

The Bastards Book of Regular Expressions

Finding Patterns in Everyday Text

Dan Nguyen

The Bastards Book of Regular Expressions

Finding Patterns in Everyday Text

Dan Nguyen

This book is for sale at <http://leanpub.com/bastards-regexes>

This version was published on 2013-04-02

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



©2013 Dan Nguyen

Contents

Regular Expressions are for Everyone	1
FAQ	2
Release notes & changelog	5
 Getting Started	 6
Finding a proper text editor	7
Why a dedicated text editor?	7
Windows text editors	7
Mac Text Editors	10
Sublime Text	12
Online regex testing sites	13
 A better Find-and-Replace	 19
How to find and replace	19
The limitations of Find-and-Replace	20
There's more than find-and-replace	22
 Your first regex	 23
Hello, word boundaries	25
Word boundaries	25
Escape with backslash	28
 Regex Fundamentals	 31
Removing emptiness	32
The newline character	32
Viewing invisible characters	34

CONTENTS

Match one-or-more with the plus sign	40
The plus operator	41
Backslash-s	46
Match zero-or-more with the star sign	47
The star sign	47
Specific and limited repetition	49
Curly braces	49
Curly braces, maximum and no-limit matching	51
Cleaning messily-spaced data	54
Anchors: A way to trim emptiness	56
The caret as starting anchor	56
The dollar sign as the ending anchor	58
Escaping special characters	61
Matching any letter, any number	63
The numeric character class	63
Word characters	66
Bracketed character classes	66
Matching ranges of characters with brackets and hyphens	67
All the characters with dot	71
Negative character sets	75
Negative character sets	75
Capture, Reuse	79
Parentheses for precedence	79
Parentheses for captured groups	81
Correcting dates with capturing groups	83
Using parentheses without capturing	90

CONTENTS

Optionality and alternation	92
Alternation with the pipe character	92
Optionality with the question mark	94
Laziness and greediness	99
Greediness	99
Laziness	101
Lookarounds	105
Positive lookahead	105
Negative lookahead	106
Positive lookbehind	107
Negative lookbehind	108
The importance of zero-width (TODO)	109
Regexes in Real Life	110
Why learn Excel?	110
The limits of Excel (todo)	110
Delimitation	110
Mixed commas and other delimiters	116
Dealing with text charts (todo)	118
Completely unstructured text (todo)	118
Moving in and out and into Excel	119
From Data to HTML (TODO)	123
Simple HTML tricks	123
Tabular data to HTML tables	126
Mocking full web pages from data	126
Visualizations	126
The Exercises	127

CONTENTS

Data Cleaning with the Stars	128
Normalized alphabetical titles	128
Make your own delimiters	129
Finding needles in haystacks (TODO)	132
Shakespeare's longest word	132
Changing phone format (TODO)	135
Telephone game	135
Ordering names and dates (TODO)	144
Year, months, days	144
Names	144
Preparing for a spreadsheet	144
Dating, Associated Press Style (TODO)	145
Scenario	145
The AP Date format	145
Real-world considerations	149
The limits of regex	150
Sorting a police blotter	152
Sloppy copy-and-paste	153
Start loose and simple	154
Conclusion	156
Converting XML to tab-delimited data	157
The payments XML	157
The pattern	159
Add more delimitation	160
Cleaning up Microsoft Word HTML (TODO)	161

CONTENTS

Switching visualizations (TODO)	162
A visualization in Excel	162
From Excel to Google Static Chart	162
From Google Static Charts to Google Interactive Charts	162
Cleaning up OCR Text (TODO)	163
Scenario	163
Cheat Sheet	164
Moving forward	165
Additional references and resources	165

Regular Expressions are for Everyone



A pre-release warning

What you're currently reading is a very alpha release of the book. I still have plenty of work in terms of writing all the content, polishing, and fact-checking it.

You're free to download it as I work on it. Just don't expect perfection.

This is my first time using [Leanpub](http://leanpub.com)^a, so I'm still trying to get the hang of its particular dialect of Markdown. At the same time, I know people want to know the general direction of the book. So rather than wait until the book is even reasonably polished, I'm just hitting "Publish" as I go.

^a<http://leanpub.com>

The shorthand term for regular expressions, “*regexes*,” is about the closest to sexy that this mini-language gets.

Which is too bad, because if I could start my programming career over, I would begin it by learning regular expressions, rather than ignoring it because it was in the optional chapter of my computer science text book. It would've saved me a lot of mind-numbing typing throughout the years. I don't even want to think about all the cool data projects I didn't even attempt because they seemed unmanageable, yet would've been made easy with basic regex knowledge.

Maybe by devoting an entire mini-book to the subject, that alone might convince people, “hey, this subject could be useful.”

But you don't have to be a programmer to benefit from knowing about regular expressions. If you have a job that deals with text-files, spreadsheets, data, writing, or webpages – which, in my estimation, covers *most* jobs involving a desk and computer – then you'll find *some* use for regular expressions. And you don't need anything fancy, other than your choice of freely-available text editors.

At worst, you'll have a find-and-replace-like tool that will occasionally save you minutes or hours of monotonous typing and typo-fixing.

But my hope is that after reading this short manual, you'll not only have that handy tool, but you'll get a greater insight into the patterns that make data *data*, whether the end product is a spreadsheet or a webpage.

FAQ

Who is the intended audience?

I claim that “regular expressions are for anyone,” but in reality, only those who deal with a lot of text will find an everyday use for them.

But by “text,” I include datasets (including spreadsheets and databases) and HTML/CSS files. It goes without saying that programmers need to know regexes. But web developers/designers, data analysts, and researchers can also reap the benefits. For this reason, I’m devoting several of the higher-level chapters in this book for demonstrating those use-cases.

How technical is this book?

This book aims to reach people who’ve never installed a separate text editor (outside of Microsoft Word). In order to reduce the intimidation factor, I do not even come close to presenting an exhaustive reference of the regular expression syntax.

Instead, I focus mostly on the regexes I use on a daily basis. I don’t get into the details of how the regex engine works under the hood, but I try to explain the logic behind the different pieces of an expression, and how they combine to form a high-level solution.

How hard are regexes compared to learning programming? Or HTML?

Incredibly complex regexes can be formed by, more or less, dumbly combining basic building blocks. So the “hard part” is memorizing the conventions.

Memorization isn’t fun, but you can print out a cheat sheet (*note: will create one for this book’s appendix*) of the syntax. The important part is to be able to describe in plain English *what you want to do*: then it’s just a matter of glancing at that cheat sheet to find the symbols you need.

For that reason, this book puts a lot of emphasis on describing problems and solutions in plain, conversational English. The actual symbols are just a detail.

How soon will my knowledge of regular expressions go obsolete?

The theory behind regular expressions is as old as [modern computing](http://en.wikipedia.org/wiki/Regular_expression)¹. It represents a formal way to describe patterns and structures in text. In other words, it’s not a fad that will go away, not as long as we have language.

¹http://en.wikipedia.org/wiki/Regular_expression

You don't need to be a programmer to use them, but if you do get into programming, every modern language has an implementation of regexes, as they are incredibly useful for virtually any application you can imagine.

The main caveat is that each language – Javascript, Ruby, Perl, .NET – has small variations. This book, however, focuses on the general uses of regexes that are more or less universal across all the major languages. (I'll be honest: I can't even remember the differences among regex flavors, because it's rarely an issue in daily usage).

What special program will I need to use regexes?

You'll need a text editor that supports regular expressions. Nearly all text-editors that are aimed towards coders support regular expressions. In the first chapter, I list the free (and powerful) text editors for all the major operating systems.

Beyond understanding the syntax, actually *using* the regular expressions requires nothing more than doing a **Find-and-Replace** in the text editor, with the “use regular expressions” checkbox checked.

What are the actual uses of regular expression?

Because regexes are as easy as **Find-and-Replace**, the first chapters of this book will show how regexes can be used to replace *patterns* of text: for example, converting a list of dates in MM/DD/YYYY format to YYYY-MM-DD. Later on, we'll show how this pattern-matching power can be used to turn unstructured blocks of text into usable spreadsheet data, and how to turn spreadsheet data into webpages.

I have hopes that by the end of this book, regexes will become a sort of “gateway drug” for you to seek out even better, more powerful ways to explore the data and information in your life. The exercises in this book can teach you how to find needles in a haystack – a name, a range of dates, a range of currency amounts, amid a dense text. But once you've done that, why settle for searching one haystack – a document, in this case – when you could apply your regex knowledge to search thousands or millions of haystacks?

Regular expressions, for all their convoluted sea-of-symbols syntax, are just *patterns*. Learning them is a small but non-trivial step toward realizing how much of our knowledge and experience is captured in patterns. And how, knowing these patterns, we can improve the way we sort and filter the information in our lives.

This book is a spinoff of the Bastards Book of Ruby, which devoted an [awkwardly-long chapter to the subject](#)². You can get a preview of what this book will cover by checking out that [\(unfinished\) chapter](#)³.

²<http://ruby.bastardsbook.com/chapters/regexes/>

³<http://ruby.bastardsbook.com/chapters/regexes/>

If you have any questions, feel free to mail me at dan@danwin.com⁴

- Dan Nguyen @[dancow](https://twitter.com/dancow)⁵, danwin.com⁶

⁴<mailto:dan@danwin.com>

⁵<https://twitter.com/dancow>

⁶<http://danwin.com>

Release notes & changelog

Note: This book is in alpha stage. Entire sections and chapters are missing. Cruel exercises have yet to be devised. Read the [intro]{#intro} for more information.

Apr. 1, 2013 - Version 0.63 Tidied up a few of the early sections

Mar. 30, 2013 - Version 0.60 Finished cheat sheet

Mar. 29, 2013 - Version 0.57 Finished chapter on optional/alternation operators

Mar. 28, 2013 - Version 0.55 Finished lesson on XML to Tab-delimited data

Mar. 21, 2013 - Version 0.51 Finished the star sign chapter

Mar. 15, 2013 - Version 0.5 Finished most of the syntax chapters. Added separate chapter for star operator.

Feb. 24, 2013 - Version 0.31 Still cranking away at the syntax lessons. Gave optional/alternation its own chapter.

Feb. 10, 2013 - Version 0.31 Moved the plus-sign lesson to its own chapter. Finished the chapter on anchor symbols.

Feb. 6, 2013 - Version 0.3 Rearranged some of the early chapters. Character sets and negative character sets are two different chapters. I think I've figured out the formatting styles that I want to use.

Jan. 28, 2013 - Version 0.22 Added more padding and stub content, removed a little more gibberish.

Jan. 28, 2013 - Version 0.2 Added some more content but mostly have structured the book into introductory syntax and then chapters devoted to real-life scenarios. Still figuring out the layout styles I want to use.

Jan. 25, 2013 - Version 0.1 The first ten chapters, some with actual content. I'm still experimenting with the whole layout and publishing process. But for now, the order of subjects seems reasonable.

Jan. 22, 2013 - Version 0x Just putting the introduction out there. Nothing to see here.

Getting Started

Finding a proper text editor

One of the nice things about regular expressions is that you don't need any special, dedicated programs to use them. Regular expressions are about matching and manipulating text patterns. And so we only need a text editor to use them.

Unfortunately, your standard word processor such as Microsoft Word won't cut it. But the text editors we can use are even simpler than Word and, more importantly, *free*.

Why a dedicated text editor?

Text editors are the best way to handle text as *raw text*. Word processors get in the way with this. Microsoft Word and even the standard TextEdit that comes with Mac OS X don't deal with just text, they deal with how to make printable documents with large headlines, bulleted lists, and italicized footnotes.

But we're not writing a résumé or a book report. All we need to do is **find** text and **replace** text.

The special text editors I list in this chapter do that *beautifully*.

While your typical word processor can do a **Find-and-Replace**, it can't do it with regular expressions. That's the key difference here.

Windows text editors

A caveat: I've used Windows PCs for most of my life, but in my recent years as a developer, I've switched to the Mac OS X platform to do my work. All the examples in this book can be done on either platform with the right text editor, even though the *look* may be different. Even so, I've tried my best in the book to provide screenshot examples from my 5-year-old Windows netbook.

Notepad++⁷ seems to be the most free and popular text editor for Windows. It has all the features we need for regular expressions, plus many others that you might use in your text-editing excursions.

⁷<http://notepad-plus-plus.org/>

```

1 <NotepadPlus>
2 <InternalCommands />
3 <Macros>
4 <Macro name="Trim Trailing and save" Ctrl="no" Alt="yes" Shift="yes" Key="83">
5 <Action type="2" message="0" wParam="42024" lParam="0" sParam="" />
6 <Action type="2" message="0" wParam="41006" lParam="0" sParam="" />
7 </Macro>
8 </Macros>
9 <UserDefinedCommands>
10 <Command name="Launch in Firefox" Ctrl="yes" Alt="yes" Shift="yes" Key="88">firefox &quot;$(FULL_CURRENT
11 <Command name="Launch in IE" Ctrl="yes" Alt="yes" Shift="yes" Key="73">iexplore &quot;$(FULL_CURRENT_PAT
12 <Command name="Launch in Chrome" Ctrl="yes" Alt="yes" Shift="yes" Key="82">chrome &quot;$(FULL_CURRENT_P
13 <Command name="Launch in Safari" Ctrl="yes" Alt="yes" Shift="yes" Key="70">safari &quot;$(FULL_CURRENT_P
14 <Command name="Get php help" Ctrl="no" Alt="yes" Shift="no" Key="112">http://www.php.net/%20$(CURRENT_WO
15 <Command name="Google Search" Ctrl="no" Alt="yes" Shift="no" Key="113">http://www.google.com/search?q=$(
16 <Command name="Wikipedia Search" Ctrl="no" Alt="yes" Shift="no" Key="114">http://en.wikipedia.org/wiki/S
17 <Command name="Open file" Ctrl="no" Alt="yes" Shift="no" Key="116">$(NPP_DIRECTORY)\notepad++.exe $(CURR
18 <Command name="Open in another instance" Ctrl="no" Alt="yes" Shift="no" Key="117">$(NPP_DIRECTORY)\notep
19 <Command name="Open containing folder" Ctrl="no" Alt="no" Shift="no" Key="0">explorer $(CURRENT_DIRECTOR
20 <Command name="Open current dir cmd" Ctrl="no" Alt="no" Shift="no" Key="0">cmd /K cd /d $(CURRENT_DIRECT
21 <Command name="Send via Outlook" Ctrl="yes" Alt="yes" Shift="yes" Key="79">outlook /a &quot;$(FULL_CURRE
22 </UserDefinedCommands>
23 <PluginCommands />
24 <ScintillaKeys />
25 </NotepadPlus>
26

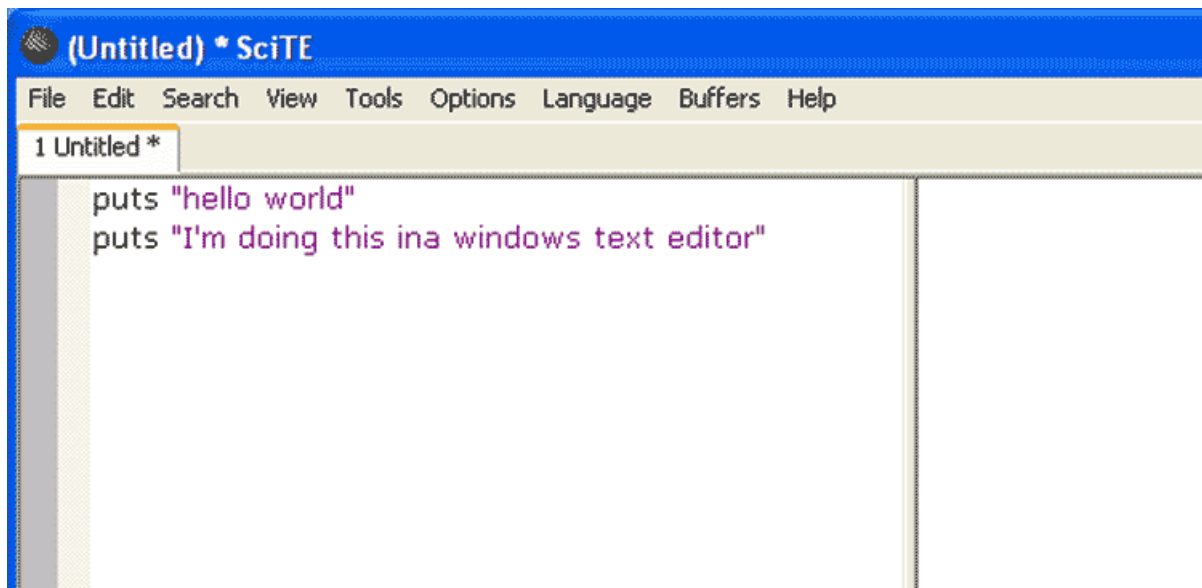
```

eXtensible Markup Language file length: 2111 lines: 26 Ln: 9 Col: 22 Sel: 0 Dos\Windows ANSI INS

Notepad++

SciTE⁸ is another free text-editor that has regex functionality. However, it uses a variation that may be different enough from the examples in this book as to cause frustration.

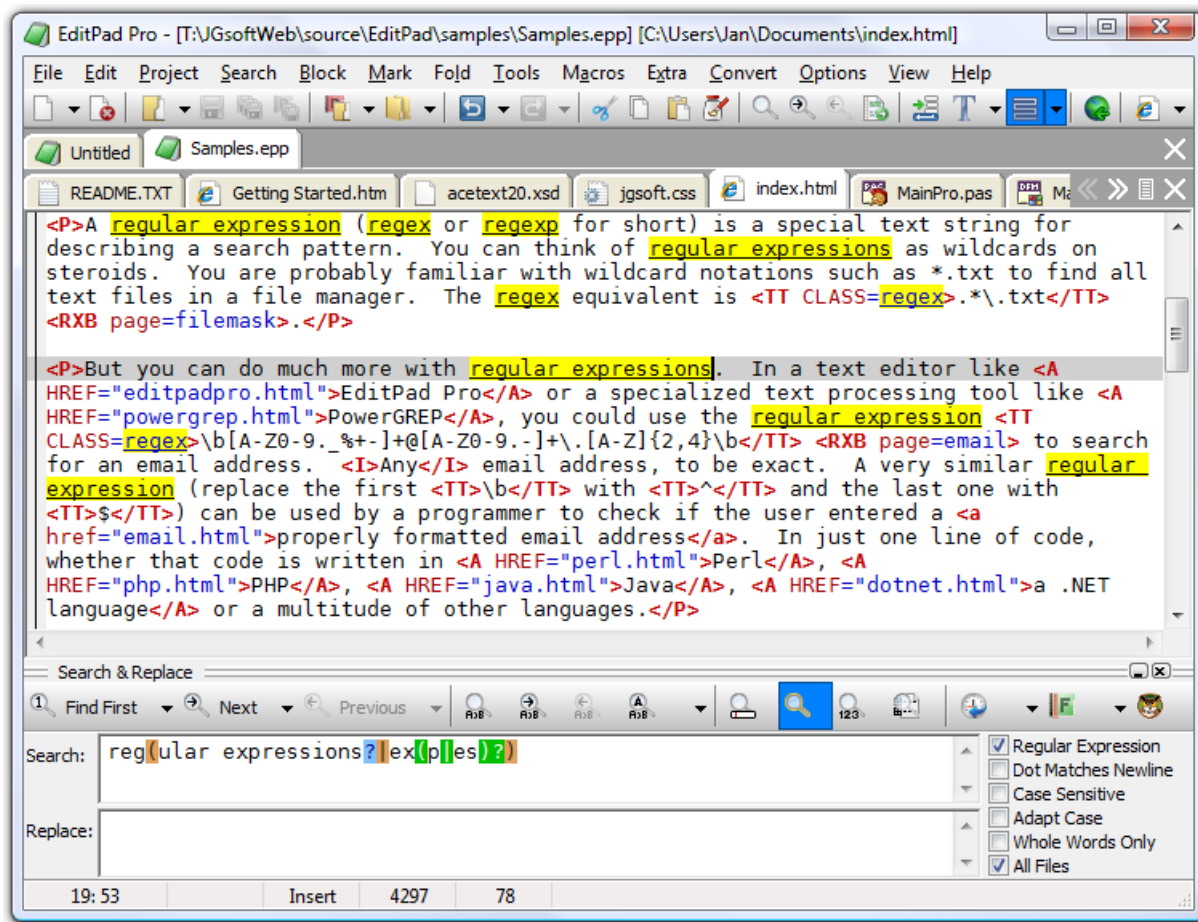
⁸<http://www.scintilla.org/SciTE.html>



SciTE in action

EditPadPro⁹ is a commercial product but considered one of the best text-editors for Windows and its regular expression support is extremely strong. It comes with a free trial period.

⁹<http://www.editpadpro.com/>



EditPadPro

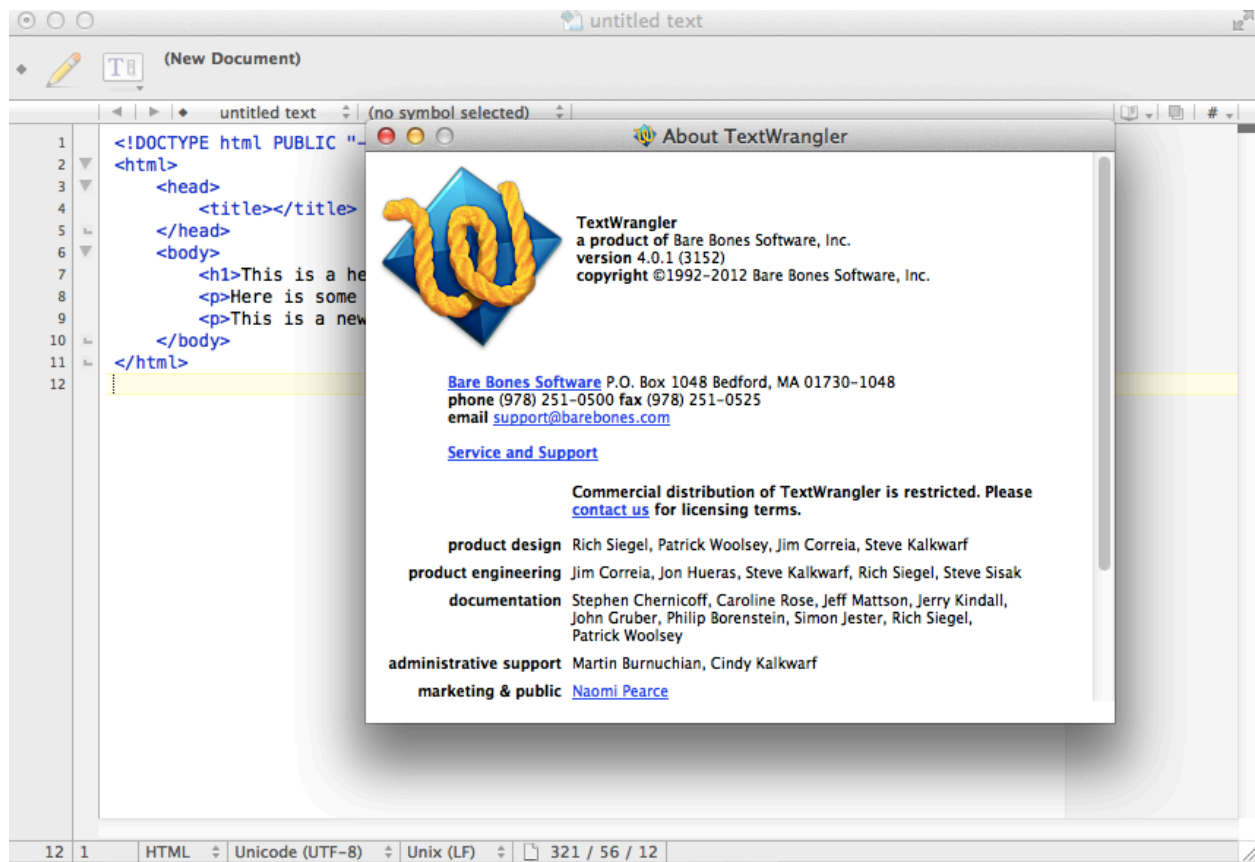
Mac Text Editors

Even in this era of Apple dominance, it's a fact of life that Apple PCs have generally less choice of software compared to mass-market Windows PCs. Luckily for us, Apple has a few solid offerings.

TextWrangler¹⁰ is a full-featured, well-designed text-editor, a sampling of Bare Bones Software's commercial product, **BBEdit**¹¹.

¹⁰<http://www.barebones.com/products/textwrangler/>

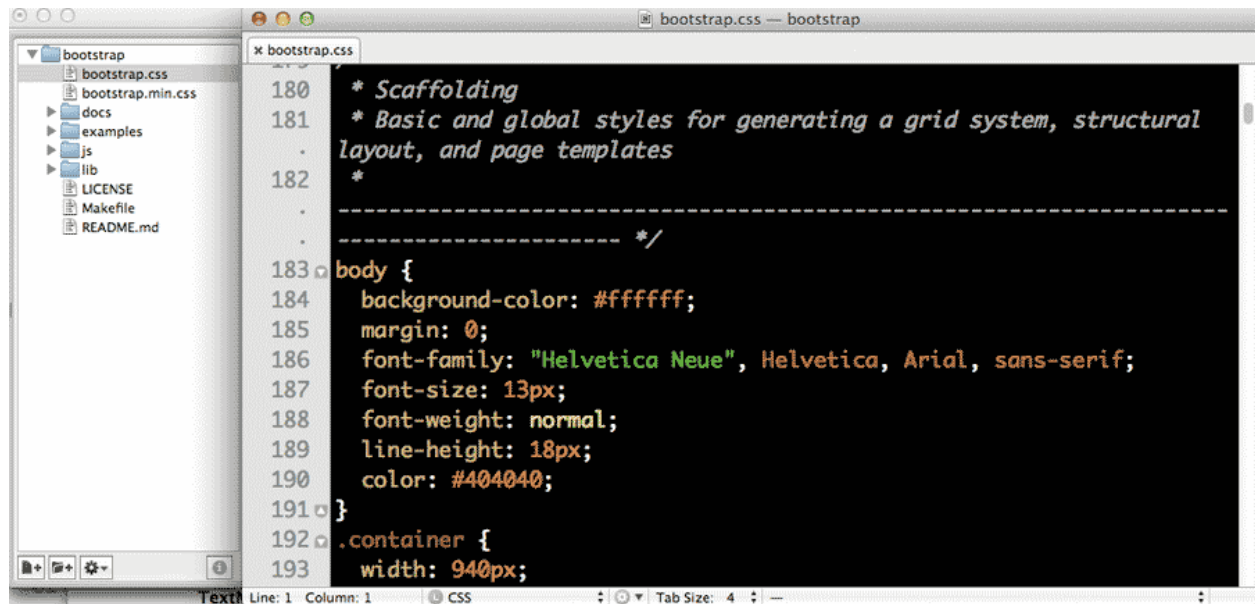
¹¹http://www.barebones.com/products/bbedit/index.html?utm_source=thedeck&utm_medium=banner&utm_campaign=bbedit



TextWrangler splash screen

TextMate¹² has long been a favorite of developers. It comes in a commercial and open-sourced version.

¹²<http://macromates.com/>



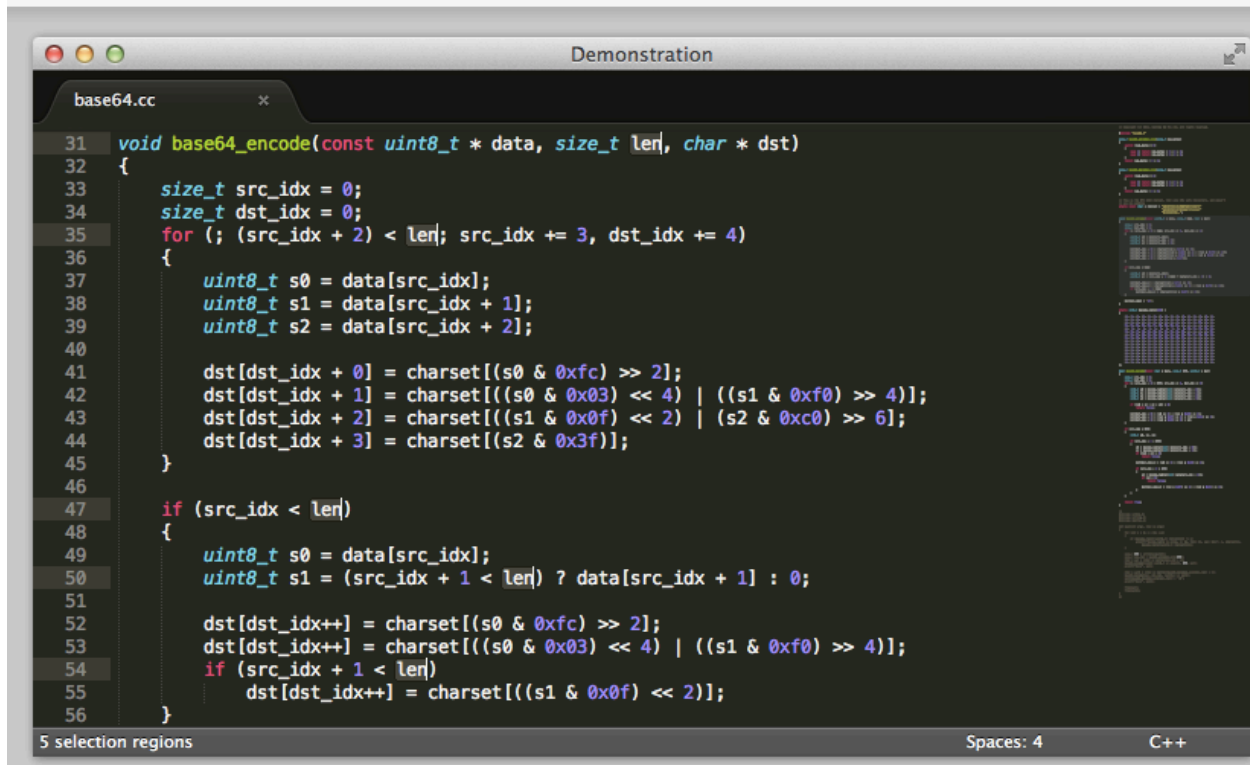
TextMate is often used as a project code editor.

Sublime Text

I'm giving Sublime Text its own section, not just because it's the editor I use for most programming, but because it is available on Windows, Macs and Linux. It also comes with a generous trial period.

Sublime Text

Sublime Text is a sophisticated text editor for code, markup and prose.
You'll love the slick user interface, extraordinary features and amazing performance.



Sublime Text 2; version 3 is on its way

Its price may seem pretty stiff, but I recommend downloading it and using it on a trial-basis. I don't know if I can advocate paying \$70 if all you intend to do is regular expressions and text formatting.

However, if you are thinking about getting into programming, then it is most definitely worth it. It's a program I spend at least two to three hours a day in, even on off-days, and makes things so smooth that it is undoubtedly worth the up-front money.

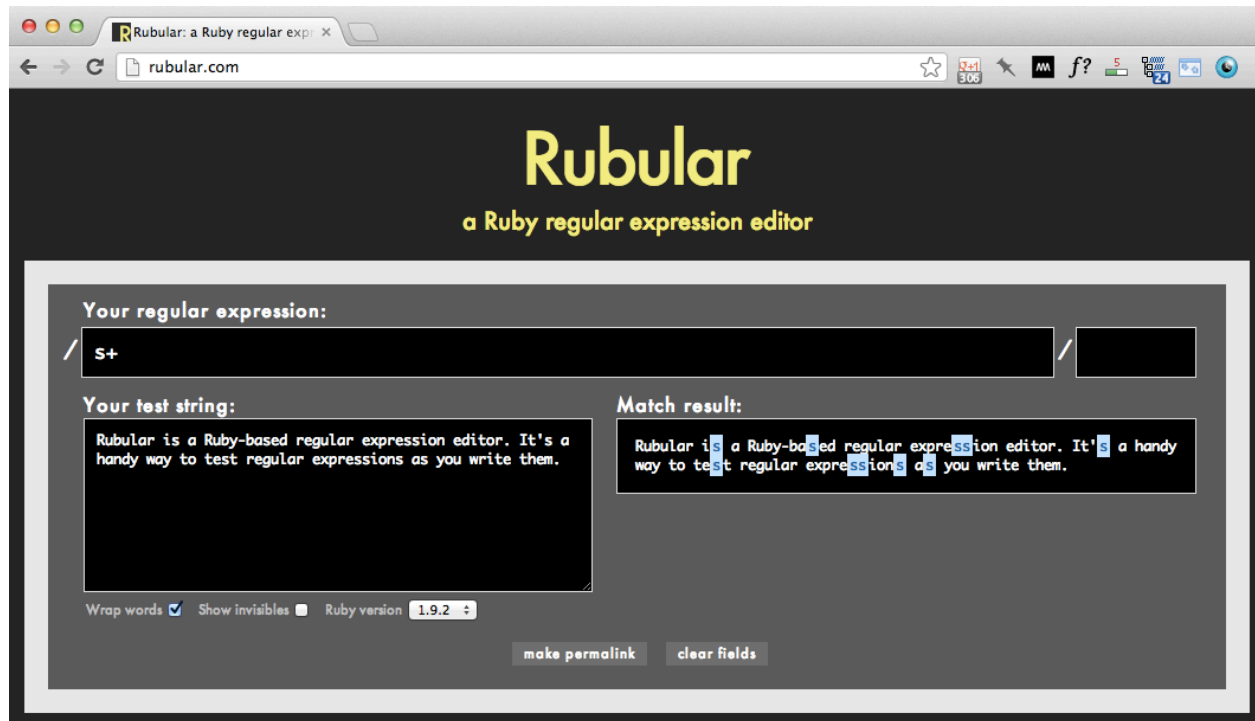
Online regex testing sites

If you don't want to install a new program yet, then you can follow along with online sites.

[Rubular](http://rubular.com/)¹³ is a great place to test regexes. It implements the Ruby programming language's variant of regexes, which, for our purposes, has everything we need and more. And despite its name, it

¹³<http://rubular.com/>

doesn't involve writing any Ruby code.



Rubular.com - It's not just for Ruby programmers

Trying out Rubular

Before you go through all the work of looking for a text-editor, downloading, and then installing it, we can play with regular expressions in our web browser with Rubular.

Point your browser to: <http://rubular.com>¹⁴

Copy and paste the following text into the Rubular text box:

The cat goes catatonic when you put it in the catapult

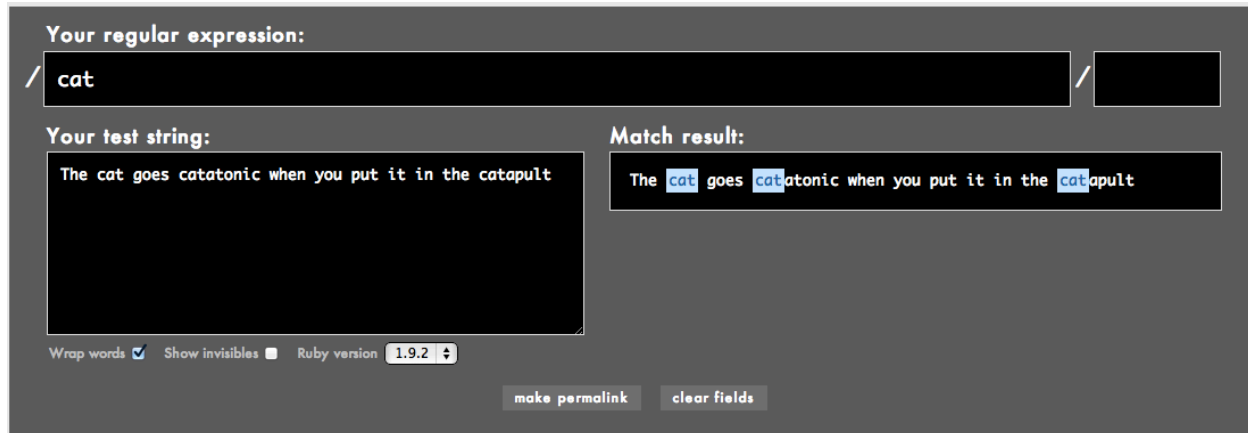
Now, in the top text input, between the two forward slashes (/ is the forward slash, not to be confused with the \ backslash), enter in the following regex:

cat

That's right, just the word "cat", literally. This is a regular expression, though not a very fancy one.

¹⁴<http://rubular.com>

In Rubular, all instances where this simple regular expression matches the text – i.e., “cat” – are highlighted:

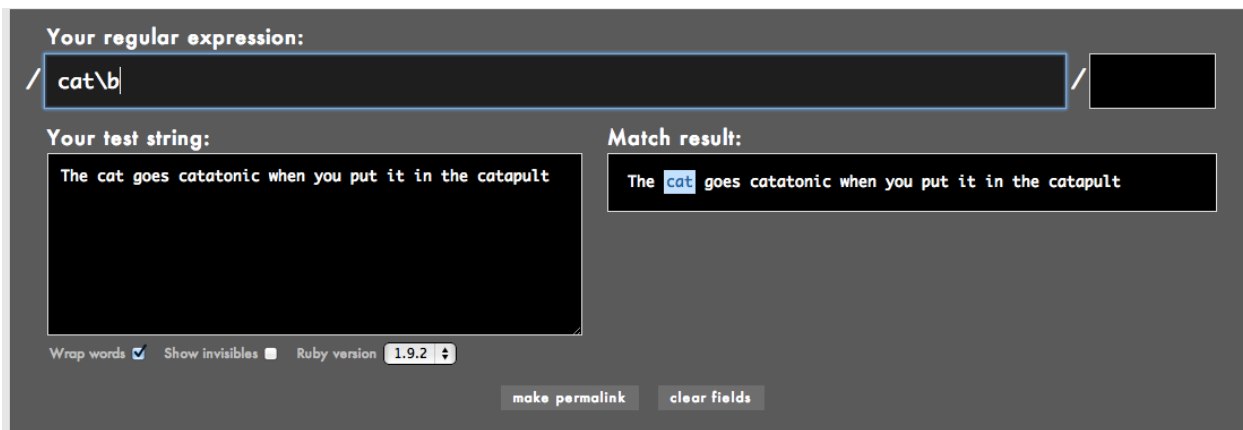


The screenshot shows the Rubular website interface. At the top, under "Your regular expression:", the text "cat" is entered in a text box. Below this, under "Your test string:", the text "The cat goes catatonic when you put it in the catapult" is entered. To the right, under "Match result:", the same text is shown with the word "cat" highlighted in blue. At the bottom, there are checkboxes for "Wrap words" (checked) and "Show invisibles" (unchecked), a "Ruby version" dropdown set to "1.9.2", and two buttons: "make permalink" and "clear fields".

'cat' on rubular.com

OK, let's add some *actual* regex syntax. Try this:

cat\b



The screenshot shows the Rubular website interface. At the top, under "Your regular expression:", the text "cat\b" is entered in a text box. Below this, under "Your test string:", the text "The cat goes catatonic when you put it in the catapult" is entered. To the right, under "Match result:", the same text is shown with the word "cat" highlighted in blue. At the bottom, there are checkboxes for "Wrap words" (checked) and "Show invisibles" (unchecked), a "Ruby version" dropdown set to "1.9.2", and two buttons: "make permalink" and "clear fields".

'cat\b' on rubular.com

Rubular will immediately highlight a new selection. In this case, it actually unselects the “cat” that is part of “catapult” and “catatonic”. As we’ll soon find out, that \b we added narrowed the selection to just the word “cat”, just in case we want to replace it with “mouse” but not end up with “mouseapult” and “mouseatonic”

Here’s a slightly more useful regular expression example. Copy and paste the following text into Rubular’s field titled **Your test string**:

Yesterday, at 12 AM, he withdrew \$600.00 from an ATM. He then spent \$200.12 on groceries. At 3:00 P.M., he logged onto a poker site and played 400 hands of poker. He won \$60.41 at first, but ultimately lost a net total of \$38.82.

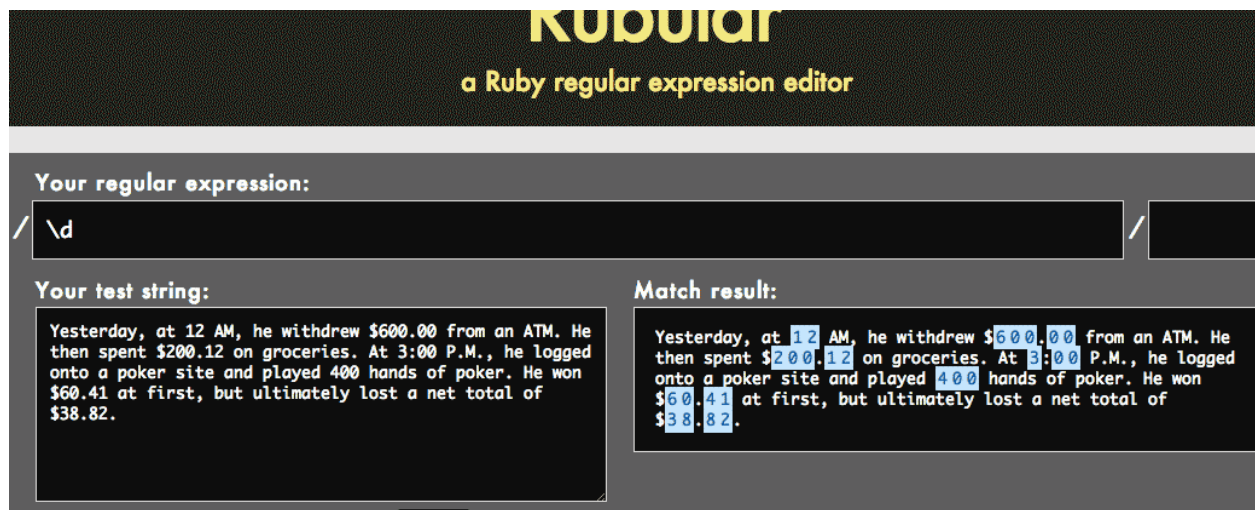


Pasting text into Rubular

In the field titled, **Your regular expression**;, type in backslash-d:

`\d`

In the field titled, **Match result**, you'll see that every numerical digit is highlighted:



Selecting digits with a regex in Rubular

Now try entering this into the regular expression field:

```
\. \d{2}
```

This matches a period followed by two numerical digits, no matter what those digits are:

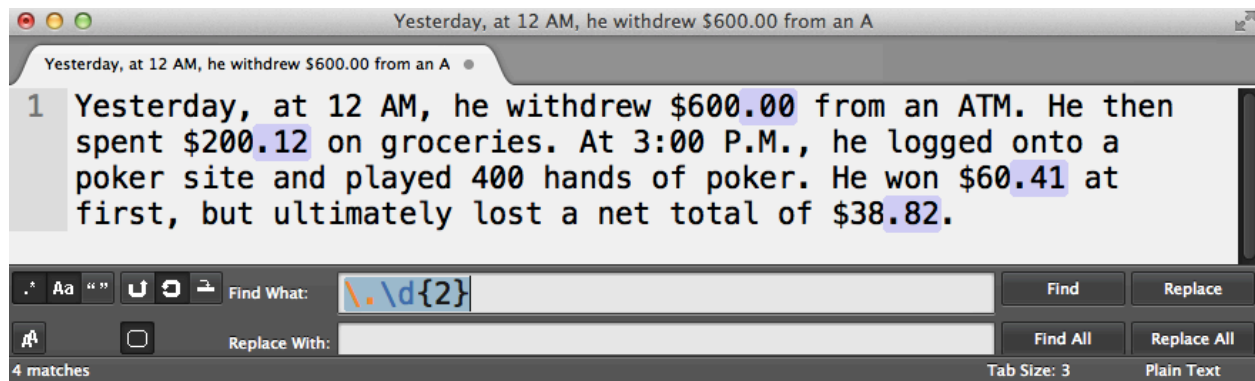
Yesterday, at 12 AM, he withdrew \$600.00 from an ATM. He then spent \$200.12 on groceries. At 3:00 P.M., he logged onto a poker site and played 400 hands of poker. He won \$60.41 at first, but ultimately lost a net total of \$38.82.

Why might we want to do this? One use might be to round all money amounts to the dollar. Notice how the highlight doesn't occur around any other instances of two-numerical digits, including "12 AM" and "400 hands of poker"

Sublime Text example

Using a regex in a text-editor is just as simple as the Rubular example. However, most of the text-editors I've mentioned here do not interactively highlight the matches, which isn't too big of a deal.

Sublime Text is an exception though, and just to show you that regexes work the same in a text editor as they do on Rubular, here's a screenshot:



A double-digit highlight in Sublime Text

For some of you, the potential of regular expressions to vastly improve how you work with text might be obvious. But if not, don't worry, this was just a basic how-to-get-started guide. Read on for many, many more practical examples and applications.

A better Find-and-Replace

As I said in the intro, regular expressions are an invaluable component of a programmer's toolbox (though there are many programmers who don't realize that).

However, the killer feature of regular expressions for me is that they're as easy-to-use as your typical word processor's **Find-and-Replace**. After we've installed a proper text-editor, we already know how to use regexes.

How to find and replace

Most readers already know how to use Find-and-Replace but I want to make sure we're all on the same page. So here's a quick review.

Using your computer's basic text editor: Notepad for Windows and TextEdit for Macs.

In modern Microsoft Word, you can bring up the **Find-and-Replace** by going to the **Edit** menu and looking for the **Find** menu. There should be an option called **Replace...**

Clicking that brings up either a side-panel or a separate dialog box with two fields.

- The top field, perhaps labeled with the word **Find**, is where you type in the term that you want to *find*.
- The bottom field, usually labeled with the word **Replace**, is where you type in the term that you want to **replace** the selection with.

So try this out:

Find t

Replace r

If you hit the **Replace All** button, all t's become r's. If you hit just **Replace**, only the first found t becomes an r.

You also have the option to specify case insensitivity, to affect the capital T and the lowercase t.

Replace All vs Replace

For the purposes of this book, we'll usually talk about replacing **all** the instances of a pattern. So assume that when I refer to **Find-and-Replace** I actually mean **Find-and-Replace-All*, unless otherwise stated.

The limitations of Find-and-Replace

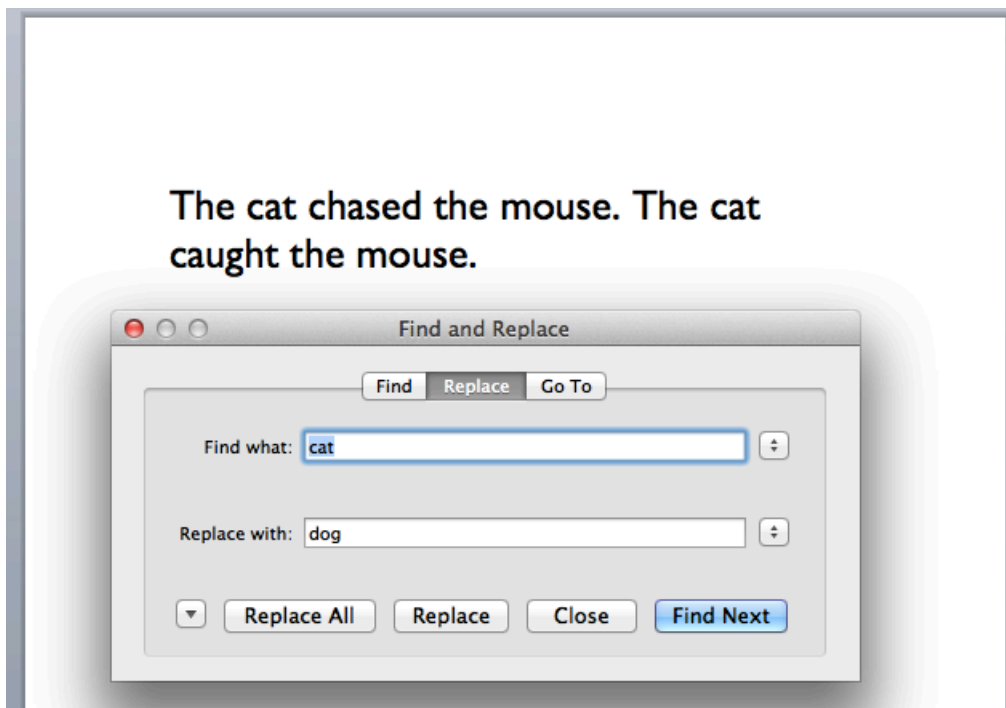
OK, now that we understand what basic **Find-and-Replace** is, let's see where it comes up short. Take a look at the following sentence:

The **cat** chased the mouse. The **cat** caught the mouse.

Suppose we want to change this protagonist to something else, such as a dog:

The **dog** chased the mouse. The **dog** caught the mouse.

Rather than replace each individual cat, we can use **Find-and-Replace** and replace both cat instances in a single action:



Find and Replace with Microsoft Word

The cat is a catch

Now what happens if we want to replace the cat in a more complicated sentence?

The **cat** chased the mouse deep down into the catacombs. The **cat** would reach a state of catharsis if it were to catch that mouse.

Using the standard **Find-and-Replace**, we end up with this:

The **dog** chased the mouse deep down into the **dogacombs**. The **dog** would reach a state of **dogharsis** if it were to **dogch** that mouse.

The find-and-replace didn't go so well there, because matching `cat` ends up matching not just `cat`, but *every* word with `cat` in it, including “`catharsis`” and “`catacombs`”.

Mixed-up date formats

Here's another problem that find-and-replace can't fix: You're doing tedious data-entry for your company sales department. It's done old school as in, just a simple list of dates and amounts:

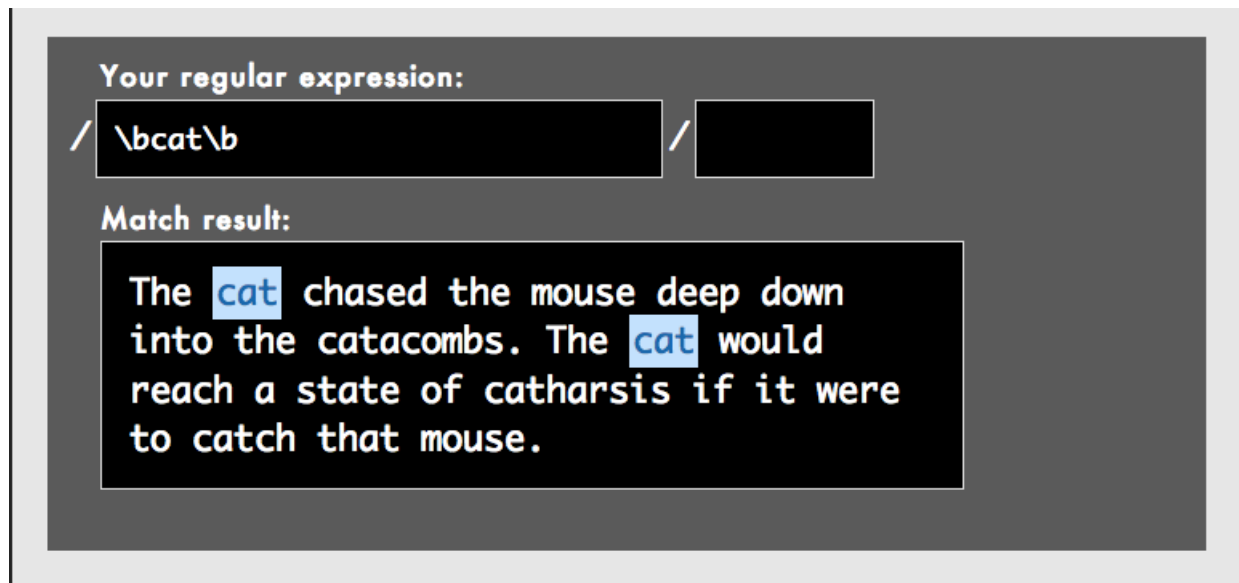
```
12/24/2012, $50.00
12/25/2012, $50.00
12/28/2012, $102.00
1/2/2012, $90.00
1/3/2012, $250.00
1/10/2012, $150.00
1/14/2012, $10.00
2/1/2013, $42.00
```

The problem is that when January 2013 rolled around, you were still in the habit of using 2012. A find-and-replace to switch 2012 to 2013 won't work here because you need to change only the numbers that happened in January.

So how does regular expressions fix this problem? Here's the big picture concept: Regexes let us work with *patterns*. We aren't limited to matching just `cat`; we can instead match these patterns:

- `cat` when it's at the beginning of a word
- `cat` when it is at the end of a word
- `cat` when it is a standalone word
- `cat`, when it appears more than 3 times in a single sentence.

In the cat-to-dog example, the pattern we want to find-and-replace is: *the word “cat”, when it stands alone and not as part of another word (like “catharsis”)*



Using Rubular.com to find just the standalone 'cat'

In the scenario of the mixed-up dates, the pattern we want to match is: *The number "2012", if it is preceded by a "1", one-or-two other numbers, and another "/"*

Without regexes, you have to fix these problems by hand. This might not be a problem if it's just a couple of words. Even manually performing a dozen **Find-and-Replaces** to fix every variation of a word is tedious, at worst.

But if you have a hundreds or thousands of fixes? Then tedious becomes painful or even impossible.

There's more than find-and-replace

Big deal, right? Being able to find-and-replace better is hardly something worth reading a whole book for. And if all I had to show you was text-replacement tricks, I'd agree.

But if you're a real data-gatherer or information-seeker, the real power of regular expressions is to find if and where *patterns exist at all*. Maybe you have a list of a million invoices that can't be put in Excel and you need to quickly filter for the six-figure amounts. Or you have a 500-page PDF and you need a fast hack to highlight proper nouns, such as names of people and places.

In these cases, you don't know precisely what you're looking for until you find it. But you *do* know the *patterns*. With regular expressions, you'll have a way to describe and actually *use* those patterns.

Your first regex

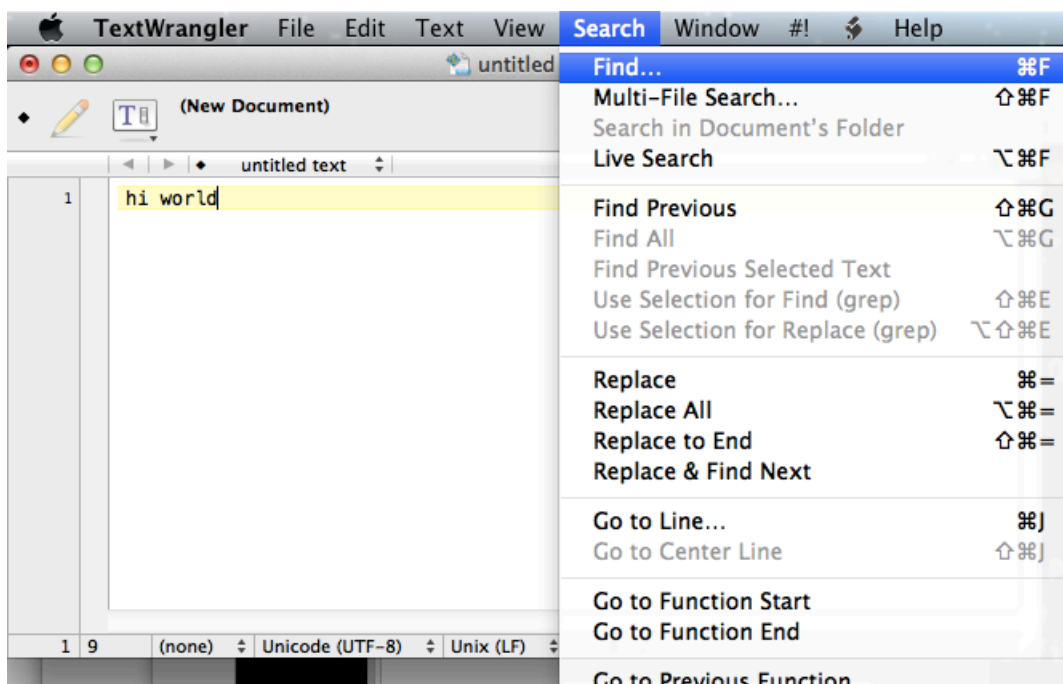
Let's try writing our first regular expression.

1. Open up your text editor and type in the following:

```
hi world
```

2. Pop open the **Find-and-Replace** dialog menu. You can do this either with the keyboard shortcut, usually **Ctrl-F** or **Cmd-F** (this is the most efficient way to do this). Or you can go up to the menubar.

Here's how to do it in Textwrangler (Mac):



search submenu

And in Notepad++ (Windows):

(Todo): Image

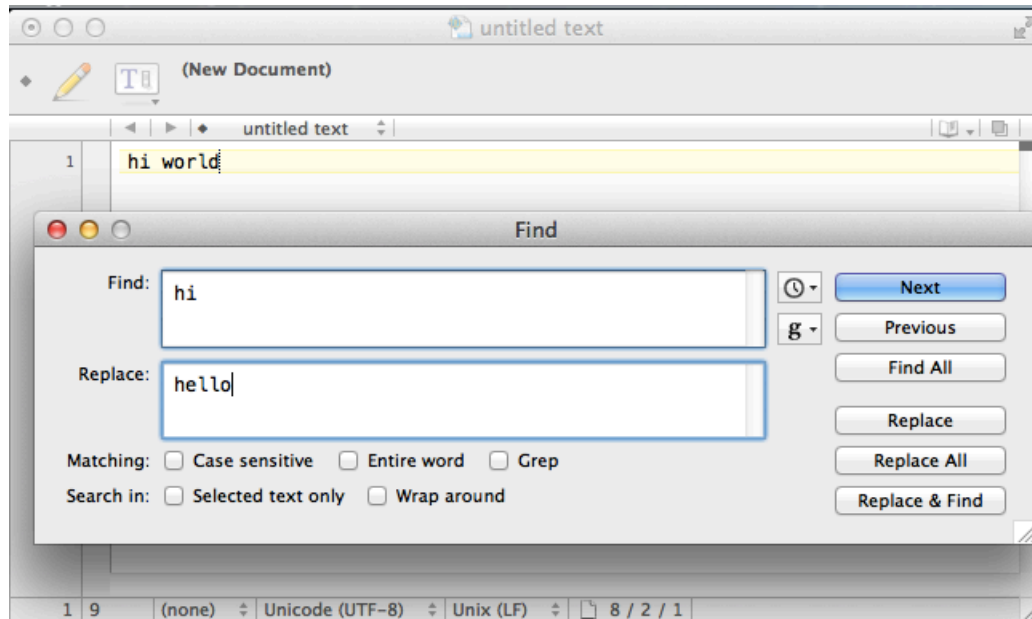
3. Our goal is to replace `hi` with `hello`. So in the **Find** field, type in:

```
hi
```

4. In the **Replace** field, type in

hello

The Find-and-Replace dialog should look like this:



find-replace-hello

1. Hit the **Replace** button. The text should now look like this:

```
hello world
```

By now, you're probably thinking, *what gives, I thought we were doing regular expressions, not find-and-replace!* Well, this technically *is* a regular expression, albeit a very simple and mostly useless one. Remember that regular expressions are *patterns*. In this case, we were simply looking for the pattern of `hi`.

Regular expressions are not necessarily all code. They usually contain *literal* values. That is, we are looking to **Find-and-Replace** the word `hi`, literally.

Hello, word boundaries

Now let's try a regex that uses actual regex syntax. In your text editor, type out this sentence:

```
hi world, it's hip to have big thighs
```

Once again, we want to replace `hi` with `hello`. But if you try the standard **Find-and-Replace**, you'll end up with:

```
hello world, it's hellop to have big thelloghs
```

So replacing the *literal* pattern of `hi` won't work. Because we don't want to just replace `hi` – we want to replace `hi` when it *stands alone*, not when it's part of another word, such as “**th**ighs”

What we need is a **word boundary**. This will be our first *real* regular expression syntax.

Word boundaries

A **word-boundary** refers to both the beginning or the end of a word.

But what is a *word*, exactly? In terms of regular expressions, any sequence of one-or-more alphanumeric characters – including letters from A to Z, uppercase and lowercase, and any numerical digit – is a *word*.

So a **word-boundary** could be a space, a hyphen, a period or exclamation mark, or the beginning or end of a line (i.e. the **Return** key).

So `dog`, `a`, `37signals`, and `under_score` are all **words**. The phrase `upside-down` actually consists of *two* words, as a **hyphen** is not a **word-character**. In this case, it would count as a **word boundary** – it ends the word `upside` and precedes the word `down`

The regex syntax to *match* a **word boundary** is `\b`

In English, we would pronounce this syntax as, “backslash-lowercase-b”

Important note: The **case** of the character matters here. `\B` and `\b` are not the same thing.

Before thinking too hard about this (e.g., *what the heck does the backslash do?*), let’s just try it out. Revert the sentence (i.e. **Undo**) to its pre-hello version.

In the **Find** field, type in:

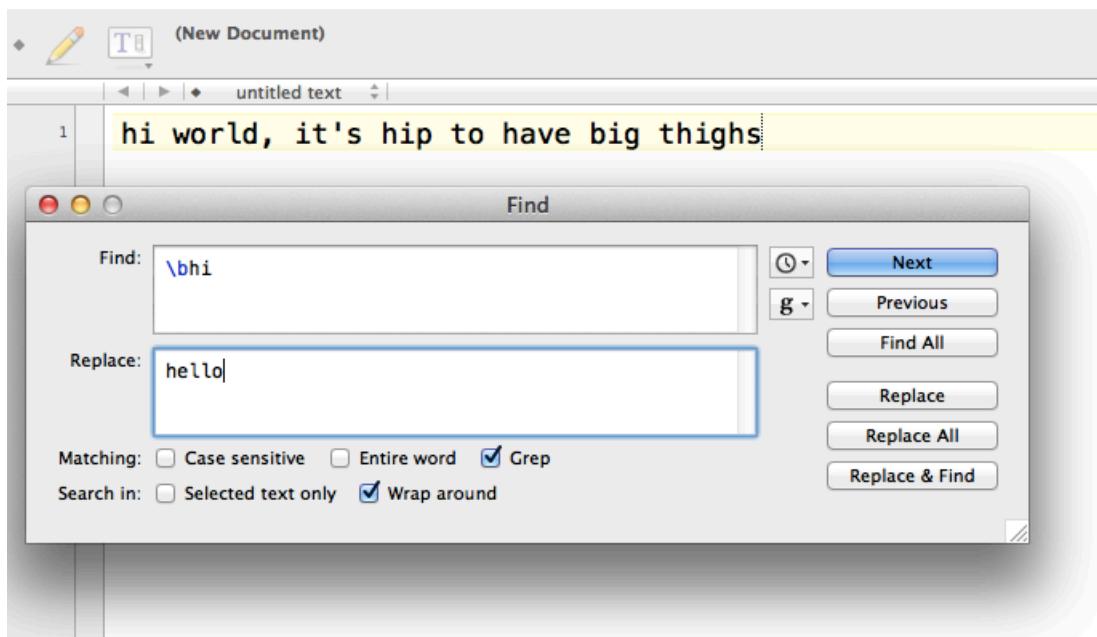
```
\bhi
```

In the **Replace** field, type in (as we did before):

```
hello
```

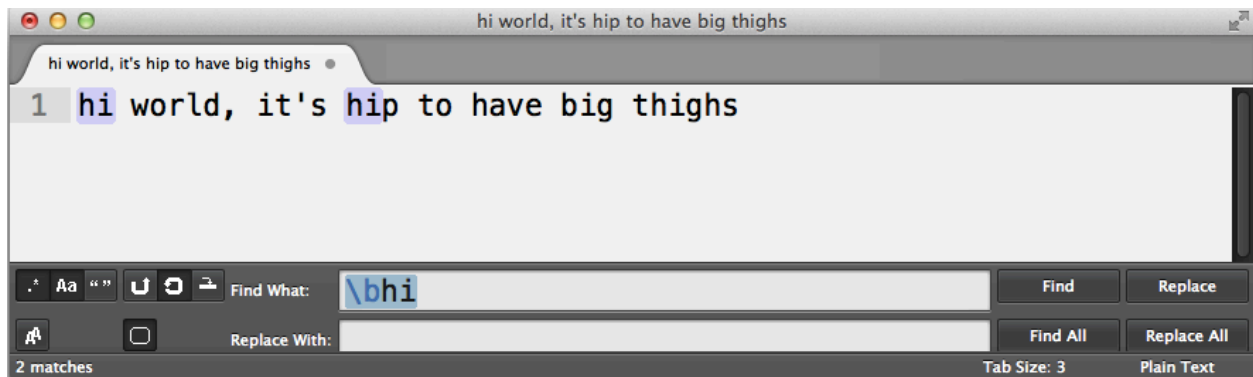
Before we hit **Replace All**, we need to tell our text-editor to enable regular expressions (or else it will look for a literal `\bhi`). In TextWrangler, this checkbox is somewhat ambiguously labeled as **Grep**. Other text editors will label the checkbox as **Regular expression**.

Here’s what it looks like in TextWrangler:



TextWrangler with regular expressions (Grep) enabled

Here's what it looks like in Sublime Text (which annoyingly uses indecipherable icons as options – mouseover them to see what they stand for):



Sublime Text with regular expressions enabled

Note: In Sublime Text, the matches are highlighted as you type in the pattern. Also, with regular expressions enabled, Sublime Text helpfully color codes the regex syntax. Notice how the `\b` is a different color from `hi`

Here's the Notepad++ version:

Note: It may be necessary to check the **Wrap around** option, which directs the text-editor to look for all matches, from wherever your cursor currently is, to the end of the document, and then back to the top.

However, you should not have to check anything else (besides whatever option enables regular expressions), including **Case sensitivity** (you want to be case *sensitive*), **Entire word**, **Selected text only**, or any variation thereof.

After we hit **Replace All**, we'll have this sentence in our text-editor:

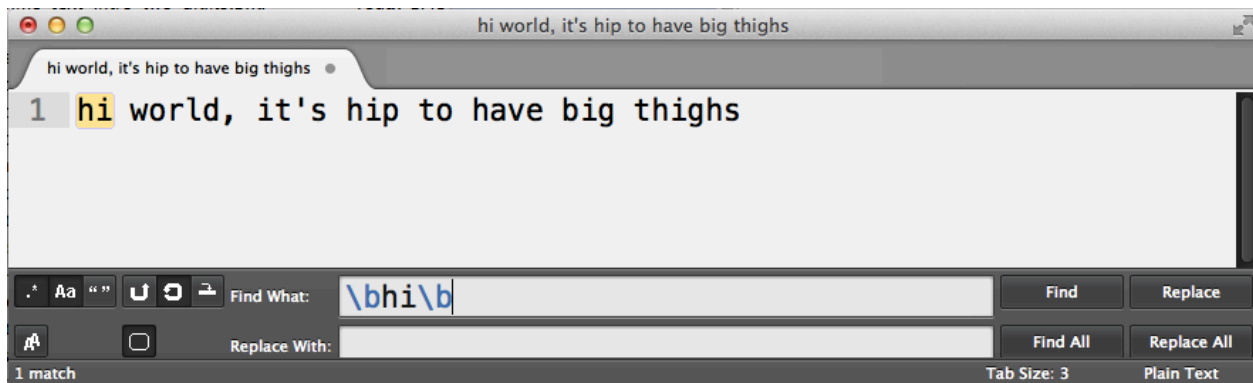
```
hello world, it's hellop to have big thighs
```

Well, that's a little better than before. When we were trying to match just `hi`, we ended up changing `thighs` to `thelloghs`.

However, with the regular expression `\bhi`, we *constrained* the pattern to match `hi` only when it was the first part of a word – which is not the case when it comes to “**thighs**”

However, that didn't quite help us with `hip`, which was changed to `hellop`

So how do we exclude the word “hip” from being caught by the pattern? With another word-boundary. Try figuring out the solution yourself before glancing at the answer below:



Double-bounded "hi" in Sublime Text

If you look at the pattern, `\bhi\b`, you might wonder, *why isn't that "h" affected by the backslash?* After all, it comes right after the `b` and seems to look like one blob of a pattern.

But as I mentioned before, the backslash affects *only* the character that comes *immediately* after it.

To reiterate this, look at how Sublime Text interprets the regular expression. The instances of `\b` are in light blue – they represent special characters in regex syntax. The letters `h` and `i`, rendered in black, are simply the *literal* alphabetical characters:

I've reprinted the regex pattern, but highlighted the parts that are actual regex syntax. The non-highlighted part – `hi` – is simply the letters `h` and `i`, literally.



Highlighted syntax in Sublime Text

Escape with backslash

We're going to see a lot of this **backslash** character (not to be confused with the *forward-slash* character, `/`). Its purpose in regex syntax is to **escape** characters.

At the beginning of this chapter, I described the pattern `hi` as the pattern for the *literal* word of `hi`. Similarly, a regex pattern that consists of `b` is going to match a *literal* `b`.

But when we have a **backslash** character *before* that `b`? Then it's not a *literal* `b` anymore, but it's a *special* kind of `b`. The backslash can be thought of *escaping* the character that *follows* it from its usual meaning.

In this case, the letter `b`, when preceded with a `\`, **will no longer match a literal `b`**.
 Instead, as we saw earlier, “backslash-`b`” is the regex way of saying, **word-boundary**.

Before moving on, let’s really ground the concept of the **backslash** in our heads. Re-enter the original sentence:

```
hi world, it's hip to have big thighs
```

And do a **Find-and-Replace** for the *literal* letter `b` and replace it with a symbol of your choosing, such as the underscore sign, `_`

Before you hit “Go”, try to predict the result. You should end up with something like this:

```
hi world, it's hip to have _ig thighs
```

What happened here? Well, because we only specified the letter `b` for our pattern, only the literal `b` in the sentence was affected. Now revert back to the original sentence and change the **Find** field to a `b` preceded by a backslash, i.e. `\b`

It’ll be harder to predict what this does. But you’ll end up with this:

```
_hi_ _world_, _it_'_s_ _hip_ _to_ _have_ _big_ _thighs_
```

What happened here? Our pattern looked for word-boundaries, which occur at the beginning and end of every word. Thus, `hi`, turns into `_hi_`.

The transformation of `it's` maybe a little more confusing. It ends up as:

```
_it_'_s_
```

This makes sense if we defined word-boundary as being the boundary of *consecutive letters*. The apostrophe, `'`, of course, is not a letter, and so it serves as much of a word-boundary as a space.

Exercise Let’s go back to the problem described in the previous chapter with the dog-and-cat sentence:

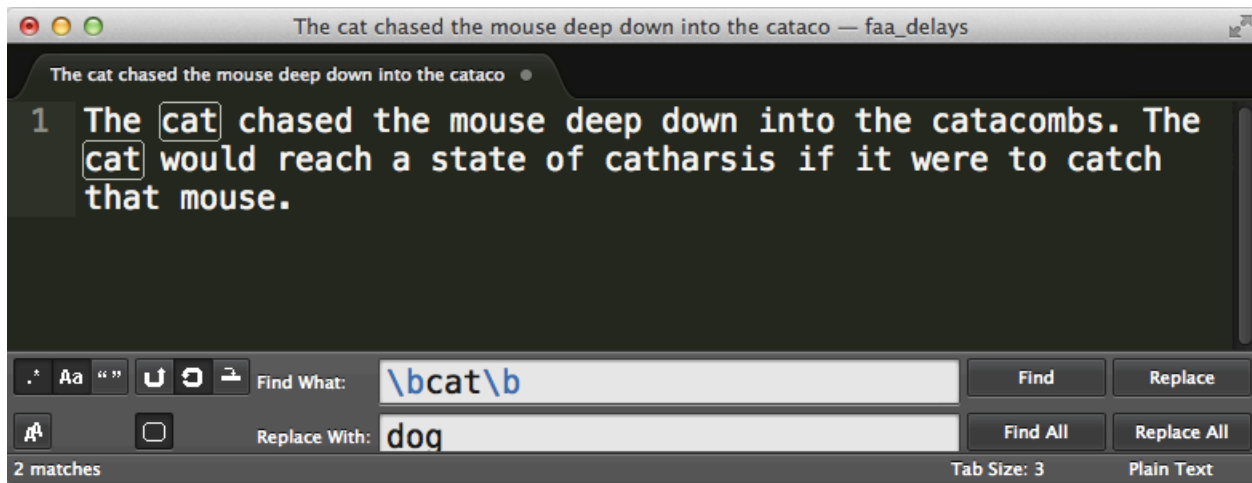
The cat chased the mouse deep down into the catacombs. The cat would reach a state of catharsis if it were to catch that mouse.

What is the regex needed to change all instances of `cat` to `dog` without affecting the words that happen to have `cat` inside of them?

Answer

Find `\bcat\b`

Replace `dog`



The regex match as done in Sublime Text 2

Congratulations on learning your first regex syntax. Word boundaries are a very useful pattern to match, and not just for finding cats. Sometimes we want to match just the last – or the first – digit of a string of characters, for instance.

That **backslash** character will be a recurring character in our regex exploration, and one that has different effects when combined with different characters and contexts. For now, it's good enough to tell it apart from its *forward*-facing sibling.

Regex Fundamentals

Removing emptiness

It's funny how empty space can cause us so much trouble. In the typewriter days, you were out of luck if you decided that a double-spaced paper should be single-spaced. Today, electronic word processing makes it easy to remove empty lines. Regular expressions make it even easier.

The newline character

What happens when you hit **Enter/Return**? You create a **new line** in your document and your cursor jumps to the beginning of that line so that you can begin filling in that new line.

The “character” created by hitting **Enter/Return** is often referred to as the **newline character**. This is how we represent it in regex syntax:

`\n`

Major caveat: Newlines are a simple concept. However, different operating systems and regex flavors treat them differently. In *most* cases, we can simply use `\n`.

However, you may find that that doesn't work. If so, try `\r` (think of `r` as standing for **R**eturn)

If you are using **TextWrangler**, you can basically substitute `\r` every time that I refer to `\n`. Other Mac text editors, including **TextMate** and **Sublime Text**, use `\n` to represent newlines.

Let's try *adding* a newline character. Given the following phrase:

Hello,world

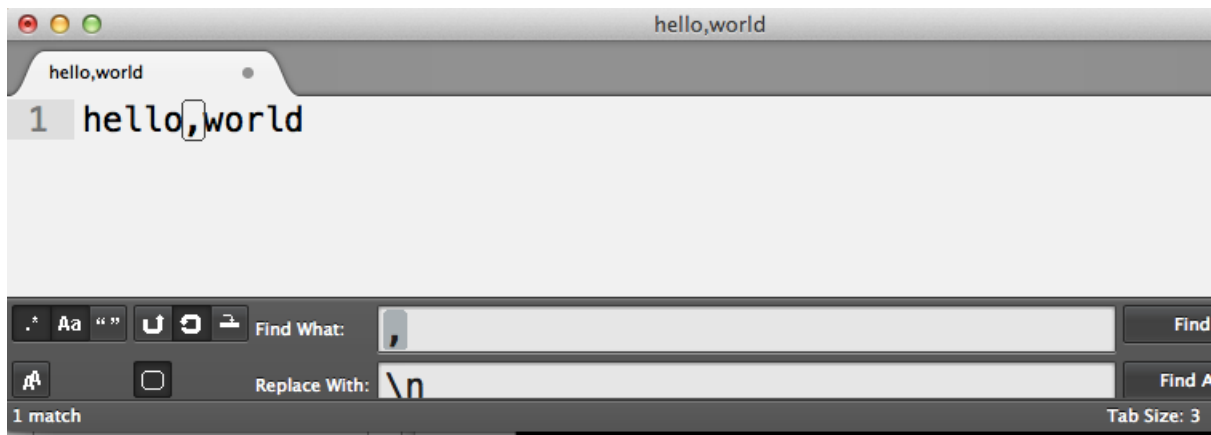
Replace that **comma** with a **newline**

Don't do it the old-fashioned way (i.e. deleting the comment and hitting **Return**). Use **Find-and-Replace**:

Find ,

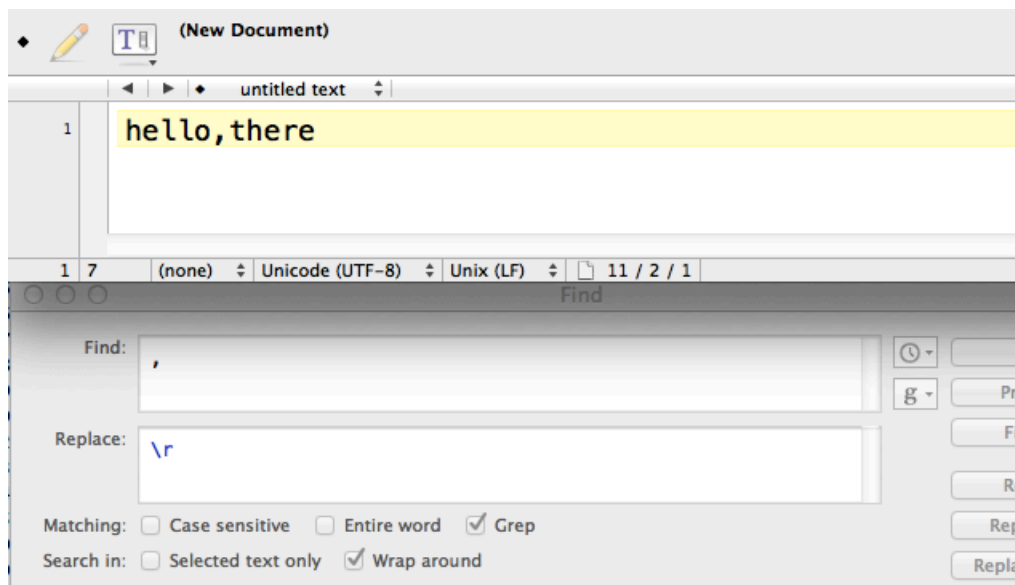
Replace \n

In Sublime Text, this is what your **Find-and-Replace** should look like:



Sublime Text, comma-to-newline

In TextWrangler, you have to use `\r` instead of `\n`:



TextWrangler, comma-to-newline

Hit **Replace** and you end up with:

Hello
world

One important thing to note: `\n` (or `\r`) is one of the few special characters that have meaning in the **Replace** action. For example, using `\b` will not do anything. But `\n` will replace whatever you specify in the **Find** the field with a newline character

Now let's do the opposite: let's replace a **newline** character with a **comma**. Start with:

```
Hello  
world
```

To *replace* the newline, simply reverse the operations that we did above:

Find `\n`

Replace ,

Viewing invisible characters

What's the difference between an empty line – created by hitting **Return** two times in a row – and an “empty” line, in which you hit **Return**, then the **space bar** a few times, and then **Return** again?

This is what the first kind of empty line looks like, using regex syntax:

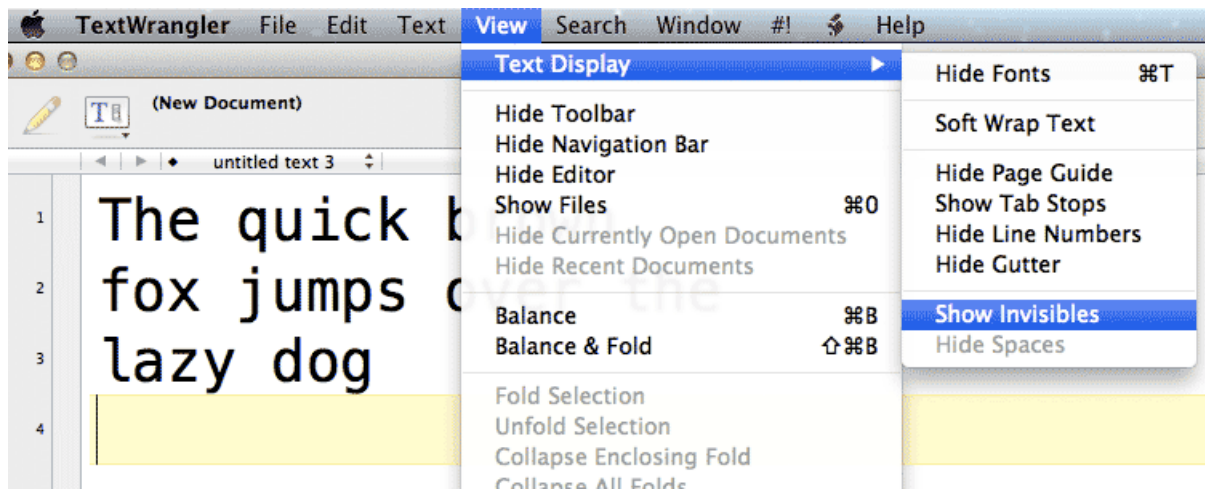
```
\n\n
```

And this is the second kind of “empty” line:

```
\n  \n
```

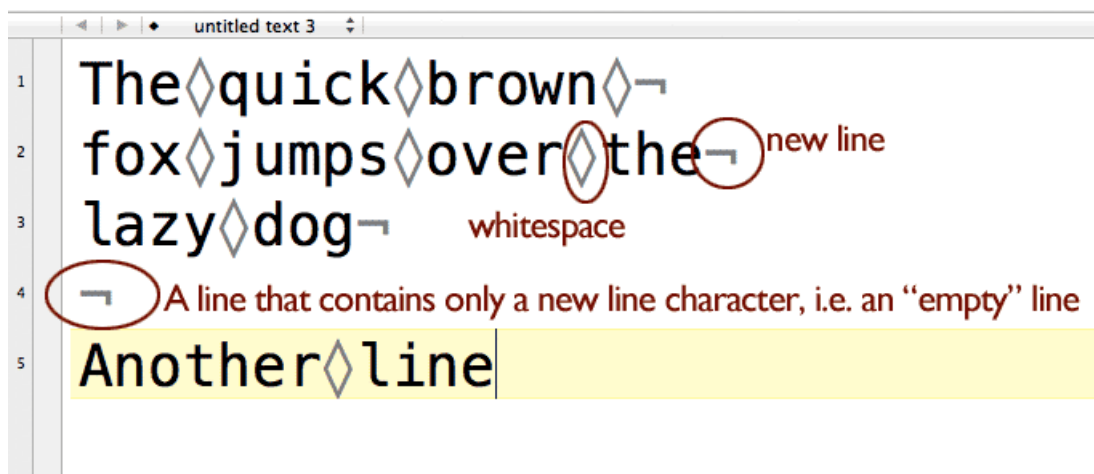
Of course, this is not the way it looks in a text editor. Both kinds of empty lines will look the same. But to a program, or to the regex engine that attempts to find consecutive newline characters, the two examples above are distinctly different.

Text editors (usually) have an option to show “**invisible characters**”. This option visually depicts white space with slightly-grayed symbols. In **TextWrangler**, here's the menu option:



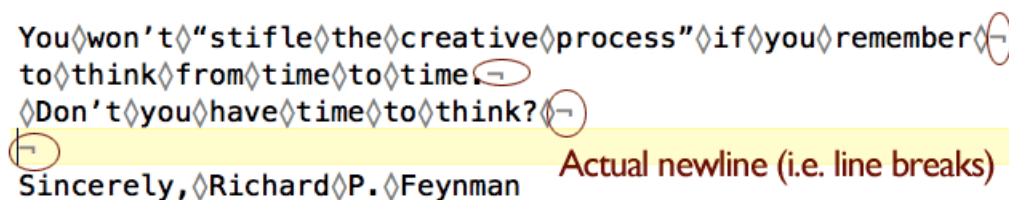
Finding the "Show Invisibles" option in TextWrangler

Here's what the "invisible characters" look like:



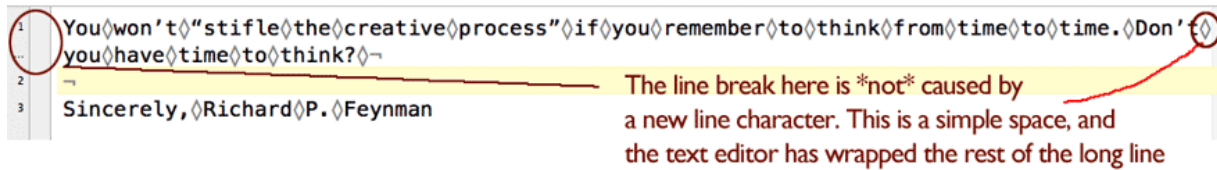
Invisible characters revealed, in TextWrangler

The now-visible "invisible characters" don't add anything to the actual text file. But they are useful for determining when there are actual line breaks (i.e. someone hit the **Return/Enter** key at the end of each line):



Multiple lines in TextWrangler

...as opposed to one long line of text that is word-wrapped by the text-editor:



A line of text word-wrapped in TextWrangler

Note: *I don't recommend showing invisible characters by default because they clutter up the view. But I'll use it from time-to-time to visualize the actual structure of the text.*

Exercise: Fix double-spacing

Given the double-spaced text below:

The quick brown
fox jumps over the
lazy dog

Make it single-spaced:

The quick brown
fox jumps over the
lazy dog

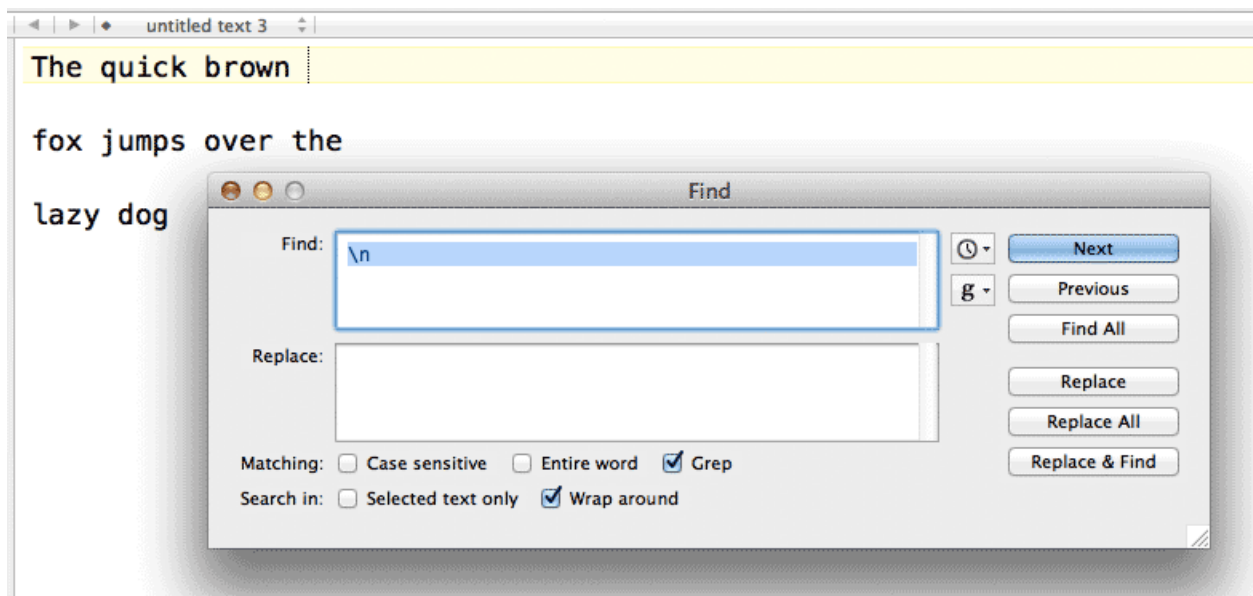
Answer First, let's describe what we want to fix in plain English: *We want to delete all lines that are empty.*

So pop open your text editor's Find-and-Replace tab, enable the regex checkbox, and type `\n` into the **Find** field. And let's **Replace** it with *nothing*; that is, don't put anything into the **Replace** field:

Here's what this looks like in Notepad++

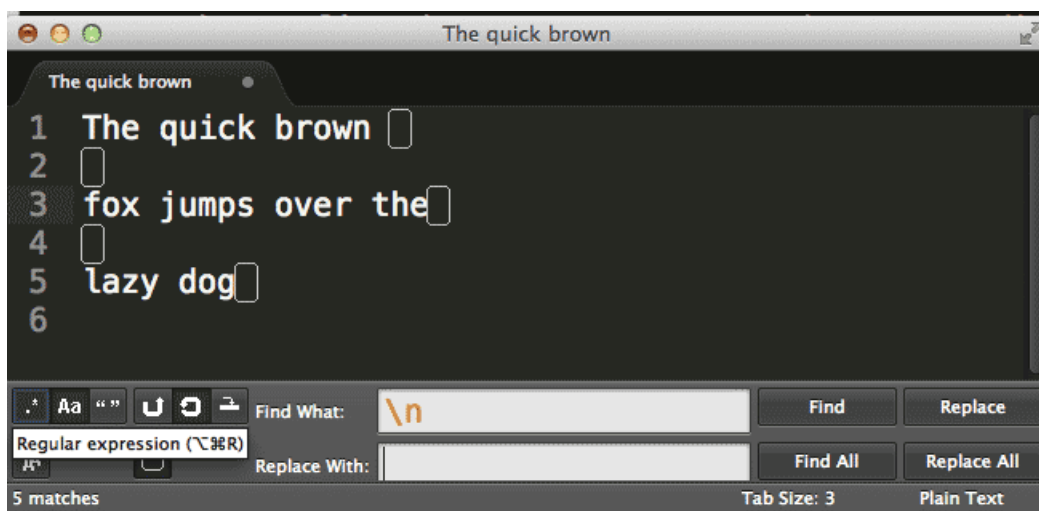
Todo: TKIMG

And in TextWrangler:



TextWrangler

And in Sublime Text 2:



Sublime Text 2

After hitting the **Replace All** button, your text window should look like this:

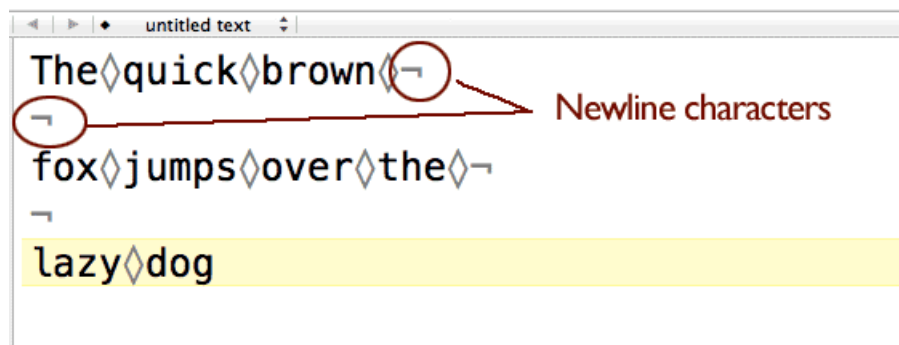


Replacement in Sublime Text 2

So we ended up replacing all the newline characters. But the result is not a *single-spaced document*, but all the words on a *single line*.

So we need to modify our pattern. If you turn on “show invisibles”, you’ll get a hint.

Answer #2 Remember when I said to think of the “empty lines” as not being *empty*, per se, but lines that contained only a **newline** character? That means immediately *before* an empty line is another newline character:



Seeing a pattern of new line characters

So what is the pattern for a text-filled line that is followed by an “empty line”? **Two** consecutive newline characters.

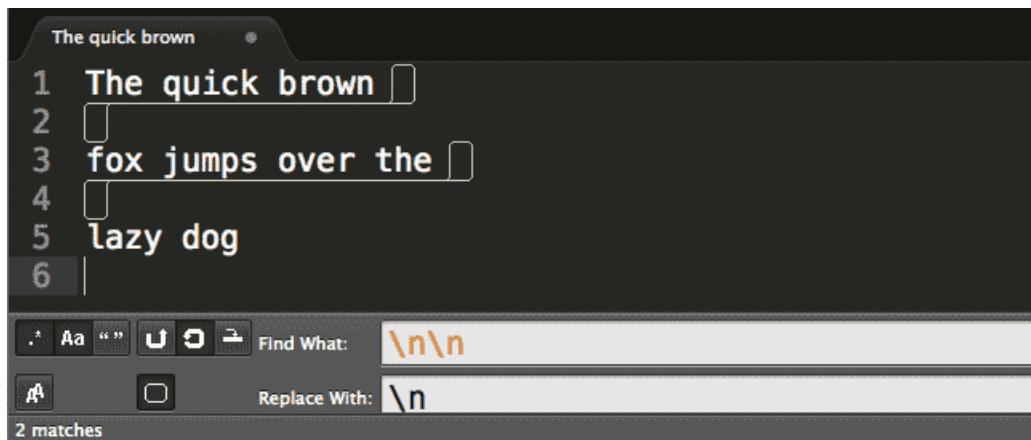
In the **Find** field, change the pattern to:

`\n\n`

And change the **Replace** field to:

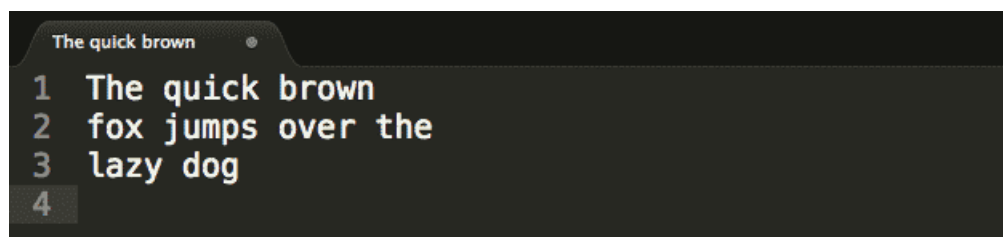
`\n`

In other words, we will be replacing all occurrences of `\n\n` with just a single `\n`, effectively changing this double-spaced document into a single-spaced document:



Double newline replacement in Sublime Text 2

We end up with this:



Single spaced result

So what happens when we want to replace a triple-spaced document? Or a quadrupled-spaced document? It's easy enough to add as many `\n` characters as you need, if annoying.

But what happens if a document contains double-spaced lines in one part, and triple-spaced lines in another? What if we don't really know how many kinds of line-spacing a text-file has? Are we resigned to doing trial and error with **Find-and-Replace**?

No. Of course regular expressions have a way to easily deal with this problem – or else I wouldn't bother learning them! Read on to the next chapter to find out how to make our pattern flexible enough to handle any of the above variations.

Match one-or-more with the plus sign

In the last chapter, we cleaned up double-spaced lines. But what if, instead of just double-spaced lines, you had a mix of spacings, where the lines could be double, triple, or quadruple-spaced?

This letter excerpt comes from Richard Feynman's published collection of letters, "Perfectly Reasonable Deviations from the Beaten Track"¹⁵

```
1 Dear Mr. Stanley,  
2  
3  
4  
5 I don't know how to answer your question -- I see no contradiction.  
6  
7  
8 All you have to do is, from time to time -- in spite of everything,  
9  
10 just try to examine a problem in a novel way.  
11  
12  
13  
14  
15 You won't "stifle the creative process" if you remember to think  
16  
17 from time to time. Don't you have time to think?
```

A letter with varied spacing between lines

If you **Find-and-Replace** with the `\n\n` pattern, you'll end up with this:

¹⁵http://www.amazon.com/gp/product/B007ZDDDO0/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=B007ZDDDO0&linkCode=as2&tag=danwincom-20

```
1 Dear Mr. Stanley,  
2  
3 I don't know how to answer your question -- I see no contradiction.  
4  
5 All you have to do is, from time to time -- in spite of everything,  
6 just try to examine a problem in a novel way.  
7  
8  
9 You won't "stifle the creative process" if you remember to think  
10 from time to time. Don't you have time to think?
```

Not completely single-spaced

Which requires you to run the pattern again and again. It seems like regular expressions should have a way to eliminate such trivial repetition, right? Before we find out how, let's state, in English, the pattern that we want to match:

We want to replace every instance of where there are at least two-or-more consecutive newline characters.

The plus operator

The plus-sign, +, is used to match *one or more* instances of the pattern that *precedes* it. For example:

a+

– matches any occurrence of one-or-more consecutive *letter* a characters, whether it be a, aaa, or aaaaaaaaaa

The following regex pattern:

\n+

– matches *one-or-more* consecutive **newline** characters, as the \n counts as a single pattern that is modified by the + symbol.

In other words, the + operator allows us match **repeating** patterns.

Let's use the plus sign to change the repeated newline characters in Richard Feynman's letter.

Exercise: Varied spacing

Given this text:

Dear Mr. Stanley,

I don't know how to answer your question -- I see no contradiction.

All you have to do is, from time to time -- in spite of everything,
just try to examine a problem in a novel way.

You won't "stifle the creative process" if you remember to think
from time to time. Don't you have time to think?

Convert it to:

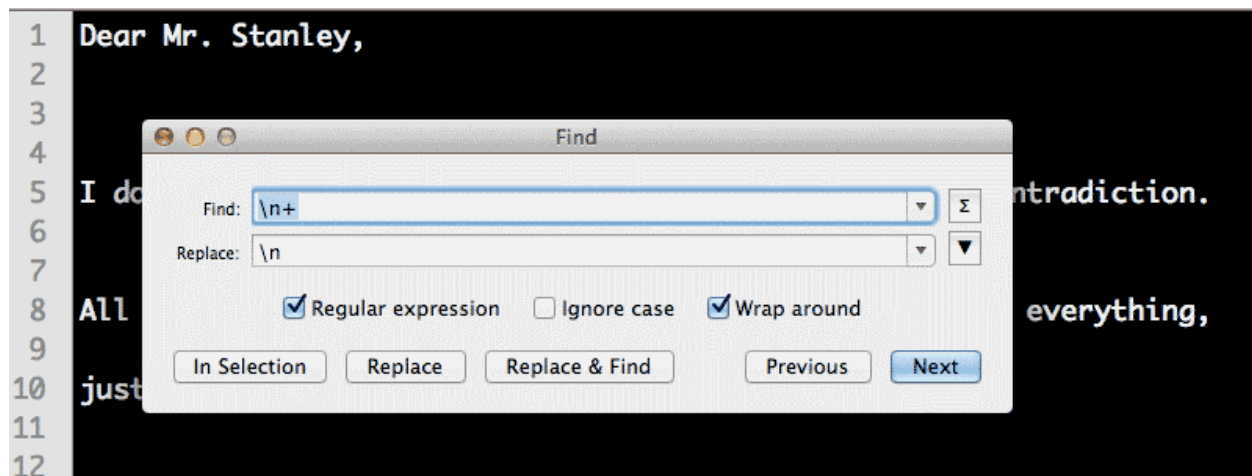
Dear Mr. Stanley,
I don't know how to answer your question -- I see no contradiction.
All you have to do is, from time to time -- in spite of everything,
just try to examine a problem in a novel way.
You won't "stifle the creative process" if you remember to think
from time to time. Don't you have time to think?

Answer

Find \n+

Replace \n

This is what our **Find-and-Replace** dialog box will look like:



Find-and-Replace in TextMate

Before we move on, let's do a little reflection: if the + in `\n+` modifies the pattern to match "one or more occurrences of a newline character" doesn't that affect *all* lines that were single-spaced to begin with?

Technically, yes. But since the **Replace** value is set to replace the pattern with a single `\n`, then those single-spaced lines don't see a net-change.

Try it out yourself. Copy this single-spaced block of text:

```
The quick brown fox  
jumps over  
the lazy  
dog
```

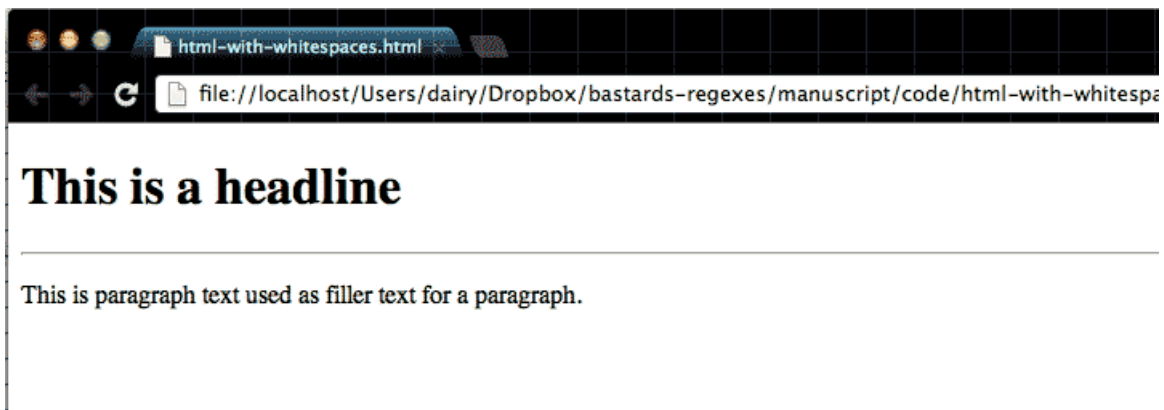
And set **Find** to: `\n+` and **Replace** to: `\n`

You should see no change at all in the text (though technically, all newline characters are replaced with a newline character, there's just no difference).

Exercise: Replace consecutive space characters in HTML

The HTML behind every web page is just plain text. However, the text in raw HTML may have a totally different structure than what you actually see.

Here's a simple webpage:



Simple webpage

It looks like it's two lines of text, right? If you were to copy-and-paste the webpage to a text-editor, you'd get this:

This is a headline

This is paragraph text used as filler text for a paragraph.

However, if we view the webpage's **source code**:



The relevant HTML code is circled

What happened to all the extra whitespace in the phrase, "This is paragraph text used as filler text

for a paragraph”? Web browsers *collapse* **consecutive whitespace**. If there are *one-or-more* space characters, the web browser renders those whitespaces as just *one* whitespace.

And newline characters don’t appear at all; they’re treated as normal whitespace characters.

(**Note:** *So how is there a line break between the headline and paragraph? That’s caused by the HTML tags. But this isn’t a HTML lesson so I’ll skip the details.*)

This approach to rendering whitespace is referred to as **insignificant whitespace**.

So why do we care? Well, we normally don’t. But if you ever get into web-scraping – writing a program to automatically download web pages to turn them into data – your program will return the text content of that paragraph as:

```
This is          paragraph
text
used as        filler text
for
a
paragraph.
```

Usually, during web-scraping, it’s fine to store the raw text in its original form. But sometimes you just want it as a readable sentence:

```
This is paragraph text used as filler text for a paragraph.
```

Write the regex pattern(s) that will clean up the insignificant whitespace.

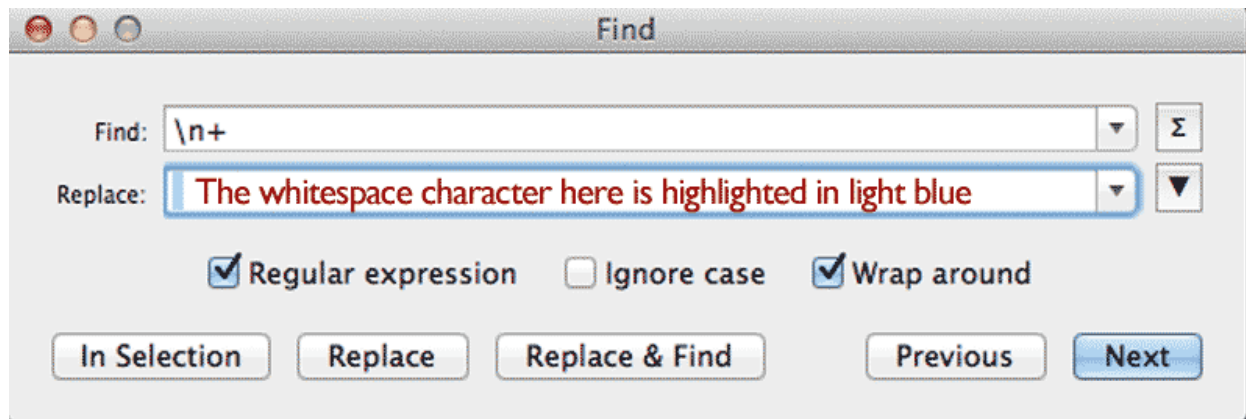
Answer

We represent the whitespace character in our pattern as, well, a whitespace. So just use your **spacebar**. Let’s start by replacing all consecutive **newline** characters with a whitespace:

Find \n+

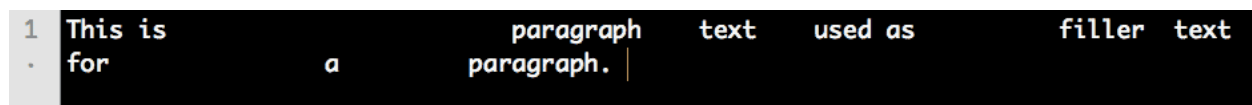
Replace

Realize that there is an actual **whitespace** character in the **Replace** box, **not** *nothing*. It’s easier to see in the text editor:



Whitespace character in Find-and-Replace

The result is this word-wrapped line:



A single line with too many whitespaces

To fix those whitespaces, use the same pattern above except **Find** a whitespace character instead of `\n`, and again, **Replace** with a single whitespace character.

Backslash-s

Most regex languages have a shortcut symbol that handle both white spaces and newline characters (including text files that use `\r` to represent newlines):

`\s`

You can perform the above exercise in one **Find-and-Replace** by just using `\s+` in the **Find** pattern to **Replace** with a single whitespace character.

The plus operator is very useful and we'll be using it many of our patterns to come. Don't get the impression that it's only useful with whitespaces – the `+` can be combined with any character or pattern. Sometimes the problem is that the `+` matches *too much*, so we'll learn in later chapters how to control the number of repetitions, rather than relying on the plus-operator's one-and-all approach.

Match zero-or-more with the star sign

The plus operator is handy for matching an unknown number of repeating characters or sequences. But sometimes you don't even know if there's even a *single* occurrence of a character.

The star sign

The star-sign, *, is used to match *zero or more* of the pattern that *precedes* it. For example:

`e*`

– matches any occurrence of **zero-or-more** consecutive *letter* `e` characters.

The pattern:

`be*g`

– matches:

`beg`

`beeeeg`

`bg`

Exercise: Baaa!

The paragraph below has a variety of sheep-based expressions:

The first sheep says “Baa!” Another sheep says “Baaaa!!!”. In reply, the sheep says, “Baaaaaa”. Another “Baaa” is heard. Finally, there’s a “Baaaa!”

Convert all of them to, simply, “Baa!”

The first sheep says “Baa!” Another sheep says “Baa!”. In reply, the sheep says, “Baa!”. Another “Baa!” is heard. Finally, there’s a “Baa!”

If we use the **plus operator**, we have:

Find Baa+!+

Replace Baa!

The first sheep says “Baa!” Another sheep says “Baa!”. In reply, the sheep says, “Baaaaaa”. Another “Baaa” is heard. Finally, there’s a “Baa!”

Because the second + depends on there being *at least one exclamation mark* in the pattern – i.e. Baa!! – it fails to match a Baaa that *doesn’t* end with an exclamation mark.

Answer

Find Baa+!*

Replace Baa!

If you know the plus sign operator, then the star sign is just a slightly more flexible variation.

Specific and limited repetition

The **plus** and **star** operators are a nice catch-all for repeated patterns. But as you can imagine, sometimes you'll need to match something that is repeated more than *once* but less than infinity.

Curly braces

To capture **n** repetitions of a pattern – where **n** is the number of repetitions – enclose **n** inside **curly braces**

The following pattern:

`a{4}`

– will match *exactly* four a characters:

`"B**aaaa**"`, said the sheep

This is the simplest form of the **curly braces** notation, let's practice using it before learning the other forms.

Exercise: zeros to millions

In the following list of numbers:

100
2000
500000
4000000
2000000
1000000
8000
9000000

Replace the **zeroes** with **millions** for the numbers that have **exactly six zeroes**:


```

100
2000
500000
4 million
2 million
1 million
8000
9 million

```

Answer

Find `0{6}`

Replace `million`

Bonus exercise

Perform the same exercise as above, but with this list of numbers:

```

900
8000000
90000000
70000
100000000
2000000
50000000000

```

If you use the exact solution as in the prior example, you'll end up with this:

```

900
8 million
9 million0
70000
1 million00
2 million
5 million0000

```

Oops. That pattern matched and replaced the *first* six zeroes. What we need is to replace the *last* six zeroes:

900
 8 million
 90 million
 70000
 100 million
 2 million
 50000 million

Hint: Remember the very [first regex syntax]{#word-boundary} that we learned about?

Answer

Find `0{6}\b`

Replace `million`

Curly braces, maximum and no-limit matching

If we want to match **m-number** of repetitions *but* fewer than **n-number**, we include both **m** and **n** in the curly braces notation:

The following pattern:

`0{3,7}`

– will match any text string containing *at least* 3 but no more than 7 consecutive zeroes:

000100000000

However, if you omit the second parameter (the *maximum* number of repetitions) *but* leave in the **comma**, like so:

`0{3,}`

– the regex will match a text string containing at least 3 *or more* consecutive zeroes.

This open-ended form of the curly braces notation is very useful for narrowing down a certain sequence of characters to find and replace.

Exercise: Baa

An essay about sheep has inconsistently spelled “Baa”:

Sheep in Northwest America say “baaaaa.” Sheep in Mongolia also say “baaaaaaaa.”
 The deepest conversation I ever had with a sheep went like this:
 “Hello!” “Baaaaa” “You’ve been a bad sheep!” “Baaaaaaaaa!”

Fix it so that all instances of sheep-talk are represented simply as baa

Answer

To get an idea of the flexibility curly braces offer us, try doing this exercise with the `plus` operator:

Find `a+`

Replace `aa`

Because `+` affects *one-or-more* instances, all single-`a`’s are caught and erroneously replaced with `aa`:

Sheep in Northwest Americaa saay “baa.” Sheep in Mongoliala aalso saay “baa.” The
 deepest conversaation I ever haad with aa sheep went like this:
 “Hello!” “Baa” “You’ve been aa baad sheep!” “Baa!”

But with curly braces, we can limit what get’s replaced to just the instances in which `a`’s are repeated *at least 3 times*:

Find `a{3,}`

Replace `aa`

Exercise: Markdown

Todo: ## Markdown, decrementing # example

Turn

```
# Act I
### Scene 1
#### Soliloquy A
#### Soliloquy B
# Act 2
### Scene 1
### Scene 2
#### Soliloquy A
# Act 3
### Scene 1
```

Into

```
# Act I
## Scene 1
### Soliloquy A
### Soliloquy B
# Act 2
## Scene 1
## Scene 2
### Soliloquy A
# Act 3
## Scene 1
```

Answer

Find $^{*}\{2\}$

Replace #

What happens without the anchor?

```
# Act I
## Scene 1
## Soliloquy A
## Soliloquy B
# Act 2
## Scene 1
## Scene 2
## Soliloquy A
# Act 3
## Scne 1
```

Cleaning messily-spaced data

Sometimes you come across a text or PDF file that contains data that you want to put in a spreadsheet:

Red Apples Green Oranges Blue Pears
 \$10 \$50 \$100

If you attempt to paste that into Excel, however, you end up with:

	A	B	C
1	Red Apples Green Oranges Blue Pears		
2	\$10 \$50 \$100		
3			
4			
5			

The data is just a bunch of text crammed into a single Excel cell

That may *look* OK, but that's not what we want. We want each datapoint to occupy its own cell, like so:

	A	B	C	D	E
1	Red Apples	Green Oranges	Blue Pears		
2	\$10	\$50	\$100		
3					

Each datapoint is separated by a column in Excel

In order for this to happen, Excel needs some help. We have to *tell* it that where *there are three spaces or more* that's where a datapoint ends. The easiest way to do that is to *replace* those extra spaces with a symbol – i.e. a **delimiter** – that Excel will interpret as a column break.

The most common symbol is a comma. Using the **curly braces** notation, we can replace the consecutive spaces with a comma:

Find {3,}

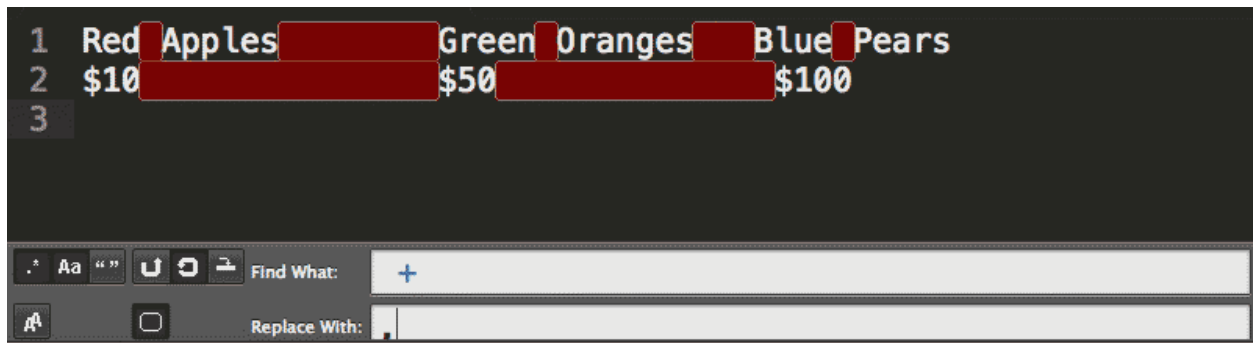
Replace ,

Which converts the text to:

Red Apples,Green Oranges,Blue Pears
 \$10,\$50,\$100

If you save that text as a .CSV (comma-separated values) file, Excel will see those commas and will know that those commas are meant to represent column breaks.

What if we didn't know how to use curly braces and instead used the + operator? Then we would've put commas in *every* space:



The red areas represent the matched whitespace

This of course means commas between each word:

Red,Apples,Green,Oranges,Blue,Pears
 \$10,\$50,\$100

And if Excel interprets each comma as a column break, we end up with:

	A	B	C	D	E	F
1	Red	Apples	Green	Oranges	Blue	Pears
2	\$10	\$50	\$100			
3						

Extra, unwanted column breaks

When you don't need a **plus**, the **curly braces** are a nice alternative. We'll be using them a lot when we need to perform more precise matching.

Anchors: A way to trim emptiness

In a [previous chapter]{#plus-operator}, we learned how to match consecutive newline characters. This is handy when we want to remove blank lines from a text file.

Another annoying whitespace problem occurs when there is padding between the beginning of the line and the first actual text character. For example, given these [haikus](#)¹⁶:

```
do what you feel like since the work is abandoned the law doesn't care
if you use this code you and your children's children must make your source free
```

I want them to be formatted so that the leading whitespace is removed and the lines are flush against the left-margin:

```
do what you feel like since the work is abandoned the law doesn't care
if you use this code you and your children's children must make your source free
```

These “license haikus” are courtesy of [Aaron Swartz](#)¹⁷

The caret as starting anchor

We need a way to match the *beginning of the line*. Think of it as a word-boundary ‐ remember `\b?` – except that we want a boundary for the *entire line*.

To have a regex include the beginning of the line, use the caret: `^`

The following regex:

```
^hello
```

– will match the word `hello` if `hello` occurs at the **beginning of the line**, as in the following example:

```
hello goodbye
```

However, `^hello^` will not match the `hello` in the following line:

```
goodbye hello
```

¹⁶<http://www.aaronsw.com/weblog/000360>

¹⁷<http://www.aaronsw.com/weblog/000360>

So, in order to remove unneeded spaces at the beginning of a line:

In English We're looking to match all whitespace characters that occur at the beginning of a line. We want to replace those characters with nothing (i.e. we want to delete them).

Find `^ +`

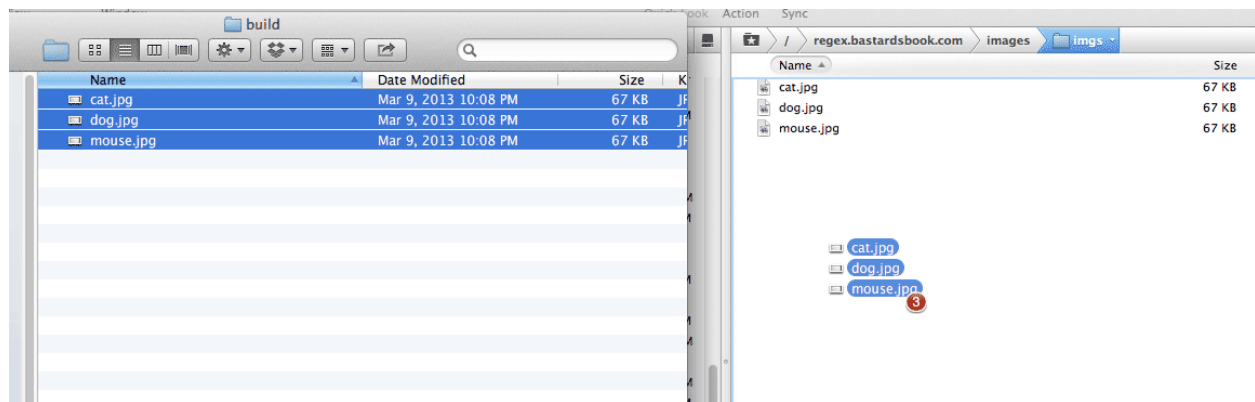
Replace (*with nothing*)

This regex only affects lines that begin with *one-or-more whitespace characters*. For those lines, those whitespace characters are removed.

Exercise: Add a common directory to a list of files

As a digital photographer, I run into this scenario all the time: I've taken a bunch of photos and want to send them to someone across the world. But with megapixels being so plentiful today, these files in their raw format can weigh in as much as 10MB to 20MB – too big to send as email attachments. And this client doesn't have a shared Dropbox.

So the easiest way to send them is to upload them to an FTP or web server. On my computer, I do this by opening the folder, selecting all the files, and dragging them to my server:



Moving files

The client can then download the files at their leisure. What was `dog.jpg` in my local hard drive can now be found at:

<http://www.example.com/images/client-joe/dog.jpg>

But how do I tell the client this? Well, first, I select all the filenames on my local hard drive and do a **copy-and-paste** into a text editor. I get something like:

dog.jpg
cat.jpg
mouse.jpg

After this it's easy enough to just copy and paste the remote path – `http://www.example.com/images/client-joe/d` – 3 or so times. But why do that when, with regular expressions, we can do it in one fell swoop?

Use the **caret** anchor to prepend the remote path to the file names:

`http://www.example.com/images/client-joe/dog.jpg`
`http://www.example.com/images/client-joe/cat.jpg`
`http://www.example.com/images/client-joe/mouse.jpg`

Answer

Simply “replace” the **caret** with the remote file path:

Find `^`

Replace `http://www.example.com/images/client-joe/`

Technically, nothing gets *replaced*, per se. The **caret** simply marks the beginning of the line and the replacement action inserts the desired text there.

This is a quick time saver when you just have to add something to a dozen or so lines.

The dollar sign as the ending anchor

If there's a way to match the beginning of a line, then there must be a way to match the end of a line.

To have a regex include the end of the line, use the dollar-sign:

`bye$`

– this will match the word `bye` if it occurs at the **end of the line**, as in the following example:

world, goodbye

However, it will not match the `bye` in the following line:

goodbye world

Exercise: Add a common filename to a list of directories

This is a variation to the previous file path problem. We have a list of directories:

```
http://example.com/events/
http://example.com/people/
http://example.com/places/
```

And we need a reference to a common filename:

```
http://example.com/events/index.html
http://example.com/people/index.html
http://example.com/places/index.html
```

Answer

Find \$

Replace index.html

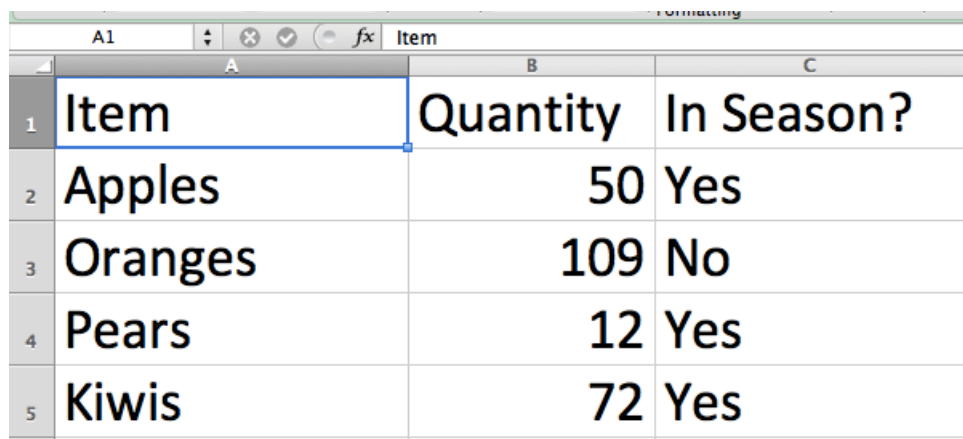
Stripping end-of-line characters

Comma-delimited files (also known as CSVs, *comma-separated values*) is a common text format for data. In order for a program like Excel to arrange CSV data into a spreadsheet, it uses the commas to determine where the “columns” are.

For example:

```
Item,Quantity,In Season?
Apples,50,Yes
Oranges,109,No
Pears,12,Yes
Kiwis,72,Yes
```

In Excel, the data looks like this:



Item	Quantity	In Season?
Apples	50	Yes
Oranges	109	No
Pears	12	Yes
Kiwis	72	Yes

CSV in Excel

CSV is a popular format and you'll run into it if you get into the habit of requesting data from governmental agencies. Unfortunately, it won't always be perfect.

A common problem – albeit trivial – will occur when exporting data from Excel to CSV. If the last column is meant to be *empty* but *isn't* (not all databases are well maintained), every line in the CSV file will have a trailing comma:

```
Item,Quantity,In Season?,
Apples,50,Yes,
Oranges,109,No,
Pears,12,Yes,
Kiwis,72,Yes,
```

This usually won't cause problems, especially in modern spreadsheets such as Excel and Google Docs. But older database import programs might protest. And if you're even a little OCD, those superfluous commas will bother you. So let's wipe them out with a single regex.

Inefficient method: remove and replace newlines If you fully grok newline characters, you realize you can fix the problem by targeting the pattern of: *a comma followed by a newline character*:

Find , \n

Replace \n

This *almost* works, except you won't catch the very last line:

```
Item,Quantity,In Season?
Apples,50,Yes
Oranges,109,No
Pears,12,Yes
Kiwis,72,Yes,
```

Why wasn't that trailing comma from the final line deleted? The pattern we used looked only for commas followed by a **newline character**. So if that final line is the *final* line, there is no other newline character. So we have to delete that comma manually.

Efficient method: leave the newlines alone Besides that nagging manual deletion step (which quickly becomes annoying if you're cleaning dozens of files), our **Find-and-Replace** just seems a little *wasteful*. All we want is to delete the trailing comma, but we end up *also* deleting (and reinserting) a newline character.

Using the `$`, however, requires no extra replacement action. It simply asserts that the pattern contains the end-of-the-line, which isn't an actual *character*, per se, in the text.

In other words, the `$` only detects the end-of-the-line, and so no replacement is actually done at that position.

Try it out yourself. In your **Find** field, match the `$` operator and then **replace** it with nothing: nothing will happen to the text (compare this to replacing `\n` with nothing).

So let's strip the trailing commas using the `$` operator:

Find ,`$`

Replace (*with nothing*)

In English We want to find the comma right before the end-of-the-line

Escaping special characters

If the **caret** and the **dollar sign** denote the beginning and the end of the line, respectively, then it follows that the following patterns don't make much sense:

```
abc^Bye
Hello$world
```

How would `abc` come before the beginning of the line yet be on the same line as `Bye`? Or how could `world` be on the same line as `Hello` and the end of the line character, `$`?

So, with what we know so far, it makes sense that the caret and dollar sign typically serve as bookends for a regex pattern. However, the following regex may be confusing:

```
He gave me \b$10+
```

This regex matches the text in bold below:

He gave me \$100 yesterday

The key character here is our friend the **backslash**. We've seen how the backslash, when preceding the letters `b` and `n`, give them *special* meaning: a word boundary and newline character, respectively.

What happens when the backslash precedes an *already* special character, such as the dollar sign? Then it's just a *literal* dollar sign, as we see in the above example.

Exercise: Remove leading dollar signs Given this list of dollar amounts in which the dollar sign is mistakenly repeated:

\$\$100

\$\$200

\$\$399

Remove the leading dollar sign with a regex that contains the **caret**.

\$100

\$200

\$399

Answer

Find `^\$`

Replace (with nothing)

Anchor characters are not only pretty easy to remember, they're very useful in many regex patterns. I use it frequently for data-cleaning, where a text file may contain unwanted spaces or junk characters at the beginning or end of each line.

Matching any letter, any number

Of course, we'll need to work with actual alphabetical letters, numbers, and symbols other than whitespace and newlines. Regular expressions have a set of shortcuts and syntax for this purpose.

The numeric character class

Using our friend the **backslash**, we can turn a literal letter into a special character to match numbers. When the letter `d` is preceded by a backslash, the pattern no longer matches the literal letter `d`, but any numerical digit from 0 to 9:

`\d+`

Matches:

There are **10010** sheep

We refer to `\d` as a “character class” or “character set” because it affects a set of characters. In this case, `\d` matches *the set (i.e. “class”) of characters that are numbers*.

Let's try out the `\d` in a simple exercise.

Exercise: Redact the numbers

Given this phrase:

The robbery was reported on January 12, 2013. At 9:40, the suspect is alleged to have entered the pizzeria at 120 Broadway and stolen 9 bags of oregano.

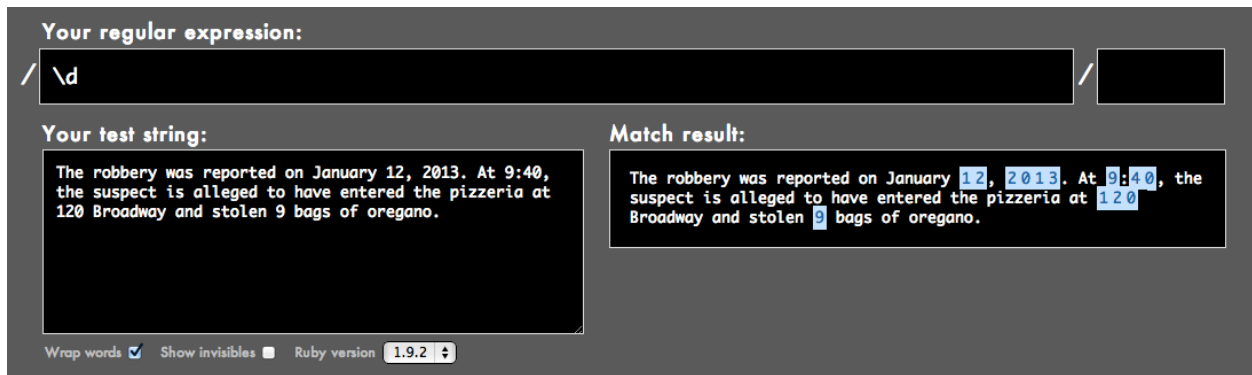
Replace all the numbers with the letter `x`

Answer

Find `\d`

Replace `x`

Using [Rubular.com](http://rubular.com)¹⁸, we can see an interactive preview of how the text is affected by our pattern:



Rubular.com's highlighting

The result of our Find-and-Replace:

The robbery was reported on January XX, XXXX. At X:XX, the suspect is alleged to have entered the pizzeria at XXX Broadway and stolen X bags of oregano.

Exercise: Masking addresses of crime reports

When releasing crime reports, some law enforcement agencies have the policy of masking the exact address of a reported incident, ostensibly to protect the privacy and safety of the victim.

Thus, given the following addresses:

1289 Houston St.
402 W. Abbey Ave.
9 S. 8th Street

The masked addresses would be:

¹⁸<http://rubular.com>

1200 block of Houston St.
400 block of W. Abbey Ave.
00 block of S. 8th Street

Answer

We basically want to “zero out” the least significant digits of an address. On a street where the address numbers are three or four digits, zeroing out the last two digits generally provides enough vagueness.

We can use `\d` to match any numerical digit and **curly braces** to target **1 to 2** digits. We also need to use a right-side **word boundary** so that the rightmost digits are matched:

Find `\d{1,2}\b`

Replace 00 block of

Note: There are competing public interests to consider here: we want to mask the exact address of the incident. However, we also don’t want to redact the address to the point that its geographical characteristic is rendered meaningless or paints a misleading characteristic.

For instance, if we changed the range of the curly braces like so:

Find `\d{1,3}`

Replace 000 block of

Then the following addresses:

1520 Broadway, New York, NY
1001 Broadway, New York, NY
1999 Broadway, New York, NY

Are effectively mapped to:

1000 block of Broadway, New York, NY
1000 block of Broadway, New York, NY
1000 block of Broadway, New York, NY

In New York, TK map image

Word characters

But who wants to just match numerical digits? For the non-discerning regex user, there's syntax for matching all the letters of the alphabet, too:

A **word-character** includes the letters A to Z – both uppercase and lowercase – as well as all the numbers, and the underscore character, _

`\w+`

Matches:

Blathering_blatherskite42!!!

Let's use `\w` in the previous exercise text.

Find `\w`

Replace `X`

The result: everything but the punctuation and spaces is X-ed out.

```
XXX XXXXXXXX XXX XXXXXXXX XX XXXXXXXX XX, XXXX. XX X:XX, XXX
XXXXXXXX XX XXXXXXXX XX XXXX XXXXXXXX XXX XXXXXXXX XX XXX XXXXXXXX
XXX XXXXXXX X XXXX XX XXXXXXXX.
```

Bracketed character classes

As you might guess, not every scenario requires such promiscuous patterns. We can specify just a few characters we want to match using **square brackets**:

`[bcd]og`

– will match bog, cog, and dog

Here's a few examples:

Match either a or b `[ab]` (order doesn't matter, so `[ba]` also works)

Match 1, 2, 9, or a space character `[1 29]`

Match a dollar sign or any of the word-characters `[$\w]`

Not so special inside the brackets

The previous example has a quirk to it: notice that the `$` is treated as a *literal dollar-sign*. It doesn't represent an end-of-line anchor as we learned about in the previous chapter. This is because *within* the brackets, nearly all the characters act just as normal, non-special characters.

There are a few exceptions to this: the backslash performs its normal escaping capacity, so `[\w]` effectively includes all word-characters. The **hyphen** and the **carat** are also special within the brackets, and we'll learn about their meanings later.

Exercise: Just the numbers

In the following list of payments, erase all the symbols that are unrelated to the actual amount:

```
$1,200.00  
$5,600.25  
$100.09  
$42,100.01
```

The decimal points are needed to preserve the fractional values, but we can do away with the dollar signs and commas.

Answer

We can remove both punctuation symbols with a character set:

Find `[$,]`

Replace (with nothing)

The result:

```
1200.00  
5600.25  
100.09  
42100.01
```

Again, note that the `$` does not have to be escaped when inside the brackets.

Matching ranges of characters with brackets and hyphens

What if you want to deal with not just two characters, but ten characters? Then bracketed character sets get unwieldy:

[abcdefghij]

As I mentioned previously, the **hyphen** acts as a special character inside the brackets.

Inside square brackets, the **hyphen** denotes a range between two characters:

- [a-z] will match any lowercase letter from a to z
- [1-5B-G] matches numbers 1 through 5 and uppercase letters B to G

Exercise: Filtering out naughtiness

If you've ever been in an online chatroom or discussion board, you've probably seen how certain bad words are censored. And sometimes, you see the side effect: users misspelling those censored words so they can continue in part to convey their true feelings.

So if "heck" is considered an improper word, the discussion board might censor it as "—". So a half-clever user might try to bypass the filter by using "h3ck", "hecck", or "hekk". And if those get blocked, there's a forest of permutations that can be used: "h33ck", "h3kk", "h3cck"

How do those system-moderators *not* spend their day creating infinitely-long lists of naughty words?

Given the following lovely *communique*:

Go heck yourself you dumb hacker. You don't know what the h3ck you are hekking doing, so go back to your H3KK-hole farm and go shuck some h3cking corn.

Use character sets to remove all the variations of heck.

Answer

There's no exact science to this and so, interpretations will vary on whether a misshapen word looks like heck enough to merit censoring.

So let's set some bounds. A variation of heck is defined as any word that:

- begins with h
- ends with k
- in which the second character is either e or 3
- in which the in-between letters is a combination of e, c, 3, k

Since regex is not by default case-insensitive, we have to include both upper and lowercase versions of the letters in our solution:

Find `[hH][eE3][cCKk]+`

Replace `----`

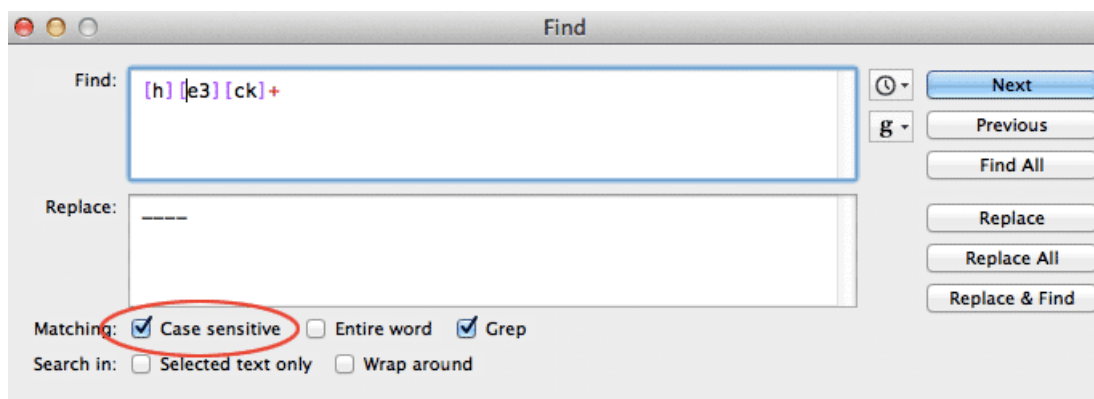
The result:

Go -- yourself you dumb hacker. You don't know what the -- you are --ing doing,
so go back to your --hole farm and go shuck some --ing corn.

Case insensitivity flags

In the regex universe, there are several ways to indicate that you want your pattern to ignore the case of letters so that `[a-d]` matches A, a, c, B, and so forth.

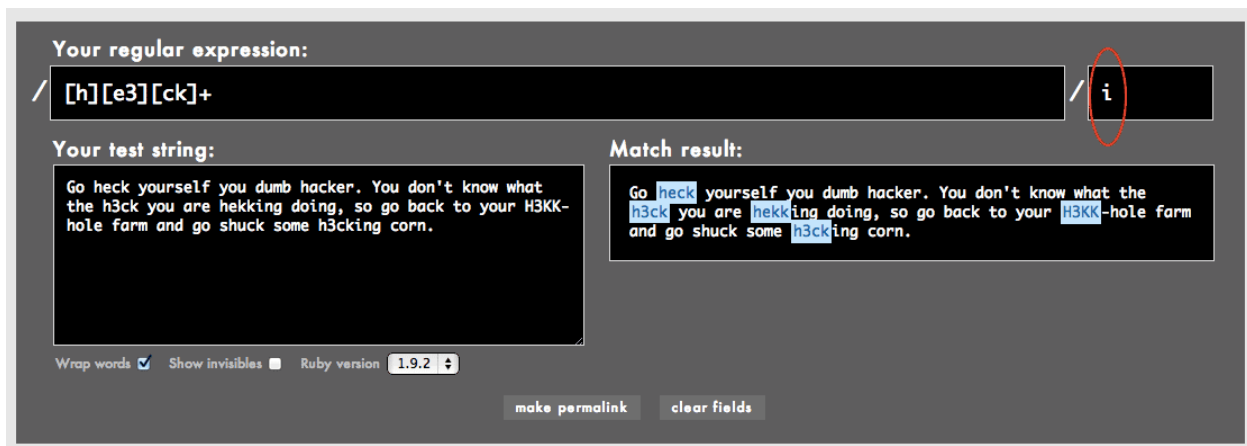
Since we're just using regex inside of a text editor, there is usually an option to select **case insensitivity**.



TextWrangler case insensitivity checkbox

[Rubular.com](http://rubular.com)¹⁹ is an emulation of how the Ruby programming language does regexes, so it uses the *flag* system. In Ruby, the letter *i* is used after the regex (which in Ruby, is enclosed with forward slashes /) to denote case-insensitivity:

¹⁹<http://rubular.com>



Rubular.com case insensitivity flag

Note: you obviously don't have to remember Ruby's implementation details if you don't intend to program in Ruby or to program at all.

If we have case-insensitivity selected, our regex in the previous exercise can be simplified to this:

```
[h] [e3] [ck] +
```

Or this, if you prefer expressing everything in all-caps:

```
[H] [E3] [CK] +
```

What you see is not what you get

If it isn't clear by now, the length of the regex does not necessarily reflect the length of text that it will match. In the above example, `[e3]` matches either `e` or `3` – but just one of those possible characters.

Thus, `he3ck` would *not* be matched. Neither would `h33ck`. That bracketed set, `[e3]`, matches exactly *one* character. However, `[ck]+` matches: `heck`, `hecc`, `heckckck`, and so forth, because of the *+ one-or-more* repetition modifier.

To hammer home the point, the following pattern:

```
[123456789] [a-z] +
```

Matches:

```
1 apple
2 oranges
```

But *not* any of the following lines:

```
12 apple
29 oranges
```

– because `[123456789]` matches just a *single* digit followed by a space.

All the characters with dot

Sometimes, we just want to match *anything*. Yes, there’s a regex for that.

The dot character – also referred to as a “period” – simply matches everything – *except* the newline characters `\r` and `\n`

For example, to match the first three characters of this line:

```
7@%x90js(las
```

The pattern is simply:

```
...
```

Or, with the **curly braces** notation:

```
.{3}
```

The **dot** matches *everything*: Latin letters, numbers, punctuation, whitespace characters, the entire Chinese alphabet. Every character falls within the dot’s purview, *except* for the **newline** character.

For example, given this gibberish:

```
1  @#$*A(SDFjkLJAbJSDKFLASDLFJxklsadfjl
2
3
4
5  asdfkl8724'"#!"$Lv hkasd!@$@#$
6
```

And applying this Find-and-Replace-All:

Find .

Replace x

We end up with:

```

1
2  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
3
4
5  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
6

```

Note that the newline characters are *preserved*. If the dot character had matched and replaced them, then our result would have been one long line of X characters:

```

1 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

The dot character is an incredibly flexible piece of our pattern matching toolset. But it's frequently abused because of how easy it is to type a single dot than a specific character set. In the next chapter, we'll learn about [negative character sets]{#negative-character-sets}. Using a combination of that technique with regular character sets is usually preferable – i.e. safer – than using the all-matching dot.

Shorthand notation

As you might have realized, the `\d` syntax we learned at the beginning of this chapter is equivalent to this bracketed set:

```
[0-9]
```

And `\w` is equivalent to:

```
[A-Za-z_]
```

You can use `\s` to handle all white space characters, including regular spaces (made with the spacebar), tab-characters (`\t`), and newlines (`\n`).

Exercise: Cleaning up Wikipedia notations

Every once in awhile when doing research, you find yourself on a Wikipedia webpage, and the content is so great that you want to insert it directly into your own paper. And hey, that's fine, because Wikipedia information is free!

But here's the *first* problem you run into: the best Wikipedia content is oftentimes the most well-annotated content. Which means when you copy-and-paste, you're getting that great text plus all those bracketed numbers, which is not what your own research paper needs:

Giant panda

From Wikipedia, the free encyclopedia
(Redirected from [Panda](#))

This article is about the mammal in the bear family. For the arboreal mammal, see [red panda](#). For other uses, see [Panda \(disambiguation\)](#).

The **panda** (*Ailuropoda melanoleuca*, lit. "black and white cat-foot"),^[2] also known as the **giant panda** to distinguish it from the unrelated [red panda](#),^[3] is a [bear](#)^[4] native to central-western and south western [China](#).^[4] It is easily recognized by the large, distinctive black patches around its eyes, over the ears, and across its round body. Though it belongs to the order [Carnivora](#), the panda's diet is 99% [bamboo](#).^[5] Pandas in the wild will occasionally eat other grasses, wild tubers, or even meat in the form of birds, rodents or carrion. In captivity, they may receive honey, eggs, fish, yams, [shrub leaves](#), oranges, or bananas along with specially prepared food.^{[6][7]}



Giant Panda entry from Wikipedia

Here's the text you get when copying-and-pasting from the first paragraph of the [Giant Panda](#)²⁰ entry:

The panda (*Ailuropoda melanoleuca*, lit. "black and white cat-foot"),^[2] also known as the giant panda to distinguish it from the unrelated red panda, is a bear^[3] native to central-western and south western China.^[4] It is easily recognized by the large, distinctive black patches around its eyes, over the ears, and across its round body. Though it belongs to the order Carnivora, the panda's diet is 99% bamboo.^[5] Pandas in the wild will occasionally eat other grasses, wild tubers, or even meat in the form of birds, rodents or carrion. In captivity, they may receive honey, eggs, fish, yams, shrub leaves, oranges, or bananas along with specially prepared food.^{[6][7]}

Write the regex needed to remove these bracketed numbers.

²⁰http://en.wikipedia.org/wiki/Giant_panda

Answer

We're only dealing with numbers so use the shorthand notation `\d`. And since we're dealing with *literal* bracket characters, we need to escape them:

Find `\[\d+\]`

Replace (*with nothing*)

What happens if you forget to **escape** those brackets and instead try the following pattern?

Find `[\d+]`

The regex engine interprets that as: *match either a digit or a plus sign* (remember that `+` is non-special inside of brackets). The resulting text is:

The panda (*Ailuropoda melanoleuca*, lit. “black and white cat-foot”),[] also known as the giant panda to distinguish it from the unrelated red panda, is a bear[] native to central-western and south western China.[]

(**Note:** Of course, cleaning up the wiki-code is just one issue when copying straight from Wikipedia. The other is proper attribution so you don't come off as a plagiarist. But that's outside of the scope of this book.)

Character classes give us a great amount of flexibility in matching text. In the [next chapter]{#negative-character-sets}, we'll learn a slight variation: specifying the characters we *don't* want to match.

Negative character sets

Now that we know how to *include* sets of characters in our patterns, let's see how to *exclude* them.

The syntax is simple and easy to remember. But it opens up a new range of regex possibilities. So this chapter will re-emphasize the concepts we've covered so far while mixing in new tricks that come from thinking negatively.

With the **square-bracket** notation, we are able to specify a *set* or *range* of characters to match:

Find `[bcd]og`

Matches “bog” or “cog” or “dog”

But what if we want to match anything *but* those terms, such as “fog” or “hog”?

Negative character sets

To exclude a set of characters from a pattern, use the same **square bracket** notation as a normal character set, but use a **caret** symbol, `^`, directly after the left-bracket.

The following regex:

`[^abcd]`

– will match any character *except* a literal a, b, c, or d

This regex:

`[^0-5]`

– matches any character that is *not* the numbers 0 through 5.

Let's revisit a previous exercise in which we removed all the non-number/decimal characters:

\$1,200.00
 \$5,600.25
 \$100.09
 \$42,100.01

Using normal character sets, the answer was:

Find [\$,]

Replace (*with nothing*)

Using **negative character sets**, we can express the pattern as:

Find [^\d.]

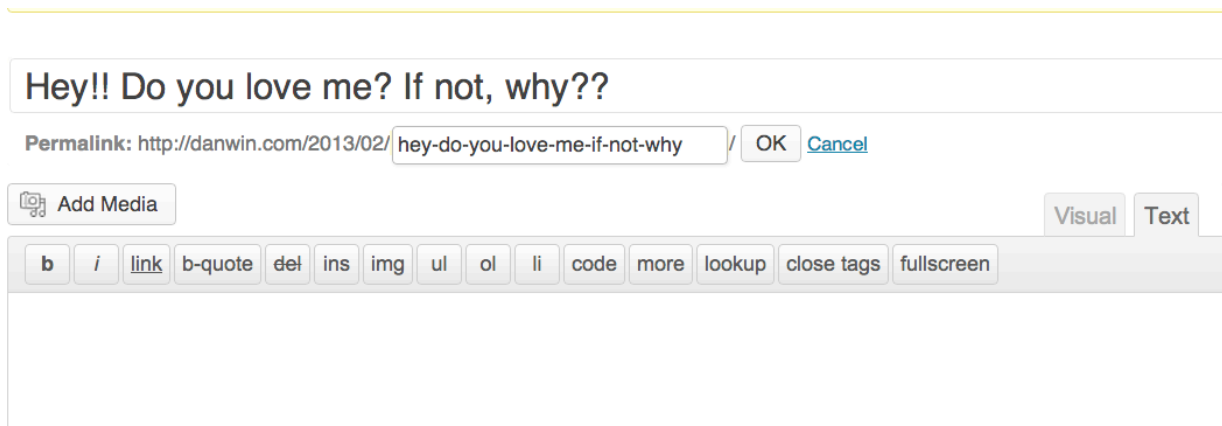
Replace (*with nothing*)

In English Match all the characters that are *not* a numerical digit or a dot.

Note: Remember that the **dot character** *is non-special* inside square brackets.

Exercise: URL slugs

If you've ever written a blog post, you may have used software that auto-generated the URL for the post from the *title*. Here's what it looks like in Wordpress:



Wordpress New Post screen

All the *non-word* characters, such as spaces and punctuation, are turned into dashes. So:

Hey!! Do you love me? If not, why??

Turns into:

hey-**do**-you-love-me-**if**-not-why

Write the regex needed to do this conversion.

Answer

There's at least two equivalent negative sets that can accomplish this:

- `[^A-Za-z0-9_]`
- `[^\w]`

You probably prefer the shorthand version:

Find `[^\w]+`

Replace -

You'll actually end up with this; we don't have a good way to eliminate that trailing slash or lowercase a phrase using regexes alone. But this is close enough:

Hey-Do-you-love-me-If-not-why-

Note: If you don't use the + in the pattern, then you won't collapse consecutive non-word-characters into one hyphen and will end up with:

Hey---Do-you-love-me--If-not--why--

More shorthand

We know that `\d` is shorthand for `[0-9]`. Negative character sets have their own shortcuts (in most of the major regex flavors that I know of, at least) and they're pretty easy to remember: take the shortcuts you already know and **capitalize the escaped character** to get the negative version:

All non-digits `\D`

All non-word-characters `\W`

All non-whitespace characters `\S`

The regex pattern for the previous exercise could also be expressed as simply as:

`\W+`

Exercise: More hecks In the previous chapter, we dealt with message-board-trolls who wanted to get around our ban of the word, heck. We assumed they would try such variations as h3ck or hekk. But using non-letter-characters, such as * or -, also serve to get the point across, e.g. h*ck and h-ck. And that's just too much implied civility for our tastes.

So, the problematic text:

Go heck yourself you dumb hacker. You don't know what the h-ck you are h-king doing, so go back to your H3cK-hole farm and go shuck some h.cking corn.

Answer This one's a little tricky. Your first instinct may be to do something like:

`h\W\Wk`

While this would match h**k and h--k, it wouldn't match h*ck or he*k. In fact, it wouldn't match heck. So we still need to match literal e, 3, c, and k characters, as well as all non-word characters. We can do this by combining square-bracket notation with negative shortcuts:

Find `h[e3\W][ck\W]k`

Replace ----

In English Match a word in which: the first character is h; the second character is either e, 3, or any non-word character; the third character is c, k, or any non-word character; the fourth character is k.

TK conclusion

Capture, Reuse

We've learned how to swiftly replace wide swaths of text.

But sometimes, the patterns we match contain text that we *don't* want to replace.

For example, given a list of names in *last-name-comma-first-name* format:

```
Smith, Shelly  
Perry, Dwayne
```

How do we change it to *first-name last-name*?

```
Shelly Smith  
Dwayne Perry
```

We still need to keep each part of the name in order to switch them around.

In this chapter, we'll learn how **parentheses** are used to “capture” a pattern that we can later refer to. This is one of the most important features of regular expressions, giving us the flexibility to re-arrange and reformat data to our needs.

Parentheses for precedence

It should come as no surprise that **parentheses**, (and), have special meaning in regex syntax, just as all the other braces (curly and square) do.

Their most basic function in a regex is to add **precedence**. If you remember arithmetic, then you'll recall how parentheses were used to change *order of operations*:

```
5 * 5 + 2    = 27  
5 * (5 + 2)  = 35
```

What kind of *order of operations* can we alter in regexes? Consider how the + operator affects the character that immediately *precedes* it:

Find ba+

Matches baaaaaaaaa

With the use of parentheses, we can affect a sequence of characters:

Find (ba)+

Matches babababa

Matches baaaaaaaaa

Note: In the text above, notice how it matches only ba and not any of the following consecutive a characters.

Exercise In the following text:

```
1 12 100 Apples 9 8 44
20 700 99 Oranges 2 98 2
```

Remove *only* the numbers (and the spaces between) that come *after* the names of the fruit:

```
1 12 100 Apples
20 700 99 Oranges
```

Answer

So the basic pattern we're looking for is: *any number of spaces, followed by any number of digits*. This can be expressed as:

`+ \d+`

(Note that there is a whitespace character before the first + operator)

However, there is more than one occurrence of this basic pattern. So we use parentheses to group the basic pattern:

``(+ \d+)``

And apply the + operator to that grouping:

``(+ \d+)+``

And for good measure, we're looking for this pattern at *end* of the line, so throw in the **end-of-the-line anchor**. Our final answer is:

Find (+ \d+)+\$

Replace (*with nothing*)

Parentheses for captured groups

Parentheses have what might be called a *major* side effect regarding the patterns they group together. When part of a regex is inside a parentheses, the regex engine contains a **backreference** to the pattern **captured** inside the parentheses.

Let's go back to the *last-name-first-name* example at the beginning of the chapter:

Smith, Shelly

Perry, Dwayne

What's the pattern?

We want to *capture* the words on both sides of the **comma**.

This is easy enough; the regex pattern is:

```
(\w+), (\w+)
```

Ah, but what do we **replace** it with?

Backreference notation

Up until now, the **Replace** part of the **Find-and-Replace** action hasn't involved many special characters. In fact, nearly none of the special characters in regex-land, such as +, [], ^, and so forth, have any meaning beyond their *literal* values.

But the **backslash** still wields its **escaping** power in the **Replace** field.



Backreference notation

Inside the **Replace** field, using a **backslash** followed by a number, will denote the value inside the corresponding captured group. In other words, \1 contains the value of the *first* captured group. And \2 contains the value for the *second* group, and so forth.

Let's consider this simple use of parentheses in a pattern:

```
(ab)(cd)(e)
```

If the **Replace** field is set to:

```
\1\2\3
```


– then the 1st backreference will contain the letters ab; the 2nd backreference contains cd, and the 3rd backreference contains e

Let's try out backreferences on the list of names:

Find `(\w+), (\w+)`

Replace `\2 \1`

This is what your text editor's **Find-and-Replace** dialog box should look like:

Todo:IMG

And the result:

Shelly Smith

Dwayne Perry

The explanation of the **Replace** field, in English, is: *the last and first names was contained in the first and second captured groups, respectively. So the replacement value is the second backreference, followed by the first backreference.*



Warning: Regex Differences Ahead

I promised you in the introduction that we would stay far away from the relatively minor differences between regex flavors. However, there is at least one nitpick we have to be aware of:

In the examples above, I said that the backreference notation involves a **backslash** and a number. *However*, some flavors of regex use a **dollar-sign** and a number.

Here's what it looks like in the TextMate editor: Todo: Image

Not a big difference, but obviously something you need to know, as you switch between different text editors and programming languages.

For the remainder of this book, we will assume the backreference operator is the **backslash**. You may have to substitute the \$ as appropriate.

Let's practice capturing groups on more complicated names. Convert the following list to this format: *first-name middle-initial last-name*

Roosevelt, Franklin D.
 Walker, Mary E.
 Berners-Lee, Tim J.
 Williams, Serena J.

The main variation here is that we need a *third* capturing group to contain the middle initial. Also, the names may contain hyphens.

Find ([\w-]+), ([\w-]+) ([A-Z].)

Replace \$2 \$3 \$1

A couple of notes:

1. The hyphen inside square brackets has to be escaped, because, the hyphen in that context is a special character that acts as a *range* (e.g. *from A to Z*, as we see in the third-capturing group).
2. The **dot** character – . – has to be escaped, because it is a special regex character. In fact, it is a very powerful character that we'll learn about in a [later chapter]{#dot-character}.

Correcting dates with capturing groups

In a previous chapter, we encountered the problem of trying to fix the year-value on a list of dates:

```
12/24/2012, $50.00
12/25/2012, $50.00
12/28/2012, $102.00
1/1/2012, $2012.00
1/2/2012, $32.00
1/6/2012, $52012.00
1/12/2012, $642.00
1/13/2012, $500.00
1/20/2013, $100.00
2/1/2013, $2012.20
```

If this list is chronological, then the data-entry person forgot to go from 2012 to 2013 when January rolled around.

Let's start with how we selectively target the January dates:

Find 1/\d+/2012

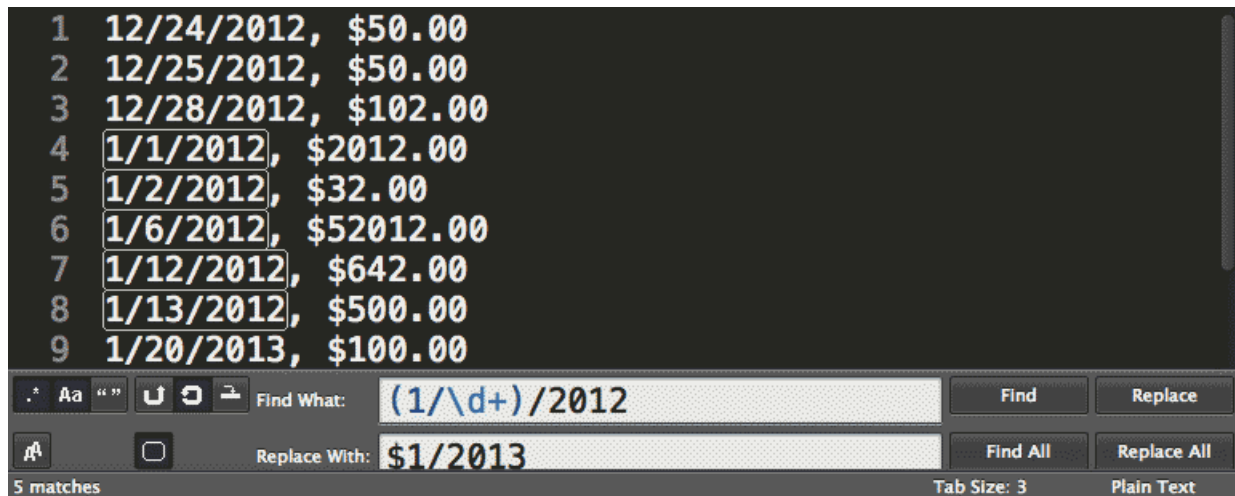
In English We are looking for a 1, followed by a forward-slash, followed by one-or-more numerical digits, and then another forward-slash and the sequence 2012.

Using a capturing group, we wrap the *non-year* part of the date in parentheses. And then we use the backreference \$1 to retrieve the value to use in the replacement:

Find (1/\d+)/2012

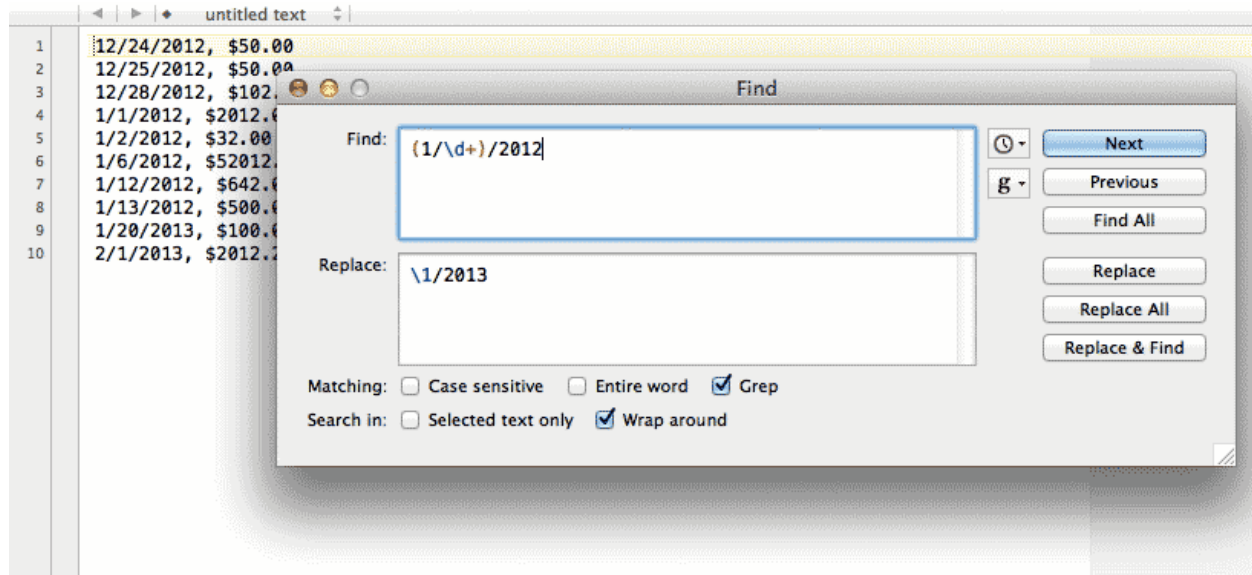
Replace \$1/2013

In Sublime Text 2, this is what the operation looks like:



Sublime Text 2

Remember that some variations of regex, including the one used in TextWrangler, the backreference involves a **backslash** instead of a dollar sign:



A backreference in TextWrangler's Find and Replace

Fixing misshapen dates is one of my most common daily uses for regexes. Dates usually contain an obvious pattern, e.g. 9/5/2010 and 4-7-99. To get them into a different format – e.g. 9/5/10 and 04/07/1999 – can often be accomplished with regex techniques.

Exercise: Change the order of date parts

In American day-to-day usage, dates are often formatted as *month-day-year*:

```
03-09-2001
11-14-2012
05-22-1978
```

However, programmers prefer dates to be in *year-month-day* format:

```
2001-03-09
2012-11-14
1978-05-22
```

Why? A common operation is to “find the earliest/latest date.” To sort date strings, the simplest method for a program is to do it an *alphabetical*-fashion. Think of how you sort words alphabetically:

Apples
Oracles
Oranges
Ore

The most important letter is the *left-most* one: A comes before O in the alphabet, so all words that begin with A will come before any word that starts with O. But what if the words start with the same letter? Then you compare the second letters, and so forth.

This is the same with dates. In the American human-readable format, 03-09-2001 would come *before* 05-22-1978, as 03 comes earlier in the numerical scale than does 05. But chronologically, of course, 05-22-1978 is several decades before 03-09-2001. In the programmer-friendly version, the “biggest” parts of the date come first, i.e. the year, *then* the month and day. This makes sorting easy, because the alphabetical-order of the date text is *the same as the chronological order* of the actual dates.

So, given the human readable version of the dates, write a regex that converts them to the programmer friendly version.

Answer

Find `(\d+)-(\d+)-(\d+)`

Replace `$3-$1-$2`

Exercise: Zero-pad the dates

The above exercise isn’t completely realistic, however, because the human-readable dates are usually written in a more shorthand way:

3-9-2001
11-14-2012
5-22-1978

The month of March, in other words, is written simply as 3, *not* as 03

Why is that extra zero there? If you think of the computer-sortable scenario, those extra zeros are vital. Consider the issue of two dates with the same year. In human readable format:

11-20-2002
3-20-2002

In machine-readable format:

2002-11-20

2002-3-20

If a program sorted the above dates, 2002-11-20 would come before 2002-3-20, because the left-most 1 is compared against the 3. In other words, the *tens* place of 11 is being compared to the *ones* place of 3, which is not what we want.

With “zero-padding”, though, this isn’t a problem:

2002-03-20

2002-11-20

Given the following dates (in human readable format):

1-12-1999

4-6-2002

3-3-1973

12-5-2004

Add zeroes where they are needed:

01-12-1999

04-06-2002

03-03-1973

12-05-2004

Answer

Your first instinct may be that this requires two regex **Find-and-Replace** operations. However, we can do it with one:

Find \b(\d)\b

Replace 0\$1

In English For every single-digit surrounded by word-boundaries, capture the digit. In the replacement text, add a 0 before the captured single digit.

Exercise: Pad the year

At the turn of the century, a lot of data systems recorded dates as:

05-14-89
12-03-98
03-15-99
02-01-00
07-07-01
03-08-02
01-22-12

The reasoning was that it's "obvious" that 98 refers to 1998 and 02 refers to 2002. Of course, it won't be so obvious as we get farther into the 21st century. So write a regex that converts the two-digit years into four-digits:

05-14-1989
12-03-1998
03-15-1999
02-01-2000
07-07-2001
03-08-2002
01-22-2012

Answer

This is a little bit tricky, not least of which because using a regex here is *not recommended in real life*. If you're working with a big dataset, you *might* have dates that include 1902 along with 2002. And so you don't want to use a dumb regex to deal with that ambiguity, you'll likely need to write a programming script with some logic and error-handling.

However, sometimes you just need a quick and dirty solution if you *know* your data doesn't contain those ambiguities. In this case, we'll need to use *two* regex operations, one each for pre-2000 and post-2000 dates.

For pre-2000:

Find ([89]\d)\$

Replace 19\$1

For post-2000:

Find ([01]\d)\$

Replace 20\$1

Exercise

```
"What in the world?" Mary asked.
"Hello!" James yelled.
"Goodbye!" Mary responded.
```

Re-arrange the lines so the speakers come *before* the dialogue.

```
Mary asked: "What in the world?"
James yelled: "Hello!"
Mary responded: "Goodbye!"
```

Answer

Let's break down the pattern into its component pieces:

1. We start with a **quotation mark**
2. We then want to capture all the text (the dialogue) *that is not a quotation mark*.
3. Then we want to capture all the text (e.g. "James yelled") until the sentence-ending **period**.

Note: Remember that the dot-character is not a special character when used inside brackets.

Find `"([^\"]+)" ([^\.]+)\.`

Replace \2: `"\1"`

The notation is a little confusing at first. In the first part of the sentence, we're representing the pattern with three quotation marks:

```
"([^\"]+)"
```

But read out the pattern to yourself: *we are capturing everything that is not a quotation mark between two quotation marks*.

The middle quotation mark is used to indicate, inside the negative character set, that we do *not* want a quotation mark in that context. We need that third quotation mark to set the bounds of the first captured group.

As for that awkward literal dot-character at the end, if we didn't include it in the pattern, it would not be *replaced*. Try leaving it out:

Find `"([^\"]+) ([^\.]+)`

Replace \2: `"\1"`

And you'll have an awkward trailing period in your lines:

Mary asked: "What in the world?".

James yelled: "Hello!".

Mary responded: "Goodbye!".

Using parentheses without capturing

What if we want to use parentheses just as a way to maintain order of operations?

To make a set of parentheses non-capturing, simply add a **question-mark** and **colon** immediately after the left-brace:

```
(?:this-pattern-isnt-captured)(this is captured)
```

Note: To re-emphasize, these symbols must show up *one after the other* in that *exact* order.

Ergo: (?:x) is not at all the same as: (? : x)

Given this text:

1. J.R. Ewing
2. A.C. Thompson
3. T.A.L. Glass

And wanting to change it to:

1. Ewing
2. Thompson
3. Smith

The pattern we seek to match is:

- A number followed by a period (captured)
- A series of uppercase letters, each followed by a period (but *not* captured)
- A string of word-characters (captured)

This is the Find-and-Replace:

Find (\d+.) ([A-Z].)+ (\w+)

Replace \1 \3

Note that we don't use the second backreference, which contains the initials (e.g. A.C., T.A.L.). It seems kind of wasteful to even store it if we don't need it. So we need a way to group a pattern we want to repeat – `[A-Z]\.` – *without capturing it*.

This is where we could use **non-capturing parentheses**:

Find `(\d+.) (?:[A-Z].)+ (\w+)`

Replace \1 \2

The only change is the addition of `?:` to the second-set of parentheses in the **Find** field. In the **Replace** field, there is no longer a 3rd backreference to use, as only two of the three parenthetical groups are capturing.

Capturing groups allow us to transform text in very powerful ways. And with great power, comes great potential to use them *badly*. We'll find out in later chapters that parentheses have several other uses, and so it behooves us to use capturing groups only when we need to, else our regex patterns become an incomprehensible tangle of parentheses.

This is a good time to read other information sources about regexes. [Regular-expressions.info](http://www.regular-expressions.info)²¹ in particular has a great summary of the nuances of captured groups.

²¹<http://www.regular-expressions.info/brackets.html>

Optionality and alternation

The two concepts in this chapter are actually more straightforward than what we've covered in past chapter. I like to think of them as ways to accommodate human error and inconsistency. Were phone numbers entered with or without dashes? Did an author use "J.F.K." or "JFK" to refer to John Fitzgerald Kennedy? Did your British and American writers use "grey" and "gray" throughout the text?

The optionality and alternation operators help us create patterns when we don't know for sure if certain characters exist in the text.

Alternation with the pipe character

The pipe character – | – is used to match *either* pattern on the left and right of it:

```
`a|b|c`
```

– will match either a, b, or c. This of course is equivalent to [abc]

However, the pipe character can be used to match groups of characters as well:

```
`happy|\d+`
```

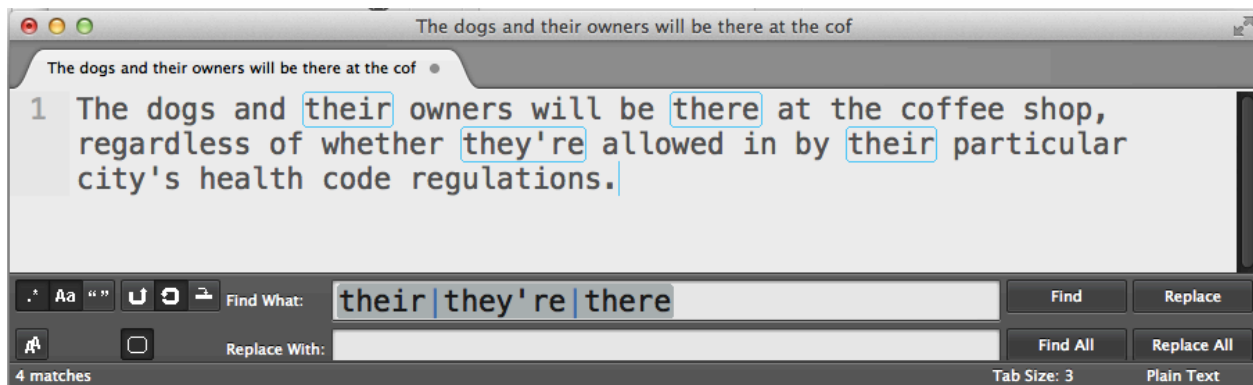
The above pattern would match either happy *or* a consecutive string of numbers.

Exercise: There, there

The simplest use of alternation can save you from having to run multiple Find operations. Considering that the uses for "their", "they're", and "there" are commonly mixed up, write a regex that would allow you to quickly locate and iterate through each of the homonyms.

Answer

Find `they're|their|there`



Matching homonyms in Sublime Text 2

Precedence with alternation

The alternation operator (the **pipe** character) has a low priority when it comes to priority of operations.

For example, the pattern `gr|ey` will match `gra` or `ey`. It would *not* match `grey` nor `gray`

To match `grey` or `gray`, we would have to use **parentheses** to group operations to our preferred order:

```
gr(a|e)y
```

Here, the “preferred” order is that the regex engine, when it reaches the third character in the pattern, looks for either an `a` or an `e`. Without the parentheses –

```
gr|ey
```

– the regex engine considers only alternating between the strings `gra` and `ey`

Exercise: Who is/are

In a particular document, you’ve noticed that the writer has consistently, but *incorrectly* used “who’s” and “who is” – for example:

She wants to know **who’s** the agents responsible for this.

Write a regex that will do a mass-replace of all instances of `who's` and `who are` – but use parentheses to create a pattern that is *not* simply, `who's|who is`

Answer

Find `who('s| is)`

Replace `who are`

Note: It's not necessary in this case, but it's a good habit to use **non-capturing parentheses** when you're not intending to capture a pattern:

```
`who(?:'s| is)`
```

Optionality with the question mark

We've learned how to match patterns even when we don't know exactly what letters or numbers they contain or how many times they occur.

But what if we're unsure if part of a pattern even exists?

The use of the `?` in a regex allows us to designate whatever pattern *precedes* it as being *optional*

`cats?`

Matches:

The **cat** is one of the **cats** The **cat**nip tastes like **catsup**

Another way to think of the question mark is that it matches *zero-or-one* occurrences.

Exercise: Match phone number input

Your web application has allowed users to enter in phone numbers in whatever format they feel like, including:

1-555-402-9800

(555)-951-8341

5551549763

1-(555)-091-5060

Write a regex using the **optionality operator** that can match any of the phone number examples above.

Answer

The solution pattern is ugly and convoluted looking. But if you take it one character at a time, from left to right, and simply put a **question mark** after each optional character, it's not at all tricky. Remember to **backslash-escape** the special characters.

Find `1?-? \(?\d{3}\)?-?\d{3}-?\d{4}`

Precedence with parentheses and optionality

As with most regex operators, the `?` only affects what directly precedes it, whether it's a single character or a complex pattern.

If what you make optional is more than just a single character, then use **parentheses** so that the entire group of characters is modified by the `?`

Given a list of dollar amounts:

\$200 \$19.20 \$610.42 \$1.5 \$15

The following pattern –

`\$\d+\.\d{2}?`

– will match only two of the listed amounts:

\$200 **\$19.20** **\$610.42** \$1.5 \$15

However, by using parentheses to make optional the literal decimal point *and* the two numerical digits, we can match all the listed amounts:

`\$\d+(\.\d{2})?`

And again, if you aren't intending to use parentheses to capture any part of the text, then you should use the non-capturing version:

`\$\d+(?:\.\d{2})?`

Which might lead you to ask: *how do you tell those question marks apart?*

For example, consider the pattern:

```
Studio(?:\d\d)?:
```

Which would match the following strings:

```
Studio 54:
```

```
Studio :
```

how does the regex engine know that the *second* question mark acts as the optional operator, but the *first* question mark acts as a way to make those parentheses non-capturing?

Think of it logically: *if* that *first* question mark made the left-side parentheses optional...how would that even make sense? The key difference is that the first question mark comes directly after a (non-literal) left-side parentheses. The engine just *knows* that you aren't intending to make it optional, because it just wouldn't make sense.

But even though it makes logical sense, it'll still be hard for us humans to read. If you're ever confused, just remember to take things character by character and you'll usually be able to suss it out.

Exercise: Mr. Ms. Mrs. Miss

Answer

```
Find M(r|r?s)\.?
```

```
Mr?r
```

Or:

```
Find who('|s)e?
```

```
its its they're there their your you're who's whose than then where wear
```

Exercise: There, there, redux

An alternative to using alternate-syntax. Use the optionality feature to match all these variations:

they're
theyre
their
there

Answer

Find `the[iry]'?[er]`

Note: while this is an amusing exercise, this regex is so unreadable that you're better off just using straight-up alternation:

`their|there|they'?re`

Exercise: Zip codes

In the United States, zip codes can come in at least two different forms: five-digit and nine-digit:

10020-9020
10018
60201-3277

In the following city listings, add commas to delimit the state, zip code, and country information.

Change:

New York 10020-9020 United States
California 90210 U.S.

To:

New York, 10020-9020, United States
California, 90210, U.S.

Answer

The optional part of the pattern is the hyphen and the four numerical digits:

Find `(\d{5})(-\d{4})?`

Replace `,\1\2,`

You might wonder what the second capturing group *captures* if nothing is actually there, as in the case of `California 90210 U.S.`, in which the optional pattern doesn't exist. For our purposes, nothing is captured in that second group and nothing is thus affected by it.

But again, you should be using non-capturing parentheses as possible. Here's an alternate solution:

Find `(\d{5}(?:-\d{4})?)`

Replace `,\1,`

If it hasn't hit you by now, regular expressions are *extremely* useful in finding terms and datapoints when you don't quite know what they look like. The alternation and optionality operators make it even more convenient to add flexibility to our searches.

Laziness and greediness

In this chapter, we'll take a closer look at how the + operator works. And we'll learn how to modify it so that it's not so "greedy."

Greediness

This book doesn't take a close look at the internals of the regular expression engine. But we can still glean some insight by observing how it behaves.

Take the + operator, often by default is considered to be **greedy**.

When we pair it with a literal character, such as a:

a+

We're telling the regex engine to essentially gobble up all consecutive a characters, as below:

The sheep goes, "Baaaaaaaa"

When paired with a non-literal character, such as a character class, the + operator's greediness can act as a convenient catch-all. Let's say we wanted to replace everything inside quotation marks, knowing that the text consists completely of **word characters** without any spaces or punctuation:

The sheep says "Cowabunga" to those who approach it.

Then the following regex does what we need:

Find "\w+"

Replace "Hello"

The regex engine starts its match when it finds the first quotation mark, then continues to the word character that immediately follows – C – and continues to capture each successive word character until it reaches the closing quotation mark.

But what happens if the quotation contains **non-word characters**?

The sheep says "Cowabunga!" to those who approach it.

In this situation, the regex engine finds the first quotation mark, continues to match Cowabunga, but then stops at !, because exclamation marks are not word characters.

In English, our regex pattern can be expressed as: *Match the text that consists of only word characters in between quotation marks*. Because Cowabunga! contains a non-word character, the regex engine quits before it reaches the closing quotation mark. In other words, the regex engine will fail to match anything and the replacement doesn't happen.

Being too greedy

In the previous chapter, we learned about the **dot** operator, which will match any kind of character. This seems like just what we need, right?

Given the following text:

The sheep says “Cowabunga!” to those who approach it.

Find `" . +"`

Replace `"Hello"`

The result is:

The sheep says “Hello” to those who approach it.

However, there’s a serious problem with this solution. Given a more complicated phrase:

When the sheep says “Cowabunga!”, the cow will reply, “Baa! Baa!”

– when we apply the previous solution, we end up with:

When the sheep says “Hello”

What happened? We got burned by **greediness**.

Let’s express the regex – `" . +"` – in English: *Match a quotation mark and then match every character until you reach another quotation mark.*

That seems straightforward. But consider the power of the **dot**: it matches *any* character, except for newline characters. And “any character” includes the quotation mark.

So in our example text:

When the sheep says “Cowabunga!”, the cow will reply, “Baa! Baa!”

There’s at least three ways that the regex engine can satisfy our pattern:

It could stop at the second quotation mark:

When the sheep says “Cowabunga!”, the cow will reply, “Baa! Baa!”

Or it could keep going until the *third* quotation mark:

When the sheep says “Cowabunga!”, the cow will reply, “Baa! Baa!”

Or it could just go for broke until the *fourth* quotation mark, after which there’s no other quotation marks to match:

When the sheep says “Cowabunga!”, the cow will reply, “Baa! Baa!”

All three matches satisfy our desired pattern: *A quotation mark, then any and all kind of characters, and then a quotation mark*. But by using the + operator, we’ve told the regex engine to grab as much of the text as possible, i.e. until the *fourth* and final quotation mark.

Hence, the term, **greediness**

Laziness

Now that we know what greediness is, **laziness** is pretty easy to understand: it’s the *opposite* of greediness.

Laziness is expressed by adding a **question mark** immediately after the + operator:

`".+?"`

Going back to our example text:

When the sheep says “Cowabunga!”, the cow will reply, “Baa! Baa!”

The regex engine will start matching at the first quotation mark. The `.+` syntax will match Cowabunga! – but when the engine reaches that second quotation mark, it effectively says, “Well, my job is done!”.

Note: The above description is a vast simplification of what’s going on under the hood. But for our purposes, it’s enough to understand that `.+?` is “reluctant” to deliver more than it needs to. In contrast, the `.+` regex is “eager” to match as much of the string as possible.

So doing a **Find-and-Replace-All** with the **laziness** operator has the desired effect:

Find `".+?"`

Replace "Hello"

Result:

When the sheep says "Hello", the cow will reply, "Hello"

Note: Just a reminder: when we use the text-editor's **Replace-All**, as opposed to just a single **Replace** action, it's the **text editor** that continues looking for matching pattern, not the regex engine. The regex engine is revved up twice to match both instances of the pattern. If we didn't use **Replace All**, the resulting text would be:

When the sheep says "Hello", the cow will reply, "Baa! Baa!"

Exercise: HTML to Markdown anchors

The **Markdown** language is a popular, alternative way to write HTML. It's not important to know HTML or Markdown for this exercise; all you need to know is what an anchor link looks like in both syntaxes:

In HTML:

```
<a href="http://example.com">Some link</a>
```

In Markdown, that same link is:

```
[Some link](http://example.com)
```

Markdown is popular because it's easier to read and write – in fact, this whole book is written in Markdown. However, web browsers don't interpret Markdown. Usually web developers write in Markdown and use a separate utility to transfer it to HTML (and back).

So, given:

```
Today, I built a <a href="http://example.com">website</a> which is now  
listed <a href="http://www.google.com">on various search engines</a>.
```

Write a regex to change it to Markdown syntax:

```
Today, I built a [website](http://example.com) which is now listed [on  
various search engines](http://www.google.com)
```

Answer

Find `(.*?)`

Replace `[\2](\1)`

Note: *This is a pretty loose answer that would work only on this particularly simple HTML example. For real-life HTML, it is generally not wise to try to parse a page with regexes only.*

Laziness and the star

The laziness operator can be used with the **star** operator for those cases when there are *zero-or-more* instances of a pattern.

Exercise: Messier HTML to Markdown anchors

Again, it's not wise to try to parse HTML with regexes alone, especially if the HTML comes from a variety of authors and styles.

As a simple example of how complicated HTML can get, here's what happens when anchor tag markup has attributes inside it:

```
Today, I built a <a href="http://example.com" style="color:green;">website</a>
which is now listed <a class="header remoteLink" href="http://www.google.com">on
various search engines</a>.
```

To convert to Markdown, all we care about is the `href` attribute. However, using our regex from the previous solution – `(.*?)` – nets us this incorrect conversion:

```
Today, I built a [website](http://example.com" style="color:green;) which
is now listed <a class="header remoteLink" href="http://www.google.com">on
various search engines</a>.
```

Use the **star** and laziness operator to convert this more-complicated HTML into Markdown.

Answer

The trick here is that the non-`href` attributes can come *before* or *after* the `href` attribute:

- ``
- ``

With the **star** operator, we can repeat a catch-all pattern for either situation. This pattern – `. * ?` – need not be capturing, since all we care about is the `href` attribute:

Find `<a.*?href="(.*?)".*?>(.*?)`

Replace `[\2](\1)`

Note: While the regex patterns we’ve learned are very flexible, do not think that there’s a masterful regex that deals with all the edge cases floating out on the Web. The exercises above are just meant to be simple examples and they *may* work for simple real life scenarios, such as when all the HTML you’re dealing with comes from one source.

If you’re wondering, “Well, then how *are* we supposed to parse HTML?” That’s something you want to learn a little programming and scripting for. Libraries such as [Nokogiri for Ruby](http://nokogiri.org/)²² and [Beautiful Soup for Python](http://www.crummy.com/software/BeautifulSoup/)²³ handle the heavy lifting of correctly parsing HTML. (If you’re wondering, “Why parse HTML in the *first place*?” Web scraping – the automatic extraction of data and useful bits from website – is a common use case.)

The difference between laziness and greediness is subtle, but it’s important to understand it when you’re trying to do a catch-all pattern that you’d prefer *not* catch *everything*. In fact, the lazy version of a regex will generally be more useful than the greedy version when it comes to real life dirty text processing.

²²<http://nokogiri.org/>

²³<http://www.crummy.com/software/BeautifulSoup/>

Lookarounds

This chapter covers **lookarounds**, a regex technique that allows us to *test* if a pattern exists without actually capturing it. The syntax can be a little confusing, and it's possible to get by without lookarounds. But they do allow for some more advanced patterns which I'll cover later in this chapter.

Positive lookahead

A positive lookahead is denoted with a **question-mark-and-equals** sign inside of **parentheses**:

```
cat(?=s)
```

– will match an instance of `cat` that is *immediately followed* by an `s` character.

The `cat` is outside while the other `cats` are inside

However, for the purposes of **Find-and-Replace**, only `cat` will be matched and replaced.

In the given text:

How much wood can a woodchuck chuck if woodchuck could chuck wood

This **Find-and-Replace** pattern:

Find `wood(?=chuck)`

Replace `stone`

Results in: > How much wood can a stonechuck chuck if stonechuck could chuck wood

Exercise

Delete the commas that are used as number delimiters below:

Original:

1,000 dachshunds, of the brown-colored variety, ran 12,000 laps.

Fixed:

1000 dachshunds, of the brown-colored variety, ran 12000 laps

Answer

Find `,(?=\d+)`

Replace *(with nothing)*

Notice how the replacement didn't affect the numbers matched with `\d+`. The positive lookahead only verifies that those numbers exist ahead of a comma; the regex engine only cares about replacing that comma.

Compare that to having to use capturing groups:

Find `,(\d+)`

Replace `$1`

Negative lookahead

A **negative lookahead** works the same as the **positive** variation, except that it looks for the specified pattern to *not exist*. Instead of an equals sign, we use an **exclamation mark**:

`cat(!=s)`

– will match an instance of `cat` that is *not immediately followed* by an `s` character.

In the given text:

How much wood can a woodchuck chuck if woodchuck could chuck wood

This **Find-and-Replace** pattern:

Find `wood(?!chuck)`

Replace `stone`

Results in: > How much stone can a woodchuck chuck if woodchuck could chuck stone

Exercise

Replace the commas that are used to separate sentence clauses with double-hyphens. Do not replace the commas used as number-delimiters:

Original:

1,000 dachshunds, of the brown-colored variety, ran 12,000 laps.

Fixed:

1000 dachshunds – of the brown-colored variety – ran 12000 laps

Answer

Find `,(?!\d+)`

Replace `--`

Positive lookbehind

The **lookbehind** is what you expect: match a pattern *only if* a given pattern *does not immediately precede it*. The following pattern matches an `s` character that is preceded by `cat`:

``(?<=cat)s``

Given this phrase:

How much wood can a woodchuck chuck if woodchuck could chuck wood

The following **Find-and-Replace**:

Find `(?<=wood)chuck`

Replace `duck`

Results in: > How much wood can a woodduck chuck if woodduck could chuck wood.

The limits of lookbehinds

The main caveat with lookbehinds is that are not supported across all regex flavors. And even for the regex flavors that *do* support lookbehinds, they only support a limited set of regex syntax.

Essentially, you may run into problems with lookbehinds that attempt to match a pattern of **variable length**.

This is generally kosher `(?<=\d),`

This likely will *not* work `(?<=\d+)`

The difference is that the first pattern is just **one digit**. The latter could be one digit or a thousand. The regex engine can't handle that variability. [Regular-expressions.info has an excellent explainer](http://www.regular-expressions.info/lookaround.html)²⁴.

In other words, use lookbehinds with caution. We'll examine some alternatives at the end of this chapter.

Exercise

Given the following date listings:

```
3/12/2010
4/6/2001
11/17/2009
```

Replace the four-digit years with two-digits:

```
3/12/2010
4/6/2001
11/17/2009
```

Answer

Find `(?<=/)\\d{2}(?=\\d{2})`

Replace *(with nothing)*

Negative lookbehind

Same concept and limits as the **positive** variation, with the difference that we're verifying the non-existence of a preceding pattern. The syntax is similar except an **exclamation mark** takes the place of the **equals sign**.

The following pattern matches an `s` character that is *not* preceded by `cat`:

²⁴<http://www.regular-expressions.info/lookaround.html>

```
`(?<!cat)s`
```

In the given text:

How much wood can a woodchuck chuck if woodchuck could chuck wood

The following **Find-and-Replace**:

Find `(?<!wood)chuck`

Replace `shuck`

Results in:

How much wood can a woodchuck shuck if woodchuck could shuck wood

Exercise

Given the following text:

Charles spent 20.5 at the chatusserie. He then withdrew 40.5 from the ATM.

Replace the **periods** used as decimal points with commas:

Charles spent 20,5 at the chatusserie. He then withdrew 40,5 from the ATM.

Answer

Find `(?<![^\d])\.`

Replace `,`

The importance of zero-width (TODO)

This section will contain some advanced lookaround examples

Regexes in Real Life

From Text to Data to Visualization (TODO)

If you work with data and believe data is something that comes from a spreadsheet or database, then this is where regular expressions can become an indispensable.

Why learn Excel?

But *why* data? Why spreadsheets?

(todo)

The limits of Excel (todo)

Delimitation

Comma-separated values (CSV)

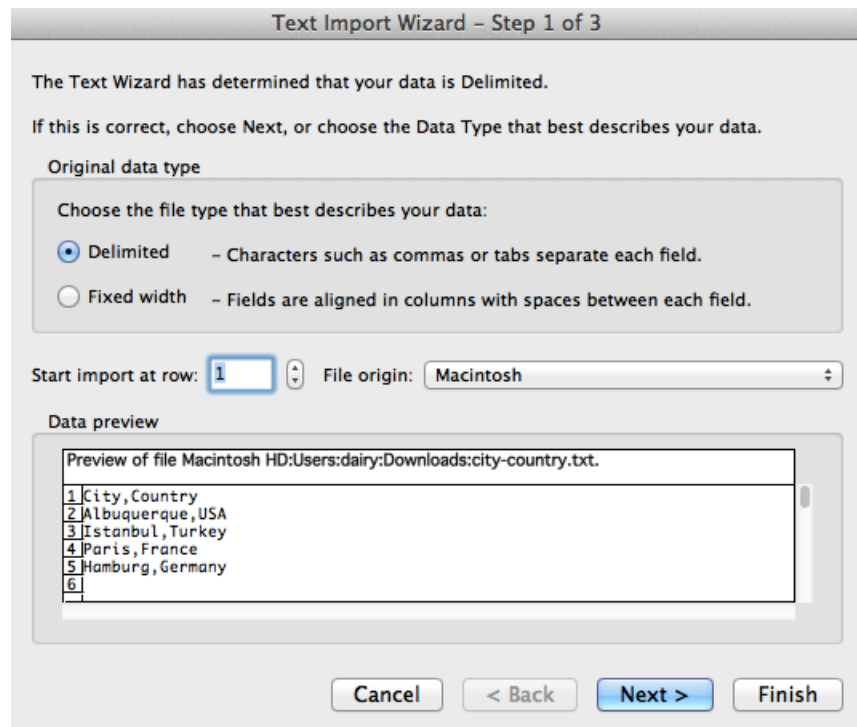
CSV files use commas to separate the data fields. Thus, when you open a CSV file using Excel, Excel uses commas to determine where the columns are.

Note: *If you don't have Excel, you may be able to follow along with another spreadsheet program, like Google Drive. However, the point of this next step is a trivial demonstration, because the point of this chapter is to show you how to manage data _without_ Excel. So all we're doing here is just a point-and-click exercise.*

1. Save the following text as a text file; you can use .txt as the file extension:

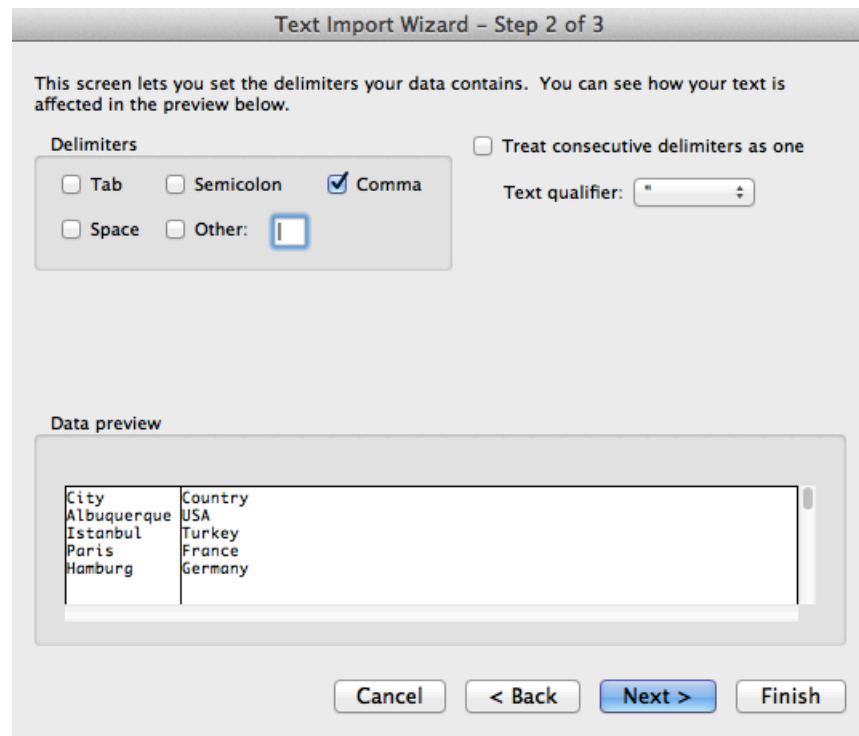
City,Country
Albuquerque,USA
Istanbul,Turkey
Paris,France
Hamburg,Germany

2. Modern spreadsheet programs will surmise that the text file is delimited. Excel, for example, will pop-up its **Text Import Wizard**:



Opening a .txt file in Excel 2011

3. Although it's pretty obvious here, Excel then asks you to tell it what delimiter character it should use. By choosing **Comma**, we can see a preview of how the columns will be arranged:



Selecting a delimiter in Excel's Text Import Wizard

4. After that, we have a text file in spreadsheet-manipulable format:

	A	B	C	D	E
1	City	Country			
2	Albuquerque	USA			
3	Istanbul	Turkey			
4	Paris	France			
5	Hamburg	Germany			
6					
7					
8					

The text as a spreadsheet

Exercise: Make CSV data from an address list

So turning CSV-delimited text to spreadsheet data is easy. but what if we want to get turn non-delimited text into a spreadsheet? We would have to convert that text into a delimited format. And this is where regexes come in.

Given this list of cities and postal codes: TODO New York, NY 10006

Convert it to this CSV format:

TODO

“Hey,” you might say, “there’s *already* a comma in that data.” True, but it’s just typical punctuation. If we were to open this list in Excel, we would end up with:

TODO:

So we need to use a simple regex to at least separate the state from the zipcode.

Answer

Find `(.+?), ([A-Z]{2}) (\d{5})`

Replace `\1,\2,\3`

Exercise: More complex addresses

Believe it or not, the easily-fixed scenario above is one that I’ve seen keep people from making perfectly usable, explorable data out of text.

However, for most kinds of text lists, the cleanup is a little more sophisticated than one extra comma. Here’s an example in which we have to deal with street names and addresses:

```
50 Fifth Ave. New York, NY 10012
100 Ninth Ave. Brooklyn, NY 11416
9 Houston St. Juneau, AK 99999
2800 Springfield Rd. Omaha, NE 55555
```

Change to:

```
50,Fifth Ave.,New York,NY,10012
100,Ninth Ave.,Brooklyn,NY,11416
9,Houston St.,Juneau,AK,99999
2800,Springfield Rd.,Omaha,NE,55555
```

Answer

This is simply breaking each part of the line into its own separate pattern:

1. Street number: consecutive digits at the beginning of the line
2. Street name: A combination of word characters and spaces until a literal period is reached.

3. City: A combination of word characters (actually, just letters) and spaces until a comma is reached.
4. State: Two uppercase letters
5. Zip: Five consecutive digits

Find `^(\\d+) ([\\w]+\\.) ([\\w]+), ([A-Z]{2}) (\\d{5})`

Replace `\\1,\\2,\\3,\\4,\\5`

Exercise: Complicated street names Street address lists can get way more complicated than this, of course. The following exercise tests how well you understand the differences between [laziness and greediness][#laziness].

Note: Don't fret if you don't get this. Fully grokking this kind of exercise requires better understanding of what's going on under the hood, which is what I've avoided presenting so far. Regular-expressions.info has a [great lesson on the internals](#)^a.

^aTK

What if our list of **street names** had periods *within* them?

100 J.D. Salinger Ave. City, ST 99999

42 J.F.K. Blvd. New York, NY 10555

Then the pattern for a street name would consist more of just word characters and spaces *until* a literal period.

So this is the pattern we have to alter:

`([\\w]+\\.)`

Now a simple solution may be just to include the literal dot inside the character set, like so:

`([\\w .]+)`

Find `^(\\d+) ([\\w .]+) ([\\w]+), ([A-Z]{2}) (\\d{5})`

Replace `\\1\\2\\3\\4\\5`

And that kind of works:

```
100,J.D. Salinger Ave.,City,ST,99999
42,J.F.K. Blvd. New,York,NY,10555
```

But notice the improper delimitation in the second line. The street name – J.F.K. Blvd. New – includes part of the city name – the New from New York.

This happens in any case where the city name consists of more than one word:

```
50 Fifth Ave. New York, NY 10012
```

Becomes:

```
50,Fifth Ave. New,York,NY,10012
```

Instead of what we had before:

```
50,Fifth Ave.,New York,NY,10012
```

Why did this happen? The subpattern `[\w .]+` was just *greedy*. We need to make it *lazier* so that the street name field doesn't unintentionally swallow part of the city name.

TODOTK: (move to laziness chapter?)

Answer The pattern for the street name is now:

```
([\w .]+?)
```

And the complete pattern is otherwise unchanged:

Find `^(\d+) ([\w .]+?) ([\w]+), ([A-Z]{2}) (\d{5})`

How did one question mark make all the difference?

Mixed commas and other delimiters

Again, just to hammer home the point: data is just *text*, with structure. Why does that structure have to be defined with commas? It *doesn't*, so good for you for realizing that.

We can basically use *any* symbol to structure our data. Tab-separated values, a.k.a. **TSV**, is another popular format. In fact, when you copy and paste from a HTML table, such as this Wikipedia HTML chart, you'll get:

TKTK

And most modern spreadsheet programs will automatically parse pasted TSV text into columns. Copy-and-pasting from the above text will get you this in Google Docs:

TKTK

Heck, you can just copy-and-paste directly from the webpage into the spreadsheet:

TKTK

Collisions

The reason why most data-providers *don't* use just “any” symbol to delimit data, though, is a practical one. What happens if you use the letter a as a delimiter – nut your data includes lots of a characters naturally?

You *can* do it, but it's not pretty.

But we don't have to dream of that scenario, we already have that problem with using comma delimiters. Consider this example list:

```
6,300 Apples from New York, NY $15,230
4,200 Oranges from Miami, FL $20,112
```

There's commas in the actual data, because they're used as a grammatical convention: 6000, for example, is 6,300.

In this case, we *don't* want to use commas as a delimiter. The pipe character, |, is a good candidate because it doesn't typically appear in this kind of list.

We can delimit this list by using this pattern:

Find `^([\d,]+) (\w+) from ([\w]+), ([A-Z]{2}) (\$[\d,]+)`

Replace `\1|\2|\3|\4|\5`

And we end up with:

```
6,300|Apples|New York|NY|$15,230
4,200|Oranges|Miami|FL|$20,112
```

Exercise: Someone else's comma-mess

Let's pretend that someone less enlightened than us tried to do the above exercise with comma-delimiters. They would end up with:

```
6,300,Apples,New York,NY,$15,230
4,200,Oranges,Miami,FL,$20,112
```

Which, when you open in Excel as CSV, looks predictably like nonsense:

	A	B	C	D	E	F	G	H
1	6 300	Apples	New York	NY		\$15	230	
2	4 200	Oranges	Miami	FL		\$20	112	
3								
4								

The result of too many commas in this CSV file

So we need to fix this mess by converting *only* the commas meant as delimiters into pipe symbols (or a delimiting character of your choice *ndash; the @ or tab character would work in this case).

Answer

Well, we obviously can't just do a simple **Find-and-Replace** affecting all commas. We need to affect only *some* of the commas.

Which ones? In this exercise, it's easier to look at the commas we *don't* want to replace:

```
6,300
$15,230
4,200
$20,112
```

So if the comma is followed by a number, we *don't* want to replace it.

There's multiple ways to do this, here's how to do it with **capturing groups** and a **negative character set**:

Find ,([\D])

Replace | \1

In English Replace all instances of commas followed by a *non-number* character (and *capture* that character) and replace them with a pipe character and that non-numbered character.

(**Note:** Remember that `\D` is a shorthand equivalent to either `[^\d]` or `[^0-9]`, though some flavors of regex may not support it.)

Answer: Using lookarounds

The more efficient way would be to use a lookahead, though, to avoid needing a backreference. Here's how to do it with a **negative lookahead**:

Find `, (?!\d)`

Replace `|`

In English Replace all commas – the ones *not* followed by a number – with a pipe character.

But you can use a **positive lookahead** too – if you combine it with a negative character set:

Find `, (?=\D)`

Replace `|`

In English Replace all commas – the ones that *are* followed by a *non-number* character – with a pipe character.

Whatever solution you use, you'll end up with:

6,300|Apples|New York|NY|\$15,230

4,200|Oranges|Miami|FL|\$20,112

Dealing with text charts (todo)

Completely unstructured text (todo)

<http://www.springsgov.com/units/police/policeblotter.asp?offset=0>

(Colorado Springs patrol reports)

```
^(\\d+) Record ID (\\w+ \\d{1,2}, \\d{4}) Incident Date (\\d{1,2}:\\d{1,2}:\\d{1,2}\\
} *\\wM) Time (\\.+? Shift [IV]+)Division (\\.+?)Title(\\.+?)Location((?:\\.|\\s|\\n)+\\
?)Summary(\\.+?)Adults\\s*Arrested ([\\w .\\-']*?)PD.+\\n.+
```

```
\\1\\n\\2\\n\\3\\n\\4\\n\\5\\n\\6\\n\\7\\n\\8\\n\\9
```

Exercise: Email headers

From: Sarah Palin <mailto:spalin@alaska.gov> To: John McCain <mailto:jmccain@mccain08.com>
 Subject: Becoming VP Date: TK

Answer

Find From: (\\.+?) <(\\.+?)>

Find To: (\\.+?) <(\\.+?)>

Find Subject: (\\.+)

(full answer TK)

Moving in and out and into Excel

TODO

Exercise: Wordiness of Hamlet

1. Break apart “Hamlet” by line per speaker
2. Import into Excel
3. TODO

Step 1. Remove all non dialogue lines

Let’s manually remove everything, including player listings, from the first line to line 55:

SCENE.- Elsinore.

a. All lines that are flush (134 lines) Examples:

<<THIS ELECTRONIC VERSION OF THE COMPLETE WORKS OF WILLIAM SHAKESPEARE IS COPYRIGHT 1990-1993 BY WORLD LIBRARY, INC., AND IS ACT III. Scene I. Elsinore. A room in the Castle. Enter King, Queen, Polonius, Ophelia, Rosencrantz, Guildenstern, and Lords. THE END

Find `^[^\s].+$$\n?`

Replace *with nothing*

b. Replace all stage directions (98 lines) Right justified text

Examples:

Enter Rosencrantz and Guildenstern.
 Exeunt [all but the Captain].
 Enter Sailors.
 Throws up [another skull].
 Exeunt marching; after the which a peal of ordnance
 are shot off.

This is tricky. We do not want this:

Ham. Why,

'As by lot, God wot,'

and then, you know,

'It came to pass, as most like it was.'

Find `^\s{15,}.+?\. *$$\n`

Replace *with nothing*

c. Remove all stage exits in dialogue (38 examples)

Example Adieu, adieu, adieu! Remember me. **Exit.**

Find `^(.{10,}) {5,}.`

Replace `\1`

d. Remove all asides (in brackets) (55 occurrences):

Example Ham. [aside] A little more than kin, and less than kind!

Find `\[.+\?\\]`

Replace *with nothing*

Concatenate dialouge

1. Play. What speech, my good lord?

Ham. I heard thee speak me a speech once, but it was never acted;
or **if** it was, not above once; **for** the play, I remember, pleas'd

Find `^ {2}(\w{1,4}\.(?: \w{1,4}\.)?) +((?:.\|\\n)+?)(?=\n^ {2}\w)`

Replace :

Remove all newlines inside speech:

Find `\\n(?:!")`

Collapse consecutive whitespace

Find `\\s{2,}`

Replace `\\s`

Exercise: Example FAA Control towers (TODO)

Step 1. Clean the data When you select-all, copy, and paste, you get this jumble:

```
FAA Contract Tower Closure List
(149 FCTs)
302202013
LOC
ID Facility Name City State
DHN DOTHAN RGNL DOTHAN AL
TCL TUSCALOOSA RGNL TUSCALOOSA AL
FYV DRAKE FIELD FAYETTEVILLE AR
TXK TEXARKANA RGNL-WEBB FIELD TEXARKANA AR
GEU GLENDALE MUNI GLENDALE AZ
```



```

...
Page 1 of 4FAA Contract Tower Closure List
(149 FCTs)
302202013
LOC
ID Facility Name City State
PIH POCATELLO RGNL POCATELLO ID
SUN FRIEDMAN MEMORIAL HAILEY ID

```

The first step is to remove all the non-data lines. The easiest way to do this is to first consider: *what are the data lines here?*

The data lines we want to keep have four fields: A 3-letter airport code, the airport's name, the city, and the two-letter state code.

So, the non-data lines are anything that: 1. *don't* begin with three capital letters, and 2. *don't* end with a two-letter state code.

(TODO)

Find `^[A-Z]`

`([A-Z]{3})+(.+?) {3,}(.+?) {3,}([A-Z]{2})`

`$1\t$2\t$3\t$4`

`http://www.faa.gov/news/media/fct_closed.pdf`

Delete useless lines

`^(?:\s.+|\s*)\n`

Change to location: `(.+?)\t([A-Z]{2})$ (city, state)`

From Data to HTML (TODO)

This chapter examines some hacks to convert delimited data into webpages. There's no new technique here, just more examples of the versatility of regular expressions as well as the importance of properly-delimited data.

HTML expertise is not required. At the heart of it, converting data into HTML is simply a visualization exercise.

Simple HTML tricks

Regexes are actually not ideal for dealing for *any* kind of HTML. But in simple situations, where you need to a quick specific conversion, a regex can be real time saver compared to doing it the old-fashioned way or writing a short program.

Plaintext to paragraphs

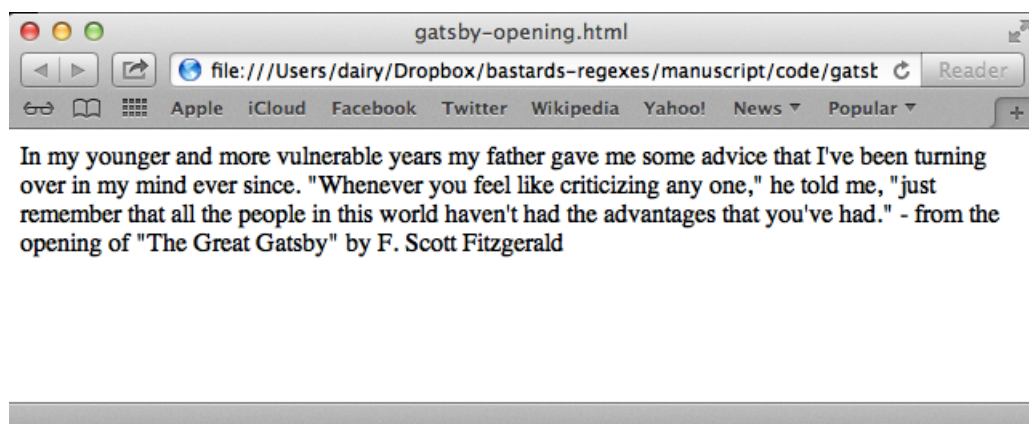
HTML doesn't respect whitespace. That is, if you create a HTML file that contains well-formatted plain text, like:

In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.

"Whenever you feel like criticizing any one," he told me, "just remember that all the people in this world haven't had the advantages that you've had."

- from the opening of "The Great Gatsby" by F. Scott Fitzgerald

Not only will the webpage contain only text, but the text won't even be properly formatted:



Unstructured text in a web browser

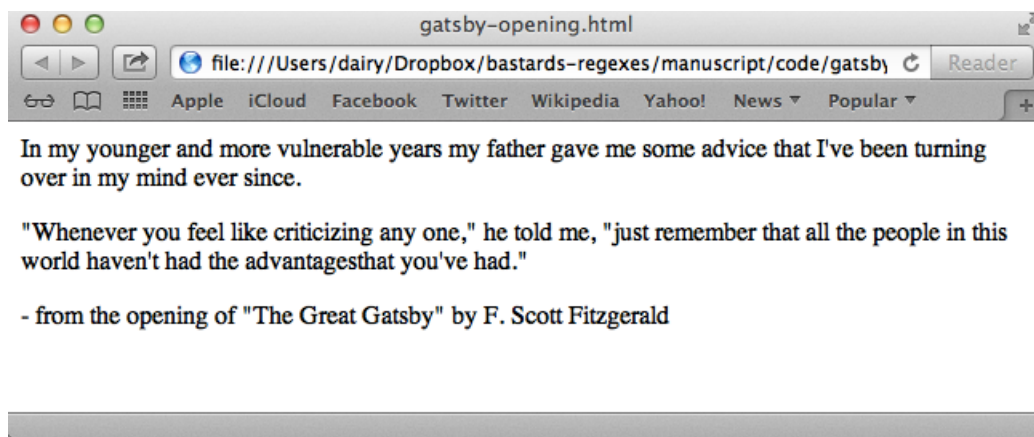
All whitespace characters, including line breaks, are rendered as simple spaces. And consecutive whitespace is rendered as a single space.

To make line paragraphs, we have to wrap each paragraph with `<p>` tags:

If each paragraph in the text file is a single line, then regex will create proper paragraphs for the browser to display:

Find `(. +)`

Replace `<p>\1</p>`



Paragraph text in a web browser

However, if there's a physical line break (rather than just the effect of word-wrap) for each line, the above regex will result in this:

The regex will have to be a little more complicated. One possible solution is:

In English Look for a sequence of *any* character, including newlines, until we reach a line that contains only whitespace (or nothing)

Find `((?:.|\n)+?)^*(?:\n)`

Replace `<p>\1</p>`

The use of non-capturing groups makes the solution a lot more complicated looking than you'd think. And it's not a bulletproof one either, but it'll work in a pinch.

Turning URLs into anchor links

Given a pipe-delimited text list of URLs and the text that describes the links, generate an HTML list of hyperlinks (a.k.a. anchor tags).

Turn this: An example site|http://example.com A search engine|http://google.com A news site|http://cnn.com
Into:

Find `(.+?)(.+)`

Replace

Stripping HTML

A common chore is removing all HTML from a document.

If you assume that all HTML markup is enclosed within opening and closing angle brackets – < and > – then this is pretty easy:

Given this:

```
<!doctype html>
```

We can use the dot character to match everything from a left-angle-bracket to a right-angle-bracket:

Find `<.+?>`

Replace *(with nothing)*

The result is imperfect: lots of unnecessary white space, plus the meta tag (the **title**, in this case), is left behind. But again, sometimes you just need a quick fix to get raw text. This is about as fast as highlighting from the web browser and copying-and-pasting to a text file:

Example Domain

Example Domain

This domain is established to be used **for** illustrative examples in documents. You **do** not need to coordinate or ask **for** permission to use this domain in examples, and it is not available **for** registration.

More information...

Stripping style from tags

Microsoft Word comes with a feature to turn a .doc file into .html. However, the translation is imperfect for a lot of real-life use cases.

For example, Word will specify the *style* of each element on the page, such as the font-color and size. When you're working with someone who had to type their blog post in Word, but you need it to have the styles of the blog, you have to remove the Word-inserted style.

Here's an example of Word-generated HTML:

(todo)

Tabular data to HTML tables

Turning an Excel spreadsheet to HTML tables

Mocking full web pages from data

Visualizations

Simple transformations

The Exercises

I've never been one to to expand my mind. And regular expressions, in my opinion, are just about the least fun thing you could learn "just for the fun of it".

To me, the "fun" part is using them. OK, maybe "fun" isn't the word...the "satisfying" part of regexes is using them to avoid really, really tedious work. And in some cases, to enable fun outputs.

So if you skimmed through the regex syntax lessons, go ahead and browse the stand alone exercises. Some of these scenarios are contrived, while some are variations of real-world data journalism problems I've run into.

Data Cleaning with the Stars

Anyone can do smart analyses when the data is nicely organized and delivered in a spreadsheet. But when it's just straight text, that's where regexes separate the resourceful from the poor chumps who try to clean up the text by hand, or give up all together.

This chapter contains a few examples of dirty data and how regexes can get them ready for the spreadsheet.

Normalized alphabetical titles

Even alphabetical-sorting can be a pain, if you're dealing with titles. The definite and indefinite articles – “the” and “a” or “an”, respectively – will give us a disproportionate number of titles in the “A” and “T” part of the alphabet:

A
A BEAUTIFUL MIND
A FISTFUL OF DOLLARS
AN AMERICAN IN PARIS
STAR TREK II: THE WRATH OF KHAN
THE APARTMENT
THE BIG LEBOWSKI
THE GODFATHER
THE GOOD, THE BAD, AND THE UGLY
THE GREAT GATSBY
THE KING AND I
THE LORD OF THE RINGS: THE RETURN OF THE KING
THE SHAWSHANK REDEMPTION
THE WRESTLER

In order for us to sort these titles alphabetically, we need to remove [definite and indefinite articles](#)²⁵ from the titles, e.g. “THE” and “A/AN”.

A simple find-and-delete for the word THE won't work:

LORD OF RINGS: RETURN OF KING

We need to remove the articles *only* if they appear at the *beginning of the title*. And, moreover, we need to *append them to the end of the title*, with a comma:

²⁵[http://en.wikipedia.org/wiki/Article_\(grammar\)](http://en.wikipedia.org/wiki/Article_(grammar))

LORD OF THE RINGS: THE RETURN OF THE KING, THE

Let's break this into two steps; first, the article: Use the beginning-of-line anchor and the alternation operator to capture just the "THE"s at the start of a movie's title:

```
^(THE|AN?)
```

Then the rest of the title can be captured with the dot operator:

```
(. +)
```

All together:

Find `^(THE|AN?) (. +)`

Replace `\2, \1`

This results in a list of titles that can be sorted alphabetically:

```
A
AMERICAN IN PARIS, AN
APARTMENT, THE
BEAUTIFUL MIND, A
BIG LEBOWSKI, THE
FISTFUL OF DOLLARS, A
GODFATHER, THE
GOOD, THE BAD, AND THE UGLY, THE
GREAT GATSBY, THE
KING AND I, THE
LORD OF THE RINGS: THE RETURN OF THE KING, THE
STAR TREK II: THE WRATH OF KHAN
```

As a bonus, notice that our regex doesn't screw up the movie titled *A* by converting it to *, A*. Our regex requires that at least one space and a non-newline character exist on the same line.

Make your own delimiters

Sometimes interesting data comes to you in clumps. Here's a sample of [movies with IMDb user ratings](http://www.imdb.com/chart/top)²⁶:

²⁶<http://www.imdb.com/chart/top>

The Godfather: Part II R 9 1974
 Pulp Fiction R 8.9 1994
 8½ NR 8.1 1963
 The Good, the Bad and the Ugly R 8.9 1966
 12 Angry Men PG 8.9 1957
 The Dark Knight PG-13 8.9 2008
 1984 R 7.1 1984
 M NR 8.5 1931
 Nosferatu U 8 1922
 Schindler's List R 8.9 1993
 Midnight Cowboy X 8 1969
 Fight Club R 8.8 1999

It'd be nice if we could do something like graph the correlation between user rating and when a movie was made, or what MPAA rating it got. But we have to first separate those values into their own fields.

But it's not trivial. The MPAA ratings can contain anywhere from one (e.g. "R") to five characters (e.g. "PG-13"). The user ratings can be integers or include a decimal point. And finally, the titles can be as short as one letter or no letters at all.

One way to approach this is to **work backwards**, because the most reliable part of the pattern is the *year* the movie was produced. The pattern to capture that is simply:

`\d{4}$`

The next part is *IMDb user rating*. At the very least it contains one digit. And it may optionally include a decimal point and a second digit. Such a pattern could be expressed as:

`\d\.\d?`

This seems a little loose to me, but I suppose we also have to prepare for the unlikely case that a movie has a perfect 10 rating, which the above pattern would be flexible enough to match.

The next datapoint to capture is the *MPAA rating*. This is the most complicated, though only in appearance. You could easily capture this field like so:

`G|PG|PG-13|R|NC-17|X|U|NR|M|MA`

If you want to challenge yourself to write something less literal, you could practice using character sets and optionality:

```
[GMNPRUX] [ACGR] ?(?:-\d{2})?
```

Finally, we have the title. This seems like the trickiest part, since the titles can be of variable length and contain any type of character, including punctuation.

However, this should work for our purposes:

```
^.+?
```

Why does this work? The `.+?` regex seems like it would inadvertently scoop up non-title parts of the line. Luckily for us, the rest of the line's data points are more or less consistently. Whatever the **dot-plus** swallows, it has to leave a minimum of structured fields in the right most part of the pattern.

So even if we have a movie named "X X 9 1999", with a rating of "R", a user rating of "8", and a production year of "2000":

```
X X 9 1999 R 8 2000
```

The title-capturing part of the regex would stop at 1999 because it has to leave at least one rating-type pattern (R), one user-rating-type pattern (8), and a four-digit number at the very end (2000).

All together now:

Find `^(.+?) ([GMNPRUX] [ACGR] ?(?:-\d{2})?) (\d\.\d?) (\d{4})$`

Replace `\1\t\2\t\3\t\4`

Or, if you prefer comma delimited style: `"\1", "\2", "\3", "\4"`

"The Godfather: Part II","R","9",1974 "Pulp Fiction","R","8.9",1994 "8½","NR","8.1",1963 "The Good, the Bad and the Ugly","R","8.9",1966 "12 Angry Men","PG","8.9",1957 "The Dark Knight","PG-13","8.9",2008 "1984","R","7.1",1984 "M","NR","8.5",1931 "Nosferatu","U","8",1922 "Schindler's List","R","8.9",1993 "Midnight Cowboy","X","8",1969 "Fight Club","R","8.8",1999"

Finding needles in haystacks (TODO)

Regexes are especially helpful in finding an answer that you don't really know, but you'll know it when you see it.

For example, dates and quantities. With some cleverness, you can choose to find dates in a certain range.

And you can do this without any programming.

In investigative work, this tool is incredibly useful, possibly the most useful

- Find extremely long words
- Find extremely long words that are not pronouns
- Certain dates, ranges
-

Shakespeare's longest word

In his collected works, the bard wrote more than TK words. Which of them was his longest?

Let's think about what we don't know:

We don't know the actual word (duh) We don't know how long the longest word is

Here's what we do know:

We know that the longest word is probably longer than 15 characters

And, more to the point:

We know the regex for finding a string of word-characters 15 characters or more:

```
\w{15,}
```

Searching for this pattern across the entirety of Shakespeare's *oeuvre*, we'll find more than 80 matches. Not bad. We can manually eyeball each word and get our answer in a minute.

But why even waste a minute? Let's just increase the threshold from 15 to 20:

```
\w{20,}
```

There is only one word that passes the threshold:

honorificabilitudinitatibus

The context:

COSTARD. O, they have liv'd long on the alms-basket of words. I marvel thy master hath not eaten thee for a word, for thou art not so long by the head as **honorificabilitudinitatibus**; thou art easier swallowed than a flap-dragon.

Hyphenation and punctuation

In the paragraph above, we see the shortcomings of our simple pattern: the `\w` group would not match a word such as `liv'd` because of the apostrophe. Nor would it match words that are hyphenated.

So let's modify our original pattern and see if there are any long compound words:

```
\b[a-zA-Z'\-]{20,}
```

This pattern matches about a dozen compound words:

```
tragical-comical-historical-pastoral
one-trunk-inheriting
to-and-fro-conflicting
wholesome-profitable
obligation-'Armigero
Castalion-King-Urinal
death-counterfeiting
candle-wasters--bring
six-or-seven-times-honour'd
water-flies-diminutives
that-way-accomplish'd
```

Longest proper noun

One of my favorite uses for regexes is to find names of people in large sets of documents. There are fancy libraries for this, natural language processing.

But sometimes I just need a quick look-see. And this is my strategy:

Proper nouns begin with capital letters.

OK, I'm not going to win any literacy awards with that definition. But it works.

Let's see how it does with Shakespeare. We'll try to find the longest name and

Longest character name

longest pronoun

Castalion-King-Urinal \b[A-Z][a-zA-Z'-]{19,}

Changing phone format (TODO)

Todo: This was moved over from a different chapter. Need to rewrite introductory guff.

Telephone game

If you've ever signed up for something on the Internet, you've undoubtedly entered your phone number in an HTML form. And you've probably noticed that some forms are smarter than others.

For example, some forms will automagically handle the formatting of the phone number. As you type, it'll insert dashes and parentheses so that it's easy for you to read. Thus, typing in:

5556667777

Will show up in your web browser (through the magic of Javascript) as this:

(555)-666-7777

However, on less sophisticated websites, the web forms might tell you explicitly how the number should be formatted. For example, it might read "Please do not use dashes." Or, "Please enter your number in XXX-YYY-ZZZZ" format. And if you don't follow the instructions, the form rejects your submission.

Hopefully, you know enough about regexes to wonder: *Why don't they just use regexes to flexibly adapt to what a user enters?* And you would be right, in my opinion, to accuse them of lazy programming.

But let's not just sulk about it. Let's see what the actual technical hurdle is in implementing an auto-formatting feature.

We'll start out with how to match the simplest form of a (standard U.S.) phone number. And then we'll add more and more variation to the possible phone number formats and adjust our regex to become more inclusive.

For each exercise, the goal is to convert any given phone number to the following format:

(555)-666-7777

Exercise: A solid line of digits The simplest form of phone number is just 10-digits in a row, with no dashes or space in between them:

```
5552415010
5559812705
9112400000
1015557745
```

What's the pattern?

Answer

In English Ten numerical digits. However, we want to separate them into three segments, so we'll need capturing groups.

Find `(\d\d\d)(\d\d\d)(\d\d\d\d)`

Replace `(\1)-\2-\3`

Curly braces for known repetition

Before we move on, let's see if there's a better way to handle a *known* quantity of repetition. In our telephone number exercise, we *know* that we need to partition out the 10 digits into 3 digits, 3 digits, and 4 digits.

We can't use the `+` operator because it matches *one-or-more*. But there is a finer-grained regex technique to capture an exact number of repetitions:



Curly braces for matching a specific quantity of repetitions

To capture n repeated instances of a pattern, enclose n (where n is the number of repetitions) inside **curly braces**

The following pattern: `> a{4}` – will match *exactly* four `a` characters

Let's use the **curly braces** to simplify our phone number regex:

Find `(\d{3})(\d{3})(\d{4})`

Replace `(\1)-\2-\3`

That's a bit easier to read; at a glance, we can immediately see how many digits we're matching in each captured group. There is some extra "visual complexity" in that we have (curly) braces nested inside other braces (parentheses, in this case). But nested braces is just something your brain will have to get used to deconstructing.

Exercise: Grouped digits, possibly separated by spaces Instead of requiring users to give us 10 straight digits, we allow them to use spaces. Why might we do this? Because it reduces mistakes on the *user* end. Sure, they might know their phone number as well as their birthdate. But using spaces lets them more easily spot typos. Compare reading 5553121121 to 555 312 1121

So now our regex has to expect this kind of input (as well as 10 straight digits):

```
5552415010
555 981 2705
9112400000
101 555 7745
```

Answer We want to match 10 numerical digits in three capturing groups. However, there may or may not be a space between each group.

This is a perfect time to use the `?` operator.

Find `(\d{3}) ?(\d{3}) ?(\d{4})`

Replace `(\1)-\2-\3`

The question mark signifies that the preceding whitespace character is *optional*. Note that the **Replace** value doesn't have to change, because the capturing group still contains the same number of digits.

Exercise: Grouped digits, possibly separated by a hyphen or a space Many people use hyphens instead of space-characters to separate their phone digits. We need to change our regex to match that possibility.

```
555-241-5010
555 981 2705
9112400000
101-555 7745
```

What's the pattern?

We want to match 10 numerical digits in three capturing groups. However, there may or may not be a space – *or a hyphen* – between each group.

We can use the `?` operator as before, but we'll throw in a character set to handle either a hyphen or a whitespace character.

Answer

Find `(\d{3})[\-]?(\d{3})[\-]?(\d{4})`

Replace `:(\1)-\2-\3`

Let's focus on the regex we use to match *either a hyphen or a whitespace character*:

`[\-]`

So there's a space in there. But what's with the *backslash-hyphen*? Recall that hyphens, when *inside square brackets*, become a special character. For example, `[a-z]`, is the character set of lowercase letters from a to z.

Thus, when we want to include a *literal* hyphen inside a square-bracketed character set, we need to **escape it with a backslash**.

One more thing to note: the square brackets act as a sort of grouping: thus, the `?` makes optional either the hyphen *or* whitespace character (and whatever other characters we want to include in those square brackets).

Exercise: Exactly 10 digits So our web-form will now correctly handle a 10-digit phone number that may or may not have hyphens/spaces.

But what happens when the user enters in *more* than 10-digits? If you apply the previous solution to the following number:

25556164040

The pattern will match this and the replacement will end up as:

(255)-561-64040

In a real-life application, we do not want our regex to be flexible enough to accept this. Why? Because this kind of input indicates that there is an error by the user. Maybe that initial 2 is a typo. Or maybe his finger slipped at the end and added an extra 0. Either way, we do not want our regex to match this. We want the user to know that he possibly screwed up.

Answer What's the pattern?

We want to match *only* phone numbers that contain exactly 10 digits (with optional spaces/hyphens) on a *single line*.

In order to do this, we make our regex more *rigid*. We can do this by using the line anchors `^` and `$` to indicate that we don't expect any other kinds of characters besides the 10-digits and the optional hyphens/spaces.

Find `^(\d{3})[\-]?(\d{3})[\-]?(\d{4})$`

Replace `(\1)-\2-\3`

If you apply it to these sample numbers:

```
25556164040
5556164040
```

You will end up with:

```
25556164040
(555)-616-4040
```

The first number is not transformed because it is not matched by the pattern.

Exercise: Dial 1? If you've ever moved from office to office, one of the perplexing questions is: *Do I have to dial 1 to make an outside call?*

So some people are used to having 1- be a part of their phone number, especially if they're using their office phone. Some of these people might add this prefix to their number out of reflex:

```
1-555-241-5010
555 981 2705
9112400000
1 101-555-7745
```

Answer

In English The same as before, except that the ten-digits *might* be preceded with the digit 1 *and also might* be preceded by a hyphen *or* a whitespace character. There's no new technique here. We just need to use the `?` operator at least once more.

Find `^1?[\-]?(\d{3})[\-]?(\d{3})[\-]?(\d{4})$`

Replace `(\1)-\2-\3`

This pattern does the job of matching an optional 1 as well as an optional hyphen or whitespace *before* the actual ten digits of the phone number. We can use the same value for the **Replace** field because we're using the same captured groups.

If you apply it to the following numbers:

```
5556164040
1 555 2929010
2 555 616 4040
```

The resulting transformation is:

```
(555)-616-4040
(555)-292-9010
2 555 616 4040
```

The last number is *not transformed* because its prefix is a 2, which is something we don't expect (in a standard U.S. number).

A more precise grouping

There's something a bit off to the previous solution. Since *both* the 1 *and* the whitespace/hyphen are optional, that means our regex would also match the following:

```
-555-212-0000
```

And turn it into:

```
(555)-212-0000
```

You might think: *Yeah, so?* Well, it's true that there's *probably* no harm done in matching (and then ignoring) a stray hyphen, sometimes you have to ask, *why is there a stray hyphen without a 1?* It may be harmless to us, but it might indicate that the user misunderstood what was supposed to be the phone field. Perhaps they're trying to enter an international format and got cutoff.

Who knows. But just to be safe, let's make our regex more *rigid* so that it *won't* match that kind of input. In other words, we want a regex that matches:

```
`1-555-212-0000`
```

But not:

```
`-555-212-0000`
```

A little obsessive-compulsive, maybe, but let's just do it for practice.

What's the pattern?

We're looking for the standard 10-digit number that *may* be preceded by a 1 *and* either a hyphen *or* a whitespace character (compare this to the previous pattern we looked for).

Remember from the previous chapter that parentheses act as a way to group parts of a pattern together. So let's try the following:

Find `^(1[\-])?(\d{3})[\-]?(\d{3})[\-]?(\d{4})$`

Replace `(\1)-\2-\3`

Apply it to the following sample input:

```
5552415010
1 555 981 2705
1-9112400000
-101 555 7745
```

And you will get:

```
()-555-241
(1 )-555-981
(1-)-911-240
-101 555 7745
```

What the? The very last number wasn't affected, which is what we wanted. But the matched patterns got replaced by non-sensical values.

The problem is our use of an extra set of parentheses *without* modifying our **Replace** value. Remember that when using **backreferences**, the numbers correspond to each sequential captured group.

In other words, `\1` no longer refers to `(\d{3})` but to `(1[\-])`, which is simply the optional 1-prefix.

There are two ways to fix this. The (intellectually) lazy (but more physically demanding) way is to alter the **Replace** field to correspond to the extra captured group:

Replace `(\2)-\3-\4`

This works because it effectively throws away the first captured group.

But if we're going to throw away that first group, why even capture it at all (remember, we're still in obsessive-compulsive mode)?

If you remember from the chapter on **capturing groups**, there was a way to designate that a set of parentheses was **non-capturing**:

`(?:whatever-we-want-to-capture)`

Remember when I said earlier in this chapter that that `?` character has multiple meanings? In this case, the `?` – when it comes *immediately* after a `(` and followed by a **colon** character – denotes that the parentheses is not intended to store a **back-reference**.

So, the cleaned-up (but more complicated-looking) solution:

Find `^(?:1[\-])?(\d{3})[\-]?(\d{3})[\-]?(\d{4})$`

Replace `(\1)-\2-\3`

When we apply it to:

```
5552415010
1 555 981 2705
1-9112400000
-101 555 7745
```

We get:

```
(555)-241-5010
(555)-981-2705
(911)-240-0000
-101 555 7745
```

The upshot of this chapter, besides introducing the `?` as the *optional* operator, is to show how a simple regex can slowly become more-and-more complicated, even as we still understand its basic purpose.

Remember that our first solution looked like this:

```
(\d{3})(\d{3})(\d{4})
```

And we ended up with:

```
^(1[\ -])?(\d{3})[\ -]?(\d{3})[\ -]?(\d{4})$
```

And yet both variations handle the same problem: how to match a U.S. 10-digit phone number.

So maybe handling all kinds of phone numbers *isn't* as simple as we thought (though every mediocre programmer should be able to figure this out). We could add a few more variations – what if there were more than one space (or hyphen) separating the numbers? what if the spaces/hyphens grouped the numbers in a different pattern than 3-3-4? – but I think we're all phoned out for now.

Ordering names and dates (TODO)

One of the most frequent questions on the the NICAR mailing list
Getting data ready for spreadsheets is one of the most tedious tasks.

Scenario

Prerequisites

- Knowing how to match character sets
- Knowing how to match repeated patterns
- Capturing groups

Year, months, days

Let's start with a simple

MM/DD/YYYY to YYYY-MM-DD

M/D/YYYY to YYYY-MM-DD

Names

Switching last name, first name

Handling middle names

Switching first name, middle initial, last name

Preparing for a spreadsheet

Dating, Associated Press Style (TODO)

The AP Stylebook basically *the* dictionary for newspaper professionals. Many newspaper jobs involved a written test of AP style. To non-journalists, the sometimes-arcane rules are probably not well known. Most writers, for example, would be satisfied with abbreviating “Arizona” with its postal code, “AZ”. But AP writers use “Ariz.”

Numbers 1 through 10 would be spelled out – “three policemen”, “seven ducklings” – unless used with ages or percentages: “Kaley, a 7-year-old” and “99 percent.”

For the longest time, “website” was spelled as “Web site”²⁷. And so on.

As you might expect, the AP has a set of rules when listing dates.

Scenario

Imagine that you have a giant text file containing news stories in which the writer didn’t adhere to AP style on dates. We need to write a regular expression that can sweep through the file and abbreviate month names *en masse*.

Prerequisites

- Alternation with the pipe character, |
- Character sets, such as \d
- Repetition with the +
- Limited repetition with curly braces, {1,2}
- Capturing groups
- **Optional:** Character sets with square brackets, [a-z]
- **Optional:** Lookahead with (?=)

The AP Date format

When spelling out the full date – February 1, 2012 – the month is abbreviated:

February 1, 2012 Feb. 1, 2012 January 19, 1999

Jan. 19, 1999 May 6, 2014

May 6, 2014 March 22, 1788

Mar. 22, 1788 June 7, 1960

June 7, 1960

²⁷<http://mashable.com/2010/04/16/ap-stylebook-website/>

Abbreviate the month

As you can see from the above examples, all months *more than 4* characters long are *abbreviated to 3 letters** with a dot. The warm months of May, June, and July are left unabbreviated.

* **Note:** September is abbreviated to **Sept.** in AP style.

The literal solution

There's only – and will always only be – 12 months, nine of which are actually affected. So without much thinking, we can match them *literally* with the use of **alternation**

Find (January|February|March|April|August|September|October|November|December)

Replace ???

Unfortunately, it's not that easy. This pattern makes a match, but what do we replace it with? Capturing the entire month's name gives us a backreference to...the *entire* month's name, which is unhelpful if we're trying to abbreviate the month.

We need to **capture** the first three characters of the long months and discard the rest. Let's also use a **bracketed character set** to match (but *not* capture) the rest of the month's name to *discard*:

Find (Jan|Feb|Mar|Apr|Aug|Sept|Oct|Nov|Dec)\w+ **Replace**
 \1.

Try it on this list of dates:

February 1, 2012
 January 19, 1999
 May 6, 2014
 March 22, 1788
 June 7, 1960

Match just full dates

There's one twist to AP Style dates: only dates that include a specific day use the abbreviated month style.

So our pattern needs to become more specific. It must not affect non-exact dates, such as:

- The president last visited the state in **October 1980**

Todo: I messed up here and forgot that abbreviations *do* occur if the date does not include the year. So I need to revise the exercises below accordingly.

In addition to the month name, our pattern needs to match a day value – one or two digits – a comma, and the four-digit year.

Let's keep the previous solution. We can use the character class of `\d` with **repetition** syntax:

Find `(Jan|Feb|Mar|Apr|Aug|Sept|Oct|Nov|Dec)\w+ (\d+, \d+)`

Replace `\1. \2`

Try it out on the following:

January 12, 2012

May 9, 2013

August 4, 1999

September 1874

June 2, 2000

February 8, 1999

March 2

December 25

April 2, 1776

Note that the second-capturing group catches both the day and the year, as well as the literal comma. This seems lazy, maybe, but we're only required to alter just the month, not anything that follows it.

Match the specific repetition

Our solution above seems to work, but it's just a little too promiscuous. Consider the following valid AP Style sentence:

Then in **September 1909**, 52 pandas launched an aerial assault on the encampment.

Because our pattern uses `\d+, \d+` – match *one-or-more* digits followed by a comma, a space, and then *one-or-more* digits – the sequence 1909, 52 is inadvertently swept up:

Then in **Sept. 1909**, 52 pandas launched an aerial assault on the encampment.

We will need to limit the repetition to make it more specific. If we assume that the day always consists of 1 to 2 digits and the year will always be 4 digits (at least for AP stories dealing with events give or take a millennium):

Find `(Jan|Feb|Mar|Apr|Aug|Sept|Oct|Nov|Dec)\w+ (\d{1,2}, \d{4})`

Replace `\1. \2`

Reduce wasteful capturing

Since we're only modifying the name of the month, why even bother capturing (and having to echo) the rest of the date?

We can use the **positive-lookahead** to test if a pattern exists *ahead* of the pattern we seek to replace. The regex engine will check if that assertion but otherwise won't try to modify it.

Using the lookahead will make our **Find** pattern a little longer, but it will simplify our **Replace** pattern. It is also more efficient, though we won't notice the difference unless we're trying to apply the pattern to hundreds of thousands of text strings at once:

Find (Jan|Feb|Mar|Apr|Aug|Sept|Oct|Nov|Dec)\w+ (?=\d{1,2}, \d{4})

Replace \1

Handling imaginary months

We're pretty much done here; the above regex will handle mass-correction of 99% of the unabbreviated full-dates that might show up in an otherwise proper Associated Press story.

But what if we didn't have just 12 months to consider? It's easy to type out such a small set of literal month names.

Let's stretch out our problem-solving muscles and find a solution that *doesn't* depend on the Gregorian(TK) month names. To keep things simple, we'll keep the same following expectations and requirements:

- Month names are capitalized and consist only of alphabet letters
- Abbreviate only the month names that are *than* 4 characters
- The day and year format is the same as before

The solution will involve a bit more special regex syntax, but the change can be described like this: *look for a word that begins with a capital letter and is followed by at least four lower-case letters.*

Let's break that down:

begins with a capital letter \b[A-Z]

followed by at least four lower-case letters [a-z]{4,}

(Note: I use the word-boundary \b to avoid matching something like "McGregor", though I guess there's no reason why there couldn't be Scottish-named months in our scenario...)

We're almost there. Remember that we still have to replace the month's name with an abbreviation consisting of the month's first three letters (we'll ignore the special case of September). So we need to *halve* the second part of the pattern above:

```
[a-z]{2}[a-z]{2,}
```

To make the abbreviation, we want to **capture** the first (uppercase) letter, plus the next two lowercase characters. So, all together:

```
\b([A-Z][a-z]{2})[a-z]{2,}
```

And now, the complete pattern, including the lookahead for the day and year digits:

Find `\b([A-Z][a-z]{2})[a-z]{2,} (?:\d{1,2}, \d{4})`

Try it out on this text:

January 12, 2012

May 9, 2013

August 4, 1999

September 1874

June 2, 2000

February 8, 1999

March 2

December 25

April 2, 1776

Then in September 1909, 52 pandas launched an aerial assault on the encampment.

Real-world considerations

So which solution is wiser: the one that explicitly lists the month names or the more abstract pattern with bracketed character sets?

In the real world, the **first** solution is smarter, given the circumstances. As far as we can predict, the names of months will not change, so why write a flexible pattern? Moreover, the flexible pattern would inadvertently capture phrases that aren't dates at all:

According to a classified report from Deep Space 9, 2335 was when panda attacks reached their peak before plateauing for the next 12 years.

However, it's good to get in the habit of searching out the most abstract pattern, for practice sake. Not every data problem will be as easily enumerable as the list of calendar months.

How to overthink a problem

Sometimes, the concrete solution isn't so simple, though.

Reliable regex solutions often require some **domain knowledge** of the problem. Recall that we had to use the pattern `\d{1,2}`, `\d{4}` – instead of the unspecific `\d+` – to capture the day and year.

But just to be safe, we might need to be even more specific:

The second-digit of a day, if there is one, is never anything else than 1, 2, or 3.

We can use the optionality operator, `?`, to limit the range of day. So instead of:

```
\d{1,2}
```

We could use:

```
[123]?\d
```

But wait. There's still more logic to consider:

- If the date is single-digit, then it can only be from 1 to 9 (i.e. not 0)
- If the date is two-digits and begins with a 3, then the ones-place digit must be either 0 or 1, as the day 32 does not exist for any month.
- If the month is February, then the tens-place digit can never be more than 2
- And what about leap years?

There is actually a way, I *think*, to capture this logic with regex syntax only. It would require **conditionals**, something I avoided covering in the book because of its convoluted syntax. The actual regex pattern would likely be several lines long.

I say *likely* because I haven't – and will not – attempt it. And neither should you.

The limits of regex

So is the AP style date-enforcer a futile goal in the real world?

Well, imagine that your job is to create *something* that could process a news organization's thousands (perhaps millions) of articles and reliably abbreviate the dates.

If you had the technical chops to be trusted with that task, you would use the far more versatile (and easier to use) techniques that general programming makes available to deal with such complicated logic. If you tried to, instead, solve it with a single regex, you would either go insane or, hopefully, just be fired.

As powerful and flexible as regexes are, they are simply one tool. To paraphrase the aphorism, "When all you have is a hammer, everything looks like a nail." So part of understanding regexes is knowing that certain problems are not simply "nails."

Knowing is half the battle

For example, here's a much more likely problem that you'll have to fix: the writers don't even bother spelling out the dates and instead, use 10/25/1980 as a shorthand for October 25, 1980.

How can a single regex "know" that 10 corresponds to October (nevermind properly abbreviate it)? Again, I'm not going to even guess what that regex (or series of regexes) would be.

But that's the kind of problem that you may want to solve, right? It may be frustrating that all of this regex practice won't give you the direct solution. But at least you *recognize* that this is a problem that *might* be solvable with regexes and *something else*. And that's a big shift in mindset.

Remember when I said regexes were a gateway drug into programming? Learning programming is a topic for another book. But as we go through these exercises, I'll try to get you to think about how your regex abilities can be further expanded.

Sorting a police blotter

The Colorado Springs Police Department posts six months worth of crime incident reports at its website. The data clearly comes from a database but we only have the text of the webpage to work with:

Colorado Springs Police Department Police Blotter <i>"Safeguarding our Community as our Family."</i>	
There are 10 records displayed per page with the most current first. Records 1 to 10 of 576	
Next Last Use to move from page to page (also at bottom of page).	
Record ID	17848
Incident Date	March 7, 2013
Time	11:20:00 AM
Division	Gold Hill -- Shift I
Title	Warrant
Location	Cascade and Sierra Madre
Summary	Officers received a tip a wanted party was in the downtown area. The party is Justin Burns who was wanted out of Texas for sexual assault on a child. Officer Keller located the suspect near the area of Cascade and Sierra Madre. His identity was confirmed and he was transported to CJC.
Adults Arrested	Burns, Justin
PD Contact & Number	Lt. Feese - 385-2125
<hr/>	
Record ID	17844
Incident Date	March 6, 2013

Colorado Springs Police blotter

We have to essentially reverse-engineer the output on the webpage to recreate the spreadsheet that the reports originate from. Unfortunately, copying-and-pasting directly from the webpage does not preserve any of the structure.

However, there are enough patterns in the text glob to do what we need with just regular expressions. The end product will look very, *very* convoluted. But I'll show you how it's actually just a bunch of simple pieces all put together.

Note: This is just a proof of concept to show you how far you can get with intuitively recognizing patterns in text. If you are actually trying to make usable data from these pages for a real-life project, *this is not how you should do it*. Learning a little bit of how to program in order to automatically web-crawl and parse the pages – using a scraping library more powerful and easy-to-use than regexes – is the ideal way to do this.

But let's see how far we can get with regexes.

Sloppy copy-and-paste

Visit one of the pages at the [Colorado Springs online police blotter](#)²⁸. Each page should have 10 reports.

If you copy-and-paste the entire webpage into a textfile, though, you get a mess like this (the names have been changed from the actual report but the (*amazing*) events are verbatim):

17836 Record ID March 5, 2013 Incident Date 6:59:00 AM Time Sand Creek – Shift IDivision OtherTitle190X Carmel DriveLocationSand Creek Officers were sent to the Westwind Apartments. The reporting party said her boyfriend was cut with a sword by a neighbor. Medical personnel staged and officers responded to the scene. Investigation revealed that Sean Doe went to Guy Incognito’s apartment with a Samurai sword. Doe swung the sword at Incognito’s face when Incognito opened his door. Incognito caught the blade in both hands and he was able wrestle the sword away. Incognito and Doe were bleeding extensively. Officers arrested Doe at his apartment. Incognito and Doe were transported to Memorial Central for treatment. Both are in stable condition and awaiting surgery.SummarySean DoeAdults Arrested Lt. Strassburg-Aldal - 444-7270PD Contact & Number

What happened? It’s a consequence of how the HTML is structured, a topic that we don’t need to concern ourselves with. All that matters is we have a jumble.

But luckily, the jumble isn’t too bad. The problem is that the data labels, such as “**Record ID**” and “**Adults Arrested**” have been thrown into the text with no separation. However, as long as those data labels stay the same for each record, then we can use them to create a usable regex.

Below is the jumbled text again, except that I’ve **bolded** the data labels:

17836 **Record ID** March 5, 2013 **Incident Date** 6:59:00 AM **Time** Sand Creek – Shift **IDivision** Other**Title**1900 Block of Carmel Drive**Location**Sand Creek Officers were sent to the Westwind Apartments. The reporting party said her boyfriend was cut with a sword by a neighbor. Medical personnel staged and officers responded to the scene. Investigation revealed that Sean Doe went to Guy Incognito’s apartment with a Samurai sword. Doe swung the sword at Incognito’s face when Incognito opened his door. Incognito caught the blade in both hands and he was able wrestle the sword away. Incognito and Doe were bleeding extensively. Officers arrested Doe at his apartment. Incognito and Doe were transported to Memorial Central for treatment. Both are in stable condition and awaiting surgery.**Summary**Sean Doe**Adults Arrested** Lt. Strassburg-Aldal - 444-7270**PD Contact** ** & **Number**

It appears the data labels come *after* their associated data, e.g:

Time Sand Creek – Shift IDivision

²⁸<http://www.springsgov.com/units/police/policeblotter.asp>

Start loose and simple

Let's try this the sloppiest way we can: use the **dot** operator with the **lazy-star** operator, `.*?` to **capture** everything that is *not* one of the actual labels:

```
Find (.*?)Record ID(.*?)Incident Date(.*?)Time(.*?)Division(.*?)Title(.*?)Location(.*?)Summary(.*?)
      Contact\s*& Number
```

Note: There's apparently always a line break between "PD Contact" and "& Number"

Why do we use the **star** instead of the **plus**? Some of those fields, such as "**Adults Arrested**", may be empty.

However, this pattern is not quite there. Some of the records have multiple line breaks within certain fields, such as the "**Summary**". The dot operator does not match **newline** characters. So we simply need to change all the instances of `(.*?)` to `((?:.|\\n)*?)`. That looks more complicated than it sounds, because I threw in a non-capturing group. But basically, we've changed the pattern to match either a **dot** (i.e. all newline characters) *or* a newline character (i.e. *everything*).

```
Find ((?:.|\\n)*?)Record ID((?:.|\\n)*?)Incident Date((?:.|\\n)*?)Time((?:.|\\n)*?)Division((?:.|\\n)*?)
      Contact *\\n& Number
```

```
Replace \\1\\n\\2\\n\\3\\n\\4\\n\\5\\n\\6\\n\\7\\n\\8\\n\\9
```

Using the above regex on our sample text, we end up with nine separate lines, each one corresponding to a data field (we don't really need the labels at this point):

```
17836
March 5, 2013
6:59:00 AM
Sand Creek -- Shift I
Other
190X Carmel Drive
Sand Creek Officers were sent to the Westwind Apartments. The reporting par\
ty said her boyfriend was cut with a sword by a neighbor. Medical personnel\
staged and officers responded to the scene. Investigation revealed that Se\
an Doe went to Guy Incognito's apartment with a Samurai sword. Doe swung th\
e sword at Incognito's face when Incognito opened his door. Incognito caught\
the blade in both hands and he was able wrestle the sword away. Incognito\
and Doe were bleeding extensively. Officers arrested Doe at his apartment.\
Incognito and Doe were transported to Memorial Central for treatment. Both\
are in stable condition and awaiting surgery.
Sean Doe
Lt. Strassburg-Aldal - 444-7270
```

I used newlines in the replacement pattern just to produce something easier to read. You could've used tab characters which would allow you to paste the result directly into Excel:

A	B	C	D	E	F	G	H	I
17836	March 5, 2013	6:59:00 AM	Sand Creek - Other - Shift I		190X Carmel Drive	Sand Creek Officers were sent to the Westwind Apartments. The reporting party said her boyfriend was cut with a sword by a neighbor. Medical personnel staged and officers responded to the scene. Investigation revealed that Sean Doe went to Guy Incognito's apartment with a Samurai sword. Doe swung the sword at Incognito's face when Incognito opened his door. Incognito caught the blade in both hands and he was able to wrestle the sword away. Incognito and Doe were bleeding extensively. Officers arrested Doe at his apartment. Incognito and Doe were transported to Memorial Central for treatment. Both are in stable condition and awaiting surgery.	Sean Doe	Lt. Strassburg-Aldal - 444-7270

A single tab-delimited police report pasted into Excel

Adding a report-ending delimiter

If you try to apply the regex above to the entire page of 10 reports, you'll find that they don't copy-and-paste so easy into Excel. This is because some of the reports contain extra line breaks. And copying/pasting line breaks into Excel will create problems:

	A	B	C	D	E	F	G
4		17839	March 5, 2013	6:06:00 PM	Sand Creek -- Shift II	Traffic Crash	E Platte Ave and N Murray Blvd
5	All traffic lanes on E. Platte Avenue and N. Murray Boulevard were opened at approximately 2210 hours. Three persons remain hospitalized at Memorial Hospital Central. The traffic accident remains under investigation by the Colorado Springs Police Department's Major Accident Unit. ***** On 03/05/13 at approximately 6:06PM officers from the Sand Creek Division were						***UPDATE ***
6							
7							
8							

The entire tab-delimited reports list with gratuitous line breaks included, pasted into Excel

There's a few hacks we can use to clean this up. I think the easiest approach is to adjust the **Replace** pattern:

Replace \1\t\2\t\3\t\4\t\5\t\6\t\7\t\8\t\9--END--

You can use whatever arbitrary string you want there. The important thing is to have that string denote the end of each report. An example result is:

17838 March 5, 2013 2:41:00 PM Falcon -- Shift II Burglary 3400 block of Sinton Rd
On 03-05-13, Officers from the Colorado Springs Police Department were dispatched to Sinton Rd regarding a burglary. Upon arrival they learned approximately 6 camper style vehicles/trailers had been broken into from 03-04-13 6pm to 03-05-13 12:00 pm. If you were a victim of one of these burglaries please contact the Colorado Springs Police Department at 444-7000. Lt. Buckley - 444-7240--END--

After you've run the regex-replace as we did previously, we do *another* find-and-replace: this time, we change all newline characters (either `\n` or `\r`, depending on your text editor) into regular whitespace:

Find `\n`

Replace *(just a space character)*

And finally, we do one more replacement: change the “end” string into a **newline** character:

Find `--END--`

Replace `\n`

And then we paste the final result into Excel:

	A	B	C	D	E	F	G	H
1	17839	March 5, 2013	6:06:00 PM	Sand Creek -- Shift II	Traffic Crash	E Platte Ave and N Murray Blvd	***UPDATE*** All traffic lanes on E. Platte Avenue and N. Murray Boulevard were opened at approximately 2210 hours. Three persons remain hospitalized at Memorial Hospital Central. The traffic accident remains under investigation by the Colorado Springs Police Department's Major Accident Unit. ***** On 03/05/13 at approximately 6:06PM officers from the Sand Creek Division were dispatched to a head on collision at the intersection of E Platte Ave and N Murray Blvd. There were two vehicles involved in the accident and three people were transported to local hospitals. At least one of the patients transported is in critical condition with life threatening injuries. The intersection of E Platte Ave and N Murray Blvd has been closed in all directions. Drivers are asked to avoid the area, using Galley Rd for an east-west alternative and either Academy, Wooten, or Powers as a north-south alternative. It is not known when the intersection will be opened for traffic.	N
2	17838	March 5, 2013	2:41:00 PM	Falcon -- Shift II	Burglary	3436 Sinton Rd	On 03-05-13, Officers from the Colorado Springs Police Department were dispatched to 3436 Sinton Rd regarding a burglary. Upon arrival they learned approximately 6 camper style vehicles/trailers had been broken into from 03-04-13 6pm to 03-05-13 12:00 pm. If you were a victim of one of these burglaries please contact the Colorado Springs Police Department at 444-7000.	

Ten reports, tab-delimited and pasted into Excel

Conclusion

This example is an elaborate exercise to show you how far a regex can get you. In this case, we were lucky enough that the data fields were consistently named. Just to be safe, we could've been much more specific with our pattern, such as looking for numerical digits in the date and time fields. But being lazy worked out for our purposes.

However, I cannot emphasize that in *the real world*, you would not parse this police blotter with just regexes. You would at least write a script to traverse the hundred or so pages to collect the text. And if you know that much coding, you'd also figure out a better way to get it into a spreadsheet than with our lazy regex.

But this doesn't mean the regex work here was just a foolish exercise. We learned how to find a pattern in a jumble, and that's a very worthwhile skill to have when writing a programming script.

Converting XML to tab-delimited data

This exercise covers a real-life scenario I encountered while working on the [Dollars for Docs](http://projects.propublica.org/docdollars/) project at ProPublica²⁹. As part of a lawsuit settlement, a drug company was required to post online what it paid doctors to promote its products.

They posted the records in a Flash application, which made it fairly easy to download the data as XML³⁰. However, while the XML was in a simple format, it did not import into Excel as a usable spreadsheet.

But XML is a pattern just like any other. This lesson describes how to convert simple XML into a spreadsheet-like format.

The payments XML

The screenshot shows a web browser window with the URL <http://www.cephalon.com/our-responsibility/fees-for-services-2010/fees-for-services.shtml>. The page title is "Cephalon | Fee for Services". The main content area contains a search bar and a table of payments. The table has columns: Last Name, First Name, City, State, and Value of Payment. The table is filtered for the range "\$0—\$10,000". The table lists several payments, including those to Abbott, Abate, Agha, and Ahmad. An orange circle highlights the table and the browser's developer console below it. The developer console shows a list of network requests, including "GET our_responsibility_image", "GET ga.js", "GET _utm.gif?utmwv=4.8.6&u", "GET body_background.gif", "GET footer_background.gif", "GET sitemap_bg.gif", "GET main_nav_bg.gif", "GET subnav_bg.jpg", "GET search_bg.jpg", "GET subnav_gradientbg.jpg", "GET subnav_section_bg.gif", "GET subnav_arrow.gif", "GET subnav_arrow_open.gif", and "GET spend_data.swf".

Last Name	First Name	City, State	Value of Payment
\$0—\$10,000			
Abbott	Brian	Great Falls, MT	\$150.00
Abate	Sheila	Indianapolis, IN	\$200.00
Agha	Mounzer	Pittsburgh, PA	\$3,000.00
Ahmad	Mahmood	Sherwood, AR	\$2,000.00

The Cephalon database of fees paid to healthcare professionals

²⁹<http://projects.propublica.org/docdollars/>

³⁰<http://www.propublica.org/nerds/item/reading-flash-data>

You can download a copy of the [drug company's data at its site](#)³¹. Check out my [tutorial for more details](#)³². However, the company's site has been down at the time of this writing.

But the actual data isn't important. Below is some sample data (with names changed) in the same XML format:

```
<dataset>
<data>
<row>
<value>Doe, Matthew M</value>
<value>MD</value>
<value>Lafayette, LA</value>
<value>$2500</value>
<value>Lectures</value>
</row>
<row>
<value>Smith, Mark G</value>
<value>RN</value>
<value>West Palm Beach, FL</value>
<value>$35000</value>
<value>Consulting</value>
</row>
<row>
<value>Johnson-Jackson, Tracy</value>
<value>MD</value>
<value>Buffalo, NY</value>
<value>$10050</value>
<value>Research</value>
</row>
<row>
<value>Clark, Ahmed</value>
<value>MD</value>
<value>Springfield, AR</value>
<value>$500</value>
<value>Lectures</value>
</row>
<data>
</dataset>
```

³¹<http://www.cephalon.com/HealthcareProfessionals/2009-Fees-for-services.aspx>

³²<http://www.propublica.org/nerds/item/reading-flash-data>

The pattern

The XML downloaded from the company site was flattened for some reason. If the XML had the typical whitespace formatting for its nested structure, it would look like this:

```
<dataset>
  <data>
    <row>
      <value>Doe, Matthew M</value>
      <value>MD</value>
      <value>Lafayette, LA</value>
      <value>$2500</value>
      <value>Lectures</value>
    </row>
    ...
  </data>
</dataset>
```

However, there's no other type of XML elements except for the `<row>` elements. So essentially we just have to care about breaking apart each `<row>` element and extract the values between the `<value>` elements.

This is pretty straightforward. The pattern should include the newline characters between each element:

Find `<row>\n<value>(.*?)</value>\n<value>(.*?)</value>\n<value>(.*?)</value>\n<value>(.*?)</value>`

Replace `\1\t\2\t\3\t\4\t\5`

So this:

```
<row>
<value>Johnson-Jackson, Tracy</value>
<value>MD</value>
<value>Buffalo, NY</value>
<value>$10050</value>
<value>Research</value>
</row>
```

Becomes this:

Johnson-Jackson, Tracy MD Buffalo, NY \$10050 Research

Note: Of course, it's possible to simply do a replacement of the *literal* values, e.g. replacing `</value>` with `\t` and `</row>` with `\n`, which would simplify the **Replace** operation.

Add more delimitation

Why restrict ourselves to organizing the values as the drug company does? If we want to analyze the data by U.S. state, for example, we'd want to format the location `<value>` element (the third element):

```
<value>Buffalo, NY</value>
```

– into a *city* and *state* field:

```
Buffalo\tNY
```

This doesn't require anything more than just adding a few more capturing groups so that the *name* value is divided into *lastname* and *firstname* values, and that *location* is separated into *city* and *state*:

Find `<row>\n<value>(.*?) , (.*?)</value>\n<value>(.*?)</value>\n<value>(.*?) , ([A-Z]{2})</value>\n<value>(.*?)</value>`

Replace `\1\t\2\t\3\t\4\t\5\t\6\t\7`

If you apply the above pattern to the sample data at the beginning of the chapter, you'll get seven columns of tab-delimited data:

Doe	Matthew M	MD	Lafayette	LA	\$2500	Lectures
Smith	Mark G	RN	West Palm Beach	FL	\$35000	Consulting
Johnson-Jackson	Tracy	MD	Buffalo	NY	\$10050	Research
Clark	Ahmed	MD	Springfield	AR	\$500	Lectures

You can directly paste this into Excel:

	A	B	C	D	E	F	G
1	Doe	Matthew M	MD	Lafayette	LA	\$2,500	Lectures
2	Smith	Mark G	RN	West Palm Beach	FL	\$35,000	Consulting
3	Johnson-Jackson	Tracy	MD	Buffalo	NY	\$10,050	Research
4	Clark	Ahmed	MD	Springfield	AR	\$500	Lectures
5							

The tab-delimited data pasted into Excel

Cleaning up Microsoft Word HTML (TODO)

Even though HTML is a pretty easy language to master, most people have been trained in word processors and feel more comfortable composing documents in them.

When those documents have to reside on the Web, there's usually a conversion tool from *.doc to *.html. But the conversion isn't perfect. In fact, the word processor will often add a lot of meta-junk.

With a few regular expressions though, we can get minimalist, clean HTML.

TODO

Eliminate everything except the main post

Find `(?:.|\\n)+?<div class=WordSection1>`

Replace *(with nothing)*

Eliminate after the main post

Find `</div>\\s*</body>\\s*</html>`

Replace *(with nothing)*

clean up line breaks within elements

Find `(?<=<)([^\>]*)\\n+`

Replace `\\1` and a space character

Delete all element attributes except for href

Find `<(\w+).*(?!href=" [^"]+?") [^\>]*?>`

Delete spans, op and <!>

Switching visualizations (TODO)

If visualizations are made from data and data made from text, then regular expressions should be able to play a part in creating data visualizations.

This chapter demonstrates how regexes can be used to port your data from one visualization option to another, including Excel, Google Charts, and R.

A visualization in Excel

From Excel to Google Static Chart

From Google Static Charts to Google Interactive Charts

Cleaning up OCR Text (TODO)

Image scanners and [optical-character-recognition software](#)³³ ease the process of turning paper into digital text. And as we learned in previous chapters, data is just text with a certain structure.

But the challenge of scanned text is that the conversion is messy.

Todo: Example image

Scenario

When using Tesseract (or an OCR program of your choice) on a scanned image, you'll almost always have imperfect translations. A common problem might involve numbers and letters that look-alike, such as the lower-case-“L” and the number 1.

We'll use regexes to quickly identify problematic character-groupings, such as numbers in the middle of words (e.g. he110).

Note: This is only a proof of concept. If you're digitizing large batches of documents, you'll be writing automated scripts with a variety of regex and parsing techniques. This chapter is not meant to imply that you can deal with this problem with your trusty text-editor alone.

Prerequisites

Todo:

Finding misplaced symbols inside letters.

- Scan a text
- Tesseract it
- highlight all words that fit:

```
(\b[A-Za-z]+[ ^ A-Za-z' \- ]+[A-Za-z]+)
```

- find all numbers that have a letter in the middle of them

³³http://en.wikipedia.org/wiki/Optical_character_recognition

Cheat Sheet

Syntax	Find	Replace	Original text	Replace first	Replace all
^ Beginning of line	^a	A	an aria	An aria	An aria
\$ End of line	a\$	d	an aria	an arid	an arid
\b Word boundary	\ba\b	an	a aria a	an aria	an aria an
\B Not a word boundary	a\B	o	an aria	on aria	on oria
\s Whitespace character (includes new lines)	\s	-	Hey hey hey	Hey-hey hey	Hey-hey-hey
\S Not a whitespace character	\S	-	Hey hey	-ey hey	--- ---
\n OR \r Newline character (depends on operating system)	\n	-	Hey hey hey	Hey-hey hey	Hey-hey-hey
\t Tab character	\t	,	Albany\tNY\tUS	Albany,NY\tUS	Albany,NY,US
. Match any character except newlines	a..	xyz	an apple a bear	xyzapple a bear	xyzxyzle xyzear
\w A word character: letters, numbers, and underscore	a\w	xy	an apple	xy apple	xy xyple
\W Not a word character	\W	-	The cat.	The-cat.	The-cat-
x y Either x or y	dog cat	rat	The cat and dog	The rat and dog	The rat and rat
[abc] Either of a,b,c	[abc]	x	a bearcat	x bearcat	x bexrxt
[^xyz] Anything but x,y,z	[^abc]	x	a bearcat	a bxarcat	a bxaxcax
[a-z] Any letters a through z	[A-Z]	x	The Cat	xhe Cat	xhe xat
? Zero-or-one	x?ray	laser	xray ray xxray	laser ray xxray	laser laser x laser
* Zero-or-more	x*ray	laser	xray ray xxray	laser ray xxray	laser laser laser
+ One-or-more	\w+a	x	baaaa baaaa	x baaaa	x x
+? Lazy one-or-more	\w+?a	x	baaaa baaaa	xaaa baaaa	xxa xxa
{2} Match exactly 2	ba{2}	x	babaaaa babaaaa	baxaa babaaaa	baxaa baxaa
{2,6} Match between 2 to 6 occurrences	ba{2,3}	x	babaaaa babaaaa	baxa babaaaa	baxa baxa
{2,} Match 2 or more occurrences	ba{2,}	x	babaaaa babaaaa	bax babaaaa	bax bax
a(?=bc) Positive lookahead	J(?=on)	D	Jon Jay Jones	Don Jay Jones	Don Jay Dones
a(?!bc) Negative lookahead	J(?!on)	D	Jon Jay Jones	Jon Day Jones	(same)
(?<=a)bc Positive lookbehind	(?<=\d)\.\d\d	bucks	He spent 12.42.	He spent 12 bucks.	(same)
(?<!a)bc Negative lookbehind	(?<!\d)\.	!	He spent 12.42.	He spent 12.42!	(same)
(?:abc) Non-capturing group	gr(?:a e)y	black	grey and gray	black and gray	black and black
(abc) Capturing group	(\d)(\d)(\d)	\3\2\1 OR \$3\$2\$1	123456	321456	321654

Moving forward

Thank you for taking the time to read this book and congrats if you managed to finish it without cursing me or regexes too much. If the syntax and details still seem adrift in your memory, *that's fine*. Print out a cheat sheet. Look at it whenever the chance to use regexes come up and soon they'll be as familiar as you've ever done on a computer.

The trick is to find opportunities to practice. The first thing is to use a regex-enabled text editor as often as you can, either by opening text files or pasting text/spreadsheets into them.

Then, find the little things to fix. Maybe there's a place in the document where there's 3 spaces where there should be one. Instead of clicking on that space and hitting *Backspace* twice, just pop-up the **Find-and-Replace** dialogue and look for `\s+`

Or, if you've written a lengthy article and you just *know* you've misused "its" and "it's" *somewhere*. Paste your article into a text editor and do a quick search for `it' ?s`

I promised at the beginning that you'd find regexes useful even if you don't program. *However*, I promise that with just a little programming, regexes become *even more* powerful. Using a regex to find proper nouns or large numbers is pretty useful in a 100-page document. Imagine how useful it becomes when you write a script to search 1,000 100-page documents, all at once, with that same pattern.

But the beauty of regexes is that they're useful no matter what your technical level. So don't program if you don't feel like you can invest the time, but hopefully what you've learned so far will have sparked ideas and uses you had never thought of.

Additional references and resources

This book covers about 90% of the regexes I use in a day to day basis, but it by no means covers the many powerful (and crazy) uses that are out there. More importantly, I cover virtually none of the theory or inner-workings of why the regex engine behaves as it does, which is essential as your pattern-making becomes more and more complicated.

Luckily, there are a huge amount of resources for you to go as deep as you need to, and many of these resources are free and online. Here's a short list of ones I've found useful.

- <http://www.regular-expressions.info/>³⁴ – I think I owe my regex ability almost entirely to this site. It offers both easy-to-read overviews of the concepts and indepth walkthroughs when you need to know the details. It's no surprise that regular-expressions.info is almost always the first Google result for any regex-related query.

³⁴[Regular-expressions.info](http://www.regular-expressions.info)

- [Regular Expressions Cookbook](#)³⁵ – You thought my book was long? This invaluable reference, written by the same folks behind [regular-expressions.info](#), is 612 pages.
- [Learn Regular Expressions the Hard Way](#)³⁶ – another in the well-regarded series of learn-to-code books by Zed Shaw.

[Smashing Magazine’s Crucial Concepts Behind Advanced Regular Expressions](#)³⁷ – A nice primer on regex technique and details.

- [“Do I have that right? And more importantly, what do you think?”](#)³⁸ – this is one of the most cited Stack Overflow questions ever because it so perfectly embodies the conflict between the promise of regexes and cold hard reality. The top-responder’s ranting will increasingly make sense as you continue to use regexes for *everything*.
- [Putting regular expressions to work in Word](#)³⁹ – it’s not quite true that you *can’t* do regular expressions in Word. Word does have some wildcard functionality; however, its syntax is limited and is completely foreign to the syntax used in every other major language.

³⁵http://www.amazon.com/gp/product/1449319432/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=1449319432&linkCode=as2&tag=danwincom-20

³⁶<http://regex.learncodethehardway.org/>

³⁷<http://coding.smashingmagazine.com/2009/05/06/introduction-to-advanced-regular-expressions/>

³⁸<http://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags>

³⁹<http://office.microsoft.com/en-us/support/putting-regular-expressions-to-work-in-word-HA001087304.aspx>