

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

lemmatization

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

stemming

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.

sentence
segmentation

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

One of the unsung successes in standardization in computer science has been the **regular expression** (often shortened to **regex**), 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. Regular expressions 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.

corpus

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

concatenation

The simplest kind of regular expression is a sequence of simple characters; putting characters in sequence is called **concatenation**. 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 `/url/`) (see Fig. 2.1).

Regular expressions are **case sensitive**; lower case `/s/` is distinct from upper

Regex	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.

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 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*.

Regex	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	“ <u>W</u> oodchuck”
/[abc]/	‘a’, ‘b’, or ‘c’	“In uomini, in soldat <u>i</u> ”
/[1234567890]/	any digit	“plenty of <u>7</u> to 5”

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

The regular expression /[1234567890]/ specifies 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.

Regex	Match	Example Patterns Matched
/[A–Z]/	an upper case letter	“we should call it ‘ <u>D</u> renched Blossoms’ ”
/[a–z]/	a lower case letter	“ <u>m</u> y beans were impatient to be hoed!”
/[0–9]/	a single digit	“Chapter <u>1</u> : 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.

Regex	Match (single characters)	Example Patterns Matched
/[[^] A–Z]/	not an upper case letter	“Oyfn pri <u>p</u> etchik”
/[[^] Ss]/	neither ‘S’ nor ‘s’	“ <u>I</u> have no exquisite reason for’t”
/[[^] .]/	not a period	“ou <u>r</u> resident Djinn”
/[e [^]]/	either ‘e’ or ‘ [^] ’	“look up <u>^</u> 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.

Regex	Match	Example Patterns Matched
<code>/woodchucks?/</code>	woodchuck or woodchucks	<u>“woodchuck”</u>
<code>/colou?r/</code>	color or colour	<u>“color”</u>

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 *as*” 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 *as*”. This will match *a* or *aaaaaa*, but it will also match the empty string at the start of *Off Minor* since the string *Off Minor* starts with zero *a*’s. So the regular expression for matching one or more *a* is `/aa*/`, meaning one *a* followed by zero or more *as*. 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.

Regex	Match	Example Matches
<code>/beg.n/</code>	any character between <i>beg</i> and <i>n</i>	<u>begin</u> , <u>beg’n</u> , <u>begun</u>

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.)

Regex	Match
<code>^</code>	start of line
<code>\$</code>	end of line
<code>\b</code>	word boundary
<code>\B</code>	non-word boundary

Figure 2.7 Anchors in regular expressions.

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

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*.

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 *theeee* 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)

We'll come back to precision and recall with more precise definitions in Chapter 4.

2.1.4 More Operators

Figure 2.8 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*).

Regex	Expansion	Match	First Matches
<code>\d</code>	<code>[0-9]</code>	any digit	Party_of_5
<code>\D</code>	<code>[^0-9]</code>	any non-digit	Blue_moon
<code>\w</code>	<code>[a-zA-Z0-9_]</code>	any alphanumeric/underscore	Daiyu
<code>\W</code>	<code>[^\w]</code>	a non-alphanumeric	!!!!
<code>\s</code>	<code>[\r\t\n\f]</code>	whitespace (space, tab)	in_Concord
<code>\S</code>	<code>[^\s]</code>	Non-whitespace	in_Concord

Figure 2.8 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. 2.9.

Regex	Match
<code>*</code>	zero or more occurrences of the previous char or expression
<code>+</code>	one or more occurrences of the previous char or expression
<code>?</code>	zero or one occurrence of the previous char or expression
<code>{n}</code>	exactly <i>n</i> occurrences of the previous char or expression
<code>{n,m}</code>	from <i>n</i> to <i>m</i> occurrences of the previous char or expression
<code>{n,}</code>	at least <i>n</i> occurrences of the previous char or expression
<code>{,m}</code>	up to <i>m</i> occurrences of the previous char or expression

Figure 2.9 Regular expression operators for counting.

Finally, certain special characters are referred to by special notation based on the backslash (`\`) (see Fig. 2.10). 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 `/\[`).

Regex	Match	First Patterns Matched
<code>*</code>	an asterisk “ <code>*</code> ”	“ <code>K*A*P*L*A*N</code> ”
<code>\.</code>	a period “ <code>.</code> ”	“ <code>Dr. Livingston, I presume</code> ”
<code>\?</code>	a question mark	“ <code>Why don’t they come and lend a hand?</code> ”
<code>\n</code>	a newline	
<code>\t</code>	a tab	

Figure 2.10 Some characters that need to be backslashed.

2.1.5 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 dollars:

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

Further fixes (like avoiding matching a dollar sign with no price after it) are left as an exercise for the reader.

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 use 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.

2.1.6 Substitution, Capture Groups, and ELIZA

substitution An important use of regular expressions is in **substitutions**. For example, the substitution operator `s/regex1/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 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 special commands `?:` after the open parenthesis, 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 some*.

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


```

User1:   Men are all alike.
ELIZA1: IN WHAT WAY
User2:   They're always bugging us about something or other.
ELIZA2: CAN YOU THINK OF A SPECIFIC EXAMPLE
User3:   Well, my boyfriend made me come here.
ELIZA3: YOUR BOYFRIEND MADE YOU COME HERE
User4:   He says I'm depressed much of the time.
ELIZA4: 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. After the input is uppercased, substitutions change all instances of *MY* to *YOUR*, and *I'M* to *YOU ARE*, and so on. That way when ELIZA repeats back part of the user utterance, it will seem to be referring correctly to the user. The next set of substitutions matches and replaces other patterns in the input. Here are some examples:

```

s/. * YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/. * YOU ARE (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 15.

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 yet advance the pointer we always keep to where we are in the text, so that we can then deal with the pattern if it occurs, but if it doesn't we can check for something else instead.

lookahead

zero-width

These **lookahead** assertions make use of the `(?= pattern)` 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 pointer. 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:

```
/^(?!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?