Ch4. Writing Structured Programs

CIS/STA 9665 (Sep 28)

Reminders:
Lab Report 4 Due Sep 29
Assignment 2 Due Sep 30
Assignment 3 Due Oct 7

Review Useful Content and Code from Last Class

- 1. Accessing text from the web
  - import nltk
  - from nltk import word_tokenize
  - from urllib import request
  - url="http://www.gutenberg.org/files/1125/1125.txt"
  - raw=request.urlopen(url).read().decode('utf8')
  - tokens=word_tokenize(raw)

Dealing with HTML

- url =http://news.bbc.co.uk/2/hi/health/2284783.stm
- html = request.urlopen(url).read().decode('utf8')
- from bs4 import BeautifulSoup
- raw = BeautifulSoup(html, 'html.parser').get_text()
- tokens = word_tokenize(raw)
- Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree

Reading local files

- raw=open('document.txt').read( )
- tokens=word_tokenize(raw)
- f=open('document.txt')
- for line in f:
- print(line.strip())

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Regular Expressions for Detecting Word Patterns

Operator	Behavior
`.`	Wildcard, matches any character
`^abc`	Matches some pattern *abc* at the start of a string
`abc$`	Matches some pattern *abc* at the end of a string
`[abc]`	Matches one of a set of characters
`[A-Z0-9]`	Matches one of a range of characters
`ed|ing|s`	Matches one of the specified strings (disjunction)
`*`	Zero or more of previous item, e.g. `a*`, `[a-z]*` (also known as *Kleene Closure*)
`+`	One or more of previous item, e.g. `a+`, `[a-z]+`
`?`	Zero or one of the previous item (i.e. optional), e.g. `a?`, `[a-z]?`
`{n}`	Exactly *n* repeats where n is a non-negative integer
`{n,}`	At least *n* repeats
`{,n}`	No more than *n* repeats
`{m,n}`	At least *m* and no more than *n* repeats
`a(b|c)+`	Parentheses that indicate the scope of the operators

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Regular Expressions for Detecting Word Patterns

- re.search('ed$', w)
  - Find words ending with "ed"
- re.search('^..j..t..$', w)
  - Find 8 letter word (the third letter is j and the sixth letter is t)
- re.search(^e-?mail$', w)
  - Find "email" or "e-mail"
- re.search('^[ghi][mno][jlk][def]$', w)]
  - ['gold', 'golf', 'hold', 'hole']
- re.search("^[g-o]+$",w)
  - Find out one or more letters from g h i j k l m n o

+ vs *

- re.search('^m+i+n+e+$', w)

```
['miiiiiiiiiiiinnnnnnnnnneeeeeeeee',
 'miiiiiinnnnnnnnneeeeeeee',
 'mine',
 'mmmmmmmmiiiiiiiinnnnnnneeeeeeee']
```

- re.search('^[ha]+$', w)

```
['a',
 'aaaaaaaaaaaaaaaa',
 'aaahhhh',
 'ah',
 'ahah',
 'ahahah',
 'ahh',
 'ahhahahaha',
```

 even Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

+ vs *

- re.search('^m*i*n*e*$', w)

```
['',
 'e',
 'i',
 'in',
 'm',
 'me',
 'meeeeeeeeeeee',
 'mi',
 'miiiiiiiiiiiinnnnnnnnnneeeeeeeee',
 'miiiiinnnnnnnnneeeeeee',
 'min',
 'mine',
 'mm',
 'mmm',
 'mmmm',
 'mmmmm',
 'mmmmmm',
 'mmmmmmmiiiiiiiinnnnnnnneeeeeeee',
 'mmmmmmmmm',
 'mmmmmmmmmmmmm',
 'mmmmmmmmmmmmmm',
 'n',
 'ne']
```

^matches any character other than a vowel.

- re.search('^[^aeiouAEIOU]+$',w)

- r'^[AEIOUaeiou]+|[AEIOUaeiou]+$|[^AEIOUaeiou]'
  - Match with initial vowel characters, final vowel sequences, and all consonants
  - The three way disjunction is processed left to right

\

- re.search('^[0-9]+\.[0-9]+$', w)
- re.search('^[A-Z]+\$$', w)
- re.search('^[0-9]{4}$,w)
- research(^[0-9]+-[a-z]{3,5}$,w)
- re.search('^[a-z]{5,}-[a-z]{2,3}-[a-z]{,6}$', w)
- re.search('(ed|ing)$', w)

Student's Questions in Last Class

- re.search('..j..t..',w)
  - re.search( ) checks for a match anywhere in the string
  - search the substring contains ..j..t.. (8-letter substring)

```
print(re.search('..j..t..',"aaaajjktaah"))
```

```
<re.Match object; span=(2, 10), match='aajjktaa'>
```

```
[w for w in wordlist if re.search('..j..t..',w)]
```

```
'preadjectival',
'preadjective',
'preadjustable',
'preadjustment',
'preconjecture',
'prejustification',
'prejustify',
'preobjection',
'preobjective',
'prerejection',
'presubjection',
'projectable',
```

- Why do not escape hypen?
  - Outside of character classes (that's what the "square brackets" are called (e.g.[a-z]), the hyphen has no special meaning, it is conventional not to escape hyphen

NLTK Stemmer

- porter = nltk.PorterStemmer()
- lancaster = nltk.LancasterStemmer()
- [porter.stem(t) for t in tokens]
- [lancaster.stem(t) for t in tokens]

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Lemmatization

- The WordNet lemmatizer only removes affixes if the resulting word is in its dictionary.
- This additional checking process makes the lemmatizer slower than the above stemmers.
- Notice that it doesn't handle lying, but it converts women to woman.
- The WordNet lemmatizer is a good choice if you want to compile the vocabulary of some texts and want a list of valid lemmas (or lexicon headwords).


```
wnl = nltk.WordNetLemmatizer()
```

```
[wnl.lemmatize(t) for t in tokens]
```

```
['DENNIS',
 ':',
 'Listen',
 ',',
 'strange',
 'woman',
 'lying',
 'in',
 'pond',
 'distributing',
 'sword',
 'is',
 'no',
 'basis',
 'for',
 'a',
 'system',
 'of',
 'government',
 '.',
 'Supreme',
 'executive',
 'power',
 'derives',
 'from',
 'a',
 'mandate',
 'from',
 'the',
 'mass',
 ',',
 'not',
 'from',
 'some',
 'farcical',
 'aquatic',
 'ceremony',
 '.']
```

Strings and Formats

- The curly brackets '{}' mark the presence of a replacement field: this acts as a placeholder for the string values of objects that are passed to the str.format() method.
- We can embed occurrences of '{}' inside a string, then replace them with strings by calling format() with appropriate arguments.
- A string containing replacement fields is called a format string.

```
'{}->{};'.format ('cat', 3)
```

```
'cat->3;'
```

```
'{}'.format(3)
```

```
'3'
```

```
'I want a {} right now'.format('coffee')
```

```
'I want a coffee right now'
```

```
# We can have any number of placeholders, but the str.format method must be cal
```

```
'{} wants a {} {}'.format ('Lee', 'sandwich', 'for lunch')
```

```
'Lee wants a sandwich for lunch'
```

```
'{} wants a {} {}'.format ('sandwich', 'for lunch')
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-377-339b7c04a9c2> in <module>
----> 1 '{} wants a {} {}'.format ('sandwich', 'for lunch')

IndexError: tuple index out of range
```

- template = 'Lee wants a {} right now’
- menu = ['sandwich', 'spam fritter', 'pancake’]
- for snack in menu:
- print(template.format(snack))

```
Lee wants a sandwich right now
Lee wants a spam fritter right now
Lee wants a pancake right now
```

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Lining Things Up

- It is right-justified by default for numbers, but we can precede the width specifier with a '<' alignment option to make numbers left-justified

```
'{:6}'.format(41)
```

```
'    41'
```

```
'{:<6}'.format(41)
```

```
'41    '
```

- Strings are left-justified by default, but can be right-justified with the '>' alignment option.

```
'{:6}'.format('dog')
```

```
'dog   '
```

```
'{:>6}'.format('dog')
```

```
'   dog'
```

{:.4f}

- Other control characters can be used to specify the sign and precision of floating point numbers; for example {:.4f} indicates that four digits should be displayed after the decimal point for a floating point number.

```
import math
```

```
'{:.4f}'.format(math.pi)
```

```
'3.1416'
```

{:.4%}

```
count, total = 3205, 9375
```

```
"accuracy for {} words: {:.4%}".format(total, count / total)
```

```
'accuracy for 9375 words: 34.1867%'
```

{:{width}}

```
'{:{width}}'.format('Monty Python', width=15)
```

```
'Monty Python    '
```

Writing Results to a File

```
output_file = open('output.txt', 'w')
```

```
words = set(nltk.corpus.genesis.words('english-kjv.txt'))
```

```
for word in sorted(words):
    print(word, file=output_file)
```

Objective

- How can you write well-structured, readable programs that you and others will be able to re-use easily?
- How do the fundamental building blocks work, such as loops, functions and assignment?
- What are some of the pitfalls with Python programming and how can you avoid them?

TF-IDF

- Short for *term frequency–inverse document frequency*, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus-wiki
- **Term Frequency**
- Term frequency (TF) is how often a word appears in a document, divided by how many words there are.
- TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document)

Cite: https://towardsdatascience.com/text-summarization-using-tf-idf-e64a0644ace3

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

TF-IDF

- **Inverse document frequency**
- Term frequency is how common a word is, inverse document frequency (IDF) is how unique or rare a word is.
- IDF(t) = Log_e(Total number of documents / Number of documents with term t in it)
- TF-IDF=TF(t)*IDF(t)

Cite: https://towardsdatascience.com/text-summarization-using-tf-idf-e64a0644ace3

TF-IDF

- Consider a document containing 100 words wherein the word ***apple*** appears 5 times. The term frequency (i.e., TF) for ***apple*** is then (5 / 100) = 0.05.
- Now, assume we have 10 million documents and the word *apple* appears in one thousand of these. Then, the inverse document frequency (i.e., IDF) is calculated as log(10,000,000 / 1,000) = 4.
- Thus, the TF-IDF weight is the product of these quantities: 0.05 * 4 = 0.20.

Cite: https://towardsdatascience.com/text-summarization-using-tf-idf-e64a0644ace3

Text Summarization Using TF-IDF

- Jupyter Notebook

We calculate the frequency of words in each sentence

	Sentence/Document 1	Sentence /Document 2	Sentence /Document 3
along	1		
program	1		1
word3	3		
..			
..			
..wordn-1	4		
wordn	1	2	2

Freq_table={word:count)

Freq_matrix={sent: {word:count}}

Calculate TermFrequency and generate a matrix

	Sentence/Document 1	Sentence /Document 2	Sentence /Document 3
along	1/19: tf: 0.0526315		
program	1/19: tf:0.0526315		1
word3	3		
..			
..			
..wordn-1	4		
wordn	1	2	2

tf_table={word:count/count_words_in_sentence)

tf_matrix={sent: {tf_table}}

Create a table for documents per words

	Sentence/Document 1	Sentence /Document 2	Sentence /Document 3
along	1 sentence contain the word “along” IDF=Log 10(3/1)=0.477	1	2
Word2:program	2 sentences contain the word “program” IDF=Log 10(3/2)=0.176	0	1
word3	3		
..			
..			
..wordn-1	4		
wordn	1	2	2

f_table={word:count)

word_per_doc_table={word: count_per_doc} program: 2

Create a table for TF*IDF for each word

	Sentence/Document 1	Sentence /Document 2	Sentence /Document 3
along	1 sentence contain the word “along” IDF=Log 10(3/1)=0.477 TF=0.0526315 IDF*TF=0.0251	1	2
Word2:program	2 sentences contain the word “program” IDF=Log 10(3/2)=0.176 TF=0.0526315 TF*IDF=0.00926	0	1
word3	3		
..			
..			
..wordn-1	4		
wordn	1	2	2

Score the sentences: **sum of Tf-IDF score of words in a sentence/# of words in sente**

	Sentence/Document 1	Sentence /Document 2	Sentence /Document 3
along	1 sentence contain the word “along” IDF=Log 10(3/1)=0.477 TF=0.0526315 IDF*TF=0.0251	1	2
Word2:program	2 sentences contain the word “program” IDF=Log 10(3/2)=0.176 TF=0.0526315 IF*IDF=0.00926	0	1
word3	3		
..			
score	0.0297	0.0312	0.02394

Threshold=(0.0297+0.0312+0.02394)/3=0.028

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Outline

- 1. Sequences
- 2. Questions of Style
- 3. Functions: The Foundation of Structured Programming
- 4. Doing More with Function
- 5. A Sample of Python Libraries

4.1 Sequences

So far, we have seen two kinds of sequence object: strings and lists. Another kind of sequence is called a tuple. Tuples are formed with the comma operator [1], and typically enclosed using parentheses. We've actually seen them in the previous chapters, and sometimes referred to them as "pairs", since there were always two members. However, tuples can have any number of members. Like lists and strings, tuples can be **indexed** [2] and **sliced** [3], and have **a length**

```
t = 'walk', 'fem', 3
```

```
t
```

```
('walk', 'fem', 3)
```

```
t[0]
```

```
'walk'
```

```
t[1:]
```

```
('fem', 3)
```

```
len(t)
```

```
3
```

Let's compare strings, lists and tuples directly, and do the indexing, slice, and length operation on each type:

```
raw = 'I turned off the spectroroute'
```

```
text = ['I', 'turned', 'off', 'the', 'spectroroute']
```

```
pair = (6, 'turned')
```

```
raw[2], text[3], pair[1]
```

```
('t', 'the', 'turned')
```

```
raw[-3:], text[-3:], pair[-2:]
```

```
('ute', ['off', 'the', 'spectroroute'], (6, 'turned'))
```

```
len(raw), len(text), len(pair)
```

```
(29, 5, 2)
```

Operating on Sequence Types

We can iterate over the items in a sequence s in a variety of useful ways, as shown in 4.1.

Table 4.1:

Various ways to iterate over sequences

Python Expression	Comment
`for item in s`	iterate over the items of `s`
`for item in sorted(s)`	iterate over the items of `s` in order
`for item in set(s)`	iterate over unique elements of `s`
`for item in reversed(s)`	iterate over elements of `s` in reverse
`for item in set(s).difference(t)`	iterate over elements of `s` not in `t`

Some other objects, such as a FreqDist, can be converted into a sequence (using list() or sorted()) and support iteration, e.g.

```
import nltk
```

```
raw = 'Red lorry, yellow lorry, red lorry, yellow lorry.'
```

```
text = nltk.word_tokenize(raw)
```

```
fdist = nltk.FreqDist(text)
```

```
sorted(fdist)
```

```
[',', '.', 'Red', 'lorry', 'red', 'yellow']
```

```
for key in fdist:
    print(key + ':', fdist[key], end='; ')
```

```
Red: 1; lorry: 4; ,: 3; yellow: 2; red: 1; .: 1;
```

In the next example, we use tuples to re-arrange the contents of our list. (We can omit the parentheses because the comma has higher precedence than assignment.)

```
words = ['I', 'turned', 'off', 'the', 'spectroroute']
```

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

```
words
```

```
['I', 'turned', 'the', 'spectroroute', 'off']
```

```
(words[2], words[3], words[4]) = (words[3], words[4], words[2])
```

```
words
```

```
['I', 'turned', 'spectroroute', 'off', 'the']
```

This is an idiomatic and readable way to move items inside a list. It is equivalent to the following traditional way of doing such tasks that does not use tuples (notice that this method needs a temporary variable tmp).

```
tmp = words[2]
```

```
words[2] = words[3]
```

```
words[3] = words[4]
```

```
words[4] = tmp
```

Exercise 1

```
# Exercise 1: Use tuples to rearrange the contents of a list (from: Ex=["we","take","into","account","this","fact"])
```

```
Ex=["we","take","into","account","this","fact"]
```

```
Ex[2],Ex[3],Ex[4],Ex[5] =Ex[4],Ex[5],Ex[2],Ex[3]
```

```
Ex
```

```
['we', 'take', 'this', 'fact', 'into', 'account']
```

As we have seen, Python has sequence functions such as sorted() and reversed() that rearrange the items of a sequence. There are also functions that modify the structure of a sequence and which can be handy for language processing. Thus, zip() takes the items of two or more sequences and "zips" them together into a single list of tuples. Given a sequence s, enumerate(s) returns pairs consisting of an index and the item at that index.

```
words = ['I', 'turned', 'off', 'the', 'spectroroute']
```

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

```
zip(words, tags)
```

```
<zip at 0xb314e18>
```

```
list(zip(words, tags))
```

```
[('I', 'noun'),
 ('turned', 'verb'),
 ('off', 'prep'),
 ('the', 'det'),
 ('spectroroute', 'noun')]
```

```
list(enumerate(words))
```

```
[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the'), (4, 'spectroroute')]
```

For some NLP tasks it is necessary to cut up a sequence into two or more parts. For instance, we might want to "train" a system on 90% of the data and test it on the remaining 10%. To do this we decide the location where we want to cut the data [1], then cut the sequence at that location [2].

```
text = nltk.corpus.nps_chat.words()
```

```
cut = int(0.9 * len(text))
```

```
training_data, test_data = text[:cut], text[cut:]
```

```
text == training_data + test_data
```

```
True
```

```
len(training_data) / len(test_data)
```

```
9.0
```

Exercise 2

```
# Exercise 2. Please divide names corpors into two parts. (95% of the data is used to "train" the model and 5% of data
```

```
import nltk
```

```
text = nltk.corpus.names.words()
```

```
cut = int(0.95 * len(text))
```

```
training_data, test_data = text[:cut], text[cut:]
```

Combining Different Sequence Types

- We began by talking about the commonalities in these sequence types, but the above code illustrates important differences in their roles.
- First, strings appear at the beginning and the end: this is typical in the context where our program is reading in some text and producing output for us to read.
- Lists and tuples are used in the middle, but for different purposes.
- A **list** is typically a sequence of **objects all having the same type**, of arbitrary length. We often use lists to hold sequences of words.
- In contrast, **a tuple** is typically a collection of **objects of different types**, of fixed length. **We often use a tuple to hold a record, a collection of different fields relating to some entity**.
- This distinction between the use of lists and tuples takes some getting used to, so here is another example:

tuples vs lists

- A good way to decide when to use tuples vs lists is to ask whether the interpretation of an item depends on its position.
- For example, a tagged token combines two strings having different interpretation, and we choose to interpret the first item as the token and the second item as the tag.
- Thus we use tuples like this: ('grail', 'noun'); a tuple of the form ('noun', 'grail') would be nonsensical since it would be a word noun tagged grail.
- In contrast, the elements of a text are all tokens, and position is not significant.
- Thus we use lists like this: ['venetian', 'blind']; a list of the form ['blind', 'venetian'] would be equally valid. The linguistic meaning of the words might be different, but the interpretation of list items as tokens is unchanged.

Tuples vs Lists

- The distinction between lists and tuples has been described in terms of usage.
- However, there is a more fundamental difference: in Python, lists are **mutable**, while **tuples are immutable**.
- In other words, here are some of the operations on lists that do in-place modification of the list.

Combining Different Sequence Types

- Let's combine our knowledge of these three sequence types, together with list comprehensions, to perform the task of sorting the words in a string by their length.

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

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

```
wordlens.sort()
```

```
' '.join(w for (_, w) in wordlens)
```

```
'I off the turned spectroroute'
```

Generator Expressions

- We've been making heavy use of list comprehensions, for compact and readable processing of texts. Here's an example where we tokenize and normalize a text:

```
text = '''"When I use a word," Humpty Dumpty said in rather a scornful tone,
  "it means just what I choose it to mean - neither more nor less."'''
```

```
[w.lower() for w in nltk.word_tokenize(text)]
```

```
['``',
 'when',
 'i',
 'use',
 'a',
 'word',
 ',',
 "''",
 'humpty',
 'dumpty',
 'said',
 'in',
 'rather',
 'a',
 'scornful',
 'tone',
 ',',
 '``',
 'it',
 'means',
 'just',
 'what',
 'i',
 'choose',
 'it',
 'to',
 'mean',
 '-',
 'neither',
 'more',
 'nor',
 'less',
 '.',
 "''"]
```

Suppose we now want to process these words further. We can do this by inserting the above expression inside a call to some other function , but Python allows us to omit the brackets

```
max([w.lower() for w in nltk.word_tokenize(text)])
```

```
'word'
```

```
max(w.lower() for w in nltk.word_tokenize(text))
```

```
'word'
```

The second line uses a generator expression. This is more than a notational convenience: in many language processing situations, generator expressions will be more efficient.

In [1], storage for the list object must be allocated before the value of **max()** is computed. If the text is very large, this could be slow. In [2], the data is streamed to the calling function. Since the calling function simply has to find the maximum value — the word which comes latest in lexicographic sort order — it can process the stream of data without having to store anything more than the maximum value seen so far.

4.2 Questions of Style

- Programming is as much an art as a science.
- Here we pick up on some issues of programming style that have important ramifications for the readability of your code, including
- code layout, procedural vs declarative style, and the use of loop variables.

Python Coding Style

- When writing programs you make many subtle choices about names, spacing, comments, and so on.
- When you look at code written by other people, needless differences in style make it harder to interpret the code.
- Therefore, the designers of the Python language have published a style guide for Python code, available at http://www.python.org/dev/peps/pep-0008/.
- The underlying value presented in the style guide is consistency, for the purpose of maximizing the readability of code.
- We briefly review some of its key recommendations here, and refer readers to the full guide for detailed discussion with examples.

python.org/dev/peps/pep-0008/

Python | PSF | Docs | PyPI | Jobs | Community

python™

Donate | Search | GO | Socialize

About | Downloads | Documentation | Community | Success Stories | News | Events

Tweets by @ThePSF

Python Software Founda @ThePSF

The @ThePSF 2020 board election nominations are now all available for our voting members to review: python.org/nominations/el…. Ballots will be sent out June 8th.

23h

Python Software Founda @ThePSF

PSF Election Update: The nomination page showed "closed" for 4 hours &

Embed | View on Twitter

The PSF

The Python Software Foundation is the organization behind Python. Become a member of the PSF and help advance the software and our mission.

Python »» Python Developer's Guide »» PEP Index »» PEP 8 -- Style Guide for Python Code

PEP 8 -- Style Guide for Python Code

PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

Contents

 ia, 978-0-596-51649-9

Python Coding Style

- Code layout should use **four spaces per indentation level**. You should make sure that when you write Python code in a file, you avoid tabs for indentation, since these can be misinterpreted by different text editors and the indentation can be messed up.
- Lines should be less than 80 characters long; if necessary you can **break a line inside parentheses, brackets, or braces**, because Python is able to detect that the line continues over to the next line.
- If you need to **break a line outside parentheses, brackets, or braces, you can often add extra parentheses**, and you can always add a **backslash** at the end of the line that is broken:

```
if (len(syllables) > 4 and len(syllables[2]) == 3 and
   syllables[2][2] in [aeiou] and syllables[2][3] == syllables[1][3]):
    process(syllables)
if len(syllables) > 4 and len(syllables[2]) == 3 and \
   syllables[2][2] in [aeiou] and syllables[2][3] == syllables[1][3]:
    process(syllables)
```

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Procedural vs Declarative 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:

```
tokens = nltk.corpus.brown.words(categories='news')
```

```
count = 0
```

```
total=0
```

```
for token in tokens:
    count+=1
    total+=len(token)
```

```
total/count
```

```
4.401545438271973
```

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 that are meaningless until the end. We say that this program is written in a **procedural** style, dictating the **machine operations step by step**. Now consider the following program that computes the same thing:

```
total = sum(len(t) for t in tokens)
```

```
print(total / len(tokens))
```

```
4.401545438271973
```

Another case where a loop variable seems to be necessary is for printing a counter with each line of output. Instead, we can use **enumerate(),** which processes a sequence s and produces a tuple of the form (i, s[i]) for each item in s, starting with (0, s[0]). Here we enumerate the key-value pairs of the frequency distribution, resulting in nested tuples (rank, (word, count)). We print rank+1 so that the counting appears to start from 1, as required when producing a list of ranked items.

```
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("%3d %6.2f%% %s" % (rank + 1, cumulative * 100, word))
        if cumulative > 0.25:
            break
```

```
  1   5.40% the
  2  10.42% ,
  3  14.67% .
  4  17.78% of
  5  20.19% and
  6  22.40% to
  7  24.29% a
  8  25.97% in
```

```
# Exercise 3: Use enumerate () fuction to rank the culmulatve frequency (30%) of the most commmon words in web text corpus
```

```
fd = nltk.FreqDist(nltk.corpus.webtext.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("%3d %6.2f%% %s" % (rank + 1, cumulative * 100, word))
        if cumulative > 0.3:
            break
```

```
  1   4.46% .
  2   8.02% :
  3  11.12% ,
  4  13.83% '
  5  15.80% I
  6  17.65% the
  7  19.22% to
  8  20.67% a
  9  21.81% you
 10  22.90% ?
 11  23.96% in
 12  24.99% and
 13  26.01% !
 14  26.95% #
 15  27.83% t
 16  28.66% -
 17  29.48% s
 18  30.30% on
```

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

It's sometimes tempting to use loop variables to store a maximum or minimum value seen so far. Let's use this method to find the longest word in a text.

```
text = nltk.corpus.gutenberg.words('milton-paradise.txt')
```

```
longest = ''
```

```
for word in text:
    if len(word) > len(longest):
        longest = word
```

```
longest
```

```
'unextinguishable'
```

- However, a more transparent solution uses two list comprehensions, both having forms that should be familiar by now:

```
maxlen = max(len(word) for word in text)
```

```
[word for word in text if len(word) == maxlen]
```

```
['unextinguishable',
 'transubstantiate',
 'inextinguishable',
 'incomprehensible']
```

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Exercise 4

```
# Exercise 4: Use a list comprehension to find out the longest words in "shakespeare-macbeth.txt" in gutenberg corpus.
```

```
text = nltk.corpus.gutenberg.words('shakespeare-macbeth.txt')
```

```
maxlen = max(len(word) for word in text)
```

```
[word for word in text if len(word) == maxlen]
```

Some Legitimate Uses for Counters

- There are cases where we still want to use loop variables in a list comprehension. For example, we need to use a loop variable to extract successive overlapping n-grams from a list:

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

```
n = 3
```

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

```
[['The', 'dog', 'gave'],
 ['dog', 'gave', 'John'],
 ['gave', 'John', 'the'],
 ['John', 'the', 'newspaper']]
```

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

```
: list(nltk.bigrams(sent))
```

```
: [('The', 'dog'),
   ('dog', 'gave'),
   ('gave', 'John'),
   ('John', 'the'),
   ('the', 'newspaper')]
```

```
: list(nltk.trigrams(sent))
```

```
: [('The', 'dog', 'gave'),
   ('dog', 'gave', 'John'),
   ('gave', 'John', 'the'),
   ('John', 'the', 'newspaper')]
```

```
: list(nltk.ngrams(sent,4))
```

```
: [('The', 'dog', 'gave', 'John'),
   ('dog', 'gave', 'John', 'the'),
   ('gave', 'John', 'the', 'newspaper')]
```

4.3 Functions: The Foundation of Structured Programming

Functions

- We have seen that functions help to make our work reusable and readable. They also help make it reliable. When we re-use code that has already been developed and tested, we can be more confident that it handles a variety of cases correctly. We also remove the risk that we forget some important step, or introduce a bug. The program that calls our function also has increased reliability. The author of that program is dealing with a shorter program, and its components behave transparently.

- To summarize, as its name suggests, a function captures functionality. It is a segment of code that can be given a meaningful name and which performs a well-defined task. Functions allow us to abstract away from the details, to see a bigger picture, and to program more effectively.

- The rest of this section takes a closer look at functions, exploring the mechanics and discussing ways to make your programs easier to read.

Function Inputs and Outputs

- We pass information to functions using a function's parameters, the parenthesized list of variables and constants following the function's name in the function definition. Here's a complete example:

```
def repeat(msg, num):
    return ' '.join([msg] * num)
```

```
monty = 'Monty Python'
```

```
repeat(monty, 3)
```

```
'Monty Python Monty Python Monty Python'
```

- We first define the function to take two parameters, msg and num . Then we call the function and pass it two arguments, monty and 3; these arguments fill the "placeholders" provided by the parameters and provide values for the occurrences of msg and num in the function body.

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

It is not necessary to have any parameters, as we see in the following example:

```
def monty():
    return "Monty Python"
```

```
monty()
```

```
'Monty Python'
```

A function usually communicates its results back to the calling program via the return statement, as we have just seen. To the calling program, it looks as if the function call had been replaced with the function's result, e.g.:

```
repeat(monty(), 3)
```

```
'Monty Python Monty Python Monty Python'
```

```
repeat('Monty Python', 3)
```

```
'Monty Python Monty Python Monty Python'
```

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

A Python function is not required to have a return statement. Some functions do their work as a side effect, printing a result, modifying a file, or updating the contents of a parameter to the function (such functions are called "procedures" in some other programming languages).

Consider the following three sort functions. The third one is dangerous because a programmer could use it without realizing that it had modified its input. In general, functions should modify the contents of a parameter (my_sort1()), or return a value (my_sort2()), not both (my_sort3()).

```
# good: modifies its argument, no return value

def my_sort1(mylist):
    mylist.sort()

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

def my_sort2(mylist):
    return sorted(mylist)

# bad: modifies its argument and also returns it

def my_sort3(mylist):
    mylist.sort()
    return mylist
```

Checking Parameter Types

- Python does not allow us to declare the type of a variable when we write a program, and this permits us to define functions that are flexible about the type of their arguments. For example, a tagger might expect a sequence of words, but it wouldn't care whether this sequence is expressed as a list or a tuple (or an iterator, another sequence type that is outside the scope of the current discussion).

- However, often we want to write programs for later use by others, and want to program in a defensive style, providing useful warnings when functions have not been invoked correctly. The author of the following tag() function assumed that its argument would always be a string.

```
: def tag(word):
      if word in ['a', 'the', 'all']:
          return 'det'
      else:
          return 'noun'

: tag('the')

: 'det'

: tag('knight')

: 'noun'

: tag(["'Tis", 'but', 'a', 'scratch'])

: 'noun'
```

The function returns sensible values for the arguments 'the' and 'knight', but look what happens when it is passed a list [1] — it fails to complain, even though the result which it returns is clearly incorrect. The author of this function could take some extra steps to ensure that the word parameter of the tag() function is a string. A naive approach would be to check the type of the argument using if not type(word) is str, and if word is not a string, to simply return Python's special empty value, None. This is a slight improvement, because the function is checking the type of the argument, and trying to return a "special", diagnostic value for the wrong input. However, it is also dangerous because the calling program may not detect that None is intended as a "special" value, and this diagnostic return value may then be propagated to other parts of the program with unpredictable consequences. This approach also fails if the word is a Unicode string, which has type unicode, not str. Here's a better solution, using an **assert statement together with Python's basestring type that generalizes over both unicode and str.**

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

```
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'
```

```
tag(["'Tis", 'but', 'a', 'scratch'])
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-53-cf5435388001> in <module>
----> 1 tag(["'Tis", 'but', 'a', 'scratch'])

<ipython-input-52-7a76f9a921cf> in tag(word)
      1 def tag(word):
----> 2     assert isinstance(word, str),"argument to tag() must be a string"
      3     if word in ['a', 'the', 'all']:
      4         return 'det'
      5     else:

AssertionError: argument to tag() must be a string
```

If the assert statement fails, it will produce an error that cannot be ignored, since it halts program execution. Additionally, the error message is easy to interpret. Adding assertions to a program helps you find logical errors, and is a kind of defensive programming. A more fundamental approach is to document the parameters to each function using docstrings as described later in this section.

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

Functional Decomposition

- Well-structured programs usually make extensive use of functions. When a block of program code grows longer than 10-20 lines, it is a great help to readability if the code is broken up into one or more functions, each one having a clear purpose. This is analogous to the way a good essay is divided into paragraphs, each expressing one main idea.

- Functions provide an important kind of abstraction. They allow us to group multiple actions into a single, complex action, and associate a name with it. (Compare this with the way we combine the actions of go and bring back into a single more complex action fetch.) When we use functions, the main program can be written at a higher level of abstraction, making its structure transparent, e.g.

Appropriate use of functions makes programs more readable and maintainable. Additionally, it becomes possible to reimplement a function — replacing the function's body with more efficient code — without having to be concerned with the rest of the program.

Consider the freq_words function in 4.3. It updates the contents of a frequency distribution that is passed in as a parameter, and it also prints a list of the n most frequent words.

```
import nltk
```

```
from urllib import request
from bs4 import BeautifulSoup

def freq_words(url, freqdist, n):
    html = request.urlopen(url).read().decode('utf8')
    raw = BeautifulSoup(html, 'html.parser').get_text()
    for word in nltk.word_tokenize(raw):
        freqdist[word.lower()] += 1
    result = []
    for word, count in freqdist.most_common(n):
        result = result + [word]
    print(result)
```

```
constitution = "http://www.archives.gov/exhibits/charters/constitution_transcrip
```

```
fd = nltk.FreqDist()
```

```
freq_words(constitution, fd, 30)
```

```
["'", ',', ':', ':1', 'the', ';', '(', ')', '``', '{', '}', 'of', '?', 'url',
'https', '@', 'import', 'qbiyg6', "'", 'archives', '#', 'and', '.', '[', ']',
'national', 'a', 'documents', 'founding', 'to']
```

This function has a number of problems. The function has two side-effects: it modifies the contents of its second parameter, and it prints a selection of the results it has computed. The function would be easier to understand and to reuse elsewhere if we initialize the FreqDist() object inside the function (in the same place it is populated), and if we moved the selection and display of results to the calling program. Given that its task is to identify frequent words, it should probably just return a list, not the whole frequency distribution. In 4.4 we refactor this function, and simplify its interface by dropping the freqdist parameter.

```
from urllib import request
from bs4 import BeautifulSoup

def freq_words(url, n):
    html = request.urlopen(url).read().decode('utf8')
    text = BeautifulSoup(html, 'html.parser').get_text()
    freqdist = nltk.FreqDist(word.lower() for word in nltk.word_tokenize(text))
    return [word for (word, _) in fd.most_common(n)]
```

```
freq_words(constitution, 30)
```

```
["'",
 ',',
 ':',
 ':1',
 'the',
 ';',
 '(',
 ')',
 '`',
 '{',
 '}',
 'of',
 '?',
 'url',
 'https',
 '@',
 'import',
 'qbiyg6',
 "'",
 'archives',
 '#',
 'and',
 '.',
 '[',
 ']',
 'national',
 'a',
 'documents',
```

Exercise 5

```
# Exercise 5: Choose your own webpage( in html format) and use the above code to output the 20 most common words in this web
```

```
from urllib import request
from bs4 import BeautifulSoup
from nltk.corpus import stopwords

def freq_words(url, n):
    html = request.urlopen(url).read().decode('utf8')
    text = BeautifulSoup(html, 'html.parser').get_text()
    fd = nltk.FreqDist(word.lower() for word in nltk.word_tokenize(text) if word.isalpha() and word.lower() not in stopwords
    return [word for (word, _) in fd.most_common(n)]
```

```
news="https://www.today.com/health/healthiest-places-u-s-2020-ranked-u-s-news-world-t192117"
```

```
freq_words(news,20)
```

```
['https',
 'null',
 'primary',
 'url',
 'id',
 'type',
 'width',
 'height',
 'name',
 'http',
 'false',
 'format',
 'headline',
 'assettype',
 'publicurl',
 'assetduration',
```

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9

4.4 Doing More with Functions

Functions as Arguments

- So far the arguments we have passed into functions have been simple objects like strings, or structured objects like lists. Python also lets us pass a function as an argument to another function. Now we can abstract out the operation, and apply a different operation on the same data. As the following examples show, we can pass the built-in function len() or a user-defined function last_letter() as arguments to another function:

```
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]
```

```
extract_property(len)
```

```
[4, 4, 2, 3, 5, 1, 3, 3, 6, 4, 4, 4, 2, 10, 1]
```

```
def last_letter(word):
    return word[-1]
```

```
extract_property(last_letter)
```

```
['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

The objects len and last_letter can be passed around like lists and dictionaries. Notice that parentheses are only used after a function name if we are invoking the function; when we are simply treating the function as an object these are omitted.

Python provides us with one more way to define functions as arguments to other functions, so-called lambda expressions. Supposing there was no need to use the above last_letter() function in multiple places, and thus no need to give it a name. We can equivalently write the following:

```
extract_property(lambda w: w[-1])
```

```
['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

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). A standard way to do this is to initialize an empty list, accumulate the material, then return the list, as shown in function search1() in 4.6

```
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
```

```
for item in search1('zz', nltk.corpus.brown.words()):
    print(item, end=" ")
```

```
Grizzlies' fizzled Rizzuto huzzahs dazzler jazz Pezza Pezza Pezza embezzling embezzlement pizza jaz
z Ozzie nozzle drizzly puzzle puzzle dazzling Sizzling guzzle puzzles dazzling jazz jazz Jazz jazz
Jazz jazz jazz Jazz jazz jazz jazz Jazz jazz dizzy jazz Jazz puzzler jazz jazzmen jazz jazz Jazz Ja
zz Jazz jazz Jazz jazz jazz jazz Jazz jazz jazz jazz jazz jazz jazz jazz jazz Jazz Jazz jazz j
azz nozzles nozzle puzzle buzz puzzle blizzard blizzard sizzling puzzled puzzle puzzle muzzle muzzl
e muezzin blizzard Neo-Jazz jazz muzzle piazzas puzzles puzzles embezzle buzzed snazzy buzzes puzzl
ed puzzled muzzle whizzing jazz Belshazzar Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie
Lizzie's Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie blizzard blizzards blizzard
blizzard fuzzy Lazzeri Piazza piazza palazzi Piazza Piazza Palazzo Palazzo Palazzo Piazza Piazza Pa
lazzo palazzo palazzo Palazzo Palazzo Piazza piazza piazza piazza Piazza Piazza Palazzo palazzo Pia
zza piazza pizza Piazza Palazzo palazzo dazzling puzzling Wozzek dazzling dazzling buzzing Jazz jaz
z Jazz Jazz jazz jazz jazz jazz Jazz jazz jazz jazz Fuzzy Lizzy Lizzy jazz fuzzy puzzles puzzling p
uzzling dazzle puzzle dazzling puzzled jazz jazz jazz jazzy whizzed frazzled quizzical puzzling poe
try-and-jazz poetry-and-jazz jazz jazz jazz jazz jazz jazz jazz Jazz jazz jazz jazz poetry-and-jazz
jazz jazz jazz Dizzy jazz jazz jazz jazz jazz poetry-and-jazz jazz jazz jazz jazz jazz jazz jazz ja
zz jazz jazz jazz jazz dazzled bedazzlement bedazzled Piazzo nozzles nozzles buzzing dazzles dizzy
puzzling puzzling puzzling puzzle muzzle puzzled nozzle Pozzatti Pozzatti Pozzatti puzzled Pozzatti
Pozzatti dazzling pizzicato Jazz jazz jazz jazz jazz nozzle grizzled fuzzy muzzle puzzled puzzle mu
zzle blizzard buzz dizzily drizzle drizzle drizzle sizzled puzzled puzzled puzzled fuzzed buzz buzz
buzz buzz-buzz-buzz buzzes fuzzy frizzled drizzle drizzle drizzling drizzling fuzz jazz jazz fuzz p
```

```
for item in search2('zz', nltk.corpus.brown.words()):
    print(item, end=" ")
```

```
Grizzlies' fizzled Rizzuto huzzahs dazzler jazz Pezza Pezza Pezza embezzling embezzlement pizza jazz
Ozzie nozzle drizzly puzzle puzzle dazzling Sizzling guzzle puzzles dazzling jazz jazz Jazz jazz Jazz
jazz jazz Jazz jazz jazz jazz Jazz jazz dizzy jazz Jazz puzzler jazz jazzmen jazz jazz Jazz Jazz Jazz
jazz Jazz jazz jazz jazz Jazz jazz jazz jazz jazz jazz jazz jazz jazz jazz Jazz Jazz jazz jazz nozzle
s nozzle puzzle buzz puzzle blizzard blizzard sizzling puzzled puzzle puzzle muzzle muzzle muezzin bl
izzard Neo-Jazz jazz muzzle piazzas puzzles puzzles embezzle buzzed snazzy buzzes puzzled puzzled muz
zle whizzing jazz Belshazzar Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie's Lizzie
Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie Lizzie blizzard blizzards blizzard blizzard fuzzy La
zzeri Piazza piazza palazzi Piazza Piazza Palazzo Palazzo Palazzo Piazza Piazza Palazzo palazzo palaz
zo Palazzo Palazzo Piazza piazza piazza piazza Piazza Piazza Palazzo palazzo Piazza piazza pizza Piaz
za Palazzo palazzo dazzling puzzling Wozzek dazzling dazzling buzzing Jazz jazz Jazz Jazz jazz jazz j
azz jazz Jazz jazz jazz jazz Fuzzy Lizzy Lizzy jazz fuzzy puzzles puzzling puzzling dazzle puzzle daz
zling puzzled jazz jazz jazz jazzy whizzed frazzled quizzical puzzling poetry-and-jazz poetry-and-jaz
z jazz jazz jazz jazz jazz jazz jazz Jazz jazz jazz jazz poetry-and-jazz jazz jazz jazz Dizzy jazz ja
zz jazz jazz jazz poetry-and-jazz jazz jazz jazz jazz jazz jazz jazz jazz jazz jazz jazz dazzled
bedazzlement bedazzled Piazzo nozzles nozzles buzzing dazzles dizzy puzzling puzzling puzzling puzzle
muzzle puzzled nozzle Pozzatti Pozzatti Pozzatti puzzled Pozzatti Pozzatti dazzling pizzicato Jazz ja
zz jazz jazz jazz nozzle grizzled fuzzy muzzle puzzled puzzle muzzle blizzard buzz dizzily drizzle dr
izzle drizzle sizzled puzzled puzzled puzzled fuzzed buzz buzz buzz buzz-buzz-buzz buzzes fuzzy frizz
led drizzle drizzle drizzling drizzling fuzz jazz jazz fuzz puzzle puzzling Nozze mezzo puzzled puzzl
ed dazzling muzzle muzzle muzzle buzzed whizzed sizzled palazzos puzzlement frizzling puzzled puzzled
puzzled dazzling muzzles fuzzy jazz ex-jazz sizzle grizzly guzzled buzzing fuzz nuzzled Kizzie Kizzie
Kizzie Kezziah Kizzie Kizzie Buzz's Buzz Buzz Buzz Buzz Buzz Buzz Buzz dizzy piazza buzzing Puzz
led dizziness dazzled Piazza Carrozza fuzzy dizzy buzzing buzzing puzzled puzzling puzzled puzzled Qu
izzical pizza
```

Higher-Order Functions

- Python provides some higher-order functions that are standard features of functional programming languages such as Haskell.
- We illustrate them here, alongside the equivalent expression using list comprehensions.
- Let's start by defining a function is_content_word() which checks whether a word is from the open class of content words.
- We use this function as the first parameter of filter(), which applies the function to each item in the sequence contained in its second parameter, and only retains the items for which the function returns True.

```
# Higher-Order Functions
```

```
def is_content_word(word):
    return word.lower() not in ['a', 'of', 'the', 'and', 'will', ',', '.']
```

```
sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',
      'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']
```

```
list(filter(is_content_word, sent))
```

```
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
```

```
[w for w in sent if is_content_word(w)]
```

```
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
```

Another higher-order function is map(), which applies a function to every item in a sequence. It is a general version of the extract_property() function we saw in 4.5. Here is a simple way to find the average length of a sentence in the news section of the Brown Corpus, followed by an equivalent version with list comprehension calculation:

```
lengths = list(map(len, nltk.corpus.brown.sents(categories='news')))
```

```
sum(lengths) / len(lengths)
```

```
21.75081116158339
```

```
lengths = [len(sent) for sent in nltk.corpus.brown.sents(categories='news')]
```

```
sum(lengths) / len(lengths)
```

```
21.75081116158339
```

The solutions based on list comprehensions are usually more readable than the solutions based on higher-order functions, and we have favored the former approach throughout this book.

Named Arguments

- When there are a lot of parameters it is easy to get confused about the correct order. Instead we can refer to parameters by name, and even assign them a default value just in case one was not provided by the calling program. Now the parameters can be specified in any order, and can be omitted.

```
def repeat(msg='<empty>', num=1):
    return msg * num
```

```
repeat(num=3)
```

```
'<empty><empty><empty>'
```

```
repeat(msg='Alice')
```

```
'Alice'
```

```
repeat(num=5, msg='Alice')
```

```
'AliceAliceAliceAliceAlice'
```

These are called keyword arguments. If we mix these two kinds of parameters, then we must ensure that the unnamed parameters precede the named ones. It has to be this way, since unnamed parameters are defined by position. We can define a function that takes an arbitrary number of unnamed and named parameters, and access them via an in-place list of arguments *args and an "in-place dictionary" of keyword arguments **kwargs. (Dictionaries will be presented in 3.)

```
def generic(*args, **kwargs):
    print(args)
    print(kwargs)
```

```
generic(1, "African swallow", monty="python")
```

```
(1, 'African swallow')
{'monty': 'python'}
```

```
song = [['four', 'calling', 'birds'],
        ['three', 'French', 'hens'],
        ['two', 'turtle', 'doves']]
```

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

```
[('four', 'three', 'two'),
 ('calling', 'French', 'turtle'),
 ('birds', 'hens', 'doves')]
```

```
list(zip(*song))
```

```
[('four', 'three', 'two'),
 ('calling', 'French', 'turtle'),
 ('birds', 'hens', 'doves')]
```

When *args appears as a function parameter, it actually corresponds to all the unnamed parameters of the function. Here's another illustration of this aspect of Python syntax, for the zip() function which operates on a variable number of arguments. We'll use the variable name *song to demonstrate that there's nothing special about the name *args.

It should be clear from the above example that typing *song is just a convenient shorthand, and equivalent to typing out song[0], song[1], song[2].

Here's another example of the use of keyword arguments in a function definition, along with three equivalent ways to call the function:

```
def freq_words(file, min=1, num=10):
    text = open(file).read()
    tokens = nltk.word_tokenize(text)
    freqdist = nltk.FreqDist(t for t in tokens if len(t) >= min)
    return freqdist.most_common(num)
```

```
fw = freq_words('document.txt', 4, 10)
```

```
fw
```

```
[('flies', 2),
 ('like', 2),
 ('Time', 1),
 ('arrow', 1),
 ('Fruit', 1),
 ('banana', 1),
 ('What', 1),
 ('doing', 1),
 ('late', 1),
 ('Which', 1)]
```

```
fw = freq_words('document.txt', min=4, num=10)
```

```
fw
```

```
[('flies', 2),
 ('like', 2),
 ('Time', 1),
 ('arrow', 1),
 ('Fruit', 1),
 ('banana', 1),
 ('What', 1),
 ('doing', 1),
 ('late', 1),
 ('Which', 1)]
```

```
fw=freq_words('document.txt', num=10, min=4)
```

```
fw
```

```
[('flies', 2),
 ('like', 2),
 ('Time', 1),
 ('arrow', 1),
 ('Fruit', 1),
 ('banana', 1),
 ('What', 1),
 ('doing', 1),
 ('late', 1),
 ('Which', 1)]
```

4.5 A Sample of Python Libraries

- Python has hundreds of third-party libraries, specialized software packages that extend the functionality of Python.
- NLTK is one such library.
- To realize the full power of Python programming, you should become familiar with several other libraries.
- Most of these will need to be manually installed on your computer.

CVS

- Language analysis work often involves data tabulations, containing information about lexical items, or the participants in an empirical study, or the linguistic features extracted from a corpus. Here's a fragment of a simple lexicon, in CSV format:
  - sleep, sli:p, v.i, a condition of body and mind ...
  - walk, wo:k, v.intr, progress by lifting and setting down each foot ...
  - wake, weik, intrans, cease to sleep
- We can use Python's CSV library to read and write files stored in this format. For example, we can open a CSV file called lexicon.csv [1] and iterate over its rows

```
# csv
```

```
import csv
```

```
input_file = open("lexicon.csv", "r")
```

```
for row in csv.reader(input_file):
    print(row)
```

```
['sleep\tsli:p\tv.i\ta condition of body and mind…']
['walk\two:k\tv.intr\tprogress by lifting and setting down each foot…']
['wake\tweik\tintrans\tcease to sleep']
```

- Each row is just a list of strings. If any fields contain numerical data, they will appear as strings, and will have to be converted using int() or float().

NumPy

- The NumPy package provides substantial support for numerical processing in Python.
- NumPy has a multi-dimensional array object, which is easy to initialize and access:

```
from numpy import array
```

```
cube = array([ [[0,0,0], [1,1,1], [2,2,2]],
               [[3,3,3], [4,4,4], [5,5,5]],
                 [[6,6,6], [7,7,7], [8,8,8]] ])
```

```
cube[1,1,1]
```

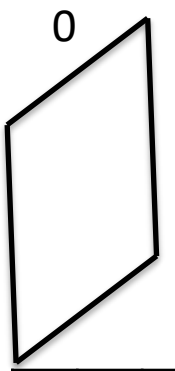
```
4
```

```
cube[2].transpose()
```

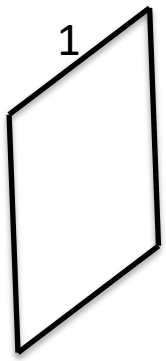
```
array([[6, 7, 8],
       [6, 7, 8],
       [6, 7, 8]])
```

```
cube[2,1:]
```

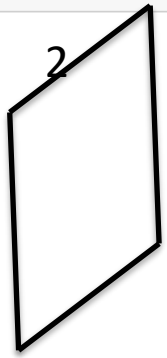
```
array([[7, 7, 7],
       [8, 8, 8]])
```



0	0	0
1	1	1
2	2	2



3	3	3
4	4	4
5	5	5



6	6	6
7	7	7
8	8	8

The code and content in this slide is cited from Natural Language Processing with Python, by Steven Bird, Ewan Klein, and Edward Loper. O'Reilly Media, 978-0-596-51649-9