

Linguistics 5200
University of Colorado, Boulder — Fall 2009
Introduction to Computational Corpus Linguistics
Course notes

Instructor: Robert Albert Felty

December 3, 2009

Contents at a glance

1	Introduction	13
2	UNIX Basics	16
3	UNIX pipes, options, and help	21
4	Regular Expressions and Globs	24
5	Common UNIX utilities	31
6	Managing code with source control	37
7	Getting started with python and the NLTK	41
8	Python lists and NLTK word frequency	45
9	Control structures and Natural Language Understanding	50
10	NLTK corpora and conditional frequency distributions	55
11	Reusing code and wordslists	58
12	Semantic relations	63
13	Processing raw text	66
14	Handling strings	70
15	Unicode and regular expressions in python	76
16	Tokenizing and normalization	80
17	More on strings and shell integration	85
18	Back to basics	90

19 More on functions	97
20 Program development and debugging	101
21 A tour of python modules	107
22 Part of speech tagging	111
23 Automatic Part of speech tagging	117
24 Supervised classification	122
25 Evaluation and Decision trees	129
26 Bayesian and Maximum entropy classifiers	134
27 Advanced Regular Expressions	138
A Practice Problem solutions	145

Contents

1	Introduction	13
1.1	How programming will make your life easier	13
1.2	Computational and Corpus Linguistics	13
1.2.1	Computational Linguistics	13
1.2.2	Corpus Linguistics	14
1.3	UNIX	14
1.3.1	Intro	14
1.3.2	Installation	14
1.4	Philosophy	14
1.4.1	KISS	14
1.4.2	TMTOWTDI	15
1.4.3	The three virtues	15
2	UNIX Basics	16
2.1	Filesystem navigation	16
2.2	Reading files	17
2.3	The PATH	18
2.4	File permissions	19
2.5	Resources	20
3	UNIX pipes, options, and help	21
3.1	Input and Output	21
3.1.1	Streams	21
3.1.2	Pipes	21
3.1.3	Subshells	22
3.2	Options Galore, and where to get help	22
3.3	Practice	23
4	Regular Expressions and Globs	24
4.1	Globs (Wildcards)	24
4.1.1	Introduction	24
4.1.2	Practice	25
4.2	Regular expressions	25

4.2.1	Character classes and anything	26
4.2.2	Quantifiers	26
4.2.3	Greediness	26
4.2.4	Grouping	27
4.2.5	Backreferences	27
4.2.6	The beginning, the end, and escaping	27
4.3	grep	28
4.3.1	Practice	28
4.4	sed and perl	28
4.5	Practice	30
5	Common UNIX utilities	31
5.1	More unix basics	31
5.1.1	network tools	31
5.1.2	Process management	31
5.1.3	Archiving and compressing	32
5.1.4	Calculator	32
5.1.5	Customizing your environment	33
5.1.6	line endings	34
5.2	Common Editors	34
5.3	basic scripting	35
5.3.1	Variables	35
6	Managing code with source control	37
6.1	Version Control	37
6.1.1	intro	37
6.1.2	example	37
6.1.3	concepts	38
6.1.4	Two types of version control	38
6.2	Subversion	38
6.2.1	Intro	38
6.2.2	Diffing	39
6.2.3	Undoing changes	39
6.2.4	Practice	40
7	Getting started with python and the NLTK	41
7.1	Why Python?	41
7.2	Programming basics	41
7.2.1	Starting Python	41
7.2.2	Algorithms	42
7.2.3	numbers	42
7.2.4	statements	42
7.2.5	functions	42

7.2.6	modules	42
7.3	NLTK basics	43
7.3.1	Getting started	43
7.3.2	Concordance	43
7.3.3	Dispersion	43
7.3.4	Lexical Diversity	43
7.4	Help	43
8	Python lists and NLTK word frequency	45
8.1	More python basics	45
8.1.1	variables	45
8.1.2	programs	45
8.1.3	strings	46
8.2	Lists	46
8.2.1	indexing	46
8.2.2	slicing	46
8.2.3	operations	47
8.2.4	len, min, and max	47
8.2.5	List methods	47
8.2.6	Sets	48
8.3	Word Frequency	48
8.3.1	Definition and uses	48
8.3.2	word frequency and the nltk	48
8.3.3	Collocations and bigrams	49
8.3.4	Other counts	49
9	Control structures and Natural Language Understanding	50
9.1	Control Structures	50
9.1.1	Conditionals	50
9.1.2	Code Blocks	52
9.1.3	For Loops	52
9.1.4	List Comprehensions	52
9.1.5	Looping with conditions	52
9.1.6	While loops	53
9.2	Natural Language Understanding	53
9.2.1	Word sense disambiguation	53
9.2.2	Pronoun resolution	54
9.2.3	Language generation	54
9.2.4	Machine Translation	54
9.2.5	Spoken Dialog Systems	54
9.2.6	Textual Entailment	54

10	NLTK corpora and conditional frequency distributions	55
10.1	On importing modules	55
10.1.1	Module syntax	55
10.2	Accessing text corpora	55
10.2.1	Available corpora	55
10.2.2	The Gutenberg Corpus	55
10.2.3	The web and chat Corpus	56
10.2.4	The Brown Corpus	56
10.2.5	Loading your own corpora	56
10.3	Conditional Frequency Distributions	56
10.3.1	Analyzing by genre	56
10.3.2	Plotting and tabulating	57
10.3.3	Generating random text	57
11	Reusing code and wordlists	58
11.1	Reusing code	58
11.1.1	Scripts / Programs	58
11.1.2	functions	58
11.1.3	Modules and Packages	59
11.2	Lexical Resources	60
11.2.1	Basic wordlist	60
11.2.2	Stoplist corpus	60
11.2.3	Names corpus	61
11.2.4	Pronouncing dictionary	61
11.2.5	Finding cognates	61
11.2.6	Automatically finding language families	61
12	Semantic relations	63
12.1	Wordnet	63
12.1.1	Synonyms	63
12.1.2	Hierarchy	63
12.2	Projects	64
12.2.1	More details	64
12.2.2	Available resources	65
13	Processing raw text	66
13.1	Accessing text from the web and disk	66
13.1.1	Retrieving text from the web	66
13.1.2	Dealing with html	66
13.1.3	Reading rss feeds	67
13.1.4	Opening local files	67
13.1.5	Handling delimited text	68
13.2	Using python to interact with the operating system	69

13.2.1	Using stdin and stdout	69
13.2.2	Handling command line arguments	69
13.2.3	Handling command line options	69
14	Handling strings	70
14.1	String basics	70
14.1.1	Creating and concatenating strings	70
14.1.2	Accessing substrings	71
14.1.3	width and precision	72
14.1.4	alignment	72
14.1.5	Strings and tuples	73
14.2	String methods	73
14.2.1	find	73
14.2.2	join / split	73
14.2.3	Differences between lists and strings	74
14.2.4	lower	74
14.2.5	replace	74
14.2.6	strip	75
14.2.7	translate	75
15	Unicode and regular expressions in python	76
15.1	Handling unicode	76
15.1.1	Fonts, glyphs, and encodings	76
15.1.2	Unicode basics	76
15.1.3	Reading non-ascii files	77
15.2	Using regular expressions in python	77
15.2.1	The re module	77
15.2.2	Compiling regular expressions	78
15.2.3	Substitution	78
16	Tokenizing and normalization	80
16.1	Miscellaneous notes on svn and stdin	80
16.1.1	Executable permissions with svn	80
16.1.2	Working with stdin	80
16.2	Applications of regular expressions	80
16.2.1	Working with word pieces	80
16.2.2	Finding word stems	82
16.2.3	Searching tokenized text	82
16.3	Normalization	82
16.3.1	stemmers	83
16.3.2	Lemmatization	83
16.4	Regular expressions for tokenization	83
16.4.1	Simple approaches to tokenization	83

16.4.2	Using the nltk tools for tokenization	84
17	More on strings and shell integration	85
17.1	Homework 7 notes	85
17.1.1	Reading files	85
17.1.2	Handling options	86
17.2	More on string formatting	87
17.2.1	More on alignment	87
17.2.2	Wrapping text	87
17.2.3	Printing vs. writing	88
17.3	Shell integration	88
17.3.1	Calling python scripts from bash scripts	88
17.3.2	Type token ratio	89
18	Back to basics	90
18.1	Homework 8 notes	90
18.1.1	An error with my homework 7 solution	90
18.1.2	Handling options	90
18.2	Sequences (Collections)	91
18.2.1	Uses for tuples	92
18.2.2	More about dictionaries	92
18.2.3	More about sets	93
18.2.4	Converting between different types of sequences	93
18.2.5	Iterating over sequences	94
18.2.6	Generator expressions	94
18.3	Style	95
18.3.1	Procedural vs. declarative style	95
18.3.2	Using counters	95
19	More on functions	97
19.1	Function basics	97
19.1.1	Variable scope	97
19.1.2	Designing effective algorithms	98
19.1.3	Refactoring code	98
19.1.4	Side effects	98
19.2	Advanced function usage	98
19.2.1	Named parameters	98
19.2.2	Higher order functions	99
20	Program development and debugging	101
20.1	Program development	101
20.1.1	More on creating modules	101
20.1.2	Debugging	102

20.2	Algorithm design	104
20.2.1	Recursion	104
21	A tour of python modules	107
21.1	Some useful modules	107
21.1.1	Timeit	107
21.1.2	Profile	108
21.1.3	Matplotlib	108
21.1.4	Numpy	109
22	Part of speech tagging	111
22.1	Using a tagger	111
22.2	Using tagged corpora	111
22.2.1	Nouns	112
22.2.2	Verbs	112
22.3	Handling exceptions	113
22.4	More with dictionaries	114
22.4.1	Indexing and inverting dictionaries	114
22.4.2	Default dictionaries	115
22.4.3	Sorting	116
23	Automatic Part of speech tagging	117
23.1	Automatic tagging	117
23.1.1	Default tagger	117
23.1.2	Regular expression tagging	117
23.1.3	Lookup Tagger	118
23.2	N-Gram tagging	119
23.2.1	Unigram tagging	119
23.2.2	Bigram tagging	120
23.3	Transformation based tagging	121
23.3.1	The Brill tagger	121
24	Supervised classification	122
24.1	Homework 10 comments	122
24.1.1	Subversion comments	122
24.1.2	More on options, parameters and arguments	122
24.2	Supervised classification	123
24.2.1	Gender identification	123
24.2.2	Choosing the right features	124
24.2.3	Document classification	126
24.2.4	Part of speech tagging	127

25	Evaluation and Decision trees	129
25.1	Evaluation	129
25.1.1	Accuracy	129
25.1.2	Precision and recall	129
25.1.3	Confusion matrices	130
25.2	Decision trees	130
25.2.1	Information gain	131
25.2.2	Practice	132
26	Bayesian and Maximum entropy classifiers	134
26.1	Final Presentations	134
26.1.1	Guidelines	134
26.1.2	Schedule	134
26.2	Bayesian Classifiers	135
26.2.1	Zero counts and smoothing	135
26.2.2	Naïvety and double counting	135
26.3	Maximum entropy classifiers	136
26.3.1	Maximizing entropy	136
26.3.2	Generative vs. conditional classifiers	136
26.4	Modeling linguistic data	137
27	Advanced Regular Expressions	138
27.1	One-liners	138
27.2	Advanced regular expression techniques	139
27.2.1	Pattern modifier flags	139
27.2.2	Groupings and backreferences	139
27.2.3	Look-ahead and look-behind	140
27.3	Substitution	141
27.3.1	Transformations	141
27.3.2	Using matches	142
27.3.3	Functions in substitutions	142
27.3.4	More regex examples	143
A	Practice Problem solutions	145
A.1	Introduction	145
A.2	UNIX Basics	145
A.2.1	Filesystem navigation	145
A.2.2	Reading files	145
A.3	UNIX pipes, options, and help	146
A.3.1	Input and Output	146
A.3.2	Practice	146
A.4	Regular Expressions and Globs	147
A.4.1	Globs (Wildcards)	147

A.4.2	grep	147
A.4.3	Practice	148
A.5	Common UNIX utilities	148
A.5.1	basic scripting	148
A.6	Managing code with source control	149
A.6.1	Subversion	149
A.7	Getting started with python and the NLTK	149
A.7.1	input	149
A.8	Python lists and NLTK word frequency	149
A.8.1	Lists	149
A.9	Control structures and Natural Language Understanding	151
A.9.1	Control Structures	151
A.10	NLTK corpora and conditional frequency distributions	152
A.10.1	Accessing text corpora	152
A.11	Reusing code and wordlists	153
A.11.1	Reusing code	153
A.11.2	Lexical Resources	153
A.12	Semantic relations	154
A.12.1	Wordnet	154
A.13	Processing raw text	154
A.13.1	Accessing text from the web and disk	154
A.14	Handling strings	155
A.14.1	String basics	155
A.14.2	String methods	156
A.15	Unicode and regular expressions in python	157
A.15.1	Using regular expressions in python	157
A.16	Tokenizing and normalization	157
A.16.1	Applications of regular expressions	157
A.17	More on strings and shell integration	158
A.17.1	Shell integration	158
A.18	Back to basics	159
A.19	More on functions	159
A.20	Program development and debugging	159
A.20.1	Program development	159
A.21	A tour of python modules	159
A.21.1	Some useful modules	159
A.22	Part of speech tagging	160
A.22.1	Using tagged corpora	160
A.23	Automatic Part of speech tagging	160
A.24	Supervised classification	160
A.24.1	Supervised classification	160
A.25	Evaluation and Decision trees	161

A.26 Bayesian and Maximum entropy classifiers	161
A.27 Advanced Regular Expressions	161
A.27.1 Substitution	161

Chapter 1

Introduction

1.1 How programming will make your life easier

Example 1.1:

I recently discovered that a portion of the sound files from the TIMIT database were grossly distorted. There are 6300 files. How can I remove all the distorted ones?

```
# this snippet removes all clipped files, by checking the  
output from sox  
for file in `find . -name "*.wav" -print`; do  
    if [[ `sox $file -n stat 2>&1 | grep -E  
        "^(Try:|Can't|(Min|Max)imum amplitude:\s+-?1\.00)"` ]];  
        then echo "$file CLIPPED";  
        mv $file $file.clipped;  
    fi;  
done
```

1.2 Computational and Corpus Linguistics

1.2.1 Computational Linguistics

Definition 1.1:

Computational linguistics is an interdisciplinary field dealing with the statistical and/or rule-based modeling of natural language from a computational perspective.

- Machine Translation
- Natural Language Processing
- Information Retrieval
- Information Extraction
- Text summary
- Automated Speech Recognition

- Text to Speech

1.2.2 Corpus Linguistics

Definition 1.2:

Corpus linguistics is the study of language as expressed in samples (corpora) or "real world" text.

- Concordance
- Collocation
- Genre

1.3 UNIX

1.3.1 Intro

What is UNIX?

Definition 1.3:

UNIX is an operating system started at Bell Labs in 1970. It is very mature and stable. Most of the internet and the web run on UNIX-type computers.

1.3.2 Installation

How do I use UNIX?

Three options (in order of ease of use)

1. Mac OSX
2. Cygwin
3. Linux

1.4 Philosophy

1.4.1 KISS

Keep it simple, stupid!

Definition 1.4:

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface. — Doug McIlroy

1.4.2 TMTOWTDI

Definition 1.5:

The Perl motto is “There’s more than one way to do it.” Divining how many more is left as an exercise to the reader.

1.4.3 The three virtues

The three principal virtues of a programmer are:

1. Laziness The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don’t have to answer so many questions about it. Hence, the first great virtue of a programmer. Also hence, this book. See also impatience and hubris. (p.609)
2. Impatience The anger you feel when the computer is being lazy. This makes you write programs that don’t just react to your needs, but actually anticipate them. Or at least pretend to. Hence, the second great virtue of a programmer. See also laziness and hubris. (p.608)
3. Hubris Excessive pride, the sort of thing Zeus zaps you for. Also the quality that makes you write (and maintain) programs that other people won’t want to say bad things about. Hence, the third great virtue of a programmer. See also laziness and impatience. (p.607)

—Larry Wall, from *Programming Perl*, 2nd. ed. (also known as the Camel book)

Chapter 2

UNIX Basics

Today we start off with learning about command line basics.

2.1 Filesystem navigation

cd Change directory to *foo*. Without arguments, changes to your home directory

ls list the contents of directory *foo*. Without arguments, list the contents of the current directory

pwd print out the current working directory (cwd)

mv Move a file to a different location

cp Make a copy of a file in a different location

rm Remove a file (deletes permanently)

mkdir Create a directory

touch Create an empty file

Pathnames

Absolute pathnames

- You can always refer to a file or folder by its absolute pathname.
- Absolute pathnames begin with /

Example 2.1:

```
$ pwd
/home/robfelty
$ cd /home/robfelty/RobsDocs/teaching
$ pwd
/home/robfelty/RobsDocs/teaching
```

Pathnames

Relative pathnames

- Relative pathnames must not begin with /
- Path is relative to current directory
- . is the current directory
- .. is the parent directory

Example 2.2:

```
$ pwd
/home/robfelty
$ cd RobsDocs/teaching
$ pwd
/home/robfelty/RobsDocs/teaching
```

Handy shortcuts

Home directory shortcut

~ is a shortcut for your home directory. E.g., your Desktop is located at ~/Desktop

TAB completion

If you start typing a command or filename, then press TAB, the shell will complete the word for you

Command history

The shell keeps a history of your commands. To scroll through them, simply press the up key. For more info on command history, including how to search it, see: <http://www.catonmat.net/blog/the-definitive-guide-to-bash-command-line-history/>

Practice

Download practiceFiles.zip from the course website and unzip it into your home directory.

1. Change the current working directory to practiceFiles, and show that you are there.
2. List the contents of the practiceFiles directory
3. Change the current working directory to the parent directory
4. Now list the contents of the practiceFiles directory

2.2 Reading files

cat Prints out an entire file (or files)

tac Prints out an entire file backwards (line by line)

head Prints out first *n* lines of a file (default 10)

tail Prints out last n lines of a file (default 10)

wc Counts number of lines, words, and bytes in a file

nl Prints out line numbers for a file

cut Prints out particular columns of a file

paste Pastes together files column-wise

split Splits a file into multiple files, each with n lines (default 1000)

more Interactively print out file, one screen worth at a time

less Fancier version of more. Less is more.

Sorting files et al.

sort Sort a file (many options)

uniq Print unique lines from a sorted input

diff Compare the contents of 2 files

Example 2.3:

Sort the CELEX file by frequency (primary key) with highest frequency coming first, and orthography (secondary key), ignoring case, and save the results in a new file

```
sort -t '\ ' -k 3,3rn -k 2,2fd celex.cd > celex.sorted
```

Practice

1. Inspect the celex.txt file one screen worth at a time
2. Count the total number of lines in the celex file
3. Print the first 10 lines of the celex file

2.3 The PATH

Finding the right PATH

Definition 2.1:

Whenever you type a command in a unix-like shell, the shell searches through the path to see if it can find such a command. You will want to make sure commonly used programs are used in your path. For example, `ls` is normally located in `/bin/ls`, but since `/bin` is in your path, you don't have to type the whole path (but you can of course).

PATH search order

If you have two programs named *foo*, one in `/usr/local/bin`, and one in `/bin`, whichever one comes first in your path will be used when you simply type *foo*. If you want to make sure that a particular one is used, specify the entire path.

Showing and changing your path

To show your path:

```
echo $PATH
```

To change your path:

```
export PATH="/some/new/path:${PATH}"
```

Search the path for a program:

`which foo` (If there is no output that means that the program is not in your path, or not installed on your computer).

2.4 File permissions

Definition 2.2:

Every file on a UNIX system has 3 sets of permissions, specifying who can read, write, and execute the file. The three sets apply to the user, group, and others

Example 2.4:

```
drwxr-xr-x  4 robfelty root      4096 Jul 10 23:02 fender4star
-rwxr-xr-x  1 robfelty robfelty 1137 Aug 19 14:12 syncWithDreamhost
lrwxrwxrwx  1 robfelty yootlers    21 Jun  9 10:43 images ->
    ../fedibblety/images/
```

File permissions

Example 2.5:

To change file permissions, you can use the commands `chown`, `chgrp`, `chmod` Change the user to john for the file `johnsfile.txt`

```
chown john johnsfile.txt
```

Change the group to johnandmary for the file `johnsfile.txt`

```
chgrp johnandmary johnsfile.txt
```

Make a file executable by all users

```
chmod a+x myFirstPythonScript.py
```

2.5 Resources

unix ref http://www.cumc.columbia.edu/computers/html/unix/unix1_01.htm

unix ref http://infohost.nmt.edu/tcc/help/unix/unix_cmd.html

delicious <http://delicious.com/robfelty/compling>

Chapter 3

UNIX pipes, options, and help

3.1 Input and Output

3.1.1 Streams

Three streams

Standard input (STDIN)

The input to a program can be specified like so: `cat < foo`

Standard output (STDOUT)

The output from a program can be redirected to a file `ls > foo.txt` To append to a file, use:
`ls >> foo.txt`

Standard error (STDERR)

Error messages are sent to a different output stream. You can redirect stderr like so: `ls 2> error.txt`

Three streams — practice

1. Write the first 10 lines of `celex.txt` to a new file called `celex.small`
2. Append the last 10 lines of `celex.txt` to `celex.small`

3.1.2 Pipes

Put this in your pipe and ...

Definition 3.1:

You can pipe the output from one program into another program

Example 3.1: head and tail

To display only the first 10 entries of a directory listing, you could do: `ls | head` How would you display the last 10?

Example 3.2:

Create a histogram of frequencies from the CELEX

```
cut -f 3 -d '\ ' celex.cd | sort -rn | uniq -c
```

Pipes — practice

1. Display lines (files) 11–20 from a directory listing of practiceFiles
2. Produce a numbered list of the files in practiceFiles

3.1.3 Subshells

Definition 3.2:

You can run a command in a subshell by using backticks ``foo``. The result of the command can be stored and used.

Example 3.3:

Remove file extensions and preceding path elements from a filename

```
echo `basename 155practiceFiles/a.txt .txt`
```

Subshells — practice

1. Move files 11–20 to a new directory tmp

3.2 Options Galore, and where to get help

Options

Most UNIX commands have a variety of options which can be specified on the command line.

short options e.g. `-h` = “print help for this command”. You can group these together, like `-hv`, means, print verbose help

long options e.g. `--help`. Long options take longer to type, but obviously are more descriptive

other Some commands take long-style options with a single dash, notably `java`.

Example 3.4:

Some options also take arguments. To print the first 25 lines of a file, we could do:

```
head -n 25 foo
```

Help

Most standard UNIX commands are quite well documented. To get help on a particular command, try:

`man foo`

Moving around a manual page: (same commands as less by default)

<space> Go forward one screen

↑ ↓ Navigate with arrow keys

k j Navigate from the home row

/ Search for a word (can use regular expressions)

n Go to next instance of found word

N Go to previous instance of found word

q Exit the manual pager

3.3 Practice

1. Count the total number of characters in the filenames in practiceFiles
2. Display the second column of celex.small
3. Create a new file with only the first and third columns of celex.small
4. Combine the celex.small with the new file you just created
5. Sort the celex.small file by orthography, ignoring case (HINT: use `-t '\'`)
6. Calculate the average number of characters per word in the devilsDictionary.txt
 - (a) First find just the number of characters
 - (b) Next find just the number of words
 - (c) Now use subshells to put this into an equation form
 - (d) Finall, pipe this equation to the basic calculator

Chapter 4

Regular Expressions and Globs

4.1 Globs (Wildcards)

4.1.1 Introduction

Definition 4.1:

Globs (wildcards) can be used by BASH, and by other programs (Microsoft Word & Excel) as shortcuts to match multiple expressions

* Match zero or more characters.

? Match any single character

[...] Match any single character from the bracketed set. A range of characters can be specified with [-]

[!...] Match any single character NOT in the bracketed set.

{a,b,...} A list (set)

N.B.

- An initial "." in a filename does not match a wildcard unless explicitly given in the pattern. In this sense filenames starting with "." are hidden. A "." elsewhere in the filename is not special.
- Pattern operators can be combined

Example 4.1:

`chapter[1-5].*` could match *chapter1.tex*, *chapter4.tex*, *chapter5.tex.old*. It would not match *chapter10.tex* or *chapter1*

Using globs in BASH

Example 4.2:

Delete all microsoft word documents in my home directory

```
rm -f ~/.doc
```

Example 4.3:

Convert all microsoft word documents in my home directory to plain text

```
for file in ~/.doc; do antiword $file `basename $file  
    .doc`.txt; done
```

Example 4.4:

Create all files a-c with extensions txt,tmp,foo,bar

```
touch {a,b,c}.{txt,tmp,foo,bar}
```

4.1.2 Practice

Practice using globs in BASH

Using the practiceFiles

1. Move all files ending in .txt to a new directory txt
2. Copy files 10-19 to a new directory 10-19
3. list permissions for files ending in .txt which do not contain numbers
4. Separate files into different directories according to their extension

4.2 Regular expressions

Regular expressions are similar to wildcards, but are much more powerful. For example, you want to find all occurrences which have any number of letters, followed by 1 number, followed by any number of letters e.g. *abc1xyz*, *a1xyz*. This is not possible using wildcards, but it is possible using regular expressions.

Regular expressions will be useful for the following purposes:

- Finding the information you need from the databases will require the use of regular expressions
- Regular expressions are a feature in many programming languages that allow one to search for a given string in a body of text, including the use of some special characters
- Problem: I want to find all CVC words in the English CELEX database Solution: `grep -E '\\\\[CVC\\\\]\\\\' celex.cd`
- Problem: I want to know how many words that start and end with the letter *k* Solution: `grep -iEc '\\k[a-z]*k\\\\' celex.cd`

Special characters: . ? + * [] {} () ^ \$

4.2.1 Character classes and anything

`.` matches any character

`[]` matches any of the characters within the brackets e.g. `[a0]` matches both *a* and *0*

Several predefined shortcuts are also possible

`[a-z]` matches all lowercase letters

`[A-Z]` matches all uppercase letters

`[a-zA-Z]` matches all uppercase and lowercase letters

`[0-9]` matches all numbers

4.2.2 Quantifiers

Special characters: `.` `?` `+` `*` `[]` `{}` `()` `|` `^` `$` `\`

`?` matches 1 or 0 of the preceding character, e.g. `colou?r` matches *color* and *colour*

`+` matches 1 or more of the preceding character, e.g. `bug +off` matches *bug off*, *bug off*, but not *bugoff*

`*` matches any number of the preceding character, e.g. `colou*r` matches *color*, *colour*, *colouur* and so on

`{}` used to specify the number of times a character should be matched. Ranges are also possible.

Example 4.5:

`a{2}` matches only *aa*

`[a-z]{2}` matches two lowercase letters, e.g. *ab*

`[a-z]{2,4}` matches 2–4 lowercase letters, e.g. *al* or *foo*

4.2.3 Greediness

Special characters: `.` `?` `+` `*` `[]` `{}` `()` `|` `^` `$` `\`

Definition 4.2:

By default, `*` and `+` are greedy, meaning that they match as much as possible. Often this is not the intended effect.

Example 4.6:

I want to strip out html tags from a document. I use the following regular expression: `<.*>` This will match `<span class='foo'>`. But it will also match `<span class='foo'>some text I don't want to get rid of</span>` Solution: use negative character classes: `<[^\<>]*>` In Perl and python, you can use `.?*>` and `.+>`

4.2.4 Grouping

Special characters: `. ? + * [] {} () | ^ $ \`

- `()` used to group sequences. Useful especially for backreferences (more on that later), and
- `|` used as an or operator, e.g. `x|y` matches either `x` or `y`

Example 4.7:

`(m|M)(in|ax)imum` matches *minimum*, *maximum*, *Minimum* and *Maximum*

4.2.5 Backreferences

Special characters `. ? + * [] {} () | ^ $ \`

Definition 4.3:

`\1` is a backreference. You can use multiple backreferences of the form `\n` where `n` is the `n`th pair of parentheses in the expression.

Example 4.8:

Say I want to find common typos involving duplicate words (such as *a a* or *the the*). I could write an expression like so `(a|the) \1` which says “match either *a* or *the* followed by a space followed by whatever was matched in the parentheses”

4.2.6 The beginning, the end, and escaping

Special characters: `. ? + * [] {} () | ^ $ \`

- `^` matches the beginning of the string Within brackets, negates the pattern, e.g. `[^xy]` matches everything but `x` or `y`
- `$` matches the end of the string
- `\` is the escape character. When you want to use one of the special characters as a normal character, it must be preceded by `\`

4.3 grep

Grep specific information

- grep will search a file on a line by line basis, and return any lines which contain the regular expression
- In the case of CELEX, we will take advantage of the fact that fields are separated by \

Grep options

Like many UNIX programs, grep has quite a few options available. For a complete list, type `man grep`

- -E extended regular expressions — allows us to use all the special characters
- -i ignore case
- -c simply print the number of matches
- -v invert match, i.e. return everything that does not match the expression

These can be used in conjunction with one another, e.g. `grep -icv 'dog' file` returns the number of lines that do not contain the word dog from the file 'file'.

4.3.1 Practice

Regular Expression practice

Practice writing some regular expressions that will find the following from CELEX:

- all words begin with 'st'
- all words that end in 'ing'
- word that begin with 'st' and ending with 'ing'
- all monosyllabic words
- all disyllabic words

4.4 sed and perl

Substitution

Definition 4.4:

Not only can you use regular expressions to match strings, but you can also replace matched strings with other strings. The easiest way to do this is with the program *sed*. *By default, sed prints out the entire input, replacing any patterns with the specified replacements* The basic form is like so:

```
sed 's/match/replace/flags' < infile > outfile
```

Example 4.9:

Input: *The blue man sat next to the green man.*

```
echo 'The blue man sat next to the green man.' |  
sed 's/man/woman/g'
```

Output: *The blue woman sat next to the green woman.*

Backreferences in replacements**Definition 4.5:**

Backreferences can be used not only in patterns, but also in replacements. This allows one to use dynamic replacements.

Example 4.10:

File replacement: I want to get rid of spaces in filenames, because they can cause problems with UNIX scripts. I can use sed.

```
mv "foo bar.txt" foo_bar.txt  
for file in *; do mv "$file" `echo $file|sed -E 's/ /_/g'`;  
done
```

More transformations

\l Makes the following character lower case

\u Makes the following character upper case

\L Makes all following characters lower case

\U Makes all following characters upper case

Note — these work in GNU (Linux) sed, but not on Mac. They work in perl on any system (and in vim).

Example 4.11:

```
echo "Minimum"|sed -r 's/(in|ax)imum/\u\1/'  
OR  
echo "Minimum"|perl -pe 's/(in|ax)imum/\u\1/'
```

4.5 Practice

Practice

1. Count the number of entries in the Devil's Dictionary
2. Print out the all the entries in the Devil's dictionary (not the definition)
3. Count the number of occurrences of the word *the* in the Devil's Dictionary
4. Count the number of indefinite articles in the Devil's Dictionary

Practice (2) details

4. Count the number of indefinite articles in the Devil's Dictionary `grep -Eic '([^[a-z]|^)(a|an)([^[a-z]|$)'` devilsDictionary.txt This is basically the same as finding *the*, except we need to search for either *a* or *an*.

Chapter 5

Common UNIX utilities

5.1 More unix basics

5.1.1 network tools

Network tools

ssh secure remote login to another computer

sftp secure file transfer to another computer (interactive)

scp secure file transfer to another computer (non-interactive)

rsync extremely powerful and smart file transfer (works both for local and remote computers — non-interactive)

5.1.2 Process management

Process management

ps Display which processes are running (non-interactive)

top Display which processes are running (interactive)

kill Kill (abort) a process using the process ID

killall Kill (abort) a process using the process name

nice Set the cpu priority for a process

ionice Set the disk usage priority for a process

nohup Keep running after logging out

& Run process in the background

Example 5.1:

Run a long process in the background and don't hog system resources

```
nohup ionice -c2 -n7 nice -n 19 prog --progOpts &
```

5.1.3 Archiving and compressing

Archiving and compressing

zip Create a zip file

unzip Extract contents from a zip file

gzip Compress a file with GNU zip

gunzip Decompress a file with GNU zip

bzip2 Compress a file with bzip compression (makes smaller files)

bunzip2 Decompress a file with bzip

tar Create and extract tar archives Common uses:

```
create tar -czvf file.tar.gz directory
```

```
extract tar -xzvf file.tar.gz
```

5.1.4 Calculator

Calculator

bc Basic interactive calculator. Usually should invoke with the **-l** option

dc Reverse polish style interactive calculator

Example 5.2:

Add the first line of one file and the last of another

```
echo "`head -n1 numbers.txt` + `tail -n1 numbers2.txt`" |bc -l
```

Example 5.3:

Add the first 4 lines of a file (which contains one number per line)

```
echo "`head -n 4 numbers.txt` +++ p" |dc
```

5.1.5 Customizing your environment

Environment variables

Definition 5.1:

Most UNIX programs pay attention to environment variables, such as the language, timezone, and PATH. To see all currently set variables, type:
export

Example 5.4:

To change a variable, do:

```
export PATH="/home/robfelty/bin:${PATH}"
```

Custom variables and aliases

Example 5.5:

You can also create and use your own variables. If you frequently want to change directories to a particular directory, you can store its name in a variable, e.g.

```
ling5200=/home/robfelty/RobsDocs/teaching/ling5200  
cd $ling5200
```

Example 5.6:

If you always want to have color listings, you can create an alias

```
alias ls='ls --color'
```

.rc files

Definition 5.2:

Many UNIX programs, including the shell (we have been using the BASH shell), have files where one can store customizations between sessions.

Common .rc files

- .bashrc
- .vimrc
- .inputrc

Every time you open a new terminal, the .bashrc file is read.

5.1.6 line endings

Line Endings

Definition 5.3:

Mac, UNIX, and DOS (Windows) use different line ending characters, which can cause lots of problems

`\r` Mac

`\n` UNIX

`\r\n` DOS

Converting between Mac, DOS, and UNIX

Most Linux distros ship with the programs *unix2dos* etc. Mac does not. Instead use the scripts provided in the `resources/utls` directory.

5.2 Common Editors

Common editors

nano (Open source version of pico). Advantages:

- user-friendly. Lists commands at bottom of screen.
- small

Disadvantages:

- Not very powerful
- Not a default install on many UNIXes

vi Two-mode editor. This is my editor of choice. advantages:

- small (in size and memory usage)
- common (found on almost all UNIX systems by default)
- powerful (great regular expression support, and nice syntax highlighting)
- fast (your fingers never have to leave the home row. No mouse required)

Disadvantages:

- steep learning curve

emacs Editor of choice for many programmers. Swiss-army knife of editors. Advantages:

- Great syntax highlighting
- Single mode editor
- Includes all sorts of tools (news readers, e-mail readers, version control interfaces, friendfeed interface)

Disadvantages:

- Uses lots of memory
- Not a default install on many UNIXes

5.3 basic scripting

Basic shell scripting

Definition 5.4:

A shell script uses the exact same syntax as the command line shell you use (we have been using BASH). In this way, you can group commands together, to reduce work.

5.3.1 Variables

Variables

Create a variable

```
FOO='hello '
```

```
BAR=`ls`
```

Make sure there are no spaces around the equal sign

Use a variable

```
echo $FOO
```

```
echo $BAR | head
```

Basic shell scripting

Example 5.7: mean characters per word

```
#!/bin/bash
LETTERS=`wc -c devilsDictionary.txt|cut -f 3 -d ' '`
WORDS=`wc -w devilsDictionary.txt|cut -f 3 -d ' '`
echo "$LETTERS / $WORDS" | bc -l
```

Example 5.8: syncing computers

```
1 #!/bin/bash
2 # this script syncs my school computer onto an
  external hard disk using rsync
3
4 # define a few constants
```

```

5 TARGET='/media/disk'
6 OPTIONS=' -avz --delete-after '
7 UMOUNT='FALSE'
8
9 echo "Executing incremental backup script"
10
11 # if /media/disk does not exist, create it,
   then mount the disk, and mark for unmounting
12 if [ ! -d /media/disk ]; then
13     echo "creating /media/disk and mounting"
14     UMOUNT='TRUE'
15     mkdir /media/disk
16     mount /dev/sdd1 /media/disk
17 fi
18 # first backup a few directories from the
   external disk to the local hard disk
19 ionice -c2 nice -n 19 rsync -avzu
   --exclude='.svn*' --exclude="*.swp"
   ${TARGET}/home/robfelty/{adam,RobDocs,pics,R,matlab}
   /home/robfelty
20 ionice -c2 nice -n 19 rsync -avzu
   --exclude='.svn*' --exclude="*.swp"
   ${TARGET}/var/celex /var
21
22 #next backup everything from the local disk to
   the external
23 ionice -c2 nice -n 19 rsync $OPTIONS /selinux
   /bin /etc /home /lib /lib64 /misc /opt /root
   /sbin /usr /var ${TARGET}/ >
   ~/fedibbletyBackupLog.txt
24
25 if [[ $UMOUNT = 'TRUE' ]]; then
26     echo "unmounting and removing /media/disk"
27     umount /media/disk
28     rmdir /media/disk
29 fi

```

Practice

1. Create a new script in ~/bin called *changeEnding*
2. Make the file executable
3. Edit the file so that it will move all .txt files to .temp Use the editor of your choice (nano, emacs, vi)

Chapter 6

Managing code with source control

6.1 Version Control

6.1.1 intro

Version Control intro

Definition 6.1:

Version control is an essential tool for programmers, providing several key functions:

1. The ability to track code changes
2. The ability to collaborate easily
3. The ability to create and potentially merge different versions of the same project

Also referred to as

- RCS (revision control system)
- SCM (source control management)

6.1.2 example

A small example

Suppose *Joe the programmer* and I are working on developing a python module.

Joe's copy

```
""" this module does X"""  
import re,sys,os,time  
foo = 1  
bar = 2
```

My copy

```
""" this module does X"""  
import re,sys,os  
foo = 1
```

```
bar = 2
another = 23
```

6.1.3 concepts

Basic concepts

revision Every time someone commits something new to the repository, a new revision is created, which is like a snapshot of the project at one particular point in time

repository The repository contains all of the project's files, and most importantly a history of all the changes to it

working copy A working copy is your own personal copy of the repository. It contains only 1 revision of the repository

6.1.4 Two types of version control

Version Control types

Centralized

All the code is stored on a central server. Whenever developers want to download the newest version, or upload some changes, they must use the server

- RCS
- CVS (concurrent version system)
- Subversion

Distributed

Every person gets a complete copy of the code, including all the history and changes

- git
- mercurial
- bazaar

6.2 Subversion

6.2.1 Intro

subversion intro

Download from <http://subversion.tigris.org>

Why subversion?

- Subversion is designed as a replacement for CVS.
- CVS was the most widely-used version control system.
- Subversion is becoming the most widely-used, and fixes lots of problems with CVS.

- free
- available for almost every operating system
- well documented
- relatively easy

subversion commands

svn help Get help on using subversion.

svn checkout Download a fresh copy of a repository

svn update Get the latest updates for your working copy

svn commit Commit some changes you have made to the repository

svn add Add a file or directory to svn (the next time you commit)

svn mv Change the location of a file

svn diff Compare your working copy to the version in the repository

class repository

I have created a subversion repository for the class.

`svn co http://robfelty.com/subversion/ling5200 myling5200`

- There is a subdirectory for each student
- You have read-only permissions on everything
- You have read-write permissions on your own directory

6.2.2 Diffing

Diffs, conflicts and logs

- You can specify particular files (or directories) in the log command `svn log slides/unix3`
- Use diff to see the differences between two different revisions `svn diff -r 14:29 slides/unix3`
- Use svn info to find the last time a file was changed `svn info slides/unix3`
- If two people edit the same file, it may result in a conflict

6.2.3 Undoing changes

Undoing changes

revert

To undo changes to your working copy, use the revert command `svn revert <file>`

merge

To bring back changes from a committed revision, use merge `svn merge -c -44 <url> <file>`

6.2.4 Practice

Practice

1. Create a new file in your own directory called `hmwk2_<name>.sh`
2. Add this new file to your working copy
3. Commit your change
4. Edit the homework file with your favorite editor (`vi`, `nano`, `emacs`)
5. Examine the differences
6. Commit your change

Chapter 7

Getting started with python and the NLTK

7.1 Why Python?

Features

interpreted No compiling necessary. Don't need to worry about memory.

object-oriented Object-oriented by design, without unnecessary baggage (like Java)

clean syntax Syntax is very simple, and forces you to use a clean style.

open source Python is free, you can view the source, and it is actively maintained.

extensible It is easy to add functionality via modules and packages, and many people share these.

popular You will be able to collaborate with many people.

Why the NLTK?

- Many common computational and corpus linguistic tasks are already implemented
- well documented
- Will allow us to understand and practice basic concepts, without a deep theoretical understanding
- Why re-invent the wheel?
- *We can use the NLTK to write an infinite number of programs*

7.2 Programming basics

7.2.1 Starting Python

- Interactive

- from the shell, type `python`
- Using IDLE
- Non-interactive
 - from the shell, type `python mypythonscript.py`
 - Using IDLE, select run > run module

7.2.2 Algorithms

What is an algorithm?

Definition 7.1:

An algorithm is a set of instructions or a recipe for a computer to carry out.

7.2.3 numbers

Division

By default, `1 / 2` yields `0` in python. This is integer division. *Solution:* `from __future__ import division`

7.2.4 statements

What is the difference between an expression and a statement?

Definition 7.2:

An expression *is* something, and a statement *does* something.

7.2.5 functions

What is a function?

Definition 7.3:

A function is a mini-program. It can take several *arguments*, and *returns* a value.

7.2.6 modules

Definition 7.4:

Python is easily *extensible*. Users can easily write programs that extend the basic functionality, and these programs can be used by other programs, by loading them as a *module*

Practice

1. load the math module
2. Round 35.4 down to the nearest integer

7.3 NLTK basics

7.3.1 Getting started

- Download the nltk from nltk.org
- Start python
- `import nltk`
- Download texts `nltk.download()`
- Load everything for the book from `nltk.book import *`

7.3.2 Concordance

Definition 7.5:

A *concordance* is a list of words from a text with their surrounding context. Syntactic structure and semantics can be inferred from a concordance.

Example 7.1:

```
text1.concordance("whiteness")
text6.concordance("holy")
text6.concordance("silly")
text6.similar("silly")
```

7.3.3 Dispersion

Definition 7.6:

A *dispersion plot* shows the location in a text in which it appears, relative to the beginning of the text

Example 7.2:

```
text4.dispersion_plot(["citizens", "democracy", "freedom", "duties",
"America"])
```

7.3.4 Lexical Diversity

Definition 7.7:

Lexical diversity is a measure of how often words are repeated in a text, computed as $\frac{\text{token}}{\text{type}}$

- Sometimes used as a measure of text difficulty
- Sometimes the inverse is used, the type / token ratio

7.4 Help

Just like UNIX, python has built-in help too.

- Try typing 'help' at the python command prompt.
- To get help on a specific command, try `help(command)`. `help(list.sort)`
- For help with the nltk, type `help(nltk)`
- You will see a list of Package contents. For more details on any of these you can type `help(nltk.<name>)`.
- For example, `help(nltk.corpus)` gives you more information about the corpora included with the nltk and
- You can also get help about a variable. `foo=2 help(foo) help(text1)`

Chapter 8

Python lists and NLTK word frequency

8.1 More python basics

8.1.1 variables

Variables

What is a variable?

Definition 8.1:

A variable is a name that refers to some value (could be a number, a string, a list etc.)

Practice

1. Store the value 42 in a variable named *foo* `foo = 42`
2. Store the value of `foo+10` in a variable named *bar* `bar = foo + 10`

8.1.2 programs

Saving and executing programs**Example 8.1:** Hello world program

```
Script File: hello.py
#!/usr/bin/env python
# this script prints 'hello, world', to stdout
print("hello, world")
Add executable permission (in BASH):
chmod a+x hello.py
Run the program:
./hello.py
```

8.1.3 strings

String Basics

- Strings must be enclosed in quotes (double or single)
- concatenate using the + operator
- repeat using the * operator
- join a list of strings into one string
- split a strings into a list

Joining and splitting

join

join a list of strings into one string

tab `'\t'.join(['foo', 'bar'])`

space `' '.join(['foo', 'bar'])`

comma `','.join(['foo', 'bar'])`

split

split a strings into a list

```
mystring = 'hello world'
mylist = mystring.split()
```

8.2 Lists

8.2.1 indexing

Indexing

Zero comes first

Python starts all indices with 0.

Practice

1. Create a list *foo*, with the following values: 25, 68, “bar”, 89.45, 789, “spam”, 0, “last item”
2. print just the first item foo
3. print just the last item foo

8.2.2 slicing

Slicing

Definition 8.2:

A list slice takes part of a list. A slice `list[i:j]` starts at the i_{th} index, and goes up to (*but does not include*) the j_{th} index.

Slicing practice**Practice**

HINT: remember that indices start at 0

1. Print the 1st to 3rd item in the list *foo*
2. Print the 3rd to last item in the list *foo*
3. Print the 2nd to the 2nd to last item in the list *foo*
4. Copy the entire *foo* list to a new list named *bar*

8.2.3 operations**Operations****Practice**

1. Change the first item in the *foo* list to 12
2. Now multiply the first item in the *foo* list by 2
3. Test whether “ham” is in the list *foo*

8.2.4 len, min, and max**Len, min, and max****practice**

1. How many items does *foo* contain?
2. What does `min(foo)` return?
3. What does `max(foo)` return?

8.2.5 List methods**Popping, appending, etc.****practice**

1. Append the value 24 to the list *foo*
2. Insert the value “twenty” to the list *foo* as the 4th item
3. Find the index of “spam” in the list *foo*
4. remove the last item from *foo*, and store it as a new variable

Extending and sorting

Practice

1. Append the following values to *foo*: 89, 23.4, 1
2. Create a new list *fooSorted* with the same contents as *foo*, but sorted

8.2.6 Sets

Python sets

In addition to lists, python also has a built-in *set* type

Definition 8.3:

A *set* is an unordered collection of unique elements

Example 8.2:

Convert list to set

```
mylist=['foo', 'bar', 'foo']  
myset = set(mylist)
```

8.3 Word Frequency

8.3.1 Definition and uses

Word frequency

Definition 8.4:

word frequency is a measure of how frequently words are used.

Uses

processing Frequent words are processed more quickly and accurately than infrequent words

historical Frequent words undergo reduction more than rare words

8.3.2 word frequency and the nltk

Calculating word frequency with the nltk

Example 8.3:

Calculating word frequency

```
freqdist1 = FreqDist(text1)  
# create a plot  
freqdist1.plot(50, cumulative=True)
```

Fine-grained selection of words

Example 8.4:

```
Selecting words by length
V = set(text1)
long_words = [w for w in V if len(w) > 15]
sorted(long_words)
# now select only frequent words longer than 7
long_frequent_words = [w for w in V if len(w) > 15 and
    freqdist1[w] > 10]
```

8.3.3 Collocations and bigrams

Collocations and bigrams

Definition 8.5:

a *bigram* is a pair of units that appear consecutively (words, phonemes, letters, etc.) *Collocations* are basically just frequent bigrams

Producing collocations

Example 8.5:

```
Using NLTK to produce a collocation
text4.collocations()
```

8.3.4 Other counts

Other counts

Example 8.6:

```
Word length frequency
# create a list of word lengths
lengths1 = [len(w) for w in text1]
lengthDist1 = FreqDist(lengths1)
# plot the distribution
lengthDist1.plot(cumulative=True)
# which word length occurs most?
lengthDist1.max()
# what proportion of words have 3 letters?
lengthDist1.freq(3)
```

Chapter 9

Control structures and Natural Language Understanding

9.1 Control Structures

9.1.1 Conditionals

Conditionals

Definition 9.1:

Conditional can be used to control the flow of a program. Code inside a conditional will only be executed when the specified condition is met.

Example 9.1:

```
temp = 50
if temp < 55:
    print "I should wear a coat today"
```

Truth values

Definition 9.2:

Nothing is false. That is:

False

None

0

[] (empty list)
{ } (empty dict)
'' (empty string)
() (empty tuple)

if, then

Practice

```
# Initialize two variables x,y to 3,4
# if x and y are equal to each other,
# print "equal", otherwise "unequal"
```

if, then

More Practice

```
# Now add a condition that if
#x is less than y, print "x is less than y"
#otherwise, print "x is greater than y"
```

Booleans

Definition 9.3:

You can combine conditions with *and* and *or*, and negate with *not*

Example 9.2:

```
if 5 < x < 10 and x not in y:
    print "x is between 5 and 10, \
        and is not in the list y"
```

9.1.2 Code Blocks

Definition 9.4:

In python, blocks are created by the use of a colon, followed by an indented section of text

Example 9.3:

```
if foo==bar:
    do something
    do another thing
    a final thing
do this regardless
```

9.1.3 For Loops

Definition 9.5:

Iterate: to do (something) over again or repeatedly. It easy to iterate over lists with for loops

practice

Multiply each item in *mylist* by 2, and print the result (don't change the values of mylist)

9.1.4 List Comprehensions

Definition 9.6:

A *list comprehension* is a concise way to operate on every element of a list

Example 9.4:

Create a new list which, in which each value of the origlist is doubled `newlist = [item * 2 for item in origlist]`

practice

For each item in *mylist*, multiply by 2 and add 3, (*do* change the values of mylist)

9.1.5 Looping with conditions

We can also use loops and conditionals together, which is very powerful

Example 9.5:

```
mylist = [4,50,8,9,20]
for item in mylist:
    if item % 4 == 0:
        print item, " is divisible by 4"
```

practice

Create a list called *ab* and put all words from *text1* into it which start with 'ab'

9.1.6 While loops

Beware of infinite loops!

It is easy to create infinite loops with `while`. Make sure that your while condition will return false at some point.

While

practice

1. Create a list *mylist* from 1:10
2. Choose a random item from this list until the item you choose is your lucky number (your choice of number from 1 to 10 (inclusive)). Count how many times did your program have to choose a number? (Use the `random.choice` function)

```
import random
mycount=0
myList=range(1,11)
myNumber=3
while random.choice(myList) != myNumber:
    mycount+=1
print "it took", mycount, "times"
```

9.2 Natural Language Understanding

Challenges in natural language understanding

What are the various challenges to getting a computer to understand and produce language?

- Word sense disambiguation
- Pronoun resolution
- Language generation
- Machine Translation
- Spoken Dialog Systems
- Textual Entailment

9.2.1 Word sense disambiguation

- Words can have multiple meanings e.g. *bank* — financial institution *bank* — edge of a river
- How do we know which sense to use?

9.2.2 Pronoun resolution

Who does *they* refer to in the following sentences?

1. The thieves stole the paintings. They were subsequently *sold*.
2. The thieves stole the paintings. They were subsequently *caught*.
3. The thieves stole the paintings. They were subsequently *found*.

9.2.3 Language generation

- Language generation is the opposite of language understanding
- One use of language generation is answering questions
- Have you ever tried a question answering program? Did it work well?

9.2.4 Machine Translation

- If we can conquer natural language understanding, we should be able to reproduce a sentence in any language
- In reality, most machine translation uses parallel texts between two languages

9.2.5 Spoken Dialog Systems

A spoken dialog system uses every part of computational linguistics

1. Automated speech recognition
2. Natural language Understanding
3. Language generation
4. Text to Speech

Try out a chatbot from the `nlk.nltk.chat.chatbots()` How do you suppose it works?

9.2.6 Textual Entailment

Another challenge for computational linguistics is evaluating simple truth statements from more complex or nuanced language.

Chapter 10

NLTK corpora and conditional frequency distributions

10.1 On importing modules

10.1.1 Module syntax

```
foo = [range(1,10), range(10,20), range(20,30)]
import pprint
pprint.pprint(foo)
from pprint import pprint
pprint(foo)
from pprint import *
bar = pformat(foo)
import pprint as howdy
from pprint import pprint as howdy
```

10.2 Accessing text corpora

10.2.1 Available corpora

To see all available corpora that nltk has by default: `help(nltk.corpus)`

10.2.2 The Gutenberg Corpus

The Gutenberg corpus has a selection of texts from Project Gutenberg

1. Import the gutenberg corpus from `nltk.corpus` `import gutenberg`
2. Show the files in the gutenberg corpus `gutenberg.fileids()`
3. Retrieve a list of words from Hamlet and store as *hamlet* `hamlet = gutenberg.words('shakespeare-h`

10.2.3 The web and chat Corpus

The web and chat corpora come from several different online texts, discussion forums, and chat rooms

practice

1. Import the web chat corpus (webtext)
2. Show the files in the webtext corpus
3. Retrieve a list of sentences from the wine file in webtext and store as *wine*
4. Print the first 10 sentences of the wine text

10.2.4 The Brown Corpus

- The brown corpus was one of the first large corpora,
- consisting of 1 million words taken from newspapers and magazines,
- first released in 1967.
- It is categorized into 15 different genres.

```
from nltk.corpus import brown
# show categories
brown.categories()
# show words from 'news' genre
brown.words(categories='news')
# show words from 'news' and romance genres
brown.words(categories=['news', 'romance'])
```

10.2.5 Loading your own corpora

```
from nltk.corpus import PlaintextCorpusReader
corpus_root = 'ling5200/resources/texts'
wordlists = PlaintextCorpusReader(corpus_root, ['celex.txt',
        'devilsDictionary.txt'])
wordlists.fileids()
# create a list of words from the Devil's Dictionary
devilswords = wordlists.words('devilsDictionary.txt')
```

10.3 Conditional Frequency Distributions

10.3.1 Analyzing by genre

Creating a list of pairs using a list comprehension

```
genres = ['romance', 'scifi']
words = ['foo', 'bar']
genre_word = [(genre, word) for genre in genres for word in
        words]
```

```

genre_word = [(genre, word)
               for genre in ['news', 'romance']
               for word in brown.words(categories=genre)]
cfd = nltk.ConditionalFreqDist(genre_word)
cfd['romance']['could']
cfd['news']['could']

```

10.3.2 Plotting and tabulating

```

# now we create pairs of word lengths for news and romance
genre_word_length = [(genre, len(word))
                     for genre in ['news', 'romance']
                     for word in brown.words(categories=genre)]
cfd = nltk.ConditionalFreqDist(genre_word_length)
cfd.plot(cumulative=True)
cfd.tabulate(samples=range(5,16))

```

10.3.3 Generating random text

```

def generate_model(cfdist, word, num=15):
    for i in range(num):
        print word,
        word = cfdist[word].max()

text = nltk.corpus.genesis.words('english-kjv.txt')
bigrams = nltk.bigrams(text)
cfd = nltk.ConditionalFreqDist(bigrams)

print cfd['living']
generate_model(cfd, 'living')

```

Chapter 11

Reusing code and wordlists

11.1 Reusing code

11.1.1 Scripts / Programs

Scripts / Programs

You can save code into a script file in order to reuse code more.

- Always start with the shbang line `#!/usr/bin/env python`
- Always put in general comments
- Make your script file executable by typing (from BASH): `$ chmod a+x myscript.py`
- Run the script by typing its path

absolute `$ /home/robfelty/ling5200/myscript.py`

relative `$ pwd /home/robfelty/ling5200 $ ./myscript.py`

11.1.2 functions

Definition 11.1:

A function is like a mini-program. It usually takes some *parameters* (sometimes called *arguments* and *returns* a value.

Scope

Variables inside a function cannot be used outside the function. These are called *local* variables

Example 11.1: the hello function

```
def hello(name):  
    return('hello, ' + name)  
  
print hello('rob')
```

Functions vs. methods

Definition 11.2:

function can be used with *any* type of variable, and has the form
`functionname(variablename)`

method can only be used with a *certain* type of variable, and has the form
`variablename.methodname()`

Example 11.2:

```
foo = 'Foo'
nums = [75, 50, 78]
print len(foo)
print len(nums)
print foo.lower()
nums.sort()
print nums
```

Function Practice

Create a function *mean* which computes the mean of a list. It should take a list as an argument, and return a number.

11.1.3 Modules and Packages

Definition 11.3:

A *module* is simply a collection of variables, functions and other code which you can re-use.
A *package* is a collection of modules

Create your own module

- Save the mean function you wrote in a file called `ling5200.py`
- Import the module into an interactive python session

The python path

Definition 11.4:

Just like BASH, python has a path where it searches for modules. The first place it searches is the current directory

Example 11.3:

```
import os
print os.getcwd() # where am I?
os.chdir('ling5200') # cd to ling5200
import sys
print sys.path # What is my path?
# prepend ling5200 directory to my path
sys.path.insert(0, '/home/robfelty/ling5200')
```

11.2 Lexical Resources

Lexicon

Definition 11.5:

A *lexicon* is a collection of words or phrases with associated information such as parts of speech and sense definitions.

A lexicon has the following parts:

lemma Aka Headword

gloss The sense definition

11.2.1 Basic wordlist

- The words corpus contains a simple list of most English words.
- Among other things, it can be used to check spelling of words

11.2.2 Stoplist corpus

Definition 11.6:

A *stoplist* is a list of function words which we wish to ignore, since function words generally tell us little about the content of a text.

Example 11.4:

```
from nltk.corpus import stopwords
# Show all languages in the stopwords corpus
stopwords.fileids()
# Get English stopwords
eng_stop = stopwords.words('english')
```

Stoplist Practice

Using a list comprehension create a list of words from Moby Dick which does not contain stopwords.

11.2.3 Names corpus

The nltk also includes a corpus of about 8000 common first names.

Example 11.5:

```
from nltk.corpus import names
# Get all male names
male_names = names.words('male.txt')
```

Use the names corpus to find all female names which start with *S*, and which don't end in *ie*.

Now add a condition that the names should also not contain an *r*

11.2.4 Pronouncing dictionary

The NLTK also include the cmudict corpus, which contains phonetic transcriptions for many english words.

```
entries = nltk.corpus.cmudict.entries()
pprint(entries[:10])
for word, pron in entries:
    if len(pron) == 3:
        ph1, ph2, ph3 = pron
        if ph1 == 'P' and ph3 == 'T':
            print word, ph2,
```

11.2.5 Finding cognates

```
from nltk.corpus import swadesh
swadesh.fileids()
```

11.2.6 Automatically finding language families

Edit distance

Definition 11.7:

Edit distance (also referred to as *Levenshtein distance*) is the minimum number of additions, deletions, and substitutions to change one string into another

Example 11.6: *cats* — *spat*

1. *sats spat* (substitute *c* with *s*)
2. *spats spat* (insert *p* between *s* and *a*)
3. *spat spat* (delete final *s*)

String similarity**Definition 11.8:**

While *Edit distance* can be from 0 to infinity, it would be nice to have a bounded similarity metric.

```
def similarity(s1,s2):
    max_len = max([len(s1), len(s2)])
    mean_len = (len(s1) + len(s2))/2
    dist = nltk.edit_distance(s1,s2)
    return((max_len - dist) / mean_len)
```

Calculating language similarity

Now we can use our similarity function to calculate the mean similarity between different pairs of languages.

```
languages = ['en', 'de', 'nl', 'es', 'fr', 'pt', 'it']
similarities = {}
totalcomb = len(languages)
for lang1 in range(totalcomb):
    for lang2 in range(lang1+1,totalcomb):
        pairs = zip(swadesh.words(languages[lang1]),
                    swadesh.words(languages[lang2]))
        label = languages[lang1] + '2' + languages[lang2]
        if not similarities.has_key(label):
            similarities[label] = 0
        for (wd1,wd2) in pairs:
            similarities[label] += similarity(wd1,wd2)
        similarities[label] = similarities[label] / len(pairs)
```

Chapter 12

Semantic relations

12.1 Wordnet

12.1.1 Synonyms

Definition 12.1:

Words which have approximately the same meaning are *synonyms* of each other. We will call the set of these words a *synset*

Example 12.1:

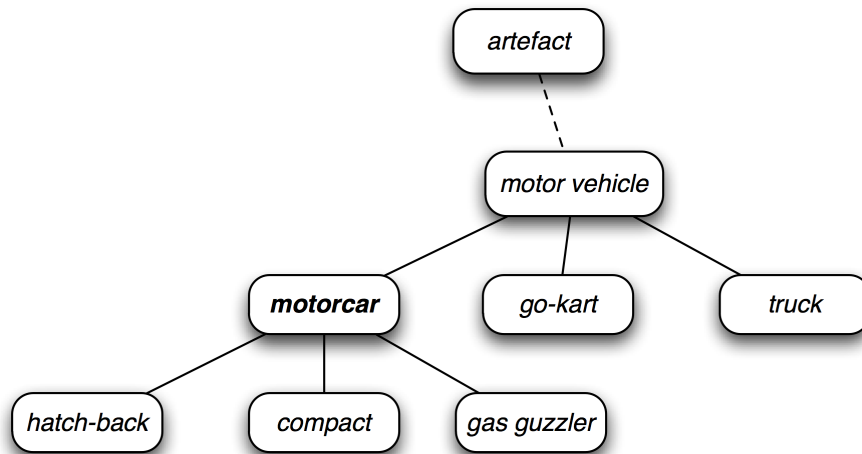
```
from nltk.corpus import wordnet
# show the synset for car
wordnet.synsets('car')
# show the words for the first definition
wordnet.synset('car.n.01').lemma_names
# show the words for the first definition
```

Synonym practice

1. Show the lemmas for the second definition of *car*
2. How many different senses of *dish* are there?
3. Test whether *dish* and *serve* are synonyms

12.1.2 Hierarchy

Wordnet organizes words into somewhat abstract categories



Definition 12.2:

A *hypernym* is a more general (superordinate) concept. A *hyponym* is a more specific (subordinate) concept.

Example 12.2: Finding hyponyms and hypernyms

```

motorcar = wordnet.synset('car.n.01')
types_of_motorcar = motorcar.hyponyms()
types_of_motorcar[26]

```

1. Print out all the lemma names from the hyponyms of motorcar

12.2 Projects

12.2.1 More details

Final projects

Time to start thinking about final projects

- Final project proposals due November 6th (about 4 weeks)
 - 1–2 pages
 - General description of project
 - List any resources you plan to use (e.g. databases, python modules)
- More details under resources > project, including
 - Project guidelines
 - Sample code
 - Sample presentation

12.2.2 Available resources

Martha Palmer will talk about available corpora, databases etc. in the linguistics department

Chapter 13

Processing raw text

13.1 Accessing text from the web and disk

13.1.1 Retrieving text from the web

- Python makes it very easy to read files from the web

Example 13.1: Read *Crime and Punishment* [gutenberg.org](http://www.gutenberg.org)

```
from urllib import urlopen
url = "http://www.gutenberg.org/files/2554/2554.txt"
raw = urlopen(url).read()
# tokenize the text
crime1 = nltk.word_tokenize(raw)
```

13.1.2 Dealing with html

- HTML has a bunch of tags which we are not interested in
- The NLTK has handy functions to remove these tags

Example 13.2: Strip out html tags

```
url = "http://www.gutenberg.org/files/2554/2554-h/2554-h.htm"
rawhtml = urlopen(url).read()
raw = nltk.clean_html(rawhtml)
crime2 = nltk.word_tokenize(raw)
```

13.1.3 Reading rss feeds

Definition 13.1:

RSS, known as *really simple syndication* is a file format for web communication, allowing rss readers to get new updates from websites

Example 13.3: Reading the ling5200 course rss feed

```
import feedparser
ling5200 =
    feedparser.parse("http://robfelty.com/teaching/ling5200Fall2009/feed")
len(ling5200.entries)
# read the latest entry
latest = ling5200.entries[0]
print latest.content[0].value
```

RSS practice

1. Create a list with the content of each entry from the ling5200 rss feed
2. strip out the html from each post and store as a new list
3. Tokenize each post

13.1.4 Opening local files

A note on paths

- For the homework problems, please use relative paths.
- Make them relative to the root of your ling5200 working copy

Example 13.4:

```
Reading in the Devil's dictionary
f=open('resources/texts/devilsDictionary.txt')
text = f.read()
```

Practice opening local files

1. Read the celex file into a string
2. Tokenize the devil's dictionary

13.1.5 Handling delimited text

Reading delimited text

- A common programming task is to read spreadsheet-like files.
- These are frequently tab-delimited or comma separated (csv) files.
- We can read these in python, and store them as a list of lists

Example 13.5: read the celex file into a list of lists

```
f=open('resources/texts/celex.txt')
celex = []
for line in f:
    line = line.rstrip('\n')
    fields = line.split('\\')
    celex.append(fields)
```

Dealing with lists of lists

Example 13.6: Accessing data in list of lists

```
# Print the 100th item of the celex
print celex[99]
# Print the orthography of the 10th item in the celex
print celex[9][1]
```

Example 13.7: Invert list of lists

```
# We invert the celex list of lists to make it accessible by
  [col][row]
celex_col_row = []
for i in xrange(len(celex[0])):
    celex_col_row.append([fields[i] for fields in celex])
```

Using numpy arrays for delimited text

- Using a of lists, we can access a row at once, but not a column
- The Numpy module allows us to do this

Example 13.8: Celex as a numpy array

```
import numpy
# Convert list of list to numpy array
celex_array = numpy.array(celex)
# Access the orthography of the first item
celex_array[0,1]
# Access the orthography of the first ten items
celex_array[0:10,1]
```

13.2 Using python to interact with the operating system

13.2.1 Using stdin and stdout

Example 13.9: Read stdin into a string

```
import sys
text = sys.stdin.read()
```

13.2.2 Handling command line arguments

Example 13.10: print out all command line arguments

```
import sys
for arg in sys.argv:
    print arg
```

argv[0]

The first argument is always the name of the file

13.2.3 Handling command line options

Example 13.11: Add three options

```
import sys, getopt
opts, args = getopt.gnu_getopt(sys.argv[1:], "ho:v", ["help",
    "output="])
```

Chapter 14

Handling strings

14.1 String basics

14.1.1 Creating and concatenating strings

Printing and internal representation

- In the interactive interpreter, when you simply type a variable, python prints out the internal representation

Example 14.1: Printing vs. Internal representation

```
foo = 'hello\nworld'  
foo  
print(foo)
```

Quotes and special characters

- You can use either single or double quotes to mark a string in python
- If you want to make a string which contains a quote, you can escape it with \

Example 14.2: single and double quotes

```
bar = "John's variable"  
aquote = 'he said "hello"'  
complex = "I need both single ' and double \" quotes"
```

Long strings

- If you have a particularly long string, you can put it in triple quotes
- Either `'''` or `"""` works fine

Example 14.3: Multi-line strings

```
multiline = ''' this is a multiple line string
when I print it out, newlines and whitespace are observed
    like so
I don't need to worry about escaping strings in a multiline
    string, unless I
want to print three in a row, like so: \'\'\'
'''
```

Concatenation

- Use `+` to combine strings
- Use `*` to repeat strings
- You can also print out multiple variables at one time, separate by a comma.
 - Note that this automatically inserts a space between them

Example 14.4: Combining strings

```
foobar = foo + bar
foofoo = foo * 2
print foo, bar
```

14.1.2 Accessing substrings

Example 14.5: Accessing part of a string

```
foo = 'hello, world'
print foo[0] #just the first character
print foo[-1] #just the last character
print foo[:3] #the first 3 characters
print foo[-3:] #the last 3 characters
```

Basic string practice

1. Store the string *John likes Mary's dog.* in *sent1*
2. Store the string *Mary said: "I like John."* in *sent2*
3. Combine the two strings with space as *sents*

14.1.3 width and precision

Definition 14.1:

- You can define how you want numbers to be printed out.
- Python's string formatting uses the same style as C.
- Formats follow the form: *form % variable*, where *form* is a string with formatting rules such as '%s'

Example 14.6: Commonly used string formatting

```
bar = 36.48903948
one = 1.0001
ten = 10
print '%s' % foo #foo is a string
print 'bar: %.2f' % bar #bar is a float. We only care about 2
    digits after the decimal point
```

Precision practice

1. Print out pi to the 3rd decimal place, with a width of 7
2. Print pi times 100 in the same fashion

14.1.4 alignment

- By specifying width, we can ensure that data will be correctly aligned

Example 14.7: Aligning by the decimal point

```
print '%7.2f' % bar #Now we print a total of 7 characters,
    with only 2 digits after the decimal point
print '%7.2f' % one
```

Example 14.8: Zero padding

```
print '%03.0f' % one #we want all our numbers to have 3
    digits, so we zero pad
```

1. Print out 23 padded with zeros to make it 4 wide
2. Print out 456.7 with a width of 10, and precision of 0

14.1.5 Strings and tuples

- It is possible to format several strings at once

Example 14.9:

You can format tuples all at once, e.g. `from math import pi, e '%s: %4.2f' % ('pi', pi)`

practice

1. Print out e and π , with appropriate labels, as in the preceding example, using a tuple

14.2 String methods

- There are a number of methods which can be performed with strings
- To see them all, type: `help(str)`

Strings are immutable

- Performing a method on a string does not change the string

Example 14.10: Strings are immutable

```
line = 'this is a line\n'
line.strip()
line
```

14.2.1 find

Definition 14.2:

- The *find* method returns the index of the first occurrence of a string, or -1 if it is not found
- Use *rfind* to find the last occurrence

practice

1. Find where *in* starts in the phrase *needle in a haystack*
2. If *needle in a haystack* contains *hay*, print *hey*

14.2.2 join / split

Definition 14.3:

- *split(sep)* splits a string into a list, using *sep* as the separator. The default is to split by whitespace.
- *sep.join(string)* joins a list of strings by *sep*.

practice

1. Split the haystack phrase into multiple words
2. Reverse the order of the words
3. Join the words back together with commas

14.2.3 Differences between lists and strings

- Lists and strings share some capabilities, e.g. index and slicing
- Some methods can only be used with strings, and some with lists
- You cannot combine strings and lists

Example 14.11: Strings vs. lists

```
# combine string and list
greeting = 'hello, '
names = ['John', 'Mary']
salutation = greeting + " and ".join(names)
print salutation
```

14.2.4 lower**Changing case****Definition 14.4:**

- The *lower()* method changes a string to lower case
- The *upper()* method changes a string to upper case
- The *title()* method changes a string to title case

practice

1. Make *ALLCAPS* all lowercase
2. Change *ALLCAPS* to title case

14.2.5 replace

Definition 14.5:

- The *replace(a, b)* method replaces all occurrences of *a* with *b*
- It does not accept regular expressions

practice

1. Replace *needle* with *noodle* in the haystack phrase What is the value of phrase now?
2. Replace *e* with *o* in the haystack phrase

14.2.6 strip**Strip****Definition 14.6:**

The *strip(chars)* removes *chars* from a string. The default is whitespace

practice

1. Strip off newline characters from end of the *haystack*
2. Strip off whitespace from *haystack*, and convert to upper case
3. Strip off whitespace from *haystack*, replace *needle* with *noodle* and convert to upper case

14.2.7 translate**Translate****Definition 14.7:**

Translate can be used to map an entire character set to a different one, using a one-to-one mapping.

Example 14.12:

I accidentally switch my keyboard to German mode, in which y and z are switched, and type my entire thesis this way without noticing.

```
thesis='verbositz in verbaliying is uglz'  
from string import maketrans  
germanToEnglish=maketrans('yz','zy')  
thesis.translate(germanToEnglish)
```

Chapter 15

Unicode and regular expressions in python

15.1 Handling unicode

15.1.1 Fonts, glyphs, and encodings

Fonts, glyphs, encodings, and code points

Definition 15.1:

There are 4 distinct parts of displaying characters

code point A numerical value representing a particular character, e.g. in ASCII 65 = A

encoding A set of code points to represent language(s), e.g. ASCII, ISO-8859-1, UTF-8

glyph An image which corresponds to a particular code point, e.g. A and **A** are different glyphs corresponding to the character *a*

font A mapping from code points to glyphs, e.g. Times New Roman, Helvetica

15.1.2 Unicode basics

What is unicode?

- Unicode is a set of character representations, which can represent every character in every language.
- There are several different character encodings to represent unicode, including UTF-8 and UTF-16. UTF-8 is currently most preferable
- Most versions of python are built to handle the first 65635 unicode characters. You shouldn't need to worry about this unless you are working with dead languages like Gothic or Homeric Greek

Entering unicode

- Unicode characters are represented as 4 hexadecimal digits

Example 15.1: Create some unicode characters

```
nacute = u'\u0144'  
print nacute  
repr(nacute)  
hello = 'hello' + nacute  
type(hello) #unicode
```

15.1.3 Reading non-ascii files

- If you know the encoding of a file, you can specify it when reading in a file
- Once the file is read in, python will treat the characters as unicode internally

Example 15.2: Reading non-ascii file

```
path = nltk.data.find(  
    'corpora/unicode_samples/polish-lat2.txt')  
import codecs  
f = codecs.open(path.path, encoding='latin2')
```

15.2 Using regular expressions in python

15.2.1 The re module

The re module

- The re module provides support for regular expressions in python
- Common methods:

search perform a regex search

match perform a regex search from the beginning of a string

sub Substitute - default is globally, optionally you can specify count

split Split a string using regex

compile Pre-compile a regex (not necessary, but can boost speed)

Flags

Flags you can use with match and search: See the python re documentation for all flags

re.I ignore case

re.M Multiline $\wedge$ matches beginning of string and newlines.

A simple search

Example 15.3: Simple regex search

```
foo = 'a cool string which is also HOT'
found = re.search(r'(hot|cold)', foo, flags=re.I)
found.groups()
```

Example 15.4: Finding geminate stops

A geminate is a doubly-long consonant.

```
from nltk.corpus import gutenberg
emma = gutenberg.words('austen-emma.txt')
gemminates = [word for word in emma if
               re.search(r'([ptkbdg])\1', word.lower())]
```

Regex search practice

1. Find all words in emma which contain any of the letters x, j, q, z
2. Find all words in emma which a q not followed by a u

15.2.2 Compiling regular expressions

- Compiling a regular expression, especially when looping over a long list, can speed up your script quite a bit

Example 15.5: Compiling a regular expression

```
# Let's replace a with e if it is not at the beginning or end
of a word
pattern = re.compile(r'(hot|cold)', flags=re.I)
re.search(pattern, foo)
```

15.2.3 Substitution

Example 15.6: Replacing word internal *a*

```
phrase = 'needle in a haystack'  
pattern = re.compile(r'(\S+?)a(\S+?)')  
newphrase = re.sub(pattern, r'\1e\2', phrase)
```

More regex practice

1. Split the raw text of Emma by newline characters (including carriage returns)
2. Replace *hot* or *cold* in *foo* with *lukewarm* (ignoring case) and store as *bar*

Chapter 16

Tokenizing and normalization

16.1 Miscellaneous notes on svn and stdin

16.1.1 Executable permissions with svn

Subversion has a way to store executable flags. If you add a file to your working copy which is executable by everyone, svn should automatically add the `svn:executable` property. You can check if it has done this

```
touch newfile.py
chmod a+x newfile.py
svn add newfile.py
svn propget 'svn:executable' newfile.py #should return *
# If it returns nothing, you can add it explicitly
svn propset 'svn:executable' '*' newfile.py #should return *
```

16.1.2 Working with stdin

- By default, stdout writes to the screen
- By default, stdin reads from the screen (until the end of file character is found, which is normally ctrl-d, (possibly ctrl-z in cygwin))
- To redirect stdin to read from a file, use `<` (in BASH)
- Generally one does not read from stdin in the interactive interpreter

16.2 Applications of regular expressions

16.2.1 Working with word pieces

The `re.findall()` method returns all non-overlapping matches from a string

Example 16.1: Counting “vowels”

```
word = 'mendacious'
vowels = re.findall(r'aeiou', word)
num_vowels = len(vowels)
```

Example 16.2: Counting real vowels

```
from nltk.corpus import cmudict
cmu = cmudict.dict()
word = cmu['mendacious']
word_str = ' '.join(word[0])
vowels = re.findall(r'[012]', word_str)
num_vowels = len(vowels)
```

Example 16.3: Count all CV pairs in rotokas

```
rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
cvs = [cv for w in rotokas_words for cv in
       re.findall(r'[ptksvr][aeiou]', w)]
cfd = nltk.ConditionalFreqDist(cvs)
cfd.tabulate()
```

Applied regular expression practice

1. Find all words in rotokas which have long vowels (a long vowel is represented as 2 of the same vowel, e.g. *uuu*)
2. Find the mean number of vowels per word in the cmudict

Example 16.4: Building an index

```
cv_word_pairs = [(cv, w) for w in rotokas_words
                  for cv in re.findall(r'[ptksvr][aeiou]', w)]
cv_index = nltk.Index(cv_word_pairs)
cv_index['su']
cv_index['po']
```

16.2.2 Finding word stems

subpatterns

?: can be used to indicate a subpattern without saving it, e.g. (?:ing|ly)

Example 16.5: Simple suffix stripping

```
re.findall(r'^.*(?:ing|ly|ed|ious|ies|ive|es|s|ment)$',  
          'processing')
```

Example 16.6: Getting both stem and suffix

```
# this will return a tuple with the result from both groupings  
re.findall(r'^(*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$',  
          'processing')
```

Stemming practice

1. Using the previous regular expression, find all words from the cmudict which have suffixes

16.2.3 Searching tokenized text

- The NLTK provides some specialized methods for searching tokenized text.
- Angle brackets <> are used to mark tokens

Example 16.7: Finding adjectives

```
# Let's find all the adjectives used to describe man  
from nltk.corpus import gutenbergl, nps_chat  
moby = nltk.Text(gutenberg.words('melville-moby_dick.txt'))  
moby.findall(r"<a> (<.*>) <man>")  
# better yet  
moby.findall(r"<a|the> (<.*>) <m[ae]n>") # returns just  
    grouping  
moby.findall(r"<a|the> <.*> <m[ae]n>") # returns entire phrase
```

16.3 Normalization

Definition 16.1:

Normalization is the process of modifying raw text to yield linguistically relevant information, including:

- converting to all lower case
- stripping affixes (stemming)
- checking that the words are in a dictionary (lemmatization)

16.3.1 stemmers

- The NLTK includes several different stemming algorithms.
- None of them are perfect.
- You should choose one based on your data, and the questions you are interested in.

Example 16.8: Using stemmers

```
porter = nltk.PorterStemmer()
lancaster = nltk.LancasterStemmer()
aquatic = grail[1818:1856]
[porters.stem(t) for t in aquatic]
[lancaster.stem(t) for t in aquatic]
```

16.3.2 Lemmatization

- The wordnet module includes a lemmatizer, which checks that each word is a valid word
- What are the differences between the output of the wordnet lemmatizer compared to the porter and lancaster stemmers?

Example 16.9: Using the wordnet lemmatizer

```
wnl = nltk.WordNetLemmatizer()
[wnl.lemmatize(t) for t in aquatic]
```

16.4 Regular expressions for tokenization

16.4.1 Simple approaches to tokenization

Example 16.10: Using re.split to tokenize

```

raw = """'When I'M a Duchess,' she said to herself, (not in a very
hopeful tone though), 'I won't have any pepper in my kitchen AT ALL.
Soup does very well without-Maybe it's always pepper that makes people
hot-tempered,'..."""
# split by space
re.split(r'\s+', raw)
# split by non-word characters
re.split(r'\W+', raw)
# match words instead of spaces
re.findall(r'\w+|\S\w*', raw)
# allow hyphens and apostrophes
re.findall(r"\w+(?:[-']\w+)*|'|[-.()+|\S\w*", raw)

```

16.4.2 Using the nltk tools for tokenization

The nltk *regexp_tokenize* function is a bit faster and easier to use than *re.findall*

Example 16.11: Using the nltk regexp tokenizer

```

text = 'That U.S.A. poster-print costs $12.40...'
pattern = r'''(?x)      # verbose regexps
    ([A-Z]\.)+          # abbreviations
    | \w+(-\w+)*         # optional internal hyphens
    | \$?\d+(\.\d+)?%?   # currency and percentages
    | \.\.\.            # ellipsis
    | [[. , ; ' ' ? ( ) : - _ `] # these are separate tokens
'''
nltk.regexp_tokenize(text, pattern)

```

Chapter 17

More on strings and shell integration

17.1 Homework 7 notes

General tips

- From now on, we will mostly be writing scripts, not using the interactive interpreter
- You still may want to use the interactive interpreter for testing small snippets of code
- Continually re-write and improve your code.
 - Look over my solutions in depth
 - If you don't understand something, please ask.
 - Feel free to copy from my solutions (e.g. update the *mean_word_len* function to my definition)
- Model the behavior of your scripts on other unix programs like *wc* and *grep*
 - Make your help function print out information in a similar way to the man pages of *wc* and *grep*
 - By default, your program should print out both word and sentence length means (like *wc* prints out line, word, and byte counts by default)

17.1.1 Reading files

- A common error in homework 7 was to concatenate files
- This might be the desired effect for some cases, but not in this homework

Example 17.1: Concatenating files

```
text=''
for file in sys.argv[1:]:
    f = file.open()
    text += f.read()
process(text)
```

Example 17.2: One file at a time

```
for file in sys.argv[1:]:
    f = file.open()
    text = f.read()
    process(text)
```

17.1.2 Handling options

Specifying options

- The *gnu_getopt* function returns command line options and arguments separately
- It is possible for the options to also have arguments.
 - E.g. the `-f` option for the `cut` command requires an argument.
 - To specify an option as requiring an argument, use a colon after the short option, and an equals sign after the long option
- The *gnu_getopt* function allows you to mix order of options and arguments. The following are all equivalent:
`./hmk7.py -s ../texts/tomSawyer.txt -w`
`./hmk7.py -s -w ../texts/tomSawyer.txt`
`./hmk7.py -sw ../texts/tomSawyer.txt`

Processing options

- It is a good idea to associate a variable with each option. For yes/no options, use a boolean variable (True/False)
- Before processing the options, set some default settings

Processing options

Example 17.3: handling default options

```
sent = False
word = False
if len(opts) == 0:
    sent = True
    word = True
for o, a in opts:
    if o in ("-h", "--help"):
        usage(sys.argv[0])
```

```

        sys.exit()
    if o in ("-s", "--sent"):
        sent = True
    if o in ("-w", "--word"):
        word = True

```

17.2 More on string formatting

17.2.1 More on alignment

- To align a string to the left, preface it with a hyphen
- You can use a variable to specify the width of string by using an asterisk

Example 17.4: Left aligning strings with variable width

```

labels = ['huckFinn', 'devilsDictionary']
mean_word = [4.5698560, 4.798790]
mean_sent = [15.689930, 16.494493098]
width = max(len(label) for label in labels)
for filename, word_len, sent_len in zip(labels, mean_word,
    mean_sent):
    print '%-*s %13.2f %13.2f' % (width, filename, word_len,
        sent_len)

```

17.2.2 Wrapping text

Example 17.5: Joining and wrapping text

```

# Print out random words from Alice in Wonderland
import random
from textwrap import fill
alice = nltk.corpus.gutenberg.raw('carroll-alice.txt').split()
random_alice = []
for i in xrange(100):
    random_alice.append(random.choice(alice))
joined = ''.join(random_alice)
wrapped = fill(joined)
print wrapped

```

17.2.3 Printing vs. writing

- There are two ways to print to stdout
 - `print 'foo'` (automatically inserts newline)
 - `sys.stdout.write('foo\n')` (Does not automatically insert newline)
- The latter method is more similar to printing to a file

Example 17.6: Printing to a file

```
filename = 'text.txt'
f = open(filename, 'w') # w is for writing
f.write('hello world\n')
f.close()
```

17.3 Shell integration

17.3.1 Calling python scripts from bash scripts

We can create a BASH script which calls a python script

Example 17.7: Calling a python script from BASH

```
#!/bin/bash
./hmk7.py ../texts/{huckFinn,tomSawyer}.txt > stats.txt
```

For a slightly more complicated example, see `text_stats`, under `resources/py`

Integration practice

1. Create a bash script which calls `text_means.py` for all files in the `resources/texts` directory which start with the letter *d*
2. Create a bash script which calls `randomExcerpt.py` on `huckFinn.txt` and `tomSawyer.txt`. Use the `-l` option to specify lengths of 100, 1000, 10000, and 100000.
 - Store the output in files called `huckFinn.500`, `tomSawyer.1000` etc.
 - Now use `-t` option of `text_means.py` to calculate the type / token ratio for each of these files

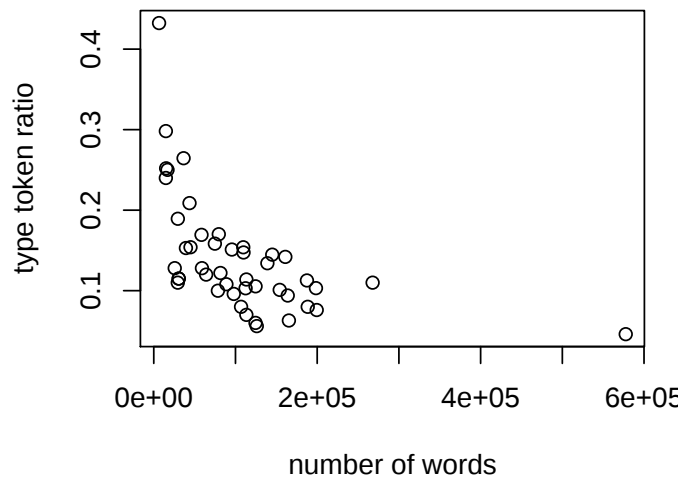


Figure 17.1: Type-token ratio as a function of text length

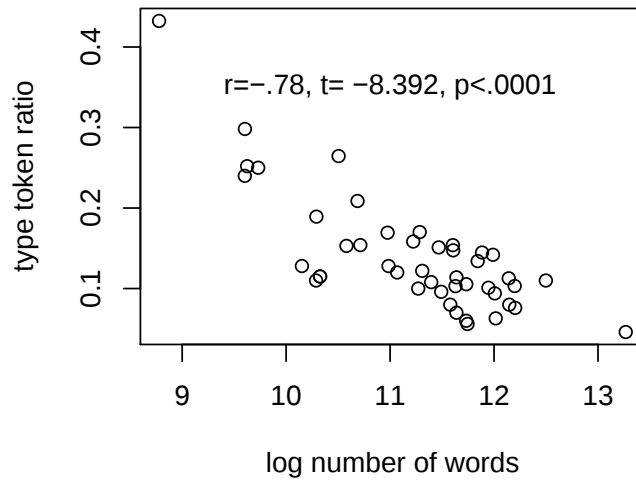


Figure 17.2: Type-token ratio as a function of log text length

17.3.2 Type token ratio

As you can see from Figures 17.1 and 17.2, the type-token ratio is highly correlated with the length of the text. For this reason, if you wish to compare the type-token ratio of multiple texts, it is best to select a random subset of each text of the same length. There are also more complicated solutions. For more information, see Baayen 2008.

Chapter 18

Back to basics

18.1 Homework 8 notes

18.1.1 An error with my homework 7 solution

An error with my homework 7 solution

- `nltk.gutenberg.sents('austen-emma.txt')` returns a list of *lists*
- `nltk.sent_tokenize()` returns a list of *strings*
- I was fooled into thinking it also returned a list of lists
- Thanks to Matt for making me aware of this
- I have corrected this in the solution in *hmk7.py* and *text_means.py*

18.1.2 Handling options

More on handling options

- With options that interact with each other, there are $\sum_1^k \frac{n!}{k!(n-k)!}$ possible combinations of output options

1 option 1

2 options 3

3 options 7

4 options 15

5 options 31

- We don't always need to consider all these combinations

More on handling options

Example 18.1: different headers based on options

```
if showheader:
    headers=['filename']
    if showword:
        headers.append('mean_word_len')
    if showsent:
        headers.append('mean_sent_len')
    if showari:
        headers.append('ari')
    if showadj:
        headers.append('adjectiv_verbs')
    format_string = '%-17s ' + '%13s ' * (len(headers)-1)
    print format_string % tuple(headers)

if showsent:
    mean_sent_length = mean_sent_len(sents)
    mean_sent_print = '%13.2f' % mean_sent_length
else: mean_sent_print = ''
if showword:
    mean_word_length = mean_word_len(words)
    mean_word_print = '%13.2f' % mean_word_length
else: mean_word_print= ''
if showari:
    ari = '%13.2f' % calc_ari(mean_sent_length, mean_word_length)
else: ari=''
if showadj:
    adjs = verb_adjectives(words)
    mean_adjs = '%13.3f' % (len(adjs) / len(sents))
else: mean_adjs=''
return '%s %s %s %s' % (mean_word_print,
                        mean_sent_print, ari, mean_adjs)
```

18.2 Sequences (Collections)

Types of sequences

Python has four different types of sequences, each of which has a combination of the following features

ordered Has the same order that you created it with

mutable Can be changed after creating it

hashable Can be used as a dictionary key

Table 18.1: Properties of different collections in python

	ordered	mutable	hashable	indexable	unique	iterable
list	✓	✓	✗	int	✗	✓
tuple	✓	✗	✓	int	✗	✓
dict	✗	✓	✓	key	✗	✓
set	✗	set	frozenset	✗	✓	✓

indexable Can access slices or individual items

iterable Can loop over it

18.2.1 Uses for tuples

Example 18.2: splitting a text

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

- Another use of tuples is for returning multiple values from a function.
- These values can then be easily unpacked.

Example 18.3: multiple return values

```
def mult(x,y):
    return(x,x+y)

a,b = mult(5,3)
```

- Use a tuple when the order of a sequence should be fixed, e.g. a tagged corpus
- Use a list when you might want to change the order. For example, you might want to sort a list of tokens

18.2.2 More about dictionaries

has_key() check if a key exists in the dictionary

keys() returns only the keys of the dictionary

values() returns only the values of the dictionary

items() returns both the keys and values of the dictionary

Example 18.4: calculating word frequency

```
text = nltk.corpus.nps_chat.words()
freq = {}
for word in text:
    if freq.has_key(word):
        freq[word] += 1
    else:
        freq[word] = 1
```

18.2.3 More about sets

Set uses

- Useful for finding unique values
- Sets are also useful for efficiently testing membership
 - For efficiency differences, see *spell_check.py*

Example 18.5: spell-checking

```
words = set(nltk.corpus.words.words())
>>> 'the' in words
True
>>> 'sdf' in words
False
```

18.2.4 Converting between different types of sequences

Simple conversions

Example 18.6: simple sequence conversions

```
list1 = [1,1,10]
list2 = ['hi', 'bye', 'aloha']
dict1 = zip(list2,list1)
tuple1 = tuple(list1)
tuple2 = tuple(dict1)
set1 = set(list1)
```

Converting dict to list of tuples

Example 18.7: sorting word frequency

```
>>> freqlist = list(freq.items())
>>> freqlist.sort()
>>> freqlist[4]
('!!!!', 18)
>>> freqlist = zip(freq.values(), freq.keys())
>>> freqlist.sort(reverse=True)
>>> freqlist[4]
(704, 'lol')
```

18.2.5 Iterating over sequences

- By default, iterating over a dict uses the keys
- It is also possible to iterate over values, or both keys and values

Example 18.8: iterating over a dict

```
for key in dict1: #keys
    print key
for value in dict1.values(): #values
    print value
for key, value in dict1.items(): #keys and values
    print key, value
```

Example 18.9: iterating over a list of tuples

We can loop over a list of tuples in a similar fashion to looping over the items in a dictionary

```
list12 = zip(list1, list2)
for x, y in list12:
    print x, y
```

18.2.6 Generator expressions

- Generator expressions are similar to list comprehension, but do not create a list
- They are useful when wanting to compute a single value, such as a sum, max, or min
- They are much more efficient

Example 18.10: a simple generator expression

```
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."'''  
# compute the longest word  
max(len(w) for w in nltk.word_tokenize(text))
```

18.3 Style

18.3.1 Procedural vs. declarative style

Procedural similar to what the computer does

Declarative similar to a human description

Example 18.11: procedural style code

```
tokens = nltk.corpus.brown.words(categories='news')  
count = 0  
total = 0  
for token in tokens:  
    count += 1  
    total += len(token)
```

Example 18.12: declarative style code

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

18.3.2 Using counters

- The enumerate function loops over a sequence using both an index and a value.
- Using the enumerate function makes the code slightly more readable

Example 18.13: looping using enumerate()

```
fd = nltk.FreqDist(nltk.corpus.brown.words())
cumulative = 0.0
for rank, word in enumerate(fd):
    cumulative += fd[word] * 100 / fd.N()
    print "%3d %6.2f%% %s" % (rank+1, cumulative, word)
    if cumulative > 25:
        break
```

Chapter 19

More on functions

19.1 Function basics

19.1.1 Variable scope

- variables created inside a function are local to that function. The program doesn't know anything about them outside the function
- Inside a function, python searches for local variables, but if the variable is not found, it then searches for variables in the body of the program
- In general, avoid using global variables inside functions, because:
 - It is difficult to read, because the variable definitions are very far from their use
 - It makes the function less flexible

Example 19.1: bad use of global variables

```
greeting = 'hello, '  
def greet(person):  
    return(greeting + person)
```

Example 19.2: passing a parameter instead of using a global variable

```
def greet(greeting, person):  
    return('%s, %s' % (greeting, person))  
  
greetings = ['hello', 'servus', 'guten tag', 'hola']  
for greeting in greetings:  
    print greet(greeting, 'rob')
```

19.1.2 Designing effective algorithms

- When creating a program, try to think of the problem in a very high-level way, and write out a basic algorithm.
- This is sometimes referred to as *pseudo-code*

Example 19.3: Algorithm for finding novel words from a wikipedia page

1. Download web page
2. Do some preprocessing (strip out html, get rid of bibliography)
3. Remove known words
4. Print out results

19.1.3 Refactoring code

Definition 19.1:

Refactoring is the process of re-writing code, to:

- improve readability
- increase flexibility
- increase efficiency

What should a function look like?

- 10-20 lines
- Should accomplish one purpose

19.1.4 Side effects

Definition 19.2:

A *side effect* is when a function modifies something outside of its scope, e.g. it prints out something, or it modifies a global variable. In general, *side effects are bad!*

19.2 Advanced function usage

19.2.1 Named parameters

- Python allows us to give names to parameters
- These are also called *keyword arguments*
- This allows us to create more user-friendly and flexible functions, including:
 - default (and optional) values
 - flexible parameter ordering

- You can mix named and unnamed parameters, but unnamed parameters must precede all named parameters
- Unnamed parameters are always mandatory
- Avoid using mutable default values (lists and dictionaries)

Example 19.4: function with named parameters

```
def greet(greeting='hello', person='john'):
    return('%s, %s' % (greeting, person))

greeting = 'servus'
print greet(greeting, 'rob')
print greet(greeting=greeting, person='rob')
print greet(person='rob', greeting=greeting)
print greet(greeting)
print greet('rob')
print greet(person='rob')
```

- Python also allows you to use specify variable number of unnamed and named parameters, using * and ** respectively
- required arguments must come first, and unnamed arguments must come before named arguments

Example 19.5: variable length parameters

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

generic(1, 10, "African swallow", monty="python")
```

19.2.2 Higher order functions

- Higher order functions accept functions as arguments
- Two common higher order functions are *filter* and *map*
 - *filter* removes items from a collection (only retains items for which the function returns true)
 - *map* applies a function to every item in a collection
- You can create your own higher order functions if you want

Example 19.6: using filter

```
def is_content_word(word):
    eng_stopwords = nltk.corpus.stopwords.words('english')
    return word.lower() not in eng_stopwords and word not in
        string.punctuation

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

Example 19.7: using map

```
>>> lengths = map(len,
    nltk.corpus.brown.sents(categories='news'))
>>> sum(lengths) / len(lengths)
21.7508111616
>>> lengths = [len(w) for w in
    nltk.corpus.brown.sents(categories='news')]
>>> sum(lengths) / len(lengths)
```

Create your own higher-order function**Example 19.8: passing functions as arguments**

```
def extract_property(prop):
    return [prop(word) for word in sent]

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

extract_property(len)
extract_property(last_letter)
```

Chapter 20

Program development and debugging

20.1 Program development

20.1.1 More on creating modules

Finding example code

- One good way to learn to code is to look at other code
- You can easily view the source of any python module
- A .pyc file is compiled. There is always a corresponding .py file

Example 20.1: finding the source

```
>>> import re,nltk
>>> nltk.metrics.distance.__file__
'/usr/lib/python2.5/site-packages/nltk/metrics/distance.pyc'
>>> re.__file__
'/usr/lib/python2.6/re.pyc'
```

Special functions and variables

- Python allows you to create special “hidden” functions and variables, which can only be seen inside modules
- These are defined with a preceding underscore, e.g. `_helper()`
- If you want it be *really* hidden, use two underscores, e.g. `__very_hidden()`
- These functions will not be imported when using `from X import *`
- This is akin to private functions (methods, classes) in Java

Module + script magic

- Python makes it easy to use a .py file as both a module and a script

Example 20.2: using `__name__`

```
def greet(greeting='hello', person='john'):
    return('%s, %s' % (greeting, person))

def demo():
    greeting = 'servus'
    print greet(greeting, 'rob')

if __name__ == '__main__':
    demo()
```

Multi-module programs

- We have already been using several different modules in our programs, but we have not been combining our own modules yet
- I have re-factored homework 9 a bit to live in two modules, *wiki.py* and *novel.py*, both in the `resources/py` directory of the svn repository
- I have also updated *text_means.py* with some module magic.

Example 20.3: exploring modules

```
import novel
help(novel)
```

Multi-module practice

1. Run the demo function from the novel module
2. Find novel words in the Devil's Dictionary
3. Use the *text_means* module to compute the type-token ratio of the novel words from the Devil's dictionary

20.1.2 Debugging

Definition 20.1:

Debugging is the process of finding (and fixing) errors in your programs

- Python provides helpful information about errors by reporting the type of error and where it occurs in a *stack trace*
- The stack trace displays an error in a tree-like fashion. For example:

your main program called function X on line 19
which called function Y on line 150
which called function Z on line 200
which produced an error on line 13

Stack traces

Example 20.4: stack trace with error in function

```
>>> import novel
>>> novel.demo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "novel.py", line 83, in demo
    novel = unknown(tokens)
  File "novel.py", line 73, in unknown
    novel = filter(_is_unknown, first_pass)
  File "novel.py", line 39, in _is_unknown
    if worrd == '':
NameError: global name 'worrd' is not defined
```

In the case of Example 20.4, the error was simply a typo. Note that the error says that the *global* name is not defined. This is because python first searches for local variables, and then global variables. If neither exists, it returns an error

Stack traces

Example 20.5: stack trace with error in main

```
>>> import wiki
>>> url = ['http://en.wikipedia.org/wiki/Ben_franklin']
>>> text = wiki.get_wiki(url)
>>> text = wiki.get_wiki(url)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "wiki.py", line 15, in get_wiki
    req=Request(url=url, headers=headers)
  File "/usr/lib/python2.6/urllib2.py", line 190, in __init__
    self.__original = unwrap(url)
  File "/usr/lib/python2.6/urllib2.py", line 1030, in unwrap
    url = url.strip()
AttributeError: 'list' object has no attribute 'strip'
```

In the case of Example 20.5, the error was not actually in the function `get_wiki`, but rather in the input to the function. It was expecting a string, not a list.

Example 20.6: Example debugging session

```
>>> pdb.run('text = wiki.get_wiki(url)')
> <string>(1)<module>()->None
(Pdb) s
--Call--
> /home/robfelty/RobsDocs/teaching/ling5200/resources/py/wiki.py(5)get_wiki()
-> def get_wiki(url, clean=True, strip_refs=True, tokenize=False):
(Pdb) p clean
True
(Pdb) s
> /home/robfelty/RobsDocs/teaching/ling5200/resources/py/wiki.py(13)get_wiki()
-> headers = {'User-Agent': 'Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB;
(Pdb) p headers
*** NameError: NameError("name 'headers' is not defined",)
(Pdb) s
> /home/robfelty/RobsDocs/teaching/ling5200/resources/py/wiki.py(14)get_wiki()
-> rv:1.8.0.4) Gecko/20060508 Firefox/1.5.0.4 '''}
(Pdb) s
> /home/robfelty/RobsDocs/teaching/ling5200/resources/py/wiki.py(15)get_wiki()
-> req=Request(url=url, headers=headers)
(Pdb) p headers
{'User-Agent': 'Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; \n
rv:1.8.0.4) Gecko/20060508 Firefox/1.5.0.4'}
(Pdb) p url
['http://en.wikipedia.org/wiki/Ben_franklin']
(Pdb) type(url)
<type 'list'>
(Pdb) exit
>>>
```

20.2 Algorithm design

20.2.1 Recursion

Definition 20.2:

recursion — see recursion.

Definition 20.3:

recursion is a process repeating itself. In programming this is a function which calls itself.

- Every recursive function must have at least one base case, in which the function returns
- If there is no base case, it will repeat forever!

```

(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720

```

Example 20.7: computing the factorial

```

def factorial(n):
    if n == 1: return 1
    else: return n * factorial(n-1)

```

Factorial

Example 20.8: computing the factorial (verbose)

```

>>> def factorial(n):
...     if n == 1:
...         return 1
...     else:
...         total = n * factorial(n-1)
...         print n, total
...         return total
...
>>> factorial(5)
2 2
3 6
4 24
5 120
120

```

- One common use of recursion is to process and create tree-like structures

Example 20.9: finding number of hyponyms

```
def size1(s):
    return 1 + sum(size1(child) for child in s.hyponyms())

from nltk.corpus import wordnet as wn
dog = wn.synset('dog.n.01')
size1(dog)
```

Finding hyponyms elaborated

Example 20.10: finding number of hyponyms (verbose version)

```
def sizev(s):
    total = 1
    for child in s.hyponyms():
        thisnode = sizev(child)
        total += thisnode
        print '%3d %3d %-30s %s' % (thisnode, total,
                                   child.name, s.name)
    #print '%3d %s' % (total, s)
    return total

sizev(dog)
```

Finding hyponyms practice

Now create a new function which recursively returns all the hyponyms of dog, instead of just returning the number of hyponyms

Recursion vs. iteration

- Every recursive function can also be defined in an iterative way
- After the iterative definition executes faster
- The recursive definition is sometimes more intuitive and/or elegant

Chapter 21

A tour of python modules

21.1 Some useful modules

Installing modules

- There are many python modules available for free on the web
- Some modules are available via *easy_install*
- For example, try:
`easy_install networkx`
- For other modules you download from the internet, the standard procedure is:
 1. Unzip (or untar) the file
 2. cd to the directory
 3. `sudo python setup.py install`

21.1.1 Timeit

- The *Timer* object takes two arguments
 1. Setup code (run once) (in a string)
 2. Repeating code (in a string)
- The *timeit* method takes one argument - the number of times to repeat (default=1000000)

Example 21.1: timing list vs. set

```
from timeit import Timer
setup_list = "import random; vocab = range(1000000)"
setup_set = "import random; vocab = set(range(1000000))"
statement = "random.randint(0, 1000000) in vocab"
print Timer(statement, setup_list).timeit(1000)
print Timer(statement, setup_set).timeit(1000)
```

Timing practice

1. Time how long it takes to find the word 'leopard' in the words corpus, using (a) a list, and (b) a set

21.1.2 Profile

- The profile module is similar to timeit, in that it tells you how long parts of your code take to run
- In contrast though, the profile module will run your entire program, and give statistics about how many times a given function is called, and what percentage of the time your program is spending on each function
- This module can help you optimize your code.
- Usually every program has a few *bottlenecks*
- Knowing where these are helps you improve the speed of the code

Profiling our scripts

Example 21.2: profiling text_means

```
import profile
import text_means
profile.run(
    "text_means._main(['../texts/damnedThing.txt'])")
```

Example 21.3: profiling regexes

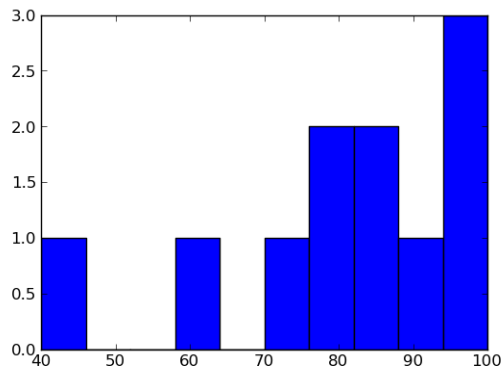
```
./sampleRegex.py
./sampleRegex.py -c
```

21.1.3 Matplotlib

Simple histogram

Example 21.4: a simple histogram of grades

```
import matplotlib.pyplot as plt
grades = [99,100,95,88,85,82,81,76,74,60,40]
n,bins,patches = plt.hist(grades)
```



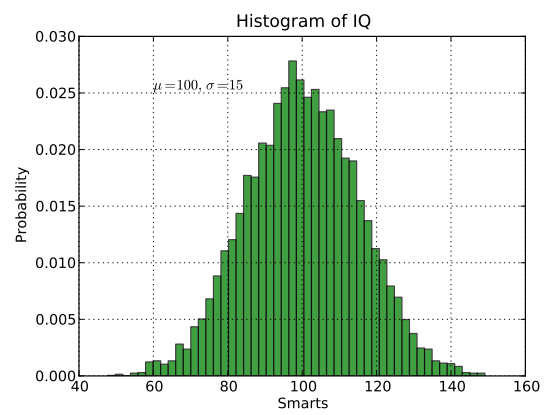
Customizing plots

- Matplotlib allows for all sorts of customizations of plots, including:
 - titles
 - axis labels
 - tick labels
 - custom text on the plot (can use $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ markup)
 - colors

Example 21.5: customized histogram

```
import numpy as np
import matplotlib.pyplot as plt
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g',
                             alpha=0.75)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
```

21.1.4 Numpy



Chapter 22

Part of speech tagging

22.1 Using a tagger

- Tagging requires the use of syntactic information
- Therefore taggers requires sentences (composed of a list of tokens)

Example 22.1: tagging a sentence

```
>>> text = nltk.word_tokenize("And now for something  
completely different")  
>>> nltk.pos_tag(text)  
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something',  
 'NN'),  
 ('completely', 'RB'), ('different', 'JJ')]
```

22.2 Using tagged corpora

Tag representations

- Tagged corpora generally represent each token as a word/tag pair, e.g. the/DT
- The NLTK represents a word/tag pair as a tuple

Example 22.2: convert tag strings to tuples

```
tagged_token = nltk.tag.str2tuple('fly/NN')
```

Reading in tagged corpora

- Any tagged corpus in the nltk has methods for getting word/tag tuples
- Not all corpora use the same tags, but the nltk also provides options to simplify the tags

Example 22.3: Getting tagged words from a corpus

```
brown_tagged = nltk.corpus.brown.tagged_words()
brown_tagged_simple =
    nltk.corpus.brown.tagged_words(simplify_tags=True)
```

Finding frequent tags

Example 22.4: finding common tags

```
from nltk.corpus import brown
brown_news_tagged = brown.tagged_words(categories='news',
                                       simplify_tags=True)
tag_fd = nltk.FreqDist(tag for (word, tag) in
                        brown_news_tagged)
tag_fd.keys()
```

22.2.1 Nouns

Definition 22.1:

Nouns generally refer to people, places, things or concepts

Example 22.5: finding syntactic context

```
word_tag_pairs = nltk.bigrams(brown_news_tagged)
before_nouns = nltk.FreqDist(a[1]
                             for (a, b) in word_tag_pairs if b[1] == 'N')
before_nouns.items()[:10]
```

22.2.2 Verbs

Definition 22.2:

Verbs describe actions and events, e.g. *run* and *think*

Example 22.6: finding common verbs

```
wsj = nltk.corpus.treebank.tagged_words(simplify_tags=True)
word_tag_fd = nltk.FreqDist(wsj)
common_verbs = [word + "/" + tag
                 for (word, tag) in word_tag_fd
                 if tag.startswith('V')]
```

1. Find the following context of nouns
2. Find the preceding context of verbs
3. Find the following context of verbs

22.3 Handling exceptions

A python mantra

It's easier to ask forgiveness later than permission first

- Notice that when python returns an error, it gives you a particular type of error.
 - E.g. `NameError: name 'pos3' is not defined`
- Sometimes we want to anticipate errors and handle them instead of having our program terminate
- This is called *exception handling*

Example 22.7: Optionally opening a file

```
file = '../texts/celex.txt'
try:
    f=open(file)
    celex=[]
    for line in f:
        fields = line.split('\\')
        celex.append(fields[1])
    f.close()
except IOError:
    pass # do nothing, i.e. ignore the error
```

Note that in Example 22.7, I would have had to check for several issues if I didn't use the try-except method. I would have to check that the file exists, and that it is readable. Using a try block I don't have to worry about this. Additionally, python is usually faster at exception handling than processing many different conditionals.

22.4 More with dictionaries

22.4.1 Indexing and inverting dictionaries

Building an index

- Building an index can help us speed-up lookups

Example 22.8: concordance without index

```
brown = nltk.corpus.brown.words()
term = 'boy'
width = 5
for index, word in enumerate(brown):
    if word == term:
        context = brown[index-width:index+width]
        print(" ".join(context))
```

Building an index

- Building an index can help us speed-up lookups

Example 22.9: concordance with index

```
brown_index = nltk.Index((word, index) for index, word in
    enumerate(brown))
for index in brown_index[term]:
    context = brown[index-width:index+width]
    print(" ".join(context))
```

Inverting a dictionary

- It is easy to find dictionary keys, but much more difficult to find dictionary values
- If we want to search a dictionary for values frequently, we might want to invert it

Example 22.10: Inverting a dictionary

```
>>> pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V',
    'furiously': 'ADV'}
>>> pos2 = dict((value, key) for (key, value) in pos.items())
>>> pos2['N']
'ideas'
```

There is a problem with this solution though. If we have non-unique values, then only the last value will be retained. Thus we are losing information, which is bad! Instead, we should consider an alternate approach storing the values in a list.

Example 22.11: Inverting a dictionary, take 2

```
pos = {'green': 'ADJ', 'colorless': 'ADJ',  
       'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}  
pos3 = {}  
for key, value in pos.items():  
    try:  
        pos3[value].append(key)  
    except KeyError:  
        pos3[value] = [key]
```

22.4.2 Default dictionaries

- Default dictionaries assign a default value when a key is accessed which has not been assigned yet
- They take one argument, which is what the default value should be
- int, str, list, dict, set, tuple

Example 22.12: frequency distribution using default dict

```
counts = nltk.defaultdict(int)  
from nltk.corpus import brown  
for (word, tag) in brown.tagged_words(categories='news'):  
    counts[tag] += 1
```

- Invert the pos dictionary using a default dictionary

Example 22.13: dictionary inversion using default dict

```
from collections import defaultdict  
pos3 = defaultdict(list)  
for key, value in pos.items():  
    pos3[value].append(key)
```

22.4.3 Sorting

Sorting dictionaries

- Dictionaries are not ordered, so we can't sort them
- We can manipulate them into a sorted list though

Example 22.14: Convert dict to sorted list

```
from operator import itemgetter
tags_counts = sorted(counts.items(), key=itemgetter(1),
                      reverse=True)
tags = [t for t, c in sorted(counts.items(),
                             key=itemgetter(1), reverse=True)]
```

Chapter 23

Automatic Part of speech tagging

23.1 Automatic tagging

23.1.1 Default tagger

- A default tagger tags every word with the same tag
- This may seem silly, but just using this step gets approximately 13% accuracy

Example 23.2: tagging a sentence

```
brown_tagged_sents = brown.tagged_sents(categories='news')
raw = '''I do not like green eggs and ham, I do not like them
      Sam I am!'''
tokens = nltk.word_tokenize(raw)
default_tagger = nltk.DefaultTagger('NN')
default_tagger.tag(tokens)
default_tagger.evaluate(brown_tagged_sents)
```

23.1.2 Regular expression tagging

- Another strategy for tagging is using regular expressions
- For example, we know that most words ending in *ing* are verbs in English

Example 23.3: Regex patterns for tagging

```
patterns = [
    (r'.*ing$', 'VBG'),          # gerunds
    (r'.*ed$', 'VBD'),          # simple past
    (r'.*es$', 'VBZ'),          # 3rd singular present
```

```

(r'.*ould$', 'MD'),          # modals
(r'.*\'s$', 'NN$'),          # possessive nouns
(r'.*s$', 'NNS'),            # plural nouns
(r'^-?[0-9]+(.[0-9]+)?$', 'CD'), # cardinal numbers
(r'.*', 'NN')                # nouns (default)
]

```

Example 23.4: Using a regular expression tagger

```

>>> brown_sents = brown.sents(categories='news')
>>> regexp_tagger = nltk.RegexpTagger(patterns)
>>> regexp_tagger.tag(brown_sents[3])
[('', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative',
'NN'), ('handful', 'NN'), ('of', 'NN'), ('such', 'NN'),
('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'),
('...', 'NN'), ('', 'NN'), ('the', 'NN'), ('jury', 'NN'),
('said', 'NN'), ('', 'NN'), ('...', 'NN'), ('considering',
'VBG'), ('the', 'NN'), ('widespread', 'NN'), ...]
>>> regexp_tagger.evaluate(brown_tagged_sents)
0.20326391789486245

```

23.1.3 Lookup Tagger

- Using already tagged text, we can find the which tags are assigned to each word most frequently

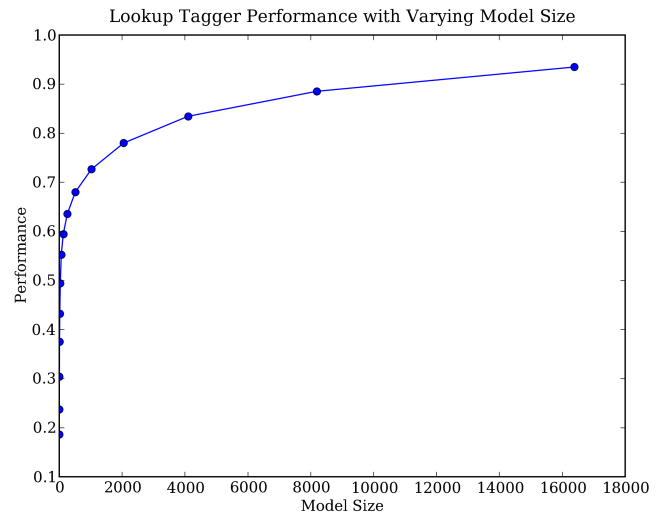
Example 23.5: Tag on 100 most frequent words

```

>>> fd = nltk.FreqDist(brown.words(categories='news'))
>>> cfd = nltk.ConditionalFreqDist(
        brown.tagged_words(categories='news'))
>>> cfd['permit'].items()
[('VB', 5), ('NN', 1)]
>>> most_freq_words = fd.keys()[:100]
>>> likely_tags = dict((word, cfd[word].max())
        for word in most_freq_words)
>>> baseline_tagger = nltk.UnigramTagger(
        model=likely_tags)
>>> baseline_tagger.evaluate(brown_tagged_sents)
0.45578495136941344

```

Lookup tagging performance



Lookup tagger performance with model size

23.2 N-Gram tagging

23.2.1 Unigram tagging

- Unigram tagging is just like lookup tagging, except that we *train* the tagger on already tagged data

Example 23.6: Unigram tagging brown news

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> unigram_tagger = nltk.UnigramTagger(brown_tagged_sents)
>>> uni_sample = unigram_tagger.tag(brown_sents[2007])
>>> unigram_tagger.evaluate(brown_tagged_sents)
0.9349006503968017
```

Separating training and test data

- It is not fair to test our tagger on the same data we train it on
- The standard practice is to train on 90% of the data, and test on 10%

Example 23.7: Training and testing

```

>>> size = int(len(brown_tagged_sents) * 0.9)
>>> size
4160
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
>>> unigram_tagger = nltk.UnigramTagger(train_sents)
>>> unigram_tagger.evaluate(test_sents)
0.81202033290142528

```

23.2.2 Bigram tagging

- Bigram tagging is similar to unigram tagging
- We use the most common bigram tags

Example 23.8: bigram tagging

```

>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bi_sample = bigram_tagger.tag(brown_sents[2007])
>>> unseen_sent = brown_sents[4203]
>>> bi_unseen = bigram_tagger.tag(unseen_sent)
>>> bigram_tagger.evaluate(test_sents)
0.10276088906608193

```

Backoff tagger

- Most taggers allow us to specify a backoff tagger
- For example, if we encounter a word in our test data which is not in the training data, we could just guess that it is a noun
- It is possible to use multiple backup taggers by specifying a backoff tagger for a backoff tagger

Example 23.9: Combining taggers

```

>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(train_sents, backoff=t0)
>>> t2 = nltk.BigramTagger(train_sents, backoff=t1)
>>> t2.evaluate(test_sents)
0.84491179108940495

```

23.3 Transformation based tagging

- Transformational taggers use sentence context from errors to write transformational rules
- This avoids the problem with n-gram taggers of having huge lookup tables
- In addition, the rules they create also have linguistic meaning

23.3.1 The Brill tagger

- The Brill tagger uses context to form transformation rules
- First, a default tagger is used (like a unigram tagger); then rules are formed from the errors

Example 23.10: sample transformation rules

The President said he will ask Congress to increase grants to states for vocational rehabilitation

We will examine the operation of two rules:

1. Replace NN with VB when the previous word is TO;
2. Replace TO with IN when the next tag is NNS.

	Phrase	to	increase	grants	to	states	for	vocational	rehabilitation
	Unigram	TO	NN	NNS	TO	NNS	IN	JJ	NN
Steps in Brill Tagging	Rule 1		VB						
	Rule 2			IN					
	Output	TO	VB	NNS	IN	NNS	IN	JJ	NN
	Gold	TO	VB	NNS	IN	NNS	IN	JJ	NN

Chapter 24

Supervised classification

24.1 Homework 10 comments

24.1.1 Subversion comments

- To copy a file using subversion, do:
`svn cp oldfile newfile`
- By copying using subversion the history from the oldfile is copied into newfile
- The history can be valuable. Consider the following
 1. File *A* gets copied to *B* at revision 100
 2. A bug in *A* is discovered, and fixed at revision 120
 3. File *A* gets copied to *C* at revision 150
- We can tell that *B* still has this bug, but *C* doesn't, by looking at the svn log of each file

24.1.2 More on options, parameters and arguments

- command line options usually use hyphens (e.g. `include-stop`)
- python options (and variables) usually use underscores (e.g. `include_stop`)
- You can use variables as values of named arguments

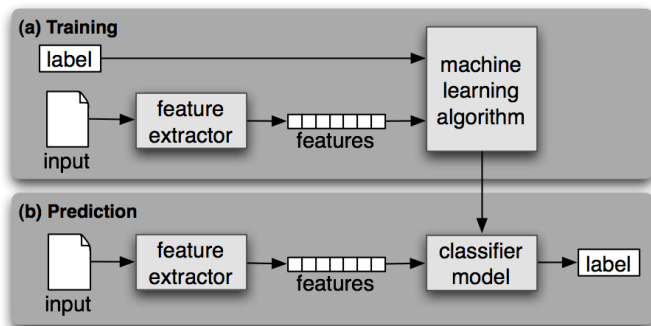
Example 24.1: on using options

```
import nltk, text_means
foo = True
ignore_stop=False
moby = nltk.corpus.gutenberg.words(
    'melville-moby_dick.txt')
# use_set=True, ignore_stop=False
text_means.mean_word_len(moby, use_set=foo,
    ignore_stop=ignore_stop)
```

24.2 Supervised classification

Definition 24.1:

A *feature set* maps from features' names to their values. The values are single items such as a string, boolean, or number (not a list or dict)



24.2.1 Gender identification

Example 24.2: gender feature extractor

```
>>> def gender_features(word):
...     feature_set = {'last_letter': word[-1],
...                   'length': len(word)}
...     return feature_set

>>> gender_features('Shrek')
{'length': 5, 'last_letter': 'k'}
```

Example 24.3: map labels to features

```
from nltk.corpus import names
import random
names = [(name, 'male')
         for name in names.words('male.txt')] +
        [(name, 'female')
         for name in names.words('female.txt')]
import random
random.shuffle(names)
featuresets = [(gender_features(n), g) for (n,g) in names]
train_set, test_set = featuresets[500:], featuresets[:500]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Example 24.4: testing out the gender classifier

```
>>> classifier.classify(gender_features('Neo'))
'male'
>>> classifier.classify(gender_features('Trinity'))
'female'
>>> print nltk.classify.accuracy(classifier, test_set)
0.752
```

Definition 24.2:

A *likelihood ratio* is the likelihood that an item with feature X has label A

Example 24.5: finding informative features

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
  last_letter = 'a'   female : male    = 38.3 : 1.0
  last_letter = 'k'   male  : female = 31.4 : 1.0
  last_letter = 'f'   male  : female = 15.3 : 1.0
  last_letter = 'p'   male  : female = 10.6 : 1.0
  last_letter = 'w'   male  : female = 10.6 : 1.
```

Gender identification practice

Add an additional feature to the *gender_features* function, and re-train the classifier. What is your accuracy now?

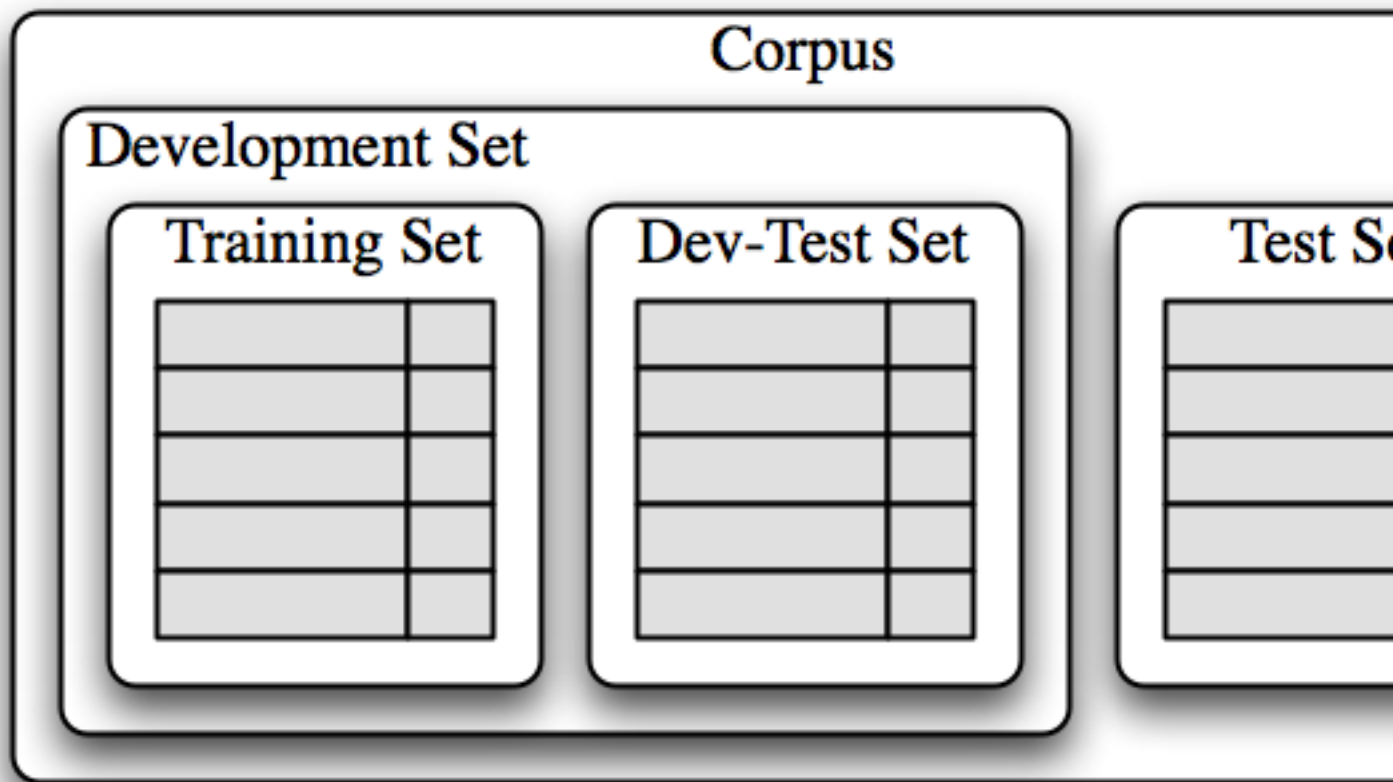
24.2.2 Choosing the right features

- The classifier is only as good as the feature set
- Adding features can improve classification, but it can also hurt it
- This is known as overfitting

model	accuracy
last letter	0.758
last letter + first letter	0.762
last letter + length	0.752

Error analysis

- We can analyze classification errors to help us choose optimal features



Example 24.6: Using devtest model

```
train_names = names[1500:]
devtest_names = names[500:1500]
test_names = names[:500]
train_set = [(gender_features(n), g) for (n,g) in train_names]
devtest_set = [(gender_features(n), g) for (n,g) in
    devtest_names]
test_set = [(gender_features(n), g) for (n,g) in test_names]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print nltk.classify.accuracy(classifier, devtest_set)
# 0.755
```

Example 24.7: Finding errors

```

errors = []
for (name, tag) in devtest_names:
    guess = classifier.classify(gender_features(name))
    if guess != tag:
        errors.append( (tag, guess, name) )

for (tag, guess, name) in sorted(errors):
    print 'correct=%-8s guess=%-8s name=%-30s' % (tag, guess,
        name)

```

24.2.3 Document classification

- We can also use features to classify document
- The movie reviews corpus is split into negative and positive reviews
- We can try to automatically categorize them using the following method:
 1. Find the 2000 most frequent words in all the reviews
 2. Write a feature extractor which considers whether each word is in a document
 3. Train a classifier based on these features

Example 24.8: Movie review categorization

```

from nltk.corpus import movie_reviews
documents = [(list(movie_reviews.words(fileid)), category)
              for category in movie_reviews.categories()
              for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)
all_words = nltk.FreqDist(w.lower() for w in
    movie_reviews.words())
word_features = all_words.keys()[:2000]
def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains(%s)' % word] = (word in
            document_words)
    return features

```

Example 24.9: Movie review categorization

```

featuresets = [(document_features(d), c) for (d,c) in
    documents]
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print nltk.classify.accuracy(classifier, test_set)
# 0.86
classifier.show_most_informative_features(5)
Most Informative Features
  contains(outstanding) = True   pos : neg = 11.1 : 1.0
    contains(seagal) = True   neg : pos =  7.7 : 1.0
contains(wonderfully) = True   pos : neg =  6.8 : 1.0
    contains(damon) = True   pos : neg =  5.9 : 1.0
    contains(wasted) = True   neg : pos =  5.8 : 1.0

```

Take home message

- Movies with Steven Seagal are bad!
- Movies with Matt Damon are good!

24.2.4 Part of speech tagging

- We can use a feature extractor to find common suffixes
- We can use these suffixes to classify parts of speech

Example 24.10: Finding most frequent suffixes

```

>>> from nltk.corpus import brown
>>> suffix_fdist = nltk.FreqDist()
>>> for word in brown.words():
...     word = word.lower()
...     suffix_fdist.inc(word[-1:])
...     suffix_fdist.inc(word[-2:])
...     suffix_fdist.inc(word[-3:])
>>> common_suffixes = suffix_fdist.keys()[:100]
>>> print common_suffixes
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a', 'of', 'the',
 'y', 'r', 'to', 'in', 'f', 'o', 'ed', 'nd', 'is', 'on',
 'l', 'g', 'and', 'ng', 'er', 'as', 'ing', 'h', 'at', 'es',
 'or', 're', 'it', '`', 'an', "'", 'm', ';', 'i', 'ly',
 'ion', ...]

```

Example 24.11: POS feature extractor

```
def pos_features(word):
    features = {}
    for suffix in common_suffixes:
        features['endswith(%s)' % suffix] =
            word.lower().endswith(suffix)
    return features
```

Example 24.12: classifying with POS features

```
>>> tagged_words = brown.tagged_words(categories='news')
>>> featuresets = [(pos_features(n), g) for (n,g) in
    tagged_words]
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:],
    featuresets[:size]
>>> classifier = nltk.DecisionTreeClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0.62705121829935351
>>> classifier.classify(pos_features('cats'))
'NNS'
```

Example 24.13: Decision tree output

```
>>> print classifier.pseudocode(depth=4)
if endswith(,) == True: return ','
if endswith(,) == False:
    if endswith(the) == True: return 'AT'
    if endswith(the) == False:
        if endswith(s) == True:
            if endswith(is) == True: return 'BEZ'
            if endswith(is) == False: return 'VBZ'
        if endswith(s) == False:
            if endswith(.) == True: return '.'
            if endswith(.) == False: return 'NN'
```

Chapter 25

Evaluation and Decision trees

25.1 Evaluation

25.1.1 Accuracy

Example 25.1: classifier accuracy

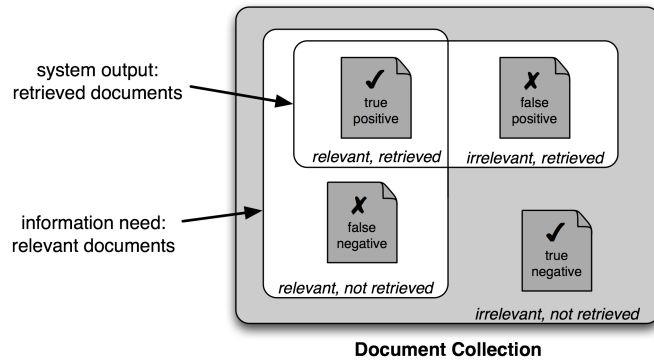
```
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print 'Accuracy: %4.2f' %
      nltk.classify.accuracy(classifier, test_set)
0.75
```

- Accuracy is the simplest method of evaluation
- However, it can be misleading depending on the distribution of the data
- E.g. if 80% of names in the names corpus are female, and we get an accuracy of 75%, then we are doing worse than just guessing female all the time

25.1.2 Precision and recall

Definition 25.1:

- *Precision* indicates the amount of returned items that were relevant $\frac{TP}{TP+FP}$
- *Recall* indicates the amount of relevant items which were returned $\frac{TP}{TP+FN}$



25.1.3 Confusion matrices

Definition 25.2:

A *confusion matrix* is a table indicating correct and incorrect responses to input. They give us a more detailed picture of our errors.

- The diagonal indicates correct responses
- The off-diagonals indicate incorrect responses

Example 25.2: creating a confusion matrix

```
from nltk.corpus import brown
def tag_list(tagged_sents):
    return [tag for sent in tagged_sents for (word, tag) in
            sent]

def apply_tagger(tagger, corpus):
    return [tagger.tag(nltk.tag.untag(sent)) for sent in
            corpus]

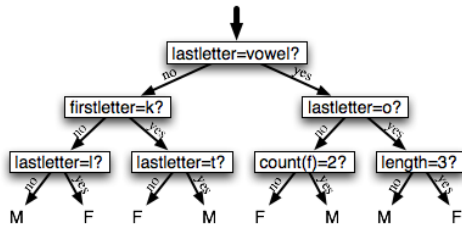
gold = tag_list(brown.tagged_sents(categories='editorial'))
test = tag_list(apply_tagger(t2,
                             brown.tagged_sents(categories='editorial')))
cm = nltk.ConfusionMatrix(gold, test)
print cm.pp(sort_by_count=True, show_percents=True,
            truncate=9)
```

25.2 Decision trees

Definition 25.3:

A *decision tree* is a simple flow chart which selects values based on inputs

- Each decision is called a *decision stump*
- Each value in a decision stump is called a *leaf*

**Definition 25.4:**

Information gain is a measure of how much more organized the input becomes when using a particular feature.

1. Create a decision stump for each feature
2. Calculate the information gain for each stump
3. Put stumps with the highest information gain at the top of the tree

25.2.1 Information gain

Entropy

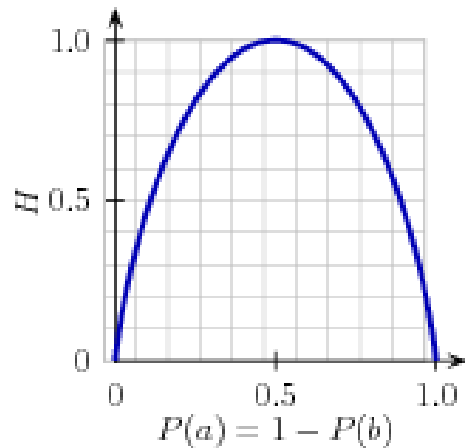
Definition 25.5:

Entropy is a measure of the uncertainty of a random variable. It is calculated as the sum of the probabilities of each possible value multiplied by the log of the probability of each possible value.

$$-\sum_{i=1}^n p(x_i) \log_2 p(x_i) \quad (25.1)$$

With binary values, like assigning gender to names, we get an entropy distribution like the following.

- If all of the names are male, the entropy is 0.
- If all of the names are female, the entropy is 0.
- Entropy is maximized when there are equal numbers of male and female names



Example 25.3: calculating entropy

```
import math
def entropy(labels):
    freqdist = nltk.FreqDist(labels)
    probs = [freqdist.freq(l) for l in nltk.FreqDist(labels)]
    return -sum([p * math.log(p,2) for p in probs])
genders = ['male', 'male', 'male', 'male']
print genders, entropy(genders)
for i in xrange(len(genders)):
    genders[i]='female'
    print genders, entropy(genders)
```

25.2.2 Practice

1. Calculate the entropy for the gender of the names corpus
2. Classify the names corpus using a decision tree classifier
 - (a) Create a list of (name, gender) tuples of the names corpus and shuffle it
 - (b) Create a feature extracting function called *gender_features* which uses the last letter and the last two letters as features
 - (c) Create a list of feature sets using the
 - (d) Divide up the feature sets into training, dev-test and test sets.
 - (e) Train a decision tree classifier on the training portion
 - (f) Calculate the accuracy of the classifier

Advantages

- Easy to interpret
- Well-suited to hierarchical categorical data like phylogeny

Disdvantages

- The amount of training material can become small for lower nodes (and lead to overfitting)
- Not good at handling independent features

Chapter 26

Bayesian and Maximum entropy classifiers

26.1 Final Presentations

26.1.1 Guidelines

- 5-10 minutes long
- Should prepare handouts or slides
 - slides in pdf or ppt format please
 - E-mail me slides by 10 a.m.
 - You can use (preferably) my computer or yours
- Use your classmates as resources for ideas
- ***PRACTICE BEFOREHAND***

26.1.2 Schedule

Tuesday, Dec. 8th

12:40

12:55

13:10

13:25

Thursday, Dec. 10th

12:40

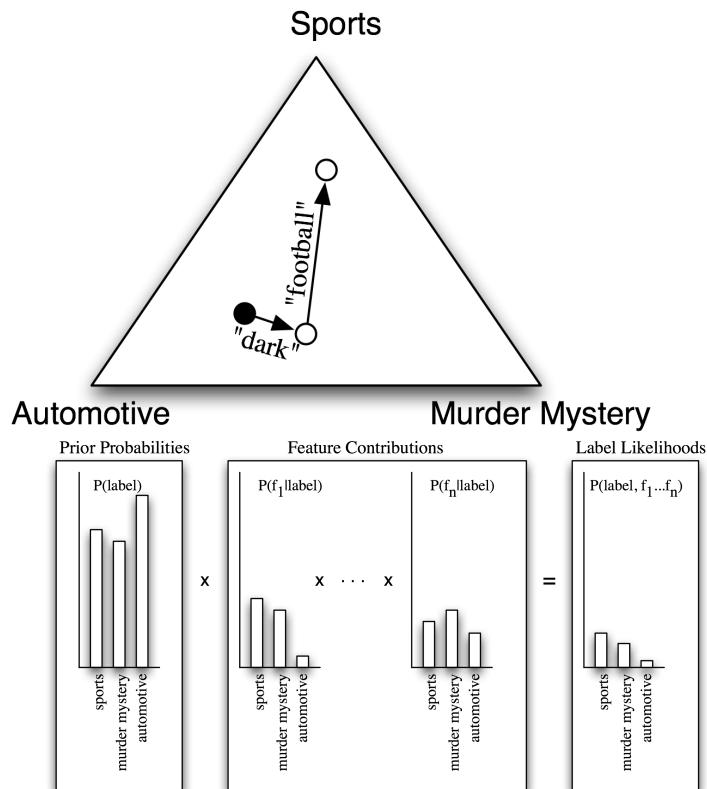
12:55

13:10

13:25

26.2 Bayesian Classifiers

- The *prior probability* of each label is calculated by determining the frequency of a label in the training set
- The contribution from each feature is then combined with this prior probability, to arrive at a likelihood estimate for each label
- The label whose likelihood estimate is the highest is then assigned to the input value.
- Features must be pre-defined, as we have done in previous examples



26.2.1 Zero counts and smoothing

- If the prior probability of any label is 0, then the total likelihood will also be 0.
- Often this is not the intention
- This happens frequently with small training sets
- *Smoothing* techniques can be used which replace 0s with an estimated likelihood
- The *Expected likelihood estimation* method adds 0.5 to the count of each feature/label frequency.

26.2.2 Naïvety and double counting

- Assume independence of features

Table 26.1: Possible probability distributions with no prior knowledge

	A	B	C	D	E	F	G	H	I	J
(i)	10	10	10	10	10	10	10	10	10	10
(ii)	5	15	0	30	0	8	12	0	6	24
(iii)	0	100	0	0	0	0	0	0	0	0

Table 26.2: Possible probability distributions knowing a 55% chance of label A

	A	B	C	D	E	F	G	H	I	J
(iv)	55	45	0	0	0	0	0	0	0	0
(v)	55	5	5	5	5	5	5	5	5	5
(vi)	55	3	1	2	9	5	0	25	0	0

- Features are calculated separately during training but combined during testing
- This results in double-counting

26.3 Maximum entropy classifiers

- Generalization of bayesian classifiers
- Do not assume independence of features
- Instead use the concept of joint-features

26.3.1 Maximizing entropy

- Row 1 adds up to 10%
- Row 1 A + C adds up to 8% (80% of the up cases)
- Column 1 adds up to 55%
- Remaining probabilities are evenly distributed

26.3.2 Generative vs. conditional classifiers

Possible questions:

1. What is the most likely label for a given input?

Table 26.3: Possible probability distributions knowing an 80% chance of label A or C when in the context of *up* which occurs 10% of the time

		A	B	C	D	E	F	G	H	I	J
(vii)	+up	5.1	0.25	2.9	0.25	0.25	0.25	0.25	0.25	0.25	0.25
	-up	49.9	4.46	4.46	4.46	4.46	4.46	4.46	4.46	4.46	4.46

2. How likely is a given label for a given input?
 3. What is the most likely input value?
 4. How likely is a given input value?
 5. How likely is a given input value with a given label?
 6. What is the most likely label for an input that might have one of two values (but we don't know which)?
- Generative classifiers can answer questions 1–6
 - Conditional classifiers can answer questions 1–2
 - *But may be more accurate*

26.4 Modeling linguistic data

Descriptive models

- Describe correlational relationships
- Sufficient for applied tasks such as document classification or question answering
- Can be used as a starting point to explanatory lines of investigation

Explanatory models

- Postulate causal relationships
- Help us understand linguistic structure

Chapter 27

Advanced Regular Expressions

27.1 One-liners

One liners

It is possible to specify python code on the command line using the -c option

Example 27.1: Python one-liner regex substitution

```
python -c 'import sys,re;
[sys.stdout.write(re.sub(r"(\W)(an?|the)(\W)",
"\1foo\3", line)) for line in sys.stdin]' < damnedThing.txt
```

Example 27.2: Perl one-liner regex substitution

```
perl -pe 's/(\W)(an?|the)(\W)/\1foo\3/' < damnedThing.txt
```

Replacing line by line

- Unless you are searching for multi-line patterns, you should process files one line at a time
- Processing the file all at once requires lots of memory to be used
- The default in perl is to use one line at a time
- When searching for multi-line patterns, use the re.M flag

Working on a whole file

Example 27.3: Python regex on whole text

```
python -c 'import sys,re; text=sys.stdin.read(); text =
re.sub(r"(?m)(\W)(an?|the)(\W)", "\1foo\3", text); print
text' < damnedThing.txt
```

Example 27.4: Perl regex on whole text

```
perl -e 'my $text = do { local ( $/ ); <>};  
    $text=~s/(\W)(an?|the)(\W)/\1foo\3/gm; print $text;' <  
damnedThing.txt
```

27.2 Advanced regular expression techniques

27.2.1 Pattern modifier flags

- There are several flags you can use to alter the behavior of your regex pattern
 - m** Treat ^and \$ as beginning and end of line, not string
 - s** . will match newline characters
 - i** ignore case
 - x** allow whitespace and comments in pattern
 - re.U** (python only) treat text as unicode
- In python, use | to separate multiple flags

Example 27.5: Using multiple modifier flags

```
re.match("Foo", "foo", re.I | re.U | re.M)
```

Example 27.6: Using embedded flags

We want to find strings that contain foo twice in them. We don't care about case

```
$ echo "Foo the f0o" | perl -ne 'print "$&\n" if  
    /( ?i)foo.*foo/'  
Foo the f0o  
$ echo "Foo the f0o" | perl -ne 'print "$&\n" if  
    /( ?i)foo.*(?-i)foo/'
```

27.2.2 Groupings and backreferences

Backreferences in patterns

Example 27.7: Using embedded flags

- We want to find strings that contain *foo* twice
- We don't care about case, except that the case of the second occurrence should be the same as the first.
- We use embedded modifiers to ignore case of the first *foo*
- Then we use a backreference to match the grouping

```
$ echo "Foofoo" | perl -ne 'print "$&\n" if
/(?i)(foo)(?-i)\1/'
$ echo "FooFoo" | perl -ne 'print "$&\n" if
/(?i)(foo)(?-i)\1/'
FooFoo
$ echo "foofoo" | perl -ne 'print "$&\n" if
/(?i)(foo)(?-i)\1/'
foofoo
```

To capture or not to capture

- Sometimes you want to use a grouping (disjunction) but you don't want to use a backreference.
- You can start the grouping with `?:` to create this behavior

For example, say you want to match *foo*, *foofoo*, *foofoofoo* etc. We can use a non-capturing grouping to do this

```
$ echo "foo" | perl -ne 'print "$&\n" if /(?:foo)+/'
foo
$ echo "foofoo" | perl -ne 'print "$&\n" if /(?:foo)+/'
foofoo
```

27.2.3 Look-ahead and look-behind

- Sometimes you want to use a pattern only when occurring before or after a particular string.
- Look-ahead and look-behind can be used for this
- You can also use negative look-ahead and look-behind.

Example 27.8: negative look-ahead

You might want to capture everything between *foo* and *bar* that doesn't include *baz*. The technique is to have the regex engine look-ahead at every character to ensure that it isn't the beginning of the undesired pattern:

```
/foo # Match starting at foo
( # Capture
(?: # Complex expression:
(?:!baz) # make sure we're not at the beginning of baz
. # accept any character
```

```
)* # any number of times
) # End capture
bar # and ending at bar
/x;
```

27.3 Substitution

27.3.1 Transformations

Text transformations

\l Makes the following character lower case

\u Makes the following character upper case

\L Makes all following characters lower case

\U Makes all following characters upper case

\E End case transformation

Note — these work in GNU (Linux) sed, but not on Mac. They work in perl on any system (and in vim).

Example 27.9:

```
echo "Minimum"|sed -r 's/(in|ax)imum/\u\1/'
OR
echo "Minimum"|perl -pe 's/(in|ax)imum/\u\1/'
```

Example 27.10: Convert articles to uppercase

```
# perl version
perl -pe 's/(?:\b|^)(an?|the)(?:\b|$)/\U\1/msgi' <
    damnedThing.txt

# python version
python -c 'import sys,re; [sys.stdout.write(
    re.sub(r"(?msi)(?:\b|^)(an?|the)(?:\b|$)", lambda x:
    x.group(1).upper(), line)) for line in sys.stdin]' <
    damnedThing.txt
```

27.3.2 Using matches

- In python, a match returns a match object, which has several properties and methods
- To see all groupings returned, use `matchobj.groups()`
- To get the value of a particular group, use `matchobj.group(<number>)`
- Group 0 is always the entire match

Example 27.11: Using groupings - python

```
>>> mymatch = re.search("(nee)(dle)?", "a needle in a
    haystack")
>>> mymatch.groups()
('nee', 'dle')
>>> mymatch.group(0)
'needle'
>>> mymatch.group(1)
'nee'
>>> mymatch = re.search("(nee)(dle)?", "a nee in a haystack")
>>> mymatch.groups()
('nee', None)
```

Example 27.12: Using groupings - perl

```
my $string = "a needle in a haystack";
$string =~ m/(nee)(dle)?/;
print "$&\n"; # 'needle '
print "$1\n"; # 'nee '
print "$2\n"; # nothing
```

27.3.3 Functions in substitutions

Functions in substitutions

- Python allows you to specify a function to be used as a substitution
- The function should take one argument, which is the match object, and return a string
- Perl allows you to use code in the substitution by using the `e` flag

Functions in substitutions

Example 27.13: Function as substitution – python

```
>>> def myfunc(match):
...     'return only the first group, in upper case'
...     return(match.group(1).upper())
...
>>> new = re.sub(r"(nee)(dle)?", myfunc, "a needle in a
    haystack")
>>> new
'a NEE in a haystack'
```

Functions in substitutions

Example 27.14: Function as substitution – perl

```
my $string = "a needle in a haystack";
$string=~s/(nee)(dle)?/uc($1)/e;
```

27.3.4 More regex examples

caMel case

- caMel case is a style of writing functions and variables by alternating case.
- It is used extensively in java and javascript.
- Python tends to favor underscores over camel case
- However, you are working on a project with a Java freak, and they want you to use caMel case for your python code.
- You can easily convert your code using a regex substitution

Example 27.15: Converting to camel case

```
under = re.match(r"(\w+)_(\w+)", "my_list")
camel = under.group(1) + under.group(2).title()
```

Camel case practice

1. Convert from camel case back to underscores with lower case. E.g. *myList* should become *my_list*
2. Make your solution more general to cover cases like *myAwesomeVar*, as well as *myList*

Auto-labeling html ids

Practice problem

- When writing html, any tag can have an id associated with it.
- IDs can be used for styling (with CSS), and for interaction (with javascript).
- Let's say you want to automatically give all of your `<h1>` and `<h2>` headings an id attribute which is the same as the contents of the heading, except all lowercase, and replacing any spaces with hyphens
- For example, `<h1>Foo Bar</h1>` should be replaced with: `<h1 id='foo-bar'>Foo Bar</h1>`

Hints

- The heading may span multiple lines
- Make sure your regex is not greedy
- You may want to write a function to substitute spaces with hyphens

Appendix A

Practice Problem solutions

A.1 Introduction

A.2 UNIX Basics

A.2.1 Filesystem navigation

Download practiceFiles.zip from the course website and unzip it into your home directory.

1. Change the current working directory to practiceFiles, and show that you are there.

```
cd ~/practiceFiles
```

```
pwd
```

2. List the contents of the practiceFiles directory

```
ls
```

3. Change the current working directory to the parent directory

```
cd ..
```

4. Now list the contents of the practiceFiles directory

```
ls practiceFiles
```

A.2.2 Reading files

1. Inspect the celex.txt file one screen worth at a time

```
less celex.txt
```

2. Count the total number of lines in the celex file

```
wc -l celex.txt
```

3. Print the first 10 lines of the celex file

```
head celex.txt
```

A.3 UNIX pipes, options, and help

A.3.1 Input and Output

Streams

1. Write the first 10 lines of celex.txt to a new file called celex.small
`head celex.txt > celex.small`
2. Append the last 10 lines of celex.txt to celex.small
`tail celex.txt >> celex.small`

Pipes

1. Display lines (files) 11–20 from a directory listing of practiceFiles
`ls | head -n 20 | tail` OR
`ls | tail -n +11 | head`
2. Produce a numbered list of the files in practiceFiles
`ls | nl`

Subshells

1. Move files 11-20 to a new directory tmp
`mkdir tmp`
`mv `ls | head -n 20 | tail` tmp`

A.3.2 Practice

1. Count the total number of characters in the filenames in practiceFiles
`ls | wc -c`
2. Display the second column of celex.small
`cut -f2 -d'\ ' celex.small`
3. Create a new file with only the first and third columns of celex.small
`cut -f1,3 -d'\ ' celex.small > celex13.txt`
4. Combine the celex.small with the new file you just created
`paste celex.small celex13.txt > combinedFile.txt`

5. Sort the celex.small file by orthography, ignoring case (HINT: use -t '\')
6. Calculate the average number of characters per word in the devilsDictionary.txt
 - (a) First find just the number of characters


```
wc -c devilsDictionary.txt|cut -f 1 -d ' '
```
 - (b) Next find just the number of words


```
wc -w devilsDictionary.txt|cut -f 1 -d ' '
```
 - (c) Now use subshells to put this into an equation form


```
echo "`wc -c devilsDictionary.txt|cut -f 1 -d ' '`/ `wc -w devilsDictionary.txt|cut -f 1 -d ' '``"
```
 - (d) Finally, pipe this equation to the basic calculator


```
echo "`wc -c devilsDictionary.txt|cut -f 1 -d ' '`/ `wc -w devilsDictionary.txt|cut -f 1 -d ' '``| bc -l"
```

A.4 Regular Expressions and Globs

A.4.1 Globs (Wildcards)

Practice

Using the practiceFiles

1. Move all files ending in .txt to a new directory txt


```
mkdir txt; mv *.txt txt
```
2. Copy files 10-19 to a new directory 10-19


```
mkdir 10-19; cp 1[0-9] 10-19
```
3. list permissions for files ending in .txt which do not contain numbers


```
ls -l [a-zA-Z].txt
```

OR

```
ls -l [!0-9].txt
```
4. Separate files into different directories according to their extension


```
mkdir {tmp,foo,bar,txt}
for file in *. {tmp,txt,foo,bar}; do mv $file `echo $file| cut -f 2 -d '.'`
$file; done
```

A.4.2 grep

Practice

Practice writing some regular expressions that will find the following from CELEX:

- all words begin with 'st'


```
\\st
```

- all words that end in 'ing'
`ing\\`
- word that begin with 'st' and ending with 'ing'
`\\st[a-z]*ing\\`
- all monosyllabic words
`\\\[CV]+\]\\`
- all disyllabic words
`\\\[CV]+\]\\\[CV]+\]\\`

A.4.3 Practice

1. Count the number of entries in the Devil's Dictionary
`grep -Ec '^[A-Z]{2,},' devilsDictionary.txt`
2. Print out the all the entries in the Devil's dictionary (not the definition)
`grep -E '^[A-Z]{2,},' devilsDictionary.txt | cut -f1 -d ','`
3. Count the number of occurrences of the word *the* in the Devil's Dictionary
`grep -Eic '( |^[a-z]|^)the(^[a-z]|$)' devilsDictionary.txt`
4. Count the number of indefinite articles in the Devil's Dictionary
`grep -Eic '( |^[a-z]|^)(a|an)( |^[a-z]|$)' devilsDictionary.txt`

A.5 Common UNIX utilities

A.5.1 basic scripting

Variables

1. Create a new script in `~/bin` called *changeEnding*
`touch ~/bin/changeEnding`
2. Make the file executable
`chmod a+x ~/bin/changeEnding`
3. Edit the file so that it will move all `.txt` files to `.temp`
Use the editor of your choice (nano, emacs, vi)
`#!/bin/bash`
`for file in *.txt`
`do mv $file `basename $file txt`temp`
`done`

A.6 Managing code with source control

A.6.1 Subversion

Practice

1. Create a new file in your own directory called `hmkw2_<name>.sh`
`touch hmkw2_<name>.sh`
2. Add this new file to your working copy
`svn add hmkw2_<name>.sh`
3. Commit your change
`svn commit -m 'adding in hmkw2 file'`
4. Edit the homework file with your favorite editor (vi, nano, emacs)
5. Examine the differences
`svn diff hmkw2_<name>.sh`
6. Commit your change
`svn commit -m 'added general info to hmkw2 '`

A.7 Getting started with python and the NLTK

A.7.1 input

modules

Practice

1. load the math module
`import math`
2. Round 35.4 down to the nearest integer
`math.floor(35.4)`

A.8 Python lists and NLTK word frequency

A.8.1 Lists

indexing

Zero comes first

Python starts all indices with 0.

Practice

1. Create a list *foo*, with the following values: 25, 68, “bar”, 89.45, 789, “spam”, 0, “last item”
`foo = [25, 68, 'bar', 89.45, 789, 'spam', 0, 'last item']`

2. print just the first item *foo*
`print(foo[0])`
3. print just the last item *foo*
`print(foo[-1])`

slicing

Practice

HINT: remember that indices start at 0

1. Print the 1st to 3rd item in the list *foo*
`print(foo[:3])`
2. Print the 3rd to last item in the list *foo*
`print(foo[2:])`
3. Print the 2nd to the 2nd to last item in the list *foo*
`print(foo[1:-1])`
4. Copy the entire *foo* list to a new list named *bar*
`bar=foo[:]`

Lists of lists

operations

Practice

1. Change the first item in the *foo* list to 12
`foo[0]=12`
2. Now multiply the first item in the *foo* list by 2
`foo[0] = foo[0]*2`
3. Test whether “ham” is in the list *foo*
`'ham' in foo`

len, min, and max

practice

1. How many items does *foo* contain?
`len(foo)`
2. What does `min(foo)` return?
3. What does `max(foo)` return?
Is that what you expected?

List methods

practice

1. Append the value 24 to the list *foo*
`foo.append(24)`
2. Insert the value “twenty” to the list *foo* as the 4th item
`foo.insert(3, 'twenty')`
3. Find the index of “spam” in the list *foo*
`foo.index('spam')`
4. remove the last item from *foo*, and store it as a new variable
`lastfoo=foo.pop()`

Practice

1. Append the following values to *foo*: 89, 23.4, 1
`foo.extend([89, 23.4, 1])`
2. Create a new list *fooSorted* with the same contents as *foo*, but sorted
`fooSorted=sorted(foo)`

A.9 Control structures and Natural Language Understanding

A.9.1 Control Structures

Conditionals

Practice

```
# Initialize two variables x,y to 3,4
# if x and y are equal to each other,
# print "equal", otherwise "unequal"
if x == y:
    print "equal"
else:
    print "unequal"
```

More Practice

```
# Now add a condition that if
#x is less than y, print "x is less than y"
#otherwise, print "x is greater than y"
if x ==y:
    print "equal"
elif x < y:
    print "x is less than y"
else:
```

```
print "x is greater than y"
```

For Loops

practice

Multiply each item in *mylist* by 2, and print the result (don't change the values of *mylist*)

```
for item in mylist:  
    print (item*2)
```

List Comprehensions

practice

For each item in *mylist*, multiply by 2 and add 3, (*do* change the values of *mylist*)

```
mylist=[x * 2 + 3 for x in mylist]
```

Looping with conditions

practice

Create a list called *ab* and put all words from *text1* into it which start with 'ab'

```
ab=[]  
for word in text1:  
    if word.startswith('ab')  
        ab.append(word)
```

A.10 NLTK corpora and conditional frequency distributions

A.10.1 Accessing text corpora

The web and chat Corpus

The web and chat corpora come from several different online texts, discussion forums, and chat rooms

practice

1. Import the web chat corpus (*webtext*)

```
from nltk.corpus import webtext
```
2. Show the files in the *webtext* corpus

```
webtext.fileids()
```

3. Retrieve a list of sentences from the wine file in webtext and store as *wine*
`wine = webtext.sents('wine.txt')`
4. Print the first 10 sentences of the wine text
`print wine[:10]`

A.11 Reusing code and wordlists

A.11.1 Reusing code

functions

Create a function *mean* which computes the mean of a list It should take a list as an argument, and return a number

```
def mean(thelist):  
    return( sum(thelist) / len(thelist) )
```

A.11.2 Lexical Resources

Stoplist corpus

Using a list comprehension create a list of words from Moby Dick which does not contain stop-words.

```
moby_nostop = [w for w in text1 if w.lower()  
               not in eng_stop]
```

Names corpus

Use the names corpus to find all female names which start with *S*, and which don't end in *ie*.

```
female_names = names.words('female.txt')  
my_names = [w for w in female_names if not w.startswith('s')  
            and not w.endswith('ie')]
```

Now add a condition that the names should also not contain an *r*

```
my_names = [w for w in female_names if not w.startswith('s')  
            and not (w.endswith('ie') or 'r' in w)]
```

A.12 Semantic relations

A.12.1 Wordnet

Synonyms

1. Show the lemmas for the second definition of car
`wordnet.synset('car.n.02').lemma_names`
2. How many different senses of *dish* are there?
`print len(wordnet.synsets('dish'))`
3. Test whether *dish* and *serve* are synonyms
`serve_synsets=wordnet.synsets('serve')`
`for synset in serve_synsets:`
 `if 'dish' in synset.lemma_names:`
 `print "'dish' and 'serve' are synonyms"`

Hierarchy

1. Print out all the lemma names from the hyponyms of motorcar
`[synset.lemma_names`
 `for synset in types_of_motorcar]`
#OR a bit fancier
`sorted([lemma.name`
 `for synset in types_of_motorcar`
 `for lemma in synset.lemmas])`

A.13 Processing raw text

A.13.1 Accessing text from the web and disk

Reading rss feeds

1. Create a list with the content of each entry from the ling5200 rss feed
`posts = [post.content[0].value for post in`
`ling5200.entries]`
2. strip out the html from each post and store as a new list
`clean_posts = [nltk.clean_html(post) for post in`
`posts]`
3. Tokenize each post
`tokens = [nltk.word_tokenize(post) for post in`
`clean_posts]`

Opening local files

1. Read the celex file into a string

```
f=open('resources/texts/celex.txt')  
text = f.read()
```
2. Tokenize the devil's dictionary

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

A.14 Handling strings

A.14.1 String basics

Accessing substrings

Basic string practice

1. Store the string *John likes Mary's dog.* in *sent1*

```
sent1 = "John likes Mary's dog."
```
2. Store the string *Mary said: "I like John."* in *sent2*

```
sent2 = 'Mary said: "I like John."'
```
3. Combine the two strings with space as *sents*

```
sents = sent1 + ' ' + sent2
```

width and precision

Precision practice

1. Print out pi to the 3rd decimal place, with a width of 7

```
print('%7.3f' % pi )
```
2. Print pi times 100 in the same fashion

```
print('%7.3f' % (100 * pi) )
```

alignment

1. Print out 23 padded with zeros to make it 4 wide

```
print('%04.0f' % 23)
```
2. Print out 456.7 with a width of 10, and precision of 0

```
print('%10.0f' % 456.7)
```

Strings and tuples

- It is possible to format several strings at once

practice

1. Print out e and π , with appropriate labels, as in the preceding example, using a tuple
`'%s: %4.2f, %s: %4.2f' % ('pi', pi, 'e', e)`

A.14.2 String methods

find

practice

1. Find where *in* starts in the phrase *needle in a haystack*
`phrase='needle in a haystack'`
`phrase.find('in')`
2. If *needle in a haystack* contains *hay*, print *hey*
`if phrase.find('hay')>=0:`
`print('hey')`

join / split

practice

1. Split the haystack phrase into multiple words
`words=phrase.split()`
2. Reverse the order of the words
`words.reverse()`
3. Join the words back together with commas ', '.
`join(words)`

lower

practice

1. Make *ALLCAPS* all lowercase
`'ALLCAPS'.lower()`
2. Change *ALLCAPS* to title case
`'ALLCAPS'.title()`

replace

practice

1. Replace *needle* with *noodle* in the haystack phrase `phrase.replace('needle', 'noodle')`
What is the value of phrase now?
2. Replace *e* with *o* in the haystack phrase `phrase=phrase.replace('e', 'o')`

strip

practice

1. Strip off newline characters from end of the *haystack*
`phrase=phrase.rstrip('\r\n')`
2. Strip off whitespace from *haystack*, and convert to upper case
`phrase=phrase.strip().upper()`
3. Strip off whitespace from *haystack*, replace *needle* with *noodle* and convert to upper case
`phrase=phrase.strip().replace('needle', 'noodle').upper()`

A.15 Unicode and regular expressions in python

A.15.1 Using regular expressions in python

The re module

1. Find all words in emma which contain any of the letters *x,j,q,z*
`words=[w for w in emma if re.search(r'[xjqz]+', w)]`
2. Find all words in emma which a *q* not followed by a *u*
`words=[w for w in emma if re.search(r'q[^u]', w)]`

Substitution

1. Split the raw text of Emma by newline characters (including carriage returns)
`emmaraw=gutenberg.raw('austen-emma.txt')`
`emma_lines= re.split(r'[\r\n]', emmaraw)`
2. Replace *hot* or *cold* in *foo* with *lukewarm* (ignoring case) and store as *bar*
`pat = re.compile(r'(hot|cold)', flags=re.I)`
`bar = re.sub(pat, 'lukewarm', foo)`

A.16 Tokenizing and normalization

A.16.1 Applications of regular expressions

Working with word pieces

1. Find all words in rotokas which have long vowels (a long vowel is represented as 2 of the same vowel, e.g. *uu*)
`vvs = set([w for w in rotokas_words for vv in re.findall(r'([aeiou])\1',w)])`
2. Find the mean number of vowels per word in the cmudict
`word_strs = [' '.join(word[0]) for word in cmu.values()]`
`vowels = [v for word_str in word_strs for v in re.findall(r'[012]', word_str)]`
`len(vowels) / len(word_strs)`

Finding word stems

1. Using the previous regular expression, find all words from the cmudict which have suffixes
`cmu_suffixed_words = [w for w in cmu.keys() if re.findall(r'^(.?*)(ing|ly|ed|ious|w)']`

A.17 More on strings and shell integration

A.17.1 Shell integration

Calling python scripts from bash scripts

1. Create a bash script which calls *text_means.py* for all files in the resources/texts directory which start with the letter *d*
`#!/bin/bash`
`./text_means.py ../texts/d*.txt`
2. Create a bash script which calls *randomExcerpt.py* on huckFinn.txt and tomSawyer.txt. Use the `-l` option to specify lengths of 100, 1000, 10000, and 100000.
 - Store the output in files called huckFinn.500, tomSawyer.1000 etc.
 - Now use `-t` option of *text_means.py* to calculate the type / token ratio for each of these files

```
#!/bin/bash
for length in 100,1000,10000,100000; do
    ./randomExcerpt.py -l $length ../texts/huckFinn.txt\
    > ../texts/huckFinn.$length
    ./randomExcerpt.py -l $length ../texts/tomSawyer.txt\
    > ../texts/tomSawyer.$length
done
./text_means.py -t ../texts/*. [0-9]*
```

A.18 Back to basics

A.19 More on functions

A.20 Program development and debugging

A.20.1 Program development

More on creating modules

1. Run the demo function from the novel module

```
novel.demo()
```

2. Find novel words in the Devil's Dictionary

```
f = open(' ../texts/devilsDictionary.txt' )
```

```
devils = f.read()
```

```
tokens = nltk.word_tokenize(devils)
```

```
devils_novel = novel.unknown(tokens)
```

3. Use the *text_means* module to compute the type-token ratio of the novel words from the Devil's dictionary

```
text_means.type_token_ratio(devils_novel)
```

A.21 A tour of python modules

A.21.1 Some useful modules

Timeit

1. Time how long it takes to find the word 'leopard' in the words corpus, using (a) a list, and (b) a set
as_list = "import nltk; words = nltk.corpus.words.words()"

```
as_set = "import nltk; words = set(nltk.corpus.words.words())"
```

```
statement = "'leopard' in words"
```

```
print Timer(statement, as_list).timeit(1000)
```

```
print Timer(statement, as_set).timeit(1000)
```

A.22 Part of speech tagging

A.22.1 Using tagged corpora

Verbs

1. Find the following context of nouns
after_nouns = nltk.FreqDist(b[1]

for (a, b) in word_tag_pairs if a[1] == 'N')
2. Find the preceding context of verbs
before_verbs = nltk.FreqDist(a[1]

for (a, b) in word_tag_pairs if b[1] == 'V')
3. Find the following context of verbs
after_verbs = nltk.FreqDist(b[1]

for (a, b) in word_tag_pairs if a[1] == 'V')

A.23 Automatic Part of speech tagging

A.24 Supervised classification

A.24.1 Supervised classification

Gender identification

```
def gender_features(word):  
    feature_set = {'last_letter': word[-1],  
                  'second_letter': word[1],  
                  'length': len(word)}  
    return feature_set  
  
featuresets = [(gender_features(n), g) for (n,g) in names]  
train_set, test_set = featuresets[500:], featuresets[:500]  
classifier = nltk.NaiveBayesClassifier.train(train_set)  
print nltk.classify.accuracy(classifier, test_set)  
# 0.756
```

A.25 Evaluation and Decision trees

A.26 Bayesian and Maximum entropy classifiers

A.27 Advanced Regular Expressions

A.27.1 Substitution

More regex examples

1. Convert from camel case back to underscores with lower case. E.g. *myList* should become *my_list*
 2. Make your solution more general to cover cases like *myAwesomeVar*, as well as *myList*
- ```
camel = re.match(r"([a-z]+)([A-Z][a-z]+)([A-Z][a-z]+)?",
 "myAwesomeVar")
```

**try:**

```
under = camel.group(1) + '_' + camel.group(2).lower()\
 + '_' + camel.group(3).lower()
```

**except AttributeError:**

```
under = camel.group(1) + '_' + camel.group(2).lower()
```

Perl version:

```
echo "myAwesomeVar" | perl -pe\
's/([a-z]+)([A-Z][a-z]+)([A-Z][a-z]+)?/\L\1_\2_\3/; s/_$///;'
echo "myList" | perl -pe\
's/([a-z]+)([A-Z][a-z]+)([A-Z][a-z]+)?/\L\1_\2_\3/; s/_$///;'
```

Practice problem

- When writing html, any tag can have an id associated with it.
- IDs can be used for styling (with CSS), and for interaction (with javascript).
- Let's say you want to automatically give all of your <h1> and <h2> headings an id attribute which is the same as the contents of the heading, except all lowercase, and replacing any spaces with hyphens
- For example, <h1>Foo Bar</h1> should be replaced with: <h1 id='foo-bar'>Foo Bar</h1>

Hints

- The heading may span multiple lines
- Make sure your regex is not greedy
- You may want to write a function to substitute spaces with hyphens

Python version:

```
import sys, re
```

```
def auto_label(matchobj):
 heading = matchobj.group(2).lower()
 tag = matchobj.group(1)
 id = heading.replace(' ', '-')
 repl = "<%s id='%s'>%s</%s>" % (tag, id, heading, tag)
 return(repl)
```

```
text = sys.stdin.read()
```

```

text = re.sub(r'<(h[12])>(.*?)</\1>', auto_label, text)
print text
 Perl version:
#slurp in text to string
my $string = do { local ($/); <>};

while ($string=~s|<(h[12])>(.*?)</\1>|"<$1 id='" . gen_id($2) .
 "'>$2</$1>"|mse) {
}
print "$string\n";

sub gen_id() {
 my $id = shift;
 $id=~s/ /-/g;
 $id=lc($id);
 return($id);
}

```