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.

Contents

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[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 obselete?

The theory behind regular expressions is as old as modern computing[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.

[1] http://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.com[4]

- Dan Nguyen @dancow[5], danwin.com[6]

[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 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 resumé 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*.

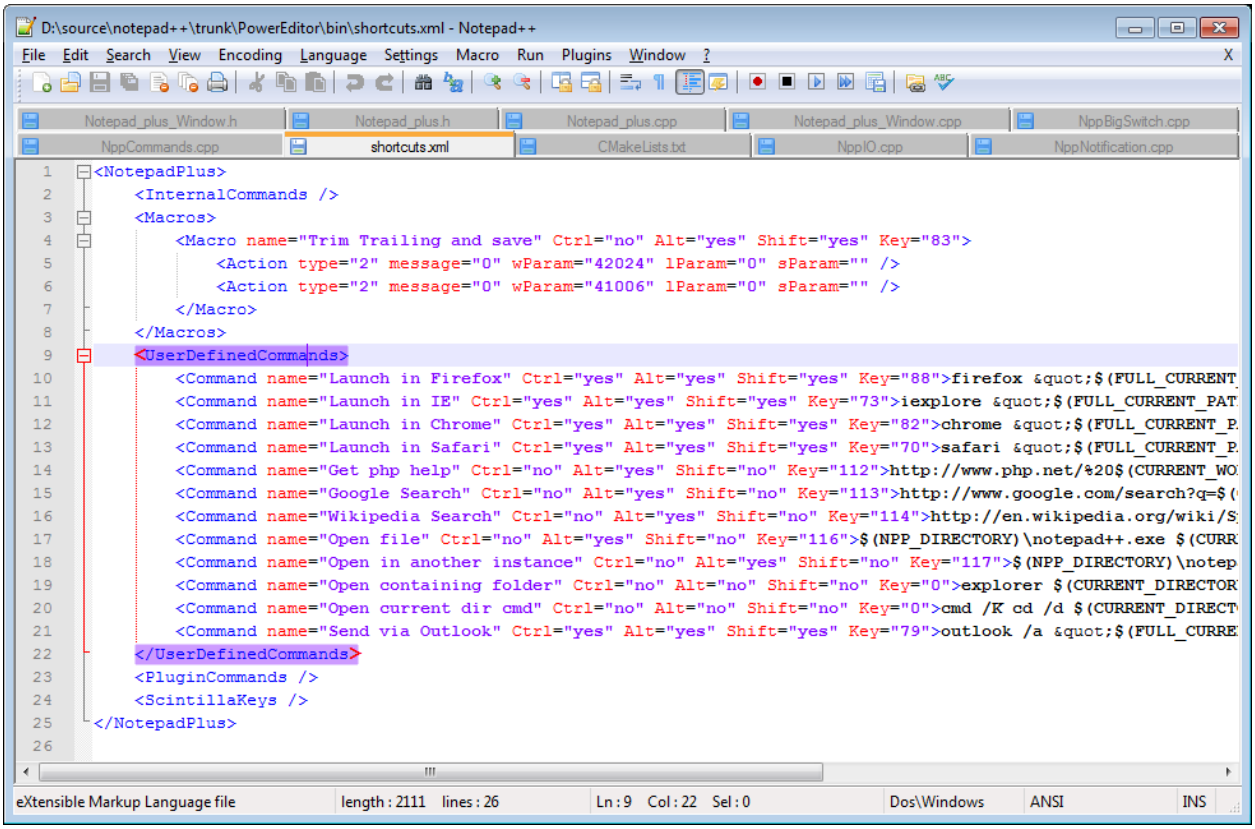
While your typical word processor can do a **Find-and-Replace**, it can't do it with regular expressions. That's the key difference here.

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.

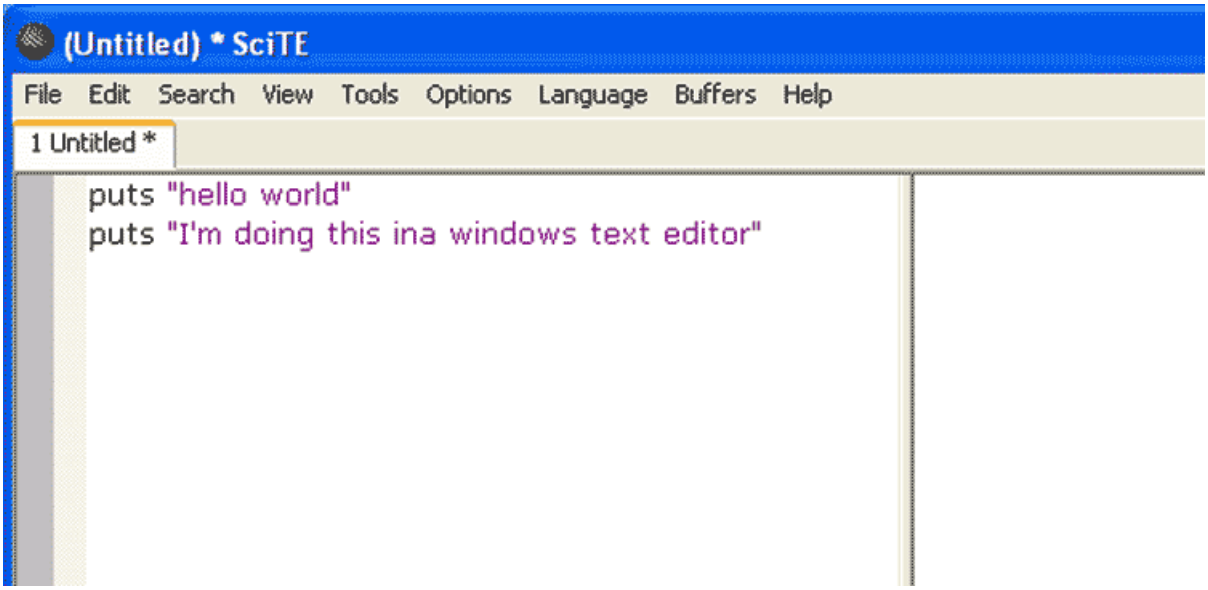
[7] http://notepad-plus-plus.org/



Notepad++

SciTE[8] 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.

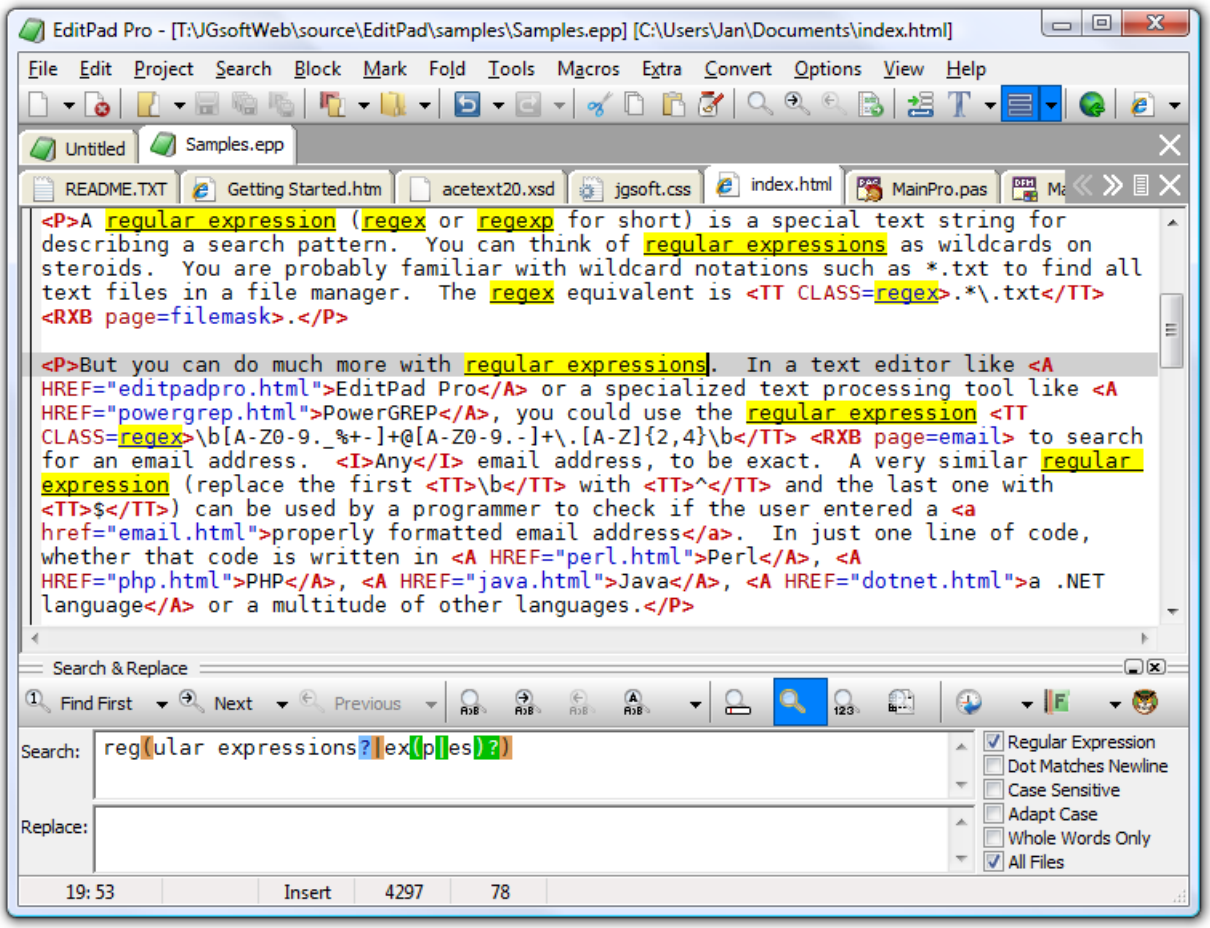
[8] http://www.scintilla.org/SciTE.html



SciTE in action

EditPadPro[9] 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

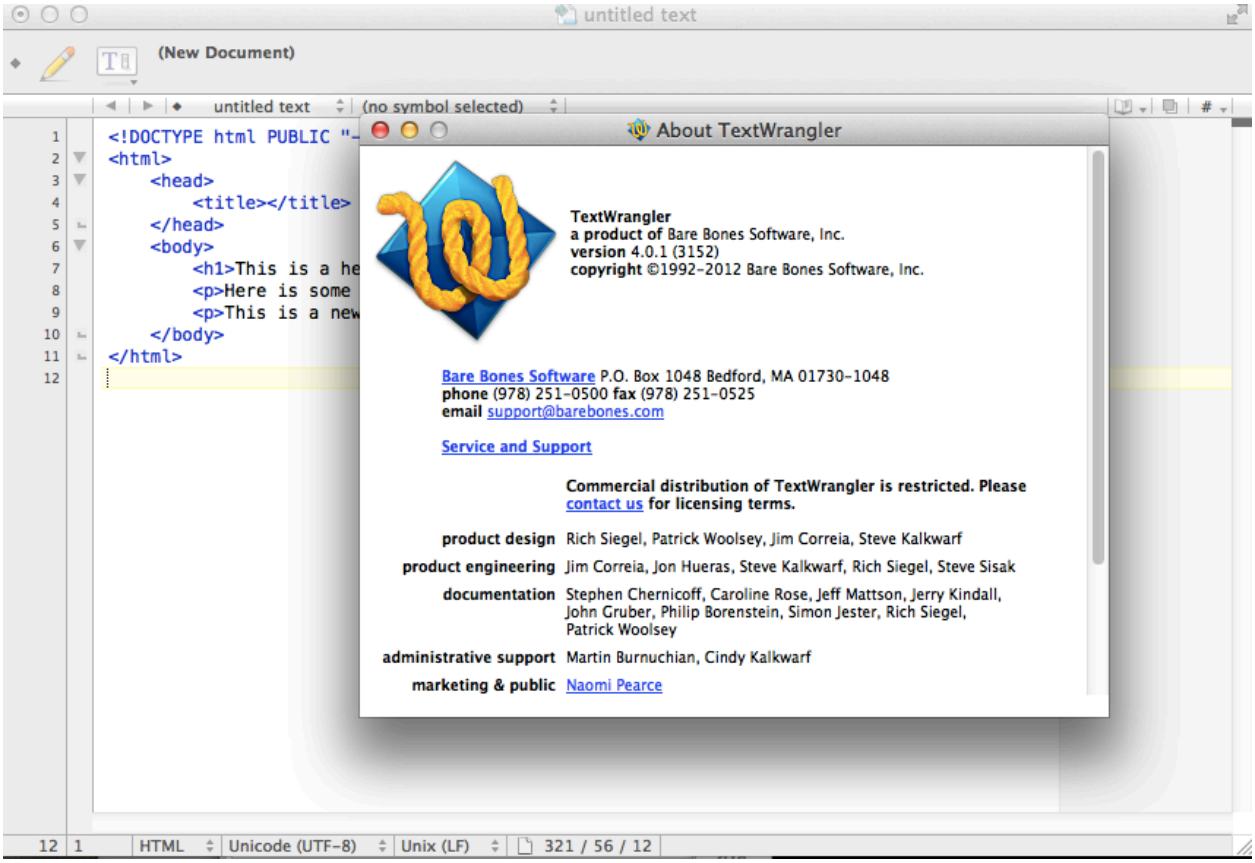
Mac Text Editors

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.

TextWrangler[10] 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/

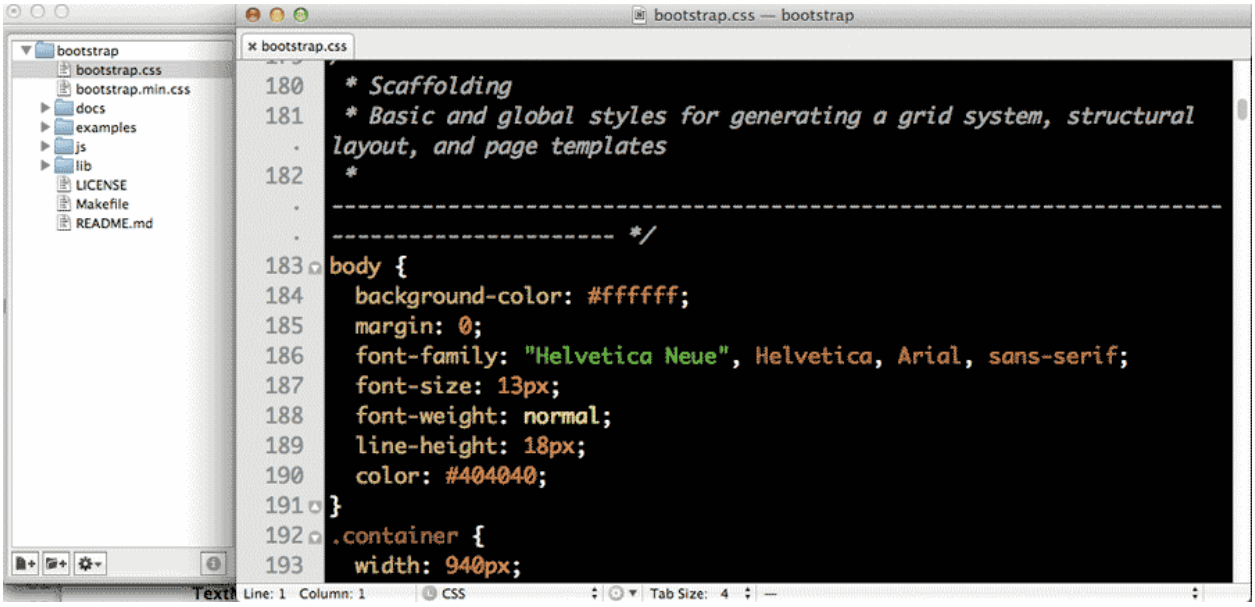
[11] http://www.barebones.com/products/bbedit/index.html?utm_source=thedeck&utm_medium=banner&utm_campaign=bbedit



TextWrangler splash screen

TextMate[12] has long been a favorite of developers. It comes in a commercial and open-sourced version.

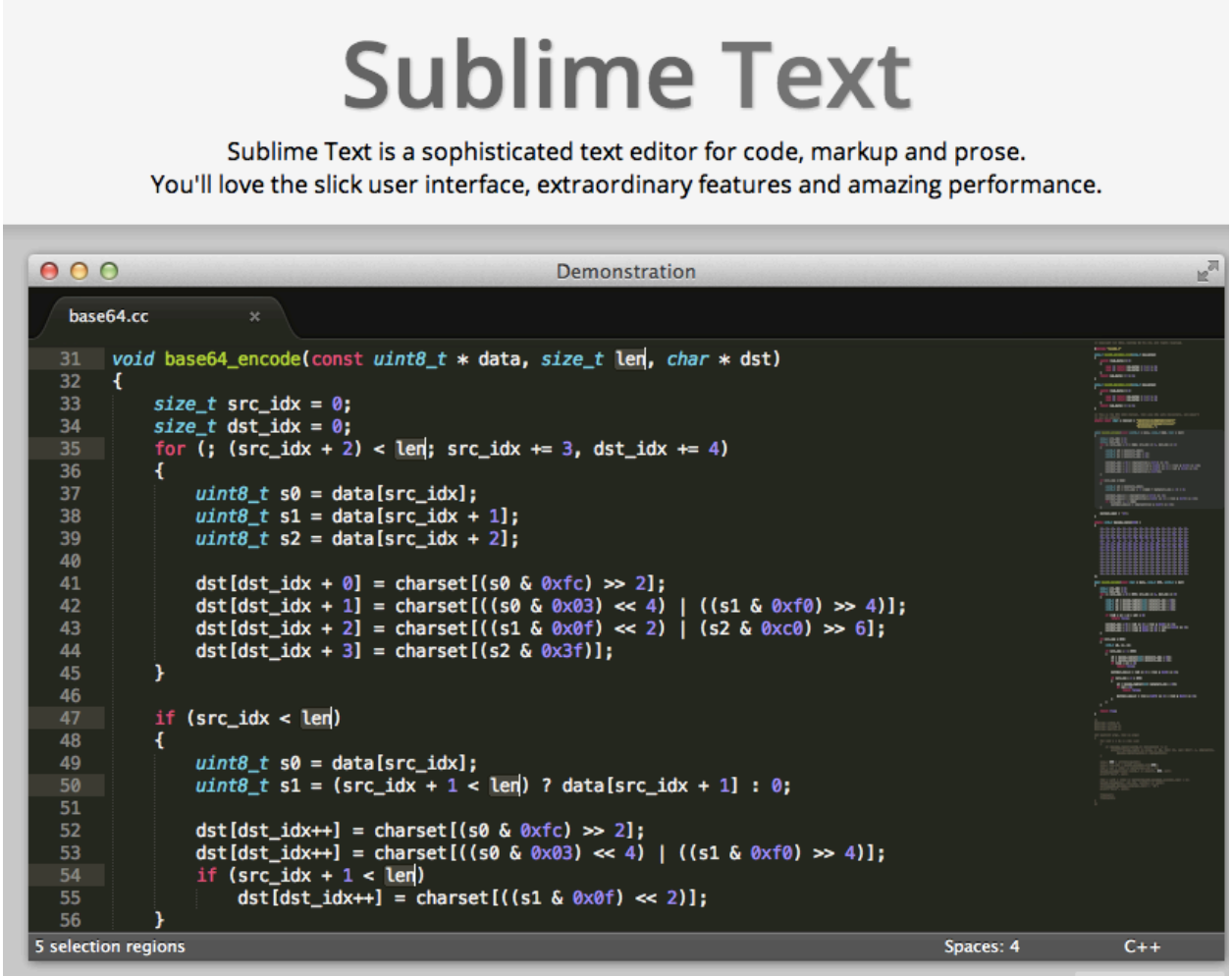
[12] http://macromates.com/



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 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.

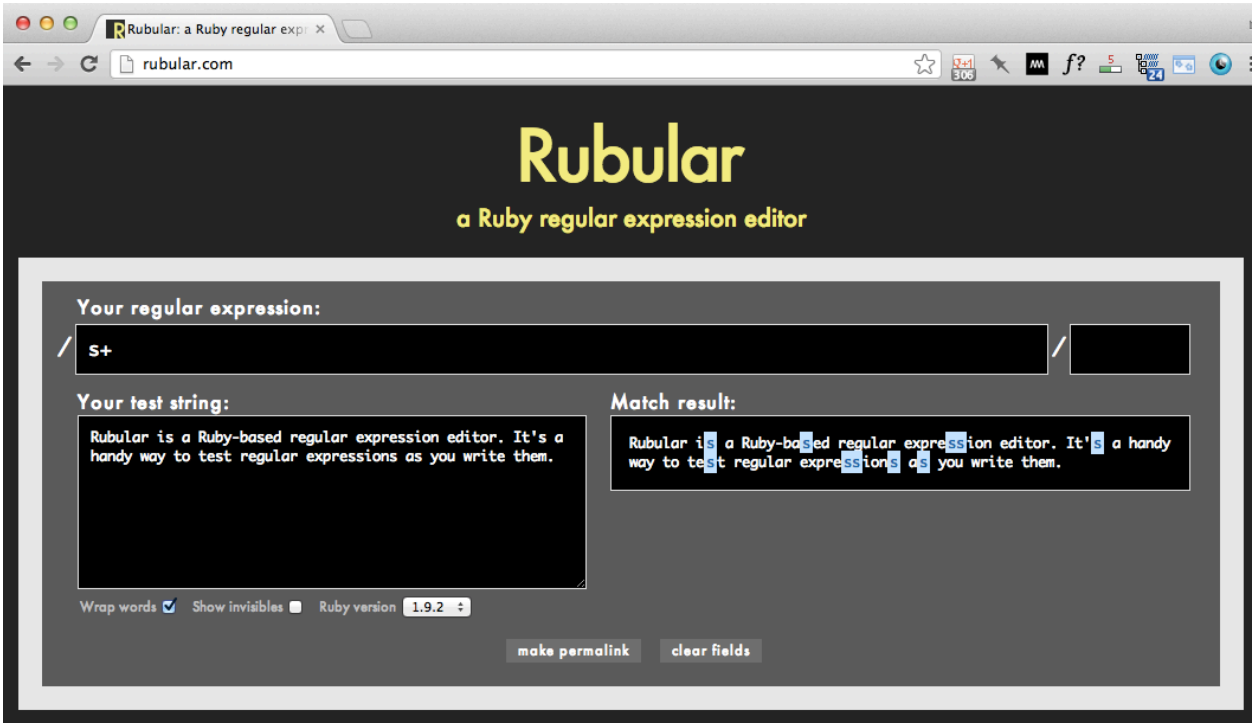
Online regex testing sites

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

Rubular[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

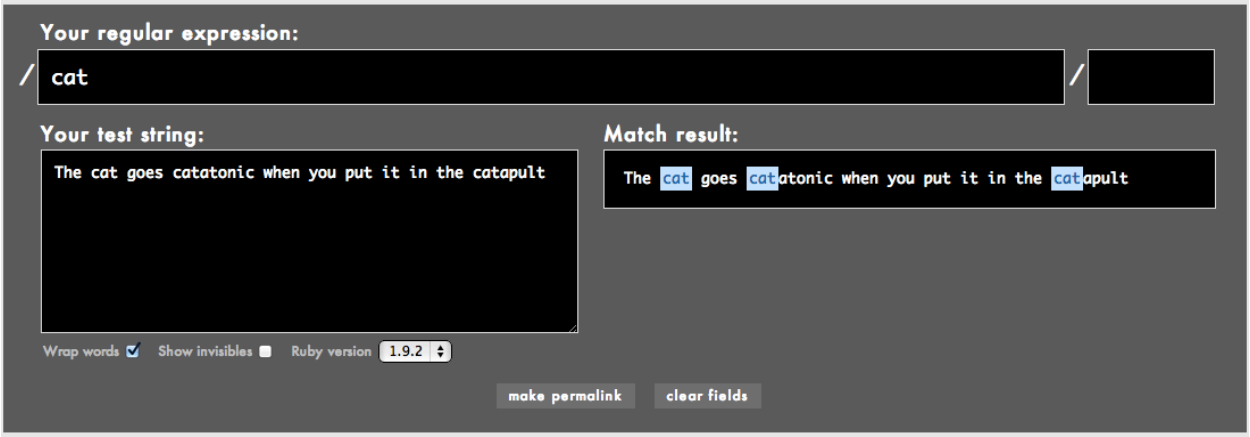
Now, in the top text input, between the two forward slashes (/ is the forward slash, not to be confused with the \ backslash), enter in the following regex:

```
cat
```

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

[14] http://rubular.com

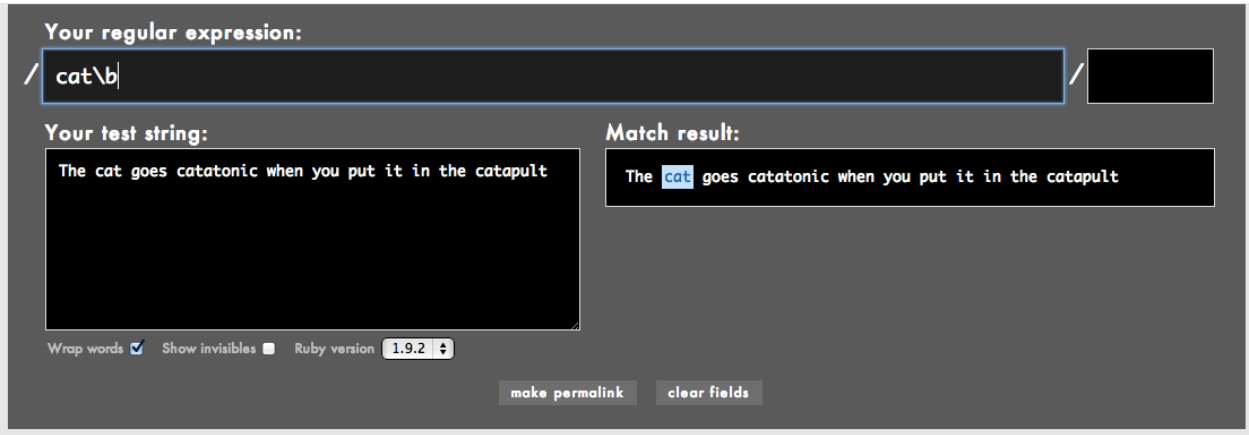
In Rubular, all instances where this simple regular expression matches the text – i.e., "cat" – are highlighted:



'cat' on rubular.com

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

```
cat\b
```



'cat\b' on rubular.com

Rubular will immediately highlight a new selection. In this case, it actually unselects the "cat" that is part of "**cat**apult" and "**cat**atonic". 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 "**mouse**apult" and "**mouse**atonic"

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

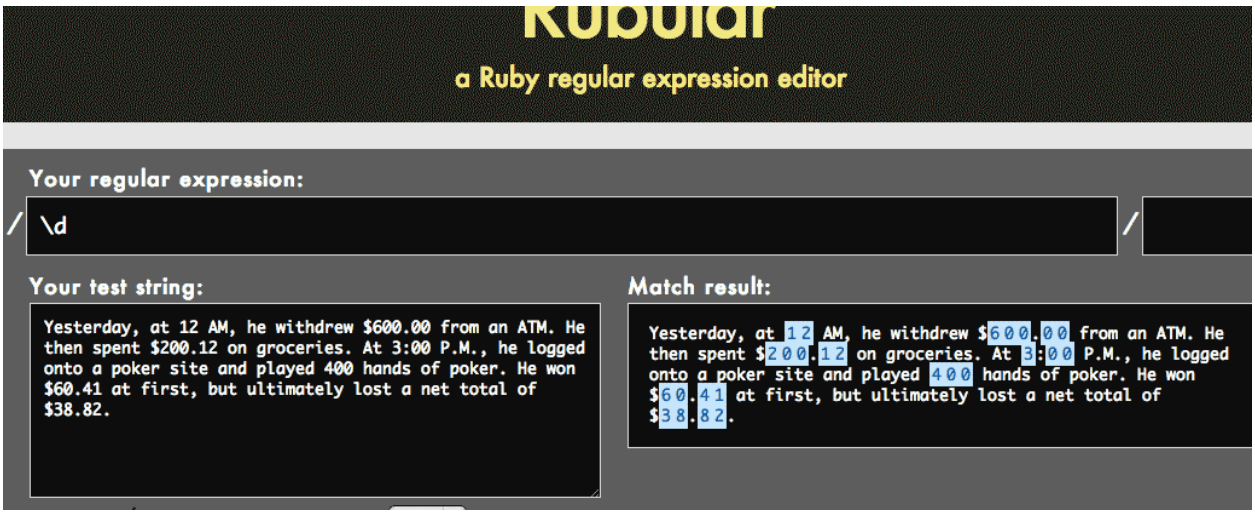
> Yesterday, at 12 AM, he withdrew $600.00 from an ATM. He then spent $200.12 on groceries. At 3:00 P.M., he logged onto a poker site and played 400 hands of poker. He won $60.41 at first, but ultimately lost a net total of $38.82.

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:

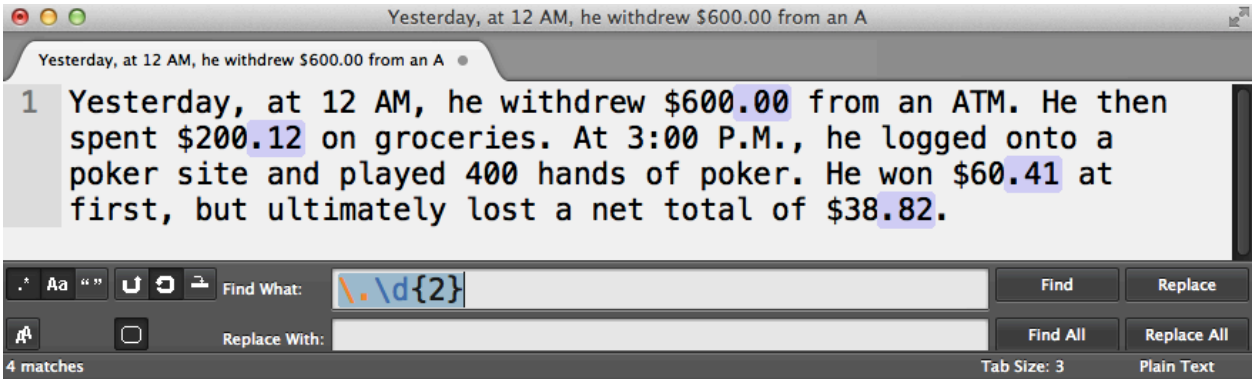
> Yesterday, at 12 AM, he withdrew $600.**00** from an ATM. He then spent $200.**12** on groceries. At 3:00 P.M., he logged onto a poker site and played 400 hands of poker. He won $60.**41** at first, but ultimately lost a net total of $38.**82**.

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 "**40**0 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

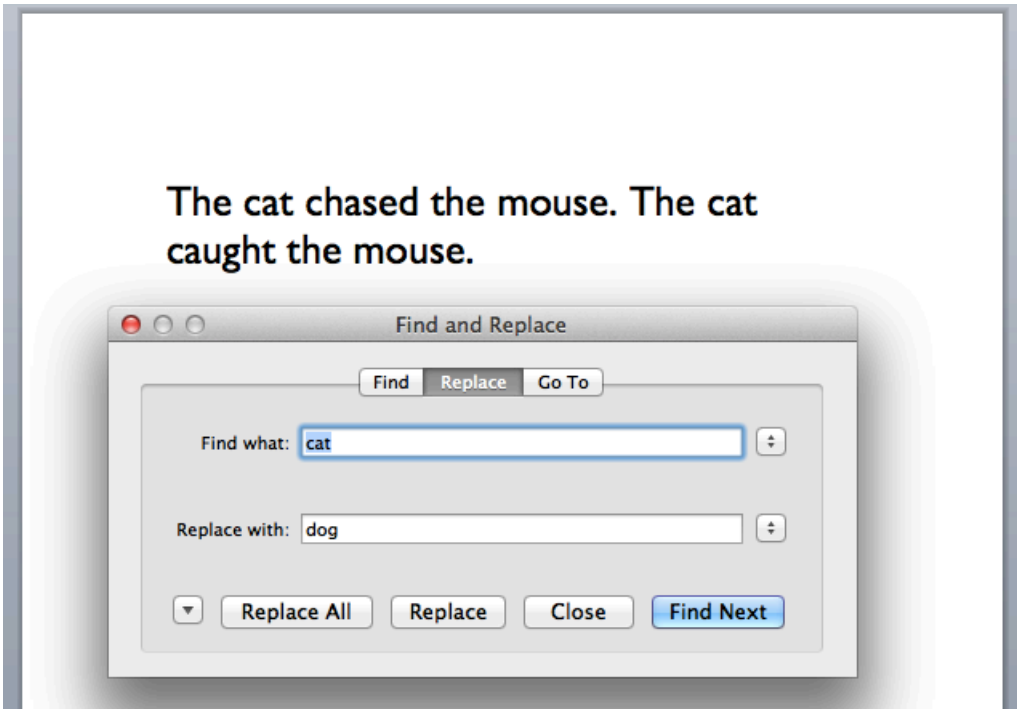
OK, now that we understand what basic **Find-and-Replace** is, let's see where it comes up short. Take a look at the following sentence:

> 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 **dog**acombs. The **dog** would reach a state of **dog**harsis if it were to **dog**ch 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 "**cat**harsis" and "**cat**acombs".

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:

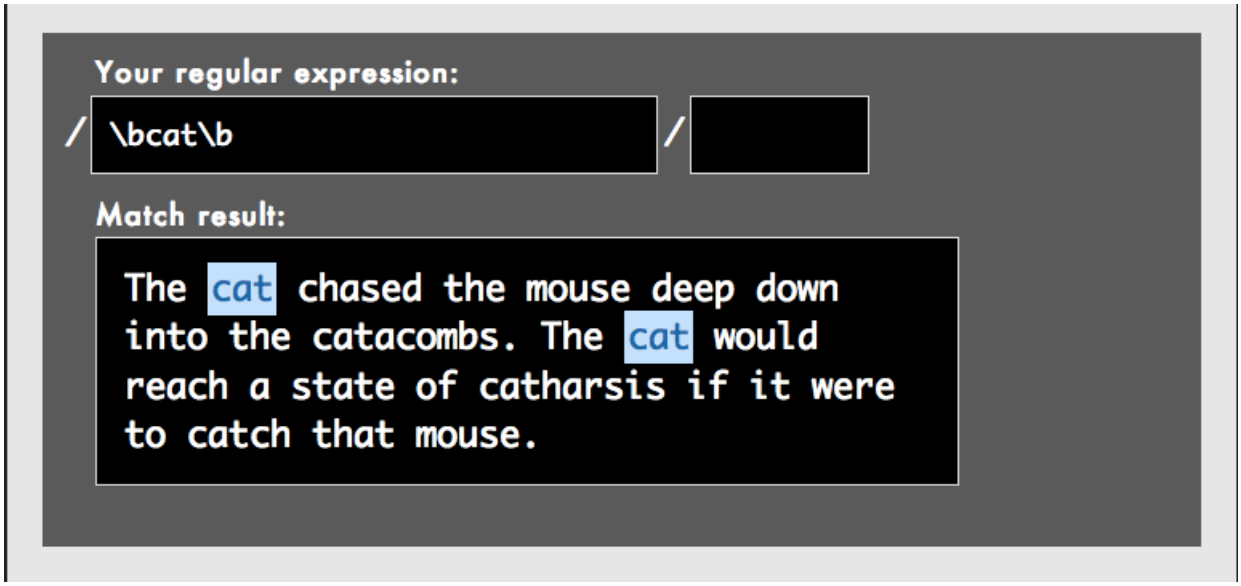
```
12/24/2012, $50.00
12/25/2012, $50.00
12/28/2012, $102.00
1/2/2012, $90.00
1/3/2012, $250.00
1/10/2012, $150.00
1/14/2012, $10.00
2/1/2013, $42.00
```

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

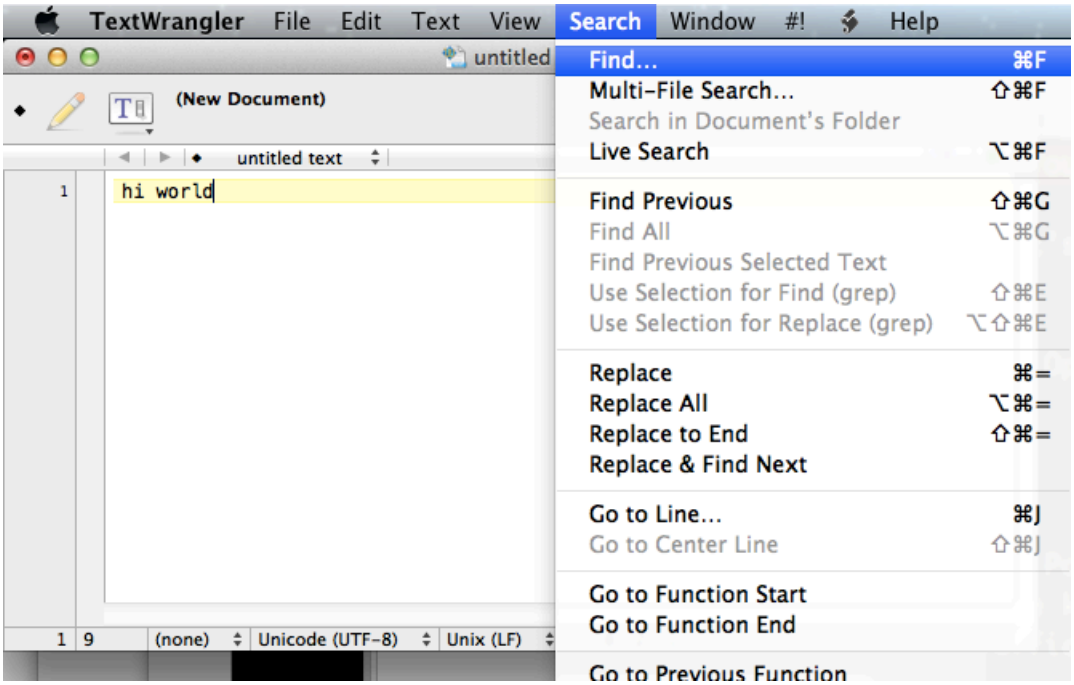
Let's try writing our first regular expression.

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

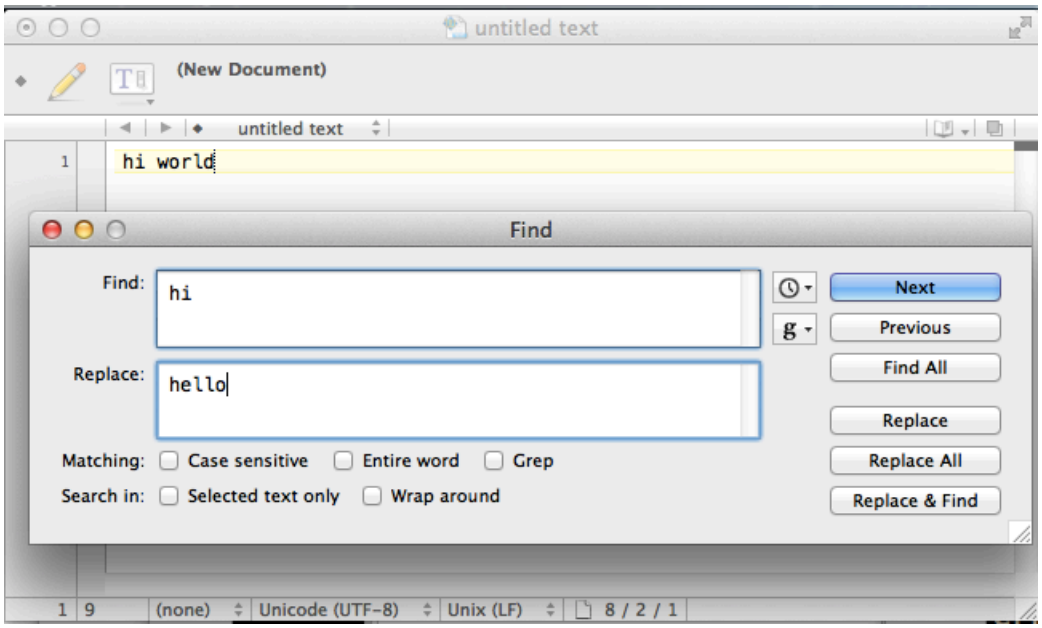
3. Our goal is to replace `hi` with `hello`. So in the **Find** field, type in:

   ```
   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 "t**hi**ghs"

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:

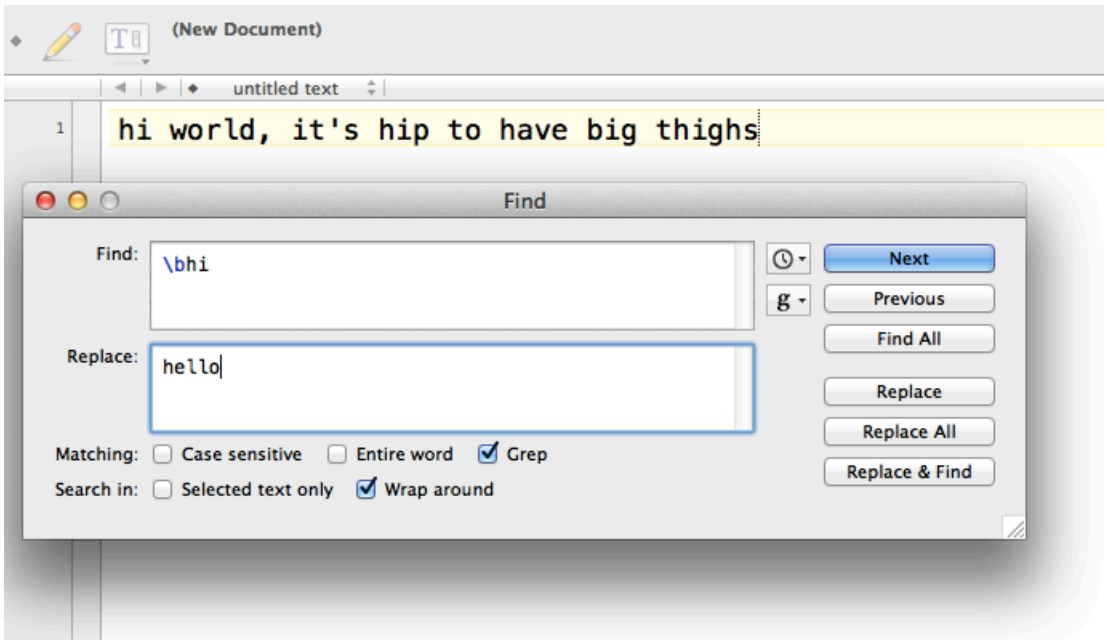
```
\bhi
```

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:

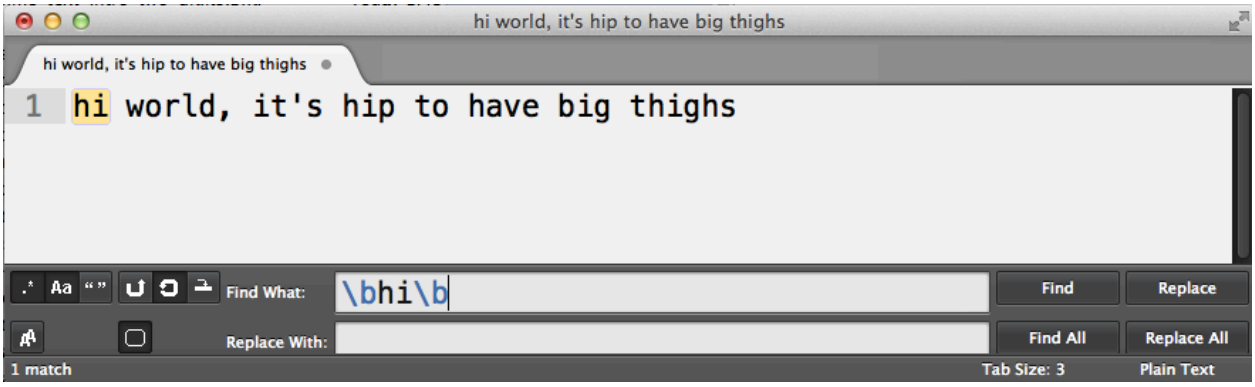
```
hello world, it's hellop to have big thighs
```

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 "t**hi**ghs"

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:

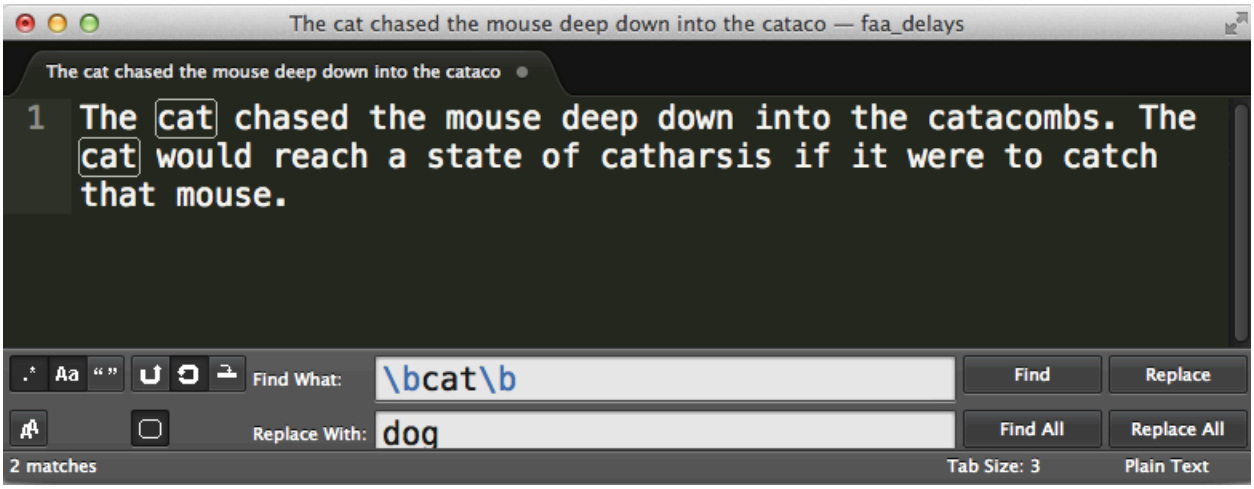
> The cat chased the mouse deep down into the catacombs. The cat would reach a state of catharsis if it were to catch that mouse.

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
```

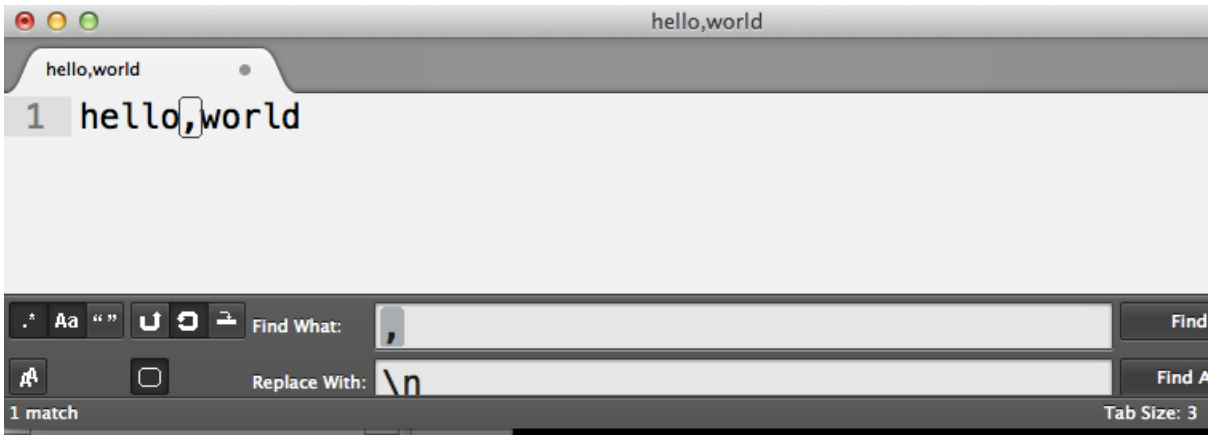
Replace that **comma** with a **newline**

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

Find `,`

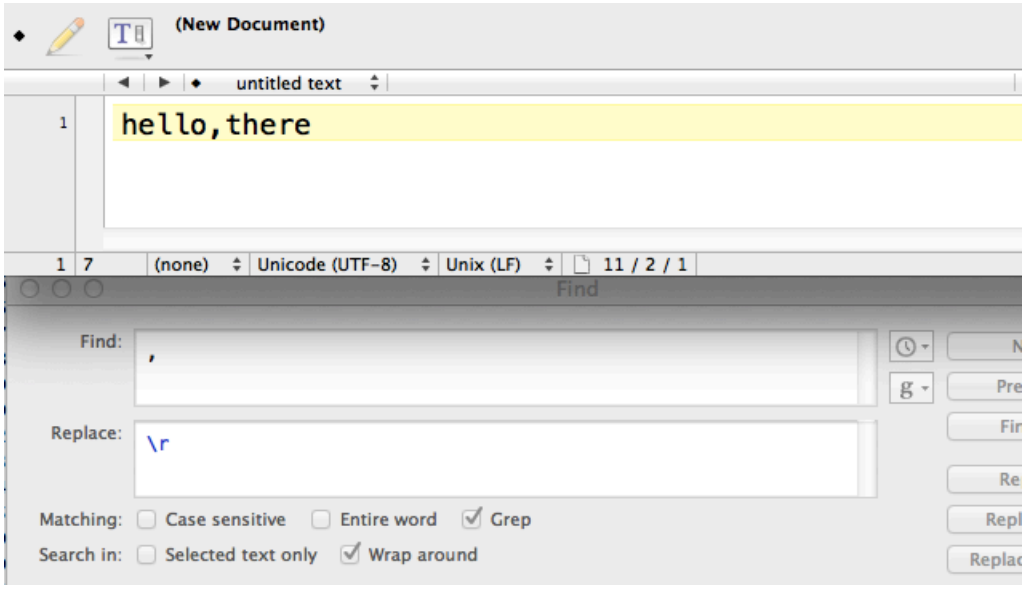
Replace `\n`

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:

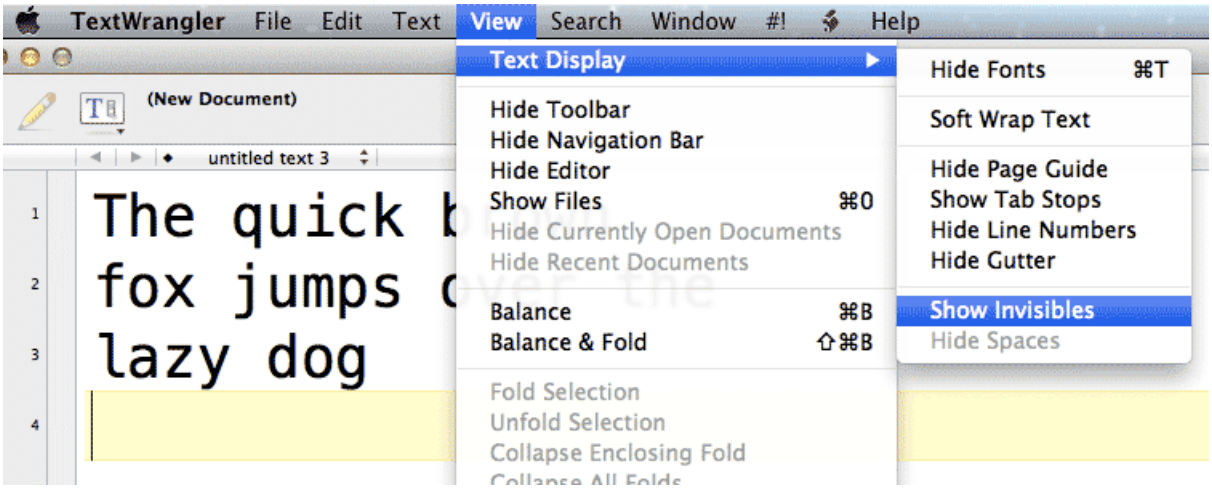
```
\n\n
```

And this is the second kind of "empty" line:

```
\n    \n
```

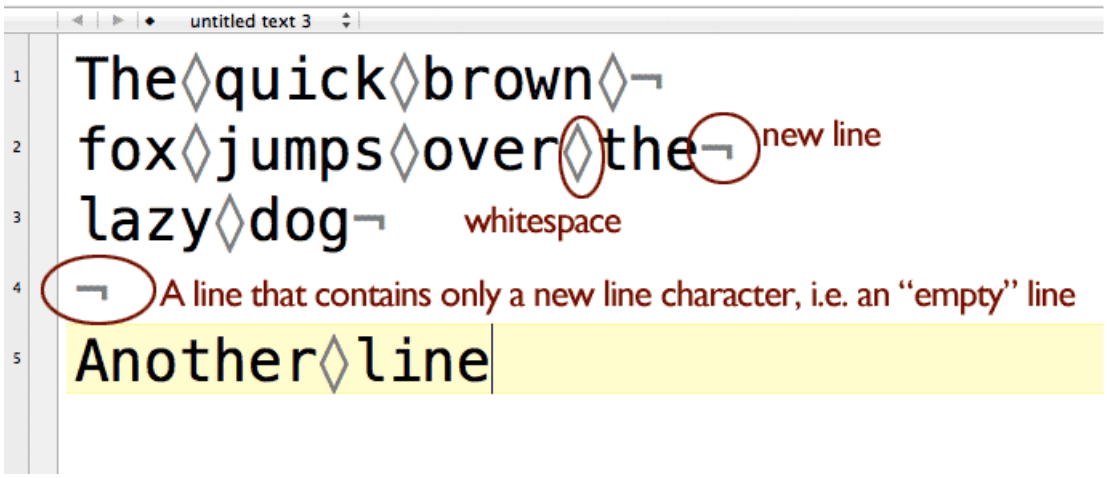
Of course, this is not the way it looks in a text editor. Both kinds of empty lines will look the same. But to a program, or to the regex engine that attempts to find consecutive newline characters, the two examples above are distinctly different.

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:



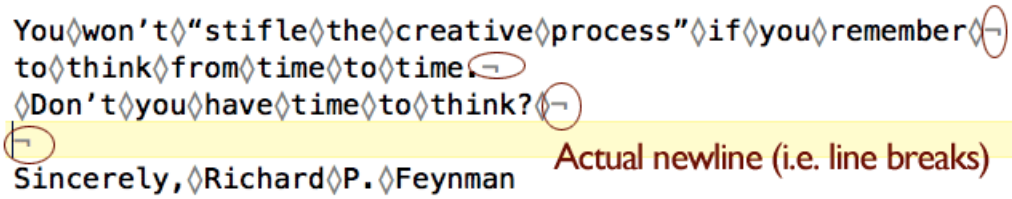
Finding the "Show Invisibles" option in TextWrangler

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



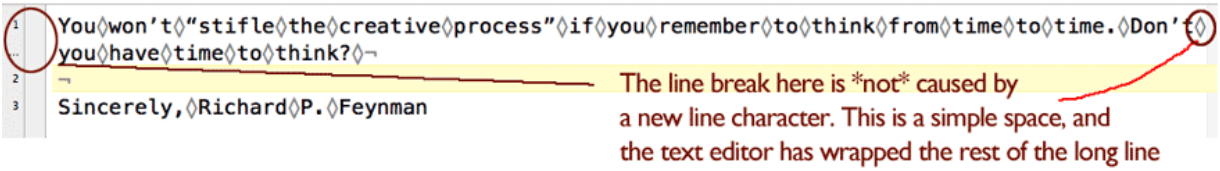
Invisible characters revealed, in TextWrangler

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
```

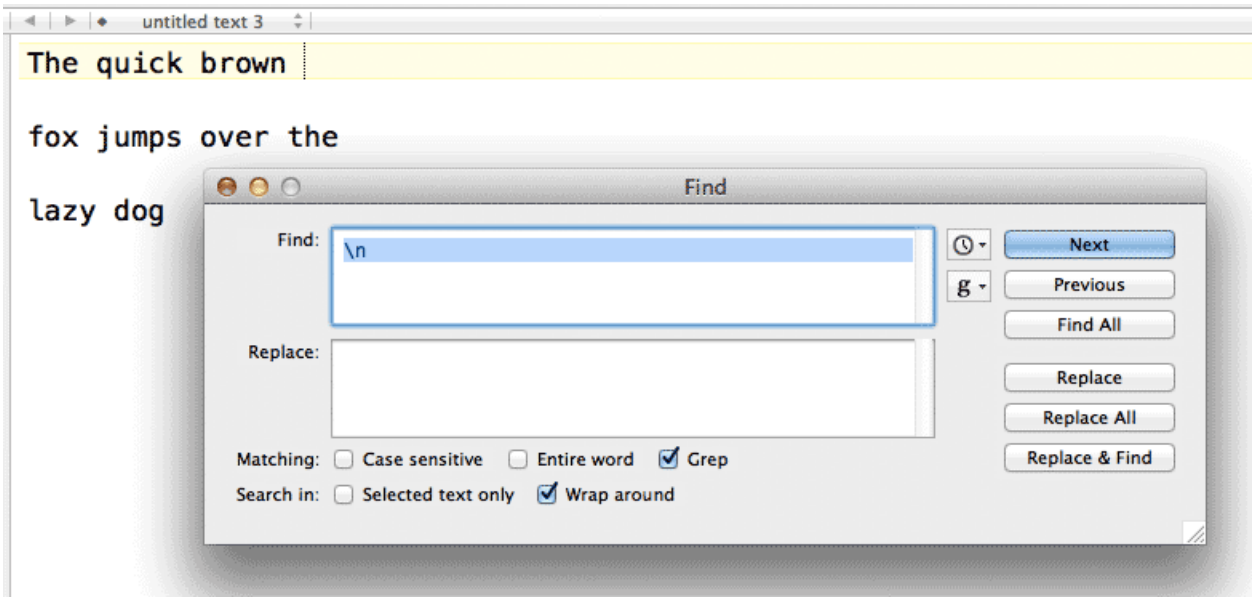
Answer First, let's describe what we want to fix in plain English: *We want to delete all lines that are empty.*

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++

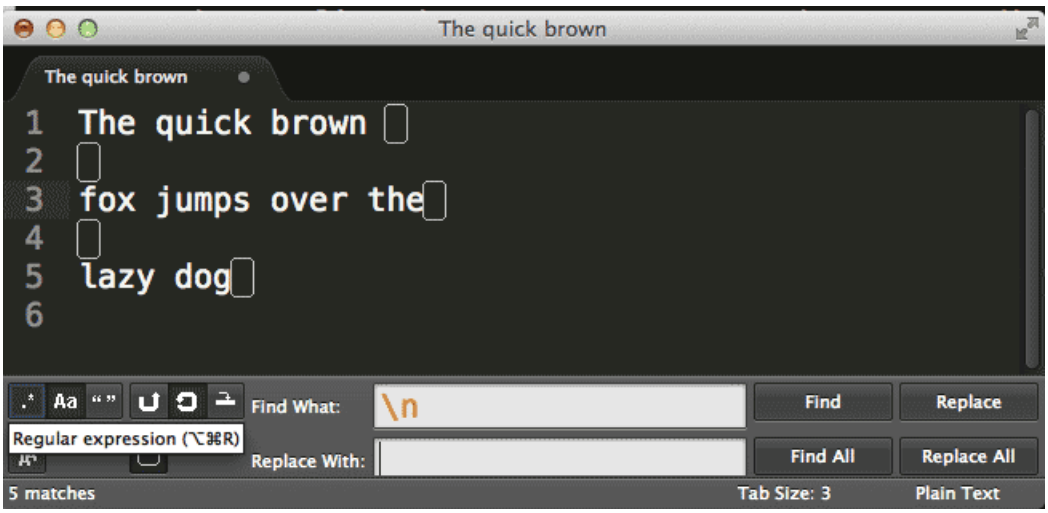
```
Todo: TKIMG
```

And in TextWrangler:



TextWrangler

And in Sublime Text 2:



Sublime Text 2

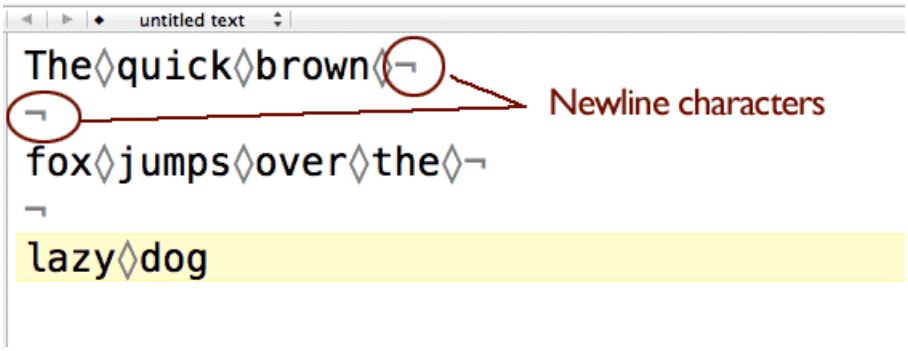
After hitting the **Replace All** button, your text window should look like this:

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.

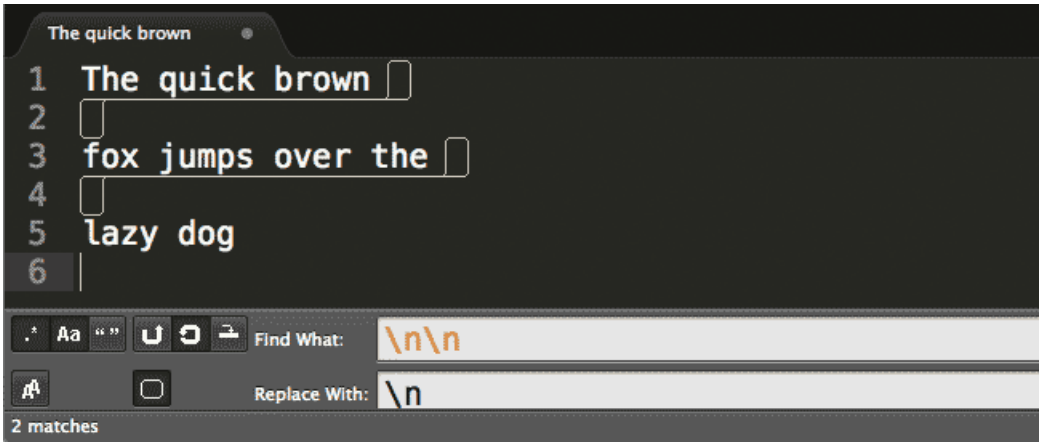
In the **Find** field, change the pattern to:

```
\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 Feymnan'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:

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

```
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?
```

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 `aaaaaaaaa`

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:

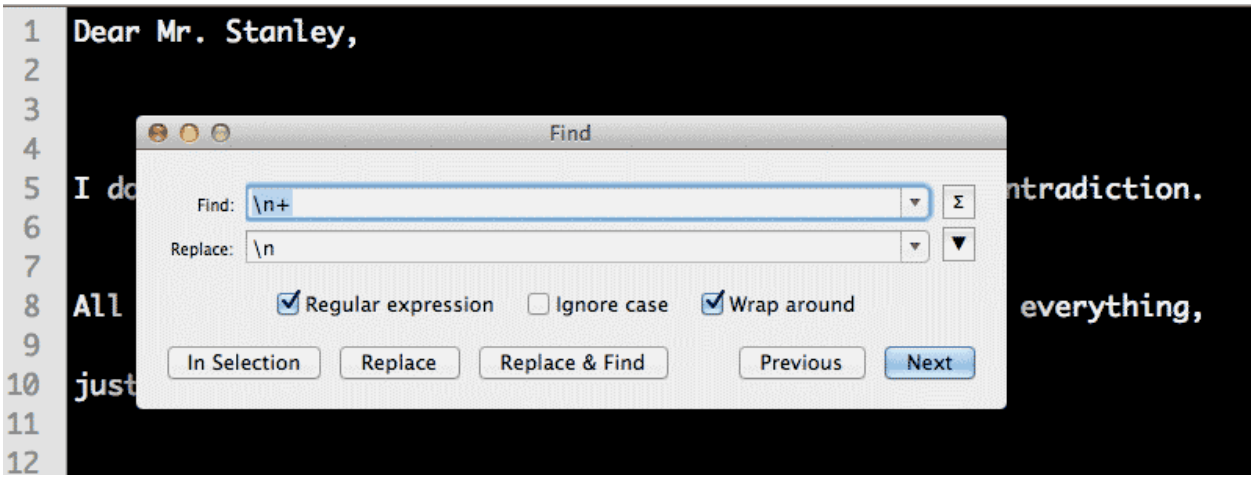
```
Dear Mr. Stanley,
I don’t know how to answer your question -- I see no contradiction.
All you have to do is, from time to time -- in spite of everything,
just try to examine a problem in a novel way.
You won’t “stifle the creative process” if you remember to think
from time to time. Don’t you have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

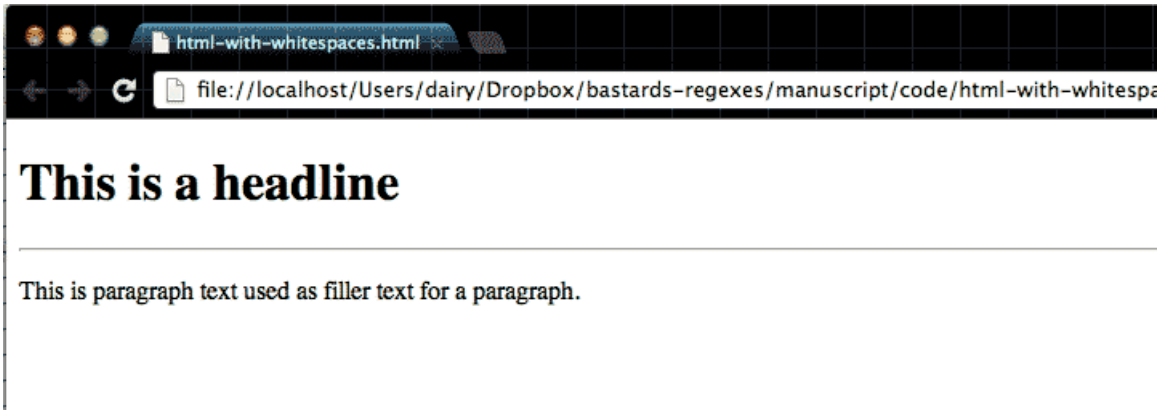
And set **Find** to: `\n+` and **Replace** to: `\n`

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:



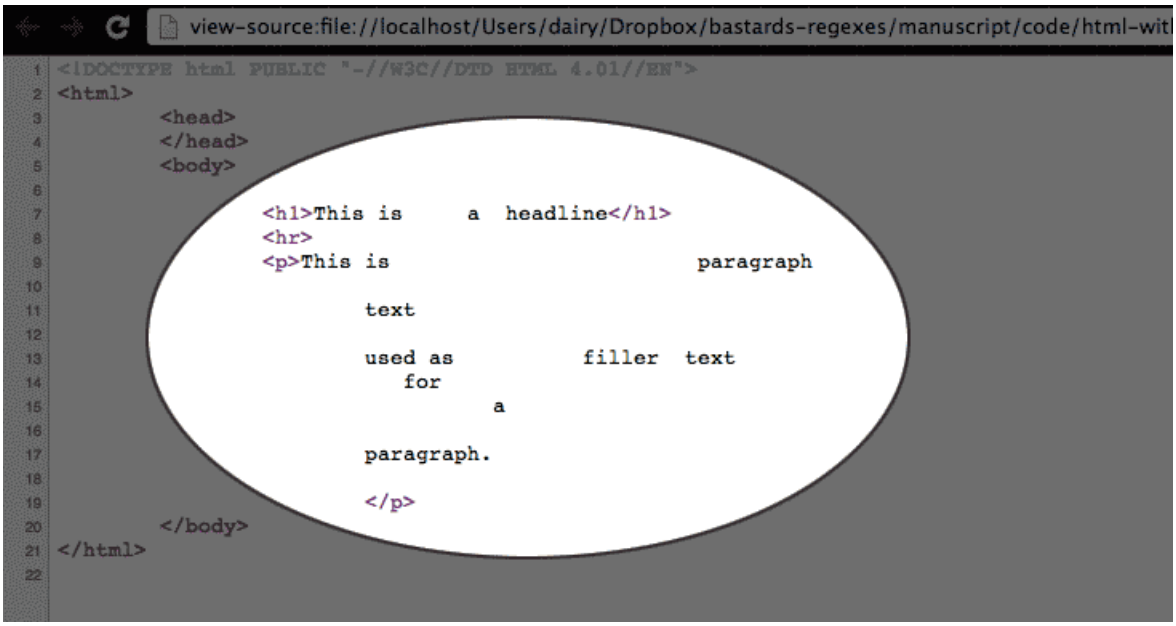
Simple webpage

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.

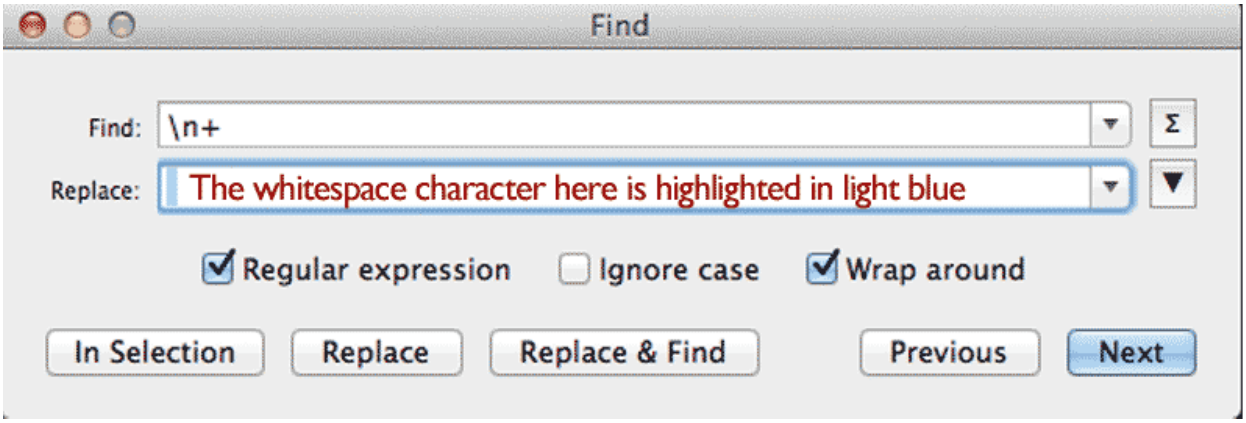
Answer

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:

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

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.