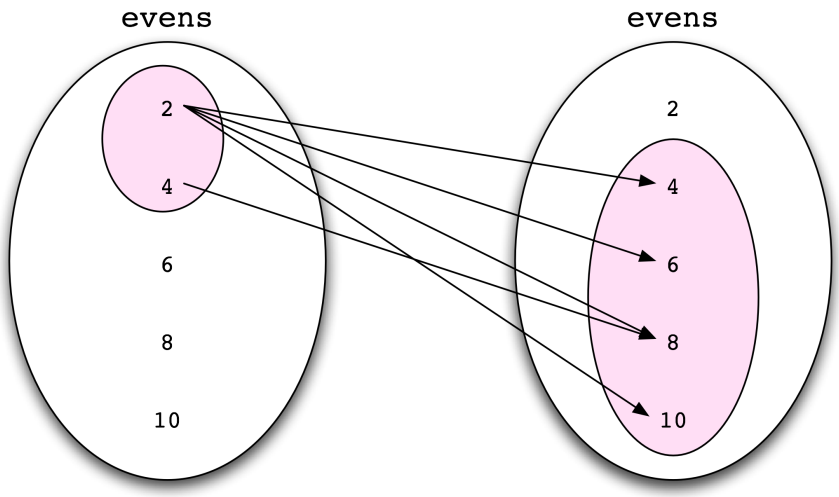


Figure 10.3: Visual Representation of a relation

Correspondingly, the set of entities in $B$ which are the second member of a pair in $R$ is called the **range** of $R$, written $ran(R)$.

We can visually represent the relation `mod` by drawing arrows to indicate elements that stand in the relation, as shown in Figure 10.3.

The domain and range of the relation are shown as shaded areas in Figure 10.3.

A relation $R \subseteq A \times B$ is a (set-theoretic) **function** just in case it meets the following two conditions:

1. For every $a \in A$ there is at most one $b \in B$ such that $\langle a, b \rangle$.
2. The domain of $R$ is equal to $A$.

Thus, the `mod` relation defined earlier is not a function, since the element `2` is paired with four items, not just one. By contrast, the relation `doubles` defined as follows *is* a function:

```
>>> odds = set([1, 2, 3, 4, 5])
>>> doubles = set([(m,n) for m in odds for n in evens if n == m * 2])
>>> doubles
set([(1, 2), (5, 10), (2, 4), (3, 6), (4, 8)])
```

If $f$ is a function $\subseteq A \times B$, then we also say that $f$ is a function from $A$ to $B$. We also write this as $f: A \rightarrow B$. If $\langle x, y \rangle \in f$, then we write $f(x) = y$. Here, $x$ is called an **argument** of $f$ and $y$ is a **value**. In such a case, we may also say that $f$ maps $x$ to $y$.

Given that functions always map a given argument to a single value, we can also represent them in Python using dictionaries (which incidentally are also known as **mapping** objects). The `update()` method on dictionaries can take as input any iterable of key/value pairs, including sets of two-membered tuples:

```
>>> d = {}
>>> d.update(doubles)
>>> d
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

A function $f\colon S_1 \times \ldots \times S_n \rightarrow T$ is called an **n-ary** function; we usually write $f(s_1, \ldots, s_n)$ rather than $f(\quad s_1, \ldots, s_n \quad)$. For sets $A$ and $B$, we write $A^B$ for the set of all functions from $A$ to $B$, that is $\{f \mid f : A \rightarrow B\}$. If $S$ is a set, then we can define a corresponding function $f_S$ called the **characteristic function** of $S$, defined as follows:

(13) $f_S(x) = \text{True}$ if $x \in S$
$f_S(x) = \text{False}$ if $x \quad S$

$f_S$ is a member of the set $\{True, False\}^S$.

It can happen that a relation meets condition (1) above but fails condition (2); such relations are called **partial functions**. For instance, let's slightly modify the definition of `doubles`:

```
>>> doubles2 = set([(m,n) for m in evens for n in evens if n == m * 2])
>>> doubles2
set([(2, 4), (4, 8)])
```

`doubles2` is a partial function since its domain is a proper subset of `evens`. In such a case, we say that `doubles2` is **defined** for `2` and `4` but **undefined** for the other elements in `evens`.

### 10.7.5 Exercises

1. ☼ Consider the relation `doubles`, where `evens` is defined as in the text earlier:

   ```
   >>> doubles = set([(m,m*2) for m in evens])
   ```

   Is `doubles` a relation over `evens`? Explain your answer.

2. ◑ What happens if you try to update a dictionary with a relation which is *not* a function?

3. ☼ Write a couple of Python functions which for any set of pairs $R$, return the domain and range of $R$.

4. ◑ Let $S$ be a family of three children, {Bart, Lisa, Maggie}. Define relations $R \subseteq S \times S$ such that:

   a. $dom(R) \subset S$;
   b. $dom(R) = S$;
   c. $ran(R) = S$;
   d. $ran(R) = S$;
   e. $R$ is a total function on $S$.
   f. $R$ is a partial function on $S$.

5. ◑ Write a Python function which for any set of pairs $R$, returns `True` if and only if $R$ is a function.

## 10.8 Further Reading

[Brent1995]
[Hunt & Thomas, 1999]