CHAPTER

2 Regular Expressions, Text Normalization, Edit Distance

```
User:  I am unhappy.
ELIZA: DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY
User:  I need some help, that much seems certain.
ELIZA: WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
User:  Perhaps I could learn to get along with my mother.
ELIZA: TELL ME MORE ABOUT YOUR FAMILY
User:  My mother takes care of me.
ELIZA: WHO ELSE IN YOU FAMILY TAKES CARE OF YOU
User:  My father.
ELIZA: YOUR FATHER
User:  You are like my father in some ways.
```

Weizenbaum (1966)

ELIZA

chatbots

The dialogue above is from **ELIZA**, an early natural language processing system that could carry on a limited conversation with a user by imitating the responses of a Rogerian psychotherapist (Weizenbaum, 1966). ELIZA is a surprisingly simple program that uses pattern matching to recognize phrases like "I need X" and translate them into suitable outputs like "What would it mean to you if you got X?". This simple technique succeeds in this domain because ELIZA doesn't actually need to *know* anything to mimic a Rogerian psychotherapist. As Weizenbaum notes, this is one of the few dialogue genres where listeners can act as if they know nothing of the world. Eliza's mimicry of human conversation was remarkably successful: many people who interacted with ELIZA came to believe that it really *understood* them and their problems, many continued to believe in ELIZA's abilities even after the program's operation was explained to them (Weizenbaum, 1976), and even today such **chatbots** are a fun diversion.

Of course modern conversational agents are much more than a diversion; they can answer questions, book flights, or find restaurants, functions for which they rely on a much more sophisticated understanding of the user's intent, as we will see in Chapter 26. Nonetheless, the simple pattern-based methods that powered ELIZA and other chatbots play a crucial role in natural language processing.

We'll begin with the most important tool for describing text patterns: the **regular expression**. Regular expressions can be used to specify strings we might want to extract from a document, from transforming "I need X" in Eliza above, to defining strings like *$199* or *$24.99* for extracting tables of prices from a document.

text normalization

tokenization

We'll then turn to a set of tasks collectively called **text normalization**, in which regular expressions play an important part. Normalizing text means converting it to a more convenient, standard form. For example, most of what we are going to do with language relies on first separating out or **tokenizing** words from running text, the task of **tokenization**. English words are often separated from each other by whitespace, but whitespace is not always sufficient. *New York* and *rock 'n' roll* are sometimes treated as large words despite the fact that they contain spaces, while sometimes we'll need to separate *I'm* into the two words *I* and *am*. For processing tweets or texts we'll need to tokenize **emoticons** like `:)` or **hashtags** like `#nlproc`.

Some languages, like Japanese, don't have spaces between words, so word tokenization becomes more difficult.

lemmatization

stemming

sentence segmentation

Another part of text normalization is **lemmatization**, the task of determining that two words have the same root, despite their surface differences. For example, the words *sang*, *sung*, and *sings* are forms of the verb *sing*. The word *sing* is the common *lemma* of these words, and a **lemmatizer** maps from all of these to *sing*. Lemmatization is essential for processing morphologically complex languages like Arabic. **Stemming** refers to a simpler version of lemmatization in which we mainly just strip suffixes from the end of the word. Text normalization also includes **sentence segmentation**: breaking up a text into individual sentences, using cues like periods or exclamation points.

Finally, we'll need to compare words and other strings. We'll introduce a metric called **edit distance** that measures how similar two strings are based on the number of edits (insertions, deletions, substitutions) it takes to change one string into the other. Edit distance is an algorithm with applications throughout language processing, from spelling correction to speech recognition to coreference resolution.

2.1 Regular Expressions

regular expression

corpus

One of the unsung successes in standardization in computer science has been the **regular expression** (**RE**), a language for specifying text search strings. This practical language is used in every computer language, word processor, and text processing tools like the Unix tools grep or Emacs. Formally, a regular expression is an algebraic notation for characterizing a set of strings. They are particularly useful for searching in texts, when we have a **pattern** to search for and a **corpus** of texts to search through. A regular expression search function will search through the corpus, returning all texts that match the pattern. The corpus can be a single document or a collection. For example, the Unix command-line tool `grep` takes a regular expression and returns every line of the input document that matches the expression.

A search can be designed to return every match on a line, if there are more than one, or just the first match. In the following examples we generally underline the exact part of the pattern that matches the regular expression and show only the first match. We'll show regular expressions delimited by slashes but note that slashes are *not* part of the regular expressions.

Regular expressions come in many variants. We'll be describing **extended regular expressions**; different regular expression parsers may only recognize subsets of these, or treat some expressions slightly differently. Using an online regular expression tester is a handy way to test out your expressions and explore these variations.

2.1.1 Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters. To search for *woodchuck*, we type `/woodchuck/`. The expression `/Buttercup/` matches any string containing the substring *Buttercup*; `grep` with that expression would return the line *I'm called little Buttercup*. The search string can consist of a single character (like `/!/`) or a sequence of characters (like `/urgl/`).

Regular expressions are **case sensitive**; lower case `/s/` is distinct from upper case `/S/` (`/s/` matches a lower case *s* but not an upper case *S*). This means that the pattern `/woodchucks/` will not match the string *Woodchucks*. We can solve this

RE	Example Patterns Matched
`/woodchucks/`	"interesting links to woodchucks and lemurs"
`/a/`	"Mary Ann stopped by Mona's"
`/!/`	"You've left the burglar behind again!" said Nori

Figure 2.1 Some simple regex searches.

problem with the use of the square braces `[` and `]`. The string of characters inside the braces specifies a **disjunction** of characters to match. For example, Fig. 2.2 shows that the pattern `/[wW]/` matches patterns containing either *w* or *W*.

RE	Match	Example Patterns
`/[wW]oodchuck/`	Woodchuck or woodchuck	"Woodchuck"
`/[abc]/`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`/[1234567890]/`	any digit	"plenty of 7 to 5"

Figure 2.2 The use of the brackets `[]` to specify a disjunction of characters.

The regular expression `/[1234567890]/` specified any single digit. While such classes of characters as digits or letters are important building blocks in expressions, they can get awkward (e.g., it's inconvenient to specify

`/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/`

to mean "any capital letter"). In cases where there is a well-defined sequence associated with a set of characters, the brackets can be used with the dash (`-`) to specify any one character in a **range**. The pattern `/[2-5]/` specifies any one of the characters *2*, *3*, *4*, or *5*. The pattern `/[b-g]/` specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples are shown in Fig. 2.3.

range

RE	Match	Example Patterns Matched
`/[A-Z]/`	an upper case letter	"we should call it 'Drenched Blossoms' "
`/[a-z]/`	a lower case letter	"my beans were impatient to be hoed!"
`/[0-9]/`	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.3 The use of the brackets `[]` plus the dash `-` to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret ^. If the caret ^ is the first symbol after the open square brace `[`, the resulting pattern is negated. For example, the pattern `/[^a]/` matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Fig. 2.4 shows some examples.

RE	Match (single characters)	Example Patterns Matched
`/[^A-Z]/`	not an upper case letter	"Oyfn pripetchik"
`/[^Ss]/`	neither 'S' nor 's'	"I have no exquisite reason for't"
`/[^.]/`	not a period	"our resident Djinn"
`/[e^]/`	either 'e' or '^'	"look up ^ now"
`/a^b/`	the pattern 'a^b'	"look up a^ b now"

Figure 2.4 The caret ^ for negation or just to mean ^. See below re: the backslash for escaping the period.

How can we talk about optional elements, like an optional *s* in *woodchuck* and *woodchucks*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question mark `/?/`, which means "the preceding character or nothing", as shown in Fig. 2.5.

RE	Match	Example Patterns Matched
`/woodchucks?/`	woodchuck or woodchucks	"woodchuck"
`/colou?r/`	color or colour	"color"

Figure 2.5 The question mark `?` marks optionality of the previous expression.

We can think of the question mark as meaning "zero or one instances of the previous character". That is, it's a way of specifying how many of something that we want, something that is very important in regular expressions. For example, consider the language of certain sheep, which consists of strings that look like the following:

baa!
baaa!
baaaa!
baaaaa!
...

Kleene *

This language consists of strings with a *b*, followed by at least two *a*'s, followed by an exclamation point. The set of operators that allows us to say things like "some number of *a*s" are based on the asterisk or `*`, commonly called the **Kleene *** (generally pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `/a*/` means "any string of zero or more *a*s". This will match *a* or *aaaaaa*, but it will also match *Off Minor* since the string *Off Minor* has zero *a*'s. So the regular expression for matching one or more *a* is `/aa*/`, meaning one *a* followed by zero or more *a*s. More complex patterns can also be repeated. So `/[ab]*/` means "zero or more *a*'s or *b*'s" (not "zero or more right square braces"). This will match strings like *aaaa* or *ababab* or *bbbb*.

For specifying multiple digits (useful for finding prices) we can extend `/[0-9]/`, the regular expression for a single digit. An integer (a string of digits) is thus `/[0-9][0-9]*/`. (Why isn't it just `/[0-9]*/`?)

Kleene +

Sometimes it's annoying to have to write the regular expression for digits twice, so there is a shorter way to specify "at least one" of some character. This is the **Kleene +**, which means "one or more occurrences of the immediately preceding character or regular expression". Thus, the expression `/[0-9]+/` is the normal way to specify "a sequence of digits". There are thus two ways to specify the sheep language: `/baaa*!/` or `/baa+!/`.

One very important special character is the period (`/./`), a **wildcard** expression that matches any single character (*except* a carriage return), as shown in Fig. 2.6.

RE	Match	Example Matches
`/beg.n/`	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.6 The use of the period `.` to specify any character.

The wildcard is often used together with the Kleene star to mean "any string of characters". For example, suppose we want to find any line in which a particular word, for example, *aardvark*, appears twice. We can specify this with the regular expression `/aardvark.*aardvark/`.

Anchors

Anchors are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret `^` and the dollar sign `$`. The caret `^` matches the start of a line. The pattern `/^The/` matches the word *The* only at the start of a line. Thus, the caret `^` has three uses: to match the start of a line, to indicate a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow grep or Python to know which function a given caret is supposed to have?) The dollar sign `$` matches the end of a line. So the pattern `␣$` is a useful pattern for matching a space at the end of a line, and `/^The dog\.$/` matches a line that contains only the phrase *The dog.* (We have to use the backslash here since we want the `.` to mean "period" and not the wildcard.)

There are also two other anchors: `\b` matches a word boundary, and `\B` matches a non-boundary. Thus, `/\bthe\b/` matches the word *the* but not the word *other*. More technically, a "word" for the purposes of a regular expression is defined as any sequence of digits, underscores, or letters; this is based on the definition of "words" in programming languages. For example, `/\b99\b/` will match the string *99* in *There are 99 bottles of beer on the wall* (because 99 follows a space) but not *99* in *There are 299 bottles of beer on the wall* (since 99 follows a number). But it will match *99* in *$99* (since *99* follows a dollar sign ($), which is not a digit, underscore, or letter).

2.1.2 Disjunction, Grouping, and Precedence

disjunction

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case, we might want to search for either the string *cat* or the string *dog*. Since we can't use the square brackets to search for "cat or dog" (why can't we say `/[catdog]/`?), we need a new operator, the **disjunction** operator, also called the **pipe** symbol `|`. The pattern `/cat|dog/` matches either the string `cat` or the string `dog`.

Precedence

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for information about pet fish for my cousin David. How can I specify both *guppy* and *guppies*? We cannot simply say `/guppy|ies/`, because that would match only the strings *guppy* and *ies*. This is because sequences like `guppy` take **precedence** over the disjunction operator `|`. To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators `(` and `)`. Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe `|` and the Kleene*. So the pattern `/gupp(y|ies)/` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

The parenthesis operator `(` is also useful when we are using counters like the Kleene*. Unlike the `|` operator, the Kleene* operator applies by default only to a single character, not to a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression `/Column␣[0-9]+␣*/` will not match any number of columns; instead, it will match a single column followed by any number of spaces! The star here applies only to the space `␣` that precedes it, not to the whole sequence. With the parentheses, we could write the expression `/(Column␣[0-9]+␣*)*/` to match the word *Column*, followed by a number and optional spaces, the whole pattern repeated zero or more times.

operator precedence

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order of RE operator precedence, from highest precedence to lowest precedence.

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

Thus, because counters have a higher precedence than sequences, `/the*/` matches *theeeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `/the|any/` matches *the* or *any* but not *thany* or *theny*.

Patterns can be ambiguous in another way. Consider the expression `/[a-z]*/` when matching against the text *once upon a time*. Since `/[a-z]*/` matches zero or more letters, this expression could match nothing, or just the first letter *o*, *on*, *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

greedy

non-greedy

*?

+?

There are, however, ways to enforce **non-greedy** matching, using another meaning of the ? qualifier. The operator ***?** is a Kleene star that matches as little text as possible. The operator **+?** is a Kleene plus that matches as little text as possible.

2.1.3 A Simple Example

Suppose we wanted to write a RE to find cases of the English article *the*. A simple (but incorrect) pattern might be:

```
/the/
```

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

```
/[tT]he/
```

But we will still incorrectly return texts with `the` embedded in other words (e.g., *other* or *theology*). So we need to specify that we want instances with a word boundary on both sides:

```
/\b[tT]he\b/
```

Suppose we wanted to do this without the use of `/\b/`. We might want this since `/\b/` won't treat underscores and numbers as word boundaries; but we might want to find *the* in some context where it might also have underlines or numbers nearby (*the_* or *the25*). We need to specify that we want instances in which there are no alphabetic letters on either side of the *the*:

```
/[^a-zA-Z][tT]he[^a-zA-Z]/
```

But there is still one more problem with this pattern: it won't find the word *the* when it begins a line. This is because the regular expression `[^a-zA-Z]`, which we used to avoid embedded instances of *the*, implies that there must be some single (although non-alphabetic) character before the *the*. We can avoid this by specifying that before the *the* we require *either* the beginning-of-line or a non-alphabetic character, and the same at the end of the line:

```
/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/
```

false positives

false negatives

The process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like *other* or *there*, and **false negatives**, strings that we incorrectly missed, like *The*. Addressing these two kinds of errors comes up again and again in implementing speech and language processing systems. Reducing the overall error rate for an application thus involves two antagonistic efforts:

- Increasing **precision** (minimizing false positives)
- Increasing **recall** (minimizing false negatives)

2.1.4 A More Complex Example

Let's try out a more significant example of the power of REs. Suppose we want to build an application to help a user buy a computer on the Web. The user might want "any machine with at least 6 GHz and 500 GB of disk space for less than $1000". To do this kind of retrieval, we first need to be able to look for expressions like *6 GHz* or *500 GB* or *Mac* or *$999.99*. In the rest of this section we'll work out some simple regular expressions for this task.

First, let's complete our regular expression for prices. Here's a regular expression for a dollar sign followed by a string of digits:

```
/$[0-9]+/
```

Note that the `$` character has a different function here than the end-of-line function we discussed earlier. Most regular expression parsers are smart enough to realize that `$` here doesn't mean end-of-line. (As a thought experiment, think about how regex parsers might figure out the function of `$` from the context.)

Now we just need to deal with fractions of dollars. We'll add a decimal point and two digits afterwards:

```
/$[0-9]+\.[0-9][0-9]/
```

This pattern only allows *$199.99* but not *$199*. We need to make the cents optional and to make sure we're at a word boundary:

```
/(^|\W)$[0-9]+(\.[0-9][0-9])?\b/
```

One last catch! This pattern allows prices like *$199999.99* which would be far too expensive! We need to limit the dollar

```
/(^|\W)$[0-9]{0,3}(\.[0-9][0-9])?\b/
```

How about disk space? We'll need to allow for optional fractions again (*5.5 GB*); note the use of `?` for making the final `s` optional, and the of `/␣*/` to mean "zero or more spaces" since there might always be extra spaces lying around:

```
/\b[0-9]+(\.[0-9]+)?␣*(GB|[Gg]igabytes?)\b/
```

Modifying this regular expression so that it only matches more than 500 GB is left as an exercise for the reader.

2.1.5 More Operators

Figure 2.7 shows some aliases for common ranges, which can be used mainly to save typing. Besides the Kleene * and Kleene + we can also use explicit numbers as counters, by enclosing them in curly brackets. The regular expression `/{3}/` means "exactly 3 occurrences of the previous character or expression". So `/a\.{24}z/` will match *a* followed by 24 dots followed by *z* (but not *a* followed by 23 or 25 dots followed by a *z*).

RE	Expansion	Match	First Matches
`\d`	`[0-9]`	any digit	Party␣of␣5
`\D`	`[ˆ0-9]`	any non-digit	Blue␣moon
`\w`	`[a-zA-Z0-9_]`	any alphanumeric/underscore	Daiyu
`\W`	`[ˆ\w]`	a non-alphanumeric	!!!!
`\s`	`[␣\r\t\n\f]`	whitespace (space, tab)	
`\S`	`[ˆ\s]`	Non-whitespace	in␣Concord

Figure 2.7 Aliases for common sets of characters.

A range of numbers can also be specified. So `/{n,m}/` specifies from *n* to *m* occurrences of the previous char or expression, and `/{n,}/` means at least *n* occurrences of the previous expression. REs for counting are summarized in Fig. 2.8.

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
`{n}`	*n* occurrences of the previous char or expression
`{n,m}`	from *n* to *m* occurrences of the previous char or expression
`{n,}`	at least *n* occurrences of the previous char or expression
`{,m}`	up to *m* occurrences of the previous char or expression

Figure 2.8 Regular expression operators for counting.

Newline

Finally, certain special characters are referred to by special notation based on the backslash (\) (see Fig. 2.9). The most common of these are the **newline** character `\n` and the **tab** character `\t`. To refer to characters that are special themselves (like `.`, `*`, `[`, and `\`), precede them with a backslash, (i.e., `/\./`, `/\*/`, `/\[/`, and `/\\/`).

RE	Match	First Patterns Matched
`\*`	an asterisk "*"	"K*A*P*L*A*N"
`\.`	a period "."	"Dr. Livingston, I presume"
`\?`	a question mark	"Why don't they come and lend a hand?"
`\n`	a newline	
`\t`	a tab	

Figure 2.9 Some characters that need to be backslashed.

2.1.6 Substitution, Capture Groups, and ELIZA

substitution

An important use of regular expressions is in **substitutions**. For example, the substitution operator `s/regexp1/pattern/` used in Python and in Unix commands like `vim` or `sed` allows a string characterized by a regular expression to be replaced by another string:

```
s/colour/color/
```

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For example, suppose we wanted to put angle brackets around all integers in a text, for example, changing *the 35 boxes* to *the <35> boxes*. We'd like a way to refer to the integer we've found so that we can easily add the brackets. To do this, we put parentheses `(` and `)` around the first pattern and use the **number** operator `\1` in the second pattern to refer back. Here's how it looks:

```
s/([0-9]+)/<\1>/
```

The parenthesis and number operators can also specify that a certain string or expression must occur twice in the text. For example, suppose we are looking for the pattern “the Xer they were, the Xer they will be”, where we want to constrain the two X’s to be the same string. We do this by surrounding the first X with the parenthesis operator, and replacing the second X with the number operator `\1`, as follows:

```
/the (.*)er they were, the \1er they will be/
```

Here the `\1` will be replaced by whatever string matched the first item in parentheses. So this will match *the bigger they were, the bigger they will be* but not *the bigger they were, the faster they will be*.

capture group

register

This use of parentheses to store a pattern in memory is called a **capture group**. Every time a capture group is used (i.e., parentheses surround a pattern), the resulting match is stored in a numbered **register**. If you match two different sets of parentheses, `\2` means whatever matched the *second* capture group. Thus

```
/the (.*)er they (.*), the \1er we \2/
```

will match *the faster they ran, the faster we ran* but not *the faster they ran, the faster we ate*. Similarly, the third capture group is stored in `\3`, the fourth is `\4`, and so on.

non-capturing group

Parentheses thus have a double function in regular expressions; they are used to group terms for specifying the order in which operators should apply, and they are used to capture something in a register. Occasionally we might want to use parentheses for grouping, but don’t want to capture the resulting pattern in a register. In that case we use a **non-capturing group**, which is specified by putting the commands `?:` after the open paren, in the form `(?: pattern )`.

```
/(?:some|a few) (people|cats) like some \1/
```

will match *some cats like some cats* but not *some cats like some a few*.

Substitutions and capture groups are very useful in implementing simple chatbots like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulates a Rogerian psychologist by carrying on conversations like the following:

User$_1$:	Men are all alike.
ELIZA$_1$:	IN WHAT WAY
User$_2$:	They’re always bugging us about something or other.
ELIZA$_2$:	CAN YOU THINK OF A SPECIFIC EXAMPLE
User$_3$:	Well, my boyfriend made me come here.
ELIZA$_3$:	YOUR BOYFRIEND MADE YOU COME HERE
User$_4$:	He says I’m depressed much of the time.
ELIZA$_4$:	I AM SORRY TO HEAR YOU ARE DEPRESSED

ELIZA works by having a series or cascade of regular expression substitutions each of which matches and changes some part of the input lines. Input lines are first uppercased. The first substitutions then change all instances of *MY* to *YOUR*, and *I’M* to *YOU ARE*, and so on. The next set of substitutions matches and replaces other patterns in the input. Here are some examples:

```
s/.* I’M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* I AM (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
```

Since multiple substitutions can apply to a given input, substitutions are assigned a rank and applied in order. Creating patterns is the topic of Exercise 2.3, and we return to the details of the ELIZA architecture in Chapter 26.

2.1.7 Lookahead Assertions

Finally, there will be times when we need to predict the future: look ahead in the text to see if some pattern matches, but not advance the match cursor, so that we can then deal with the pattern if it occurs.

lookahead

zero-width

These **lookahead** assertions make use of the `(?` syntax that we saw in the previous section for non-capture groups. The operator `(?= pattern)` is true if `pattern` occurs, but is **zero-width**, i.e. the match pointer doesn't advance. The operator `(?! pattern)` only returns true if a pattern does not match, but again is zero-width and doesn't advance the cursor. Negative lookahead is commonly used when we are parsing some complex pattern but want to rule out a special case. For example suppose we want to match, at the beginning of a line, any single word that doesn't start with "Volcano". We can use negative lookahead to do this:

```
/ˆ(?!Volcano)[A-Za-z]+/
```

2.2 Words

corpus

corpora

Before we talk about processing words, we need to decide what counts as a word. Let's start by looking at one particular **corpus** (plural **corpora**), a computer-readable collection of text or speech. For example the Brown corpus is a million-word collection of samples from 500 written English texts from different genres (newspaper, fiction, non-fiction, academic, etc.), assembled at Brown University in 1963–64 (Kučera and Francis, 1967). How many words are in the following Brown sentence?

He stepped out into the hall, was delighted to encounter a water brother.

This sentence has 13 words if we don't count punctuation marks as words, 15 if we count punctuation. Whether we treat period ("."), comma (","), and so on as words depends on the task. Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks). For some tasks, like part-of-speech tagging or parsing or speech synthesis, we sometimes treat punctuation marks as if they were separate words.

The Switchboard corpus of American English telephone conversations between strangers was collected in the early 1990s; it contains 2430 conversations averaging 6 minutes each, totaling 240 hours of speech and about 3 million words (Godfrey et al., 1992). Such corpora of spoken language don't have punctuation but do introduce other complications with regard to defining words. Let's look at one utterance from Switchboard; an **utterance** is the spoken correlate of a sentence:

utterance

I do uh main- mainly business data processing

disfluency

fragment

filled pause

This utterance has two kinds of **disfluencies**. The broken-off word *main-* is called a **fragment**. Words like *uh* and *um* are called **fillers** or **filled pauses**. Should we consider these to be words? Again, it depends on the application. If we are building a speech transcription system, we might want to eventually strip out the disfluencies.

But we also sometimes keep disfluencies around. Disfluencies like *uh* or *um* are actually helpful in speech recognition in predicting the upcoming word, because they may signal that the speaker is restarting the clause or idea, and so for speech recognition they are treated as regular words. Because people use different disfluencies they can also be a cue to speaker identification. In fact Clark and Fox Tree (2002) showed that *uh* and *um* have different meanings. What do you think they are?

Are capitalized tokens like *They* and uncapitalized tokens like *they* the same word? These are lumped together in some tasks (speech recognition), while for part-of-speech or named-entity tagging, capitalization is a useful feature and is retained.

lemma

wordform

How about inflected forms like *cats* versus *cat*? These two words have the same **lemma** *cat* but are different wordforms. A **lemma** is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense. The **wordform** is the full inflected or derived form of the word. For morphologically complex languages like Arabic, we often need to deal with lemmatization. For many tasks in English, however, wordforms are sufficient.

word type

word token

How many words are there in English? To answer this question we need to distinguish two ways of talking about words. **Types** are the number of distinct words in a corpus; if the set of words in the vocabulary is V, the number of types is the vocabulary size $|V|$. **Tokens** are the total number N of running words. If we ignore punctuation, the following Brown sentence has 16 tokens and 14 types:

> They picnicked by the pool, then lay back on the grass and looked at the stars.

When we speak about the number of words in the language, we are generally referring to word types.

Corpus	Tokens = N	Types = $|V|$
Shakespeare	884 thousand	31 thousand
Brown corpus	1 million	38 thousand
Switchboard telephone conversations	2.4 million	20 thousand
COCA	440 million	2 million
Google N-grams	1 trillion	13 million

Figure 2.10 Rough numbers of types and tokens for some English language corpora. The largest, the Google N-grams corpus, contains 13 million types, but this count only includes types appearing 40 or more times, so the true number would be much larger.

Herdan's Law

Heaps' Law

Fig. 2.10 shows the rough numbers of types and tokens computed from some popular English corpora. The larger the corpora we look at, the more word types we find, and in fact this relationship between the number of types $|V|$ and number of tokens N is called **Herdan's Law** (Herdan, 1960) or **Heaps' Law** (Heaps, 1978) after its discoverers (in linguistics and information retrieval respectively). It is shown in Eq. 2.1, where k and β are positive constants, and $0 < \beta < 1$.

$$|V| = kN^{\beta} \tag{2.1}$$

The value of β depends on the corpus size and the genre, but at least for the large corpora in Fig. 2.10, β ranges from .67 to .75. Roughly then we can say that the vocabulary size for a text goes up significantly faster than the square root of its length in words.

Another measure of the number of words in the language is the number of lemmas instead of wordform types. Dictionaries can help in giving lemma counts; dictionary **entries** or **boldface forms** are a very rough upper bound on the number of lemmas (since some lemmas have multiple boldface forms). The 1989 edition of the Oxford English Dictionary had 615,000 entries.

2.3 Corpora

Words don't appear out of nowhere. Any particular piece of text that we study is produced by one or more specific speakers or writers, in a specific dialect of a specific language, at a specific time, in a specific place, for a specific function.

Perhaps the most important dimension of variation is the language. NLP algorithms are most useful when they apply across many languages. The world has 7097 languages at the time of this writing, according to the online Ethnologue catalog (Simons and Fennig, 2018). Most NLP tools tend to be developed for the official languages of large industrialized nations (Chinese, English, Spanish, Arabic, etc.), but we don't want to limit tools to just these few languages. Furthermore, most languages also have multiple varieties, such as dialects spoken in different regions or by different social groups. Thus, for example, if we're processing text in African American Vernacular English (**AAVE**), a dialect spoken by millions of people in the United States, it's important to make use of NLP tools that function with that dialect. Twitter posts written in AAVE make use of constructions like *iont* (*I don't* in Standard American English (**SAE**)), or *talmbout* corresponding to SAE *talking about*, both examples that influence word segmentation (Blodgett et al. 2016, Jones 2015).

AAVE

SAE

It's also quite common for speakers or writers to use multiple languages in a single communicative act, a phenomenon called **code switching**. Code switching is enormously common across the world; here are examples showing Spanish and (transliterated) Hindi code switching with English (Solorio et al. 2014, Jurgens et al. 2017):

code switching

(2.2) Por primera vez veo a @username actually being hateful! it was beautiful:)
[For the first time I get to see @username actually being hateful! it was beautiful:)]

(2.3) dost tha or ra- hega ... dont wory ... but dherya rakhe
[“he was and will remain a friend ... don't worry ... but have faith”]

Another dimension of variation is the genre. The text that our algorithms must process might come from newswire, fiction or non-fiction books, scientific articles, Wikipedia, or religious texts. It might come from spoken genres like telephone conversations, business meetings, police body-worn cameras, medical interviews, or transcripts of television shows or movies. It might come from work situations like doctors' notes, legal text, or parliamentary or congressional proceedings.

Text also reflects the demographic characteristics of the writer (or speaker): their age, gender, race, socioeconomic class can all influence the linguistic properties of the text we are processing.

And finally, time matters too. Language changes over time, and for some languages we have good corpora of texts from different historical periods.

Because language is so situated, when developing computational models for language processing, it's important to consider who produced the language, in what context, for what purpose, and make sure that the models are fit to the data.

2.4 Text Normalization

Before almost any natural language processing of a text, the text has to be normalized. At least three tasks are commonly applied as part of any normalization process:

1. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences

In the next sections we walk through each of these tasks.

2.4.1 Unix Tools for Crude Tokenization and Normalization

Let's begin with an easy, if somewhat naive version of word tokenization and normalization (and frequency computation) that can be accomplished for English solely in a single UNIX command-line, inspired by Church (1994). We'll make use of some Unix commands: `tr`, used to systematically change particular characters in the input; `sort`, which sorts input lines in alphabetical order; and `uniq`, which collapses and counts adjacent identical lines.

For example let's begin with the 'complete words' of Shakespeare in one textfile, `sh.txt`. We can use `tr` to tokenize the words by changing every sequence of non-alphabetic characters to a newline ('A-Za-z' means alphabetic, the -c option complements to non-alphabet, and the -s option squeezes all sequences into a single character):

```
tr -sc 'A-Za-z' '\n' < sh.txt
```

The output of this command will be:

```
THE
SONNETS
by
William
Shakespeare
From
fairest
creatures
We
...
```

Now that there is one word per line, we can sort the lines, and pass them to `uniq -c` which will collapse and count them:

```
tr -sc 'A-Za-z' '\n' < sh.txt | sort | uniq -c
```

with the following output:

```
1945 A
72 AARON
19 ABBESS
25 Aaron
6 Abate
1 Abates
5 Abbess
6 Abbey
```

```
3 Abbot
...
```

Alternatively, we can collapse all the upper case to lower case:

```
tr -sc ’A-Za-z’ ’\n’ < sh.txt | tr A-Z a-z | sort | uniq -c
```

whose output is

```
14725 a
   97 aaron
    1 abaissiez
   10 abandon
    2 abandoned
    2 abase
    1 abash
   14 abate
    3 abated
    3 abatement
    ...
```

Now we can sort again to find the frequent words. The `-n` option to `sort` means to sort numerically rather than alphabetically, and the `-r` option means to sort in reverse order (highest-to-lowest):

```
tr -sc ’A-Za-z’ ’\n’ < sh.txt | tr A-Z a-z | sort | uniq -c | sort -n -r
```

The results show that the most frequent words in Shakespeare, as in any other corpus, are the short **function words** like articles, pronouns, prepositions:

```
27378 the
26084 and
22538 i
19771 to
17481 of
14725 a
13826 you
12489 my
11318 that
11112 in
    ...
```

Unix tools of this sort can be very handy in building quick word count statistics for any corpus.

2.4.2 Word Tokenization

tokenization

The simple UNIX tools above were fine for getting rough word statistics but more sophisticated algorithms are generally necessary for **tokenization**, the task of segmenting running text into words.

While the Unix command sequence just removed all the numbers and punctuation, for most NLP applications we’ll need to keep these in our tokenization. We often want to break off punctuation as a separate token; commas are a useful piece of information for parsers, periods help indicate sentence boundaries. But we’ll often want to keep the punctuation that occurs word internally, in examples like *m.p.h,*, *Ph.D.*, *AT&T*, *cap’n*. Special characters and numbers will need to be kept in prices

($45.55) and dates (`01/02/06`); we don't want to segment that price into separate tokens of "45" and "55". And there are URLs (`http://www.stanford.edu`), Twitter hashtags (`#nlproc`), or email addresses (`someone@cs.colorado.edu`).

Number expressions introduce other complications as well; while commas normally appear at word boundaries, commas are used inside numbers in English, every three digits: `555,500.50`. Languages, and hence tokenization requirements, differ on this; many continental European languages like Spanish, French, and German, by contrast, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, `555 500,50`.

clitic

A tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes, for example, converting `what're` to the two tokens `what are`, and `we're` to `we are`. A clitic is a part of a word that can't stand on its own, and can only occur when it is attached to another word. Some such contractions occur in other alphabetic languages, including articles and pronouns in French (`j'ai`, `l'homme`).

Depending on the application, tokenization algorithms may also tokenize multiword expressions like `New York` or `rock 'n' roll` as a single token, which requires a multiword expression dictionary of some sort. Tokenization is thus intimately tied up with **named entity detection**, the task of detecting names, dates, and organizations (Chapter 18).

Penn Treebank tokenization

One commonly used tokenization standard is known as the **Penn Treebank tokenization** standard, used for the parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC), the source of many useful datasets. This standard separates out clitics (*doesn't* becomes *does* plus *n't*), keeps hyphenated words together, and separates out all punctuation (to save space we're showing visible spaces '␣' between tokens, although newlines is a more common output):

Input: `"The San Francisco-based restaurant," they said,`
`"doesn't charge $10".`

Output: `"␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣`
`"␣does␣n't␣charge␣$␣10␣"␣.`

In practice, since tokenization needs to be run before any other language processing, it needs to be very fast. The standard method for tokenization is therefore to use deterministic algorithms based on regular expressions compiled into very efficient finite state automata. For example, Fig. 2.11 shows an example of a basic regular expression that can be used to tokenize with the `nltk.regexp_tokenize` function of the Python-based Natural Language Toolkit (NLTK) (Bird et al. 2009; `http://www.nltk.org`).

Carefully designed deterministic algorithms can deal with the ambiguities that arise, such as the fact that the apostrophe needs to be tokenized differently when used as a genitive marker (as in *the book's cover*), a quotative as in *'The other class', she said*, or in clitics like *they're*.

Word tokenization is more complex in languages like written Chinese, Japanese, and Thai, which do not use spaces to mark potential word-boundaries.

hanzi

In Chinese, for example, words are composed of characters (called **hanzi** in Chinese). Each character generally represents a single unit of meaning (called a **morpheme**) and is pronounceable as a single syllable. Words are about 2.4 characters long on average. But deciding what counts as a word in Chinese is complex. For example, consider the following sentence:

(2.4) 姚明进入总决赛
"Yao Ming reaches the finals"

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Figure 2.11 A python trace of regular expression tokenization in the NLTK (Bird et al., 2009) Python-based natural language processing toolkit, commented for readability; the `(?x)` verbose flag tells Python to strip comments and whitespace. Figure from Chapter 3 of Bird et al. (2009).

As Chen et al. (2017) point out, this could be treated as 3 words ('Chinese Treebank' segmentation):

(2.5) 姚明 进入 总决赛
YaoMing reaches finals

or as 5 words ('Peking University' segmentation):

(2.6) 姚 明 进入 总 决赛
Yao Ming reaches overall finals

Finally, it is possible in Chinese simply to ignore words altogether and use characters as the basic elements, treating the sentence as a series of 7 characters:

(2.7) 姚 明 进 入 总 决 赛
Yao Ming enter enter overall decision game

In fact, for most Chinese NLP tasks it turns out to work better to take characters rather than words as input, since characters are at a reasonable semantic level for most applications, and since most word standards result in a huge vocabulary with large numbers of very rare words (Li et al., 2019).

word segmentation

However, for Japanese and Thai the character is too small a unit, and so algorithms for **word segmentation** are required. These can also be useful for Chinese in the rare situations where word rather than character boundaries are required. The standard segmentation algorithms for these languages use neural **sequence models** trained via supervised machine learning on hand-segmented training sets; we'll introduce sequence models in Chapter 8.

2.4.3 Byte-Pair Encoding for Tokenization

There is a third option to tokenizing text input. Instead of defining tokens as words (defined by spaces in orthographies that have spaces, or more complex algorithms), or as characters (as in Chinese), we can use our data to automatically tell us what size tokens should be. Perhaps sometimes we might want tokens that are space-delimited words (like *spinach*) other times it's useful to have tokens that are larger than words (like *New York Times*), and sometimes smaller than words (like the morphemes *-est* or *-er*. A morpheme is the smallest meaning-bearing unit of a language; for example the word *unlikeliest* has the morphemes *un-*, *likely*, and *-est*; we'll return to this on page 20.

subword

One reason it's helpful to have **subword** tokens is to deal with unknown words.

2018). BPE and wordpiece both assume that we already have some initial tokenization of words (such as by spaces, or from some initial dictionary) and so we never tried to induce word parts across spaces. By contrast, the SentencePiece model works from raw text; even whitespace is handled as a normal symbol. Thus it doesn't need an initial tokenization or word-list, and can be used in languages like Chinese or Japanese that don't have spaces.

2.4.4 Word Normalization, Lemmatization and Stemming

normalization

Word **normalization** is the task of putting words/tokens in a standard format, choosing a single normal form for words with multiple forms like `USA` and `US` or `uh-huh` and `uhhuh`. This standardization may be valuable, despite the spelling information that is lost in the normalization process. For information retrieval or information extraction about the US, we might want see information from documents whether they mention the `US` or the `USA`.

case folding

Case folding is another kind of normalization. Mapping everything to lower case means that *Woodchuck* and *woodchuck* are represented identically, which is very helpful for generalization in many tasks, such as information retrieval or speech recognition. For sentiment analysis and other text classification tasks, information extraction, and machine translation, by contrast, case can be quite helpful and case folding is generally not done. This is because maintaining the difference between, for example, `US` the country and `us` the pronoun can outweigh the advantage in generalization that case folding would have provided for other words.

For many natural language processing situations we also want two morphologically different forms of a word to behave similarly. For example in web search, someone may type the string *woodchucks* but a useful system might want to also return pages that mention *woodchuck* with no *s*. This is especially common in morphologically complex languages like Russian, where for example the word *Moscow* has different endings in the phrases *Moscow*, *of Moscow*, *to Moscow*, and so on.

Lemmatization is the task of determining that two words have the same root, despite their surface differences. The words *am*, *are*, and *is* have the shared lemma *be*; the words *dinner* and *dinners* both have the lemma *dinner*. Lemmatizing each of these forms to the same lemma will let us find all mentions of words in Russian like Moscow. The lemmatized form of a sentence like *He is reading detective stories* would thus be *He be read detective story*.

morpheme
stem
affix

How is lemmatization done? The most sophisticated methods for lemmatization involve complete **morphological parsing** of the word. **Morphology** is the study of the way words are built up from smaller meaning-bearing units called **morphemes**. Two broad classes of morphemes can be distinguished: **stems**—the central morpheme of the word, supplying the main meaning— and **affixes**—adding "additional" meanings of various kinds. So, for example, the word *fox* consists of one morpheme (the morpheme *fox*) and the word *cats* consists of two: the morpheme *cat* and the morpheme *-s*. A morphological parser takes a word like *cats* and parses it into the two morphemes *cat* and *s*, or a Spanish word like *amaren* ('if in the future they would love') into the morphemes *amar* 'to love', *3PL*, and *future subjunctive*.

The Porter Stemmer

stemming
Porter stemmer

Lemmatization algorithms can be complex. For this reason we sometimes make use of a simpler but cruder method, which mainly consists of chopping off word-final affixes. This naive version of morphological analysis is called **stemming**. One of the most widely used stemming algorithms is the Porter (1980). The Porter stemmer applied to the following paragraph:

```
This was not the map we found in Billy Bones’s chest, but
an accurate copy, complete in all things-names and heights
and soundings-with the single exception of the red crosses
and the written notes.
```

produces the following stemmed output:

```
Thi wa not the map we found in Billi Bone s chest but an
accur copi complet in all thing name and height and sound
with the singl except of the red cross and the written note
```

cascade

The algorithm is based on series of rewrite rules run in series, as a **cascade**, in which the output of each pass is fed as input to the next pass; here is a sampling of the rules:

ATIONAL → ATE (e.g., relational → relate)
ING → ε if stem contains vowel (e.g., motoring → motor)
SSES → SS (e.g., grasses → grass)

Detailed rule lists for the Porter stemmer, as well as code (in Java, Python, etc.) can be found on Martin Porter’s homepage; see also the original paper (Porter, 1980).

Simple stemmers can be useful in cases where we need to collapse across different variants of the same lemma. Nonetheless, they do tend to commit errors of both over- and under-generalizing, as shown in the table below (Krovetz, 1993):

Errors of Commission		**Errors of Omission**	
organization	organ	European	Europe
doing	doe	analysis	analyzes
numerical	numerous	noise	noisy
policy	police	sparse	sparsity

2.4.5 Sentence Segmentation

Sentence segmentation

Sentence segmentation is another important step in text processing. The most useful cues for segmenting a text into sentences are punctuation, like periods, question marks, and exclamation points. Question marks and exclamation points are relatively unambiguous markers of sentence boundaries. Periods, on the other hand, are more ambiguous. The period character “.” is ambiguous between a sentence boundary marker and a marker of abbreviations like *Mr.* or *Inc.* The previous sentence that you just read showed an even more complex case of this ambiguity, in which the final period of *Inc.* marked both an abbreviation and the sentence boundary marker. For this reason, sentence tokenization and word tokenization may be addressed jointly.

In general, sentence tokenization methods work by first deciding (based on rules or machine learning) whether a period is part of the word or is a sentence-boundary marker. An abbreviation dictionary can help determine whether the period is part of a commonly used abbreviation; the dictionaries can be hand-built or machine-learned (Kiss and Strunk, 2006), as can the final sentence splitter. In the Stanford CoreNLP toolkit (Manning et al., 2014), for example sentence splitting is rule-based, a deterministic consequence of tokenization; a sentence ends when a sentence-ending punctuation (., !, or ?) is not already grouped with other characters into a token (such as for an abbreviation or number), optionally followed by additional final quotes or brackets.

CHAPTER

8 Part-of-Speech Tagging

parts of speech

Dionysius Thrax of Alexandria (*c.* 100 B.C.), or perhaps someone else (it was a long time ago), wrote a grammatical sketch of Greek (a "*technē*") that summarized the linguistic knowledge of his day. This work is the source of an astonishing proportion of modern linguistic vocabulary, including words like *syntax*, *diphthong*, *clitic*, and *analogy*. Also included are a description of eight **parts of speech**: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article. Although earlier scholars (including Aristotle as well as the Stoics) had their own lists of parts of speech, it was Thrax's set of eight that became the basis for practically all subsequent part-of-speech descriptions of most European languages for the next 2000 years.

Schoolhouse Rock was a series of popular animated educational television clips from the 1970s. Its Grammar Rock sequence included songs about exactly 8 parts of speech, including the late great Bob Dorough's *Conjunction Junction*:

> *Conjunction Junction, what's your function?*
> *Hooking up words and phrases and clauses...*

Although the list of 8 was slightly modified from Thrax's original, the astonishing durability of the parts of speech through two millennia is an indicator of both the importance and the transparency of their role in human language.[1]

POS

Parts of speech (also known as **POS**, **word classes**, or **syntactic categories**) are useful because they reveal a lot about a word and its neighbors. Knowing whether a word is a **noun** or a **verb** tells us about likely neighboring words (nouns are preceded by determiners and adjectives, verbs by nouns) and syntactic structure (nouns are generally part of noun phrases), making part-of-speech tagging a key aspect of parsing (Chapter 13). Parts of speech are useful features for labeling **named entities** like people or organizations in **information extraction** (Chapter 18), or for coreference resolution (Chapter 22). A word's part of speech can even play a role in speech recognition or synthesis, e.g., the word *content* is pronounced *CONtent* when it is a noun and *conTENT* when it is an adjective.

This chapter introduces parts of speech, and then introduces two algorithms for **part-of-speech tagging**, the task of assigning parts of speech to words. One is generative— Hidden Markov Model (HMM)—and one is discriminative—the Maximum Entropy Markov Model (MEMM). Chapter 9 then introduces a third algorithm based on the recurrent neural network (RNN). All three have roughly equal performance but, as we'll see, have different tradeoffs.

8.1 (Mostly) English Word Classes

Until now we have been using part-of-speech terms like **noun** and **verb** rather freely. In this section we give a more complete definition of these and other classes. While word classes do have semantic tendencies—adjectives, for example, often describe

[1] Nonetheless, eight isn't very many and, as we'll see, recent tagsets have more.

properties and nouns *people*— parts of speech are traditionally defined instead based on syntactic and morphological function, grouping words that have similar neighboring words (their **distributional** properties) or take similar affixes (their morphological properties).

closed class
open class

Parts of speech can be divided into two broad supercategories: **closed class** types and **open class** types. Closed classes are those with relatively fixed membership, such as prepositions—new prepositions are rarely coined. By contrast, nouns and verbs are open classes—new nouns and verbs like *iPhone* or *to fax* are continually being created or borrowed. Any given speaker or corpus may have different open class words, but all speakers of a language, and sufficiently large corpora, likely share the set of closed class words. Closed class words are generally **function words** like *of*, *it*, *and*, or *you*, which tend to be very short, occur frequently, and often have structuring uses in grammar.

function word

Four major open classes occur in the languages of the world: **nouns**, **verbs**, **adjectives**, and **adverbs**. English has all four, although not every language does. The syntactic class **noun** includes the words for most people, places, or things, but others as well. Nouns include concrete terms like *ship* and *chair*, abstractions like *bandwidth* and *relationship*, and verb-like terms like *pacing* as in *His pacing to and fro became quite annoying*. What defines a noun in English, then, are things like its ability to occur with determiners (*a goat, its bandwidth, Plato's Republic*), to take possessives (*IBM's annual revenue*), and for most but not all nouns to occur in the plural form (*goats, abaci*).

noun

proper noun

Open class nouns fall into two classes. **Proper nouns**, like *Regina*, *Colorado*, and *IBM*, are names of specific persons or entities. In English, they generally aren't preceded by articles (e.g., *the book is upstairs*, but *Regina is upstairs*). In written English, proper nouns are usually capitalized. The other class, **common nouns**, are divided in many languages, including English, into **count nouns** and **mass nouns**. Count nouns allow grammatical enumeration, occurring in both the singular and plural (*goat/goats, relationship/relationships*) and they can be counted (*one goat, two goats*). Mass nouns are used when something is conceptualized as a homogeneous group. So words like *snow, salt*, and *communism* are not counted (i.e., **two snows* or **two communisms*). Mass nouns can also appear without articles where singular count nouns cannot (*Snow is white* but not **Goat is white*).

common noun
count noun
mass noun

verb

Verbs refer to actions and processes, including main verbs like *draw*, *provide*, and *go*. English verbs have inflections (non-third-person-sg (*eat*), third-person-sg (*eats*), progressive (*eating*), past participle (*eaten*)). While many researchers believe that all human languages have the categories of noun and verb, others have argued that some languages, such as Riau Indonesian and Tongan, don't even make this distinction (Broschart 1997; Evans 2000; Gil 2000) .

adjective

The third open class English form is **adjectives**, a class that includes many terms for properties or qualities. Most languages have adjectives for the concepts of color (*white*, *black*), age (*old*, *young*), and value (*good*, *bad*), but there are languages without adjectives. In Korean, for example, the words corresponding to English adjectives act as a subclass of verbs, so what is in English an adjective "beautiful" acts in Korean like a verb meaning "to be beautiful".

adverb

The final open class form, **adverbs**, is rather a hodge-podge in both form and meaning. In the following all the italicized words are adverbs:

Actually, I ran *home extremely quickly yesterday*

What coherence the class has semantically may be solely that each of these words can be viewed as modifying something (often verbs, hence the name "ad-

locative
degree
manner
temporal

verb”, but also other adverbs and entire verb phrases). **Directional adverbs** or **locative adverbs** (*home*, *here*, *downhill*) specify the direction or location of some action; **degree adverbs** (*extremely*, *very*, *somewhat*) specify the extent of some action, process, or property; **manner adverbs** (*slowly*, *slinkily*, *delicately*) describe the manner of some action or process; and **temporal adverbs** describe the time that some action or event took place (*yesterday*, *Monday*). Because of the heterogeneous nature of this class, some adverbs (e.g., temporal adverbs like *Monday*) are tagged in some tagging schemes as nouns.

The closed classes differ more from language to language than do the open classes. Some of the important closed classes in English include:

> **prepositions:** on, under, over, near, by, at, from, to, with
> **particles:** up, down, on, off, in, out, at, by
> **determiners:** a, an, the
> **conjunctions:** and, but, or, as, if, when
> **pronouns:** she, who, I, others
> **auxiliary verbs:** can, may, should, are
> **numerals:** one, two, three, first, second, third

preposition
particle

Prepositions occur before noun phrases. Semantically they often indicate spatial or temporal relations, whether literal (*on it*, *before then*, *by the house*) or metaphorical (*on time*, *with gusto*, *beside herself*), but often indicate other relations as well, like marking the agent in *Hamlet was written by Shakespeare*. A **particle** resembles a preposition or an adverb and is used in combination with a verb. Particles often have extended meanings that aren’t quite the same as the prepositions they resemble, as in the particle *over* in *she turned the paper over*.

phrasal verb

A verb and a particle that act as a single syntactic and/or semantic unit are called a **phrasal verb**. The meaning of phrasal verbs is often problematically **non-compositional**—not predictable from the distinct meanings of the verb and the particle. Thus, *turn down* means something like ‘reject’, *rule out* ‘eliminate’, *find out* ‘discover’, and *go on* ‘continue’.

determiner
article

A closed class that occurs with nouns, often marking the beginning of a noun phrase, is the **determiner**. One small subtype of determiners is the **article**: English has three articles: *a*, *an*, and *the*. Other determiners include *this* and *that* (*this chapter*, *that page*). *A* and *an* mark a noun phrase as indefinite, while *the* can mark it as definite; definiteness is a discourse property (Chapter 23). Articles are quite frequent in English; indeed, *the* is the most frequently occurring word in most corpora of written English, and *a* and *an* are generally right behind.

conjunctions
complementizer

Conjunctions join two phrases, clauses, or sentences. Coordinating conjunctions like *and*, *or*, and *but* join two elements of equal status. Subordinating conjunctions are used when one of the elements has some embedded status. For example, *that* in *“I thought that you might like some milk”* is a subordinating conjunction that links the main clause *I thought* with the subordinate clause *you might like some milk*. This clause is called subordinate because this entire clause is the “content” of the main verb *thought*. Subordinating conjunctions like *that* which link a verb to its argument in this way are also called **complementizers**.

pronoun
personal
possessive
wh

Pronouns are forms that often act as a kind of shorthand for referring to some noun phrase or entity or event. **Personal pronouns** refer to persons or entities (*you*, *she*, *I*, *it*, *me*, etc.). **Possessive pronouns** are forms of personal pronouns that indicate either actual possession or more often just an abstract relation between the person and some object (*my, your, his, her, its, one’s, our, their*). **Wh-pronouns** (*what, who, whom, whoever*) are used in certain question forms, or may also act as complementizers (*Frida, who married Diego...*).

auxiliary

A closed class subtype of English verbs are the **auxiliary** verbs. Cross-linguistically, auxiliaries mark semantic features of a main verb: whether an action takes place in the present, past, or future (tense), whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, or desired (mood). English auxiliaries include the **copula** verb *be*, the two verbs *do* and *have*, along with their inflected forms, as well as a class of **modal verbs**. *Be* is called a copula because it connects subjects with certain kinds of predicate nominals and adjectives (*He is a duck*). The verb *have* can mark the perfect tenses (*I have gone*, *I had gone*), and *be* is used as part of the passive (*We were robbed*) or progressive (*We are leaving*) constructions. Modals are used to mark the mood associated with the event depicted by the main verb: *can* indicates ability or possibility, *may* permission or possibility, *must* necessity. There is also a modal use of *have* (e.g., *I have to go*).

copula

modal

interjection

negative

English also has many words of more or less unique function, including **interjections** (*oh, hey, alas, uh, um*), **negatives** (*no, not*), **politeness markers** (*please, thank you*), **greetings** (*hello, goodbye*), and the existential **there** (*there are two on the table*) among others. These classes may be distinguished or lumped together as interjections or adverbs depending on the purpose of the labeling.

8.2 The Penn Treebank Part-of-Speech Tagset

An important tagset for English is the 45-tag Penn Treebank tagset (Marcus et al., 1993), shown in Fig. 8.1, which has been used to label many corpora. In such labelings, parts of speech are generally represented by placing the tag after each word, delimited by a slash:

Tag	Description	Example	Tag	Description	Example	Tag	Description	Example
CC	coordinating conjunction	*and, but, or*	PDT	predeterminer	*all, both*	VBP	verb non-3sg present	*eat*
CD	cardinal number	*one, two*	POS	possessive ending	*'s*	VBZ	verb 3sg pres	*eats*
DT	determiner	*a, the*	PRP	personal pronoun	*I, you, he*	WDT	wh-determ.	*which, that*
EX	existential 'there'	*there*	PRP$	possess. pronoun	*your, one's*	WP	wh-pronoun	*what, who*
FW	foreign word	*mea culpa*	RB	adverb	*quickly*	WP$	wh-possess.	*whose*
IN	preposition/ subordin-conj	*of, in, by*	RBR	comparative adverb	*faster*	WRB	wh-adverb	*how, where*
JJ	adjective	*yellow*	RBS	superlatv. adverb	*fastest*	$	dollar sign	*$*
JJR	comparative adj	*bigger*	RP	particle	*up, off*	#	pound sign	#
JJS	superlative adj	*wildest*	SYM	symbol	+,%, &	“	left quote	‘ or “
LS	list item marker	*1, 2, One*	TO	“to”	*to*	”	right quote	’ or ”
MD	modal	*can, should*	UH	interjection	*ah, oops*	(	left paren	[, (, {, <
NN	sing or mass noun	*llama*	VB	verb base form	*eat*	)	right paren	],), }, >
NNS	noun, plural	*llamas*	VBD	verb past tense	*ate*	,	comma	,
NNP	proper noun, sing.	*IBM*	VBG	verb gerund	*eating*	.	sent-end punc	. ! ?
NNPS	proper noun, plu.	*Carolinas*	VBN	verb past part.	*eaten*	:	sent-mid punc	: ; ... – -

Figure 8.1 Penn Treebank part-of-speech tags (including punctuation).

(8.1) The/DT grand/JJ jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS ./.

(8.2) **There/EX** are/VBP 70/CD children/NNS **there/RB**

(8.3) Preliminary/JJ findings/NNS were/VBD **reported/VBN** in/IN today/NN **'s/POS** New/NNP England/NNP Journal/NNP of/IN Medicine/NNP ./.

Example (8.1) shows the determiners *the* and *a*, the adjectives *grand* and *other*, the common nouns *jury*, *number*, and *topics*, and the past tense verb *commented*. Example (8.2) shows the use of the EX tag to mark the existential *there* construction in English, and, for comparison, another use of *there* which is tagged as an adverb (RB). Example (8.3) shows the segmentation of the possessive morpheme *'s*, and a passive construction, 'were reported', in which *reported* is tagged as a past participle (VBN). Note that since *New England Journal of Medicine* is a proper noun, the Treebank tagging chooses to mark each noun in it separately as NNP, including *journal* and *medicine*, which might otherwise be labeled as common nouns (NN).

Corpora labeled with parts of speech are crucial training (and testing) sets for statistical tagging algorithms. Three main tagged corpora are consistently used for training and testing part-of-speech taggers for English. The **Brown** corpus is a million words of samples from 500 written texts from different genres published in the United States in 1961. The **WSJ** corpus contains a million words published in the Wall Street Journal in 1989. The **Switchboard** corpus consists of 2 million words of telephone conversations collected in 1990-1991. The corpora were created by running an automatic part-of-speech tagger on the texts and then human annotators hand-corrected each tag.

Brown

WSJ

Switchboard

There are some minor differences in the tagsets used by the corpora. For example in the WSJ and Brown corpora, the single Penn tag TO is used for both the infinitive *to* (*I like to race*) and the preposition *to* (*go to the store*), while in Switchboard the tag TO is reserved for the infinitive use of *to* and the preposition is tagged IN:

Well/UH ,/, I/PRP ,/, I/PRP want/VBP **to/TO** go/VB **to/IN** a/DT restaurant/NN

Finally, there are some idiosyncracies inherent in any tagset. For example, because the Penn 45 tags were collapsed from a larger 87-tag tagset, the **original Brown tagset**, some potentially useful distinctions were lost. The Penn tagset was designed for a treebank in which sentences were parsed, and so it leaves off syntactic information recoverable from the parse tree. Thus for example the Penn tag IN is used for both subordinating conjunctions like *if, when, unless, after*:

after/IN spending/VBG a/DT day/NN at/IN the/DT beach/NN

and prepositions like *in, on, after*:

after/IN sunrise/NN

Words are generally tokenized before tagging. The Penn Treebank and the British National Corpus split contractions and the *'s*-genitive from their stems:[2]

would/MD n't/RB
children/NNS 's/POS

The Treebank tagset assumes that tokenization of multipart words like *New York* is done at whitespace, thus tagging. *a New York City firm* as *a/DT New/NNP York/NNP City/NNP firm/NN*.

Another commonly used tagset, the Universal POS tag set of the Universal Dependencies project (Nivre et al., 2016), is used when building systems that can tag many languages. See Section 8.7.

[2] Indeed, the Treebank tag POS is used only for *'s*, which must be segmented in tokenization.