Preface

This is a rough in-progress dump of the book. The grammar will probably be bad, there will be sections missing, but you get to watch me write the book and see how I do things.

Finally, don't forget that I have href{http://learnpythonthehardway.org}{Learn Python The Hard Way, 2nd Edition} which you should read if you can't code yet.

Introduction: Giants In Tiny Pants

Do you like word problems? You know those annoying "math" questions you had to study in class simply because the SAT used them. They went something like this:

"A train leaving a station at 10:50pm from San Francisco is travelling at 40MPH. Another train leaving Chicago at 12:00pm is going 60MPH. At which point will the two trains cross paths."

Even today, after studying years of advanced mathematics, statistics, and computer science, I still can't solve these things. The reason is they're an attempt to hide an actual equation from you with convoluted English wording. Rather than just give you a straight math problem, like $ax + b = c$ they want you to "reverse engineer" the real equation from this pile of ambiguity. In an attempt to simplify mathematics these types of questions actually end up obfuscating the real symbolic language behind them.

Imagine if all mathematics was like this where every single problem you encountered, no matter how simple or complex, was only written as English because nobody understood the symbols. This would drive you crazy, even if you didn't know mathematics. Without the symbols of mathematics we'd find mathematics degenerate to one Jacques Derrida style "obscurantisme terroriste" paper after another. They'd be no better than Philosophers!

Thankfully, humans invented symbols to succinctly describe the things that symbols are best at describing, and left human languages to describe the rest.footnote{Thankfully, the things that can be described with symbols is pretty small or else nobody would be able to read.} The symbolic languages act as building blocks to construct complex structures in a short space that would take entire books in English to describe.

The power of symbolic languages is that, once you learn them, they can describe things more quickly and accurately than a human language can.

Regular Expressions Are Character Algebra

Regular expressions, or regex, is a symbolic language that describe how to identify a sequence of integers (or, more practically, characters) as matching a required set. Compared to mathematics regular expressions are pathetically tiny. There's only a few symbols to learn, and they have exact specific meanings. This makes them easy to learn, given that you actually put in the time to learn to read and write them.

Effectively a regular expression is the algebra equation to a programming language's word problem. With them you can describe things that would take mountains of code if you were to write it by hand. This is why they exist, but being symbolic languages most people simply hate them because most people are never taught how to learn a symbolic language.

How To Learn A Symbolic Language

When you read a line of code like this:

```
name.match(/^[\da-z]+\d$/);
```

You probably actually see this:

```
name.match(justabunchofrandomjunkallsmashedtogether);
```

Your eyes most likely hit that part inside the parenthesis and try to process it
as if it were one cohesive word. The truth is each character inside the parenth
esis is actually like a whole word or even a phrase. What trips you up is you're
used to reading sentences where there's spaces between whole words and periods
that end cohesive thoughts. When you try to read this, you have to stop and teas
e the this one "word" apart into what it's actually doing.

However, if you've been programming for a while, you're already processing and u
nderstanding a symbolic language, just one that's a bit closer to a human langua
ge. What you need to do is take this tiny bit of symbolic literacy skill and tur
n it into a real skill at reading symbols. To do this you need to do what you do
when learning any language: Translate it to and from the language you're curren
tly know the best.

Imagine if you could write your regular expressions like this:

```
name.match(/
        ^    # from the beginning
        [    # a set of chars containing
          \d # any digit
          a  # the letter a
          -  # through
          z  # the letter z
        ]    # end class
        +    # one or more
        \d   # any digit
        $    # to the end
        /);
```

Now it makes a bit more sense because every symbol in the regex is explained in
English. You can now say, "Match from the start a set of one or more digits or a
-z followed by any digit to the end of the string." You could even get more succ
inct and say, "Match completely one more digits or lowercase letters followed by
a digit."

In this book I'm going to teach you to read regular expressions by showing you h
ow to convert them back and forth from English such that you can break down any
that you find. You'll also learn good habits for writing regular expressions, an
d more importantly when to not use them. They're handy tools, but like any tool
they're for a particular job.
The Giants In Tiny Pants Problem

The problem that all symbolic languages have is an expert in them, using time an
d continuous evolution, can eventually describe everything as one gigantic equat
ion. They start with something simple and go bad as they tack on hack after hack
to handle more and more edge cases. In regular expressions you end up with the
infamous infamous Mail::RFC822::Address monstrosity. Here's just the top 6 lines
of all 82 lines in this one regular expression:

```
::
    (?:(?:rn)?[ t])*(?:(?:(?:[^()<>@,;:\".[] 000-031]+(?:(?:(?:rn)?[ t] )+|Z|(?=
[["()<>@,;:\".[]]))|"(?:[^"r\]|\.|(?:(?:rn)?[ t]))*"(?:(?: rn)?[ t])*)(?:.(?:(?:
rn)?[ t])*(?:[^()<>@,;:\".[] 000-031]+(?:(?:( ?:rn)?[ t])+|Z|(?=[["()<>@,;:\".[]
]))|"(?:[^"r\]|\.|(?:(?:rn)?[ t]))*"(?:(?:rn)?[ t])*))*@(?:(?:rn)?[ t])*(?:[^()<
```

```
>@,;:\".[] 000-0 31]+(?:(?:(?:rn)?[ t])+|Z|(?=[["()<>@,;:\".[]]))|[([^[]r\]|\.)*
](?:(?:rn)?[ t])*)(?:.(?:(?:rn)?[ t])*(?:[^()<>@,;:\".[] 000-031]+ ... for 82 li
nes ...
```

This isn't regular expression's fault, it's the fault of people abusing the tool when there's other tools more suitable. In this case, a simple lexer that used smaller regular expressions would work better and most likely be faster and less error prone. More importantly, a lexer could report errors and tell you where something failed to pass. Tools like lex, re2c, and Ragel all make this easier than the above.

I call this the "giant in tiny pants problem" because you're taking something that's actually a giant complex piece of code, and trying to cram all of it in the tiny pants of a regular expression. In the above example they're trying to cram an email address parser into one a regular expression, and what they end up with is not a succinct clean expression, but still a giant bursting at the seams.

Slaying Giants In Tiny Pants With Parsing

The key to using regular expressions correctly is to know where their usefulness ends and when you need to bust out a lexer. You also need to know where a lexer falls down and when a parser is the right tool. When you use regular expressions to simplify creating lexers that feed into simple parsers you then have a set of tools for cleaning and accurately parsing text without going insane.

In this book I'm going to subversively teach you parsing, but I'm going to be very practical and straight forward about it. No NFA to DFA conversion. No crazy explanations of push down finite state automata. Just practical code that gets you introduced to the basics of parsing, understanding the core theory, and then actually using them to get work done.

How To Use This Book

As with all of my "Learn The Hard Way" books, the best way to use this book is to do the work. That means sitting down with each exercise, actually typing all of the code in, reading about what you did, and doing the extra credit. You cannot learn these concepts without actually doing the work and practicing them. Otherwise you might as well just save your money and continue to suck at programming.

A good way to do the book is to go "low and slow". Rather than sit down on Saturday and cram 8 exercises out in a 10 hour marathon, you should spend 1 hour a night and do 1 or 2 exercises. Sometimes you'll blaze through exercises and other times you'll get stuck and need to study. Some exercise you might have to struggle with for a week before moving on. But, your average should be about 1 a day and consistent steady study is more important than cramming.

The reason I give this advice is you've got to train and retrain your brain to understand this new language, and the best way to do that is with repetition and practice. Otherwise you just cheat yourself and end up thinking you've learned something when you've really done nothing but become familiar with the material.

Finally, do not copy-paste the regex! The text samples I give you are fine to copy paste, but the regex code I have you type you must type them. You have to actually type it or you won't learn it. This will seem like painful work at first if you're used to copy-pasting everything, but after a few weeks it'll be easy and after three months you'll really know the stuff.

Enjoy the book, and if you need help, just email help@learncodethehardway.org and I'll give you pointers.

Exercise 0: The Setup

Teaching regular expressions can be difficult because they are typically embedded into a programming language or a tool like sed. This is compounded by differences between most of these implementations. Rather than have you get stuck in various different flavors of regex hell, I've created a very small project called Regetron for experimenting with regular expressions. It uses Python's version of regex, but those are pretty close to the 90% of regex you'll actually use.

To install Regetron simply do this:

    Install Python if you don't have it.
    Install PIP and distribute using http://pypi.python.org/pypi/distribute.
    Run pip install regetron

Alternatively, you can go to the Regetron project page and get the source to install like this:

```
$ git clone git://gitorious.org/regetron/regetron.git
$ cd regetron
$ sudo python setup.py install
```

Once you have Regetron, you try running it and doing some things with it:

```
$ regetron
Regetron! The regex teaching shell.
Type your regex at the prompt and hit enter. It'll show you the
lines that match that regex, or nothing if nothing matches.
Hit CTRL-d to quit (CTRL-z on windows).
> !data "Hello World!"
> .*W.*
0000: Hello World!
> Cats
>
```

How To Use Regetron

How this works is you give Regetron a file or a string of data, and then you type in regular expressions (regex). You can see this when I type !data "Hello World!" to setup a string to work with, then did a regex against it .*W.*. If a regex matches then it will print out the lines that it found matched. If nothing matches then it prints nothing. That's what happened when I typed Cats.

Regetron has a few commands and options as well:

    If you give it a file on the command line it will load that: regetron somefile.txt
    If you hit ENTER to make a blank line it will go into "verbose mode" which we'll use heavily in the book. To finish in verbose mode and run the regex just enter a blank line.
    The command !data EXP will run any Python expression (EXP) and and set that as the data. Try !data "LOTS OF ME" * 100 and then try regex .*ME.* to see that it duplicated the line 100 times.
    The command !help will print the available commands.
    !load FILE will load the given file for your data.
    !match toggles whether regex are run in match vs. search mode. We'll cover that later.
    If you have readline installed, then Regetron will give you a readline scrollback and edit feature.
    To exit, just use CTRL-d (or CTRL-z on Windows) and it'll exit.

Extra Credit

For this exercise just play with Regetron and make sure you're comfortable with it.