Regular expressions

Justin Myers (@myersjustinc),
news automation editor, The Associated Press

What is a regular expression?

A regular expression is a common, compact way to represent some sort of pattern in text.

You can use one to:

- Help you search within a document
- Reformat or otherwise clean up an inconsistent or "dirty" data set
- Extract structured data from plain text

Many common programming languages, text editors and search tools support regular expressions, so you can use them in whatever environment you find most comfortable.

The building blocks

As complicated as regular expressions can look, they're all ultimately made up of smaller parts:

- Literals
  - Letters and numbers refer to themselves. **`a`** refers to the lowercase letter "a", **`3`** refers to the numeral "3", etc.
- Escaping
  - A ton of characters in regular expressions have special meanings; to override that meaning for a particular character, put a backslash (**`\`**) before it.
  - For example, a period (**`.`**) matches any character, but an escaped period (**`\.`**) just matches a literal period character.
  - Going the other direction, a lowercase **`d`** matches just that letter, but an escaped d (**`\d`**) is shorthand for "any digit".
- Character classes
  - You can specify a class of characters (a list of specific characters to match) by surrounding them in square brackets; for example, **`[aeiou]`** would match any vowel.
    - These also can contain one or more ranges, where a range is a set of characters in consecutive ASCII order:
      - **`[0-9]`** would match any digit.
      - **`[A-Za-z]`** would match any letter.
      - **`[a-z_]`** would match any lowercase letter or underscore.
    - If a caret (**`^`**) is the first character within the brackets, the entire class is negated; for example, **`[^aeiou]`** matches any character *except* those five vowels.
    - If you want your class to include a literal caret (rather than negate everything else that's there), you either can escape it (**`\^`**) or put it somewhere other than the beginning; similarly, if you want to match a literal hyphen (rather than define a range), you can escape it or put it at the beginning or the end of the class so it isn't between two other characters.
  - On the extreme end of things, a period (**`.`**) matches any character at all (except newlines, in most cases).

    - Some common shorthand for frequently used classes (but might differ between programs):
        - **`\d`** is equivalent to **`[0-9]`** and matches any digit.
        - **`\s`** matches whitespace characters, like space and tab.
        - **`\w`** is equivalent to **`[A-Za-z_]`** and matches "word characters", which means letters and underscores.
- Groups
    - You can surround a part of a regular expression in parentheses—**`(`** and **`)`**—to treat it as a unit.
    - One common use for this is to provide alternatives, which are sort of like character classes but for more than one letter at a time; for example,
      `this (and|or) that` uses the pipe character (**`|`**) to provide two options for the conjunction between **`this`** and **`that`**. (Without the parentheses, the pipe would've acted on everything before and after it, so it only would have matched the phrase **`this and`** and the phrase **`or that`**.)
    - Another is to save those groups so you can refer to them later, such as in the replacement step of a find-and-replace operation. These often are referenced by their position; for example, in **`(\d\d\d)-(\d\d\d)-(\d\d\d\d)`**, you might use **`\1`** to refer to the first three digits, **`\2`** to refer to the second three digits,
      and **`\3`** to refer to the last four digits.
- Quantifiers
    - Several symbols exist to let you specify how many times in a row to match a given character, class or group.
    - A question mark (**`?`**) matches zero or one occurrence; for example, **`colou?r`** makes the **`u`** optional, so it would match either the American spelling **`color`** or the Commonwealth spelling **`colour`**.
    - An asterisk (**`*`**) matches zero or more occurrences; for example, **`snakes*`** matches **`snake`**, **`snakes`** and **`snakesssssssss`**.
    - A plus sign (**`+`**) matches one or more occurrences; for example, **`ba(na)+`** matches **`bana`**, **`banana`**, **`bananana`**, and so on—but it does not match **`ba`** by itself.
    - A number in braces (**`{}`**) matches only that many occurrences; for example, **`vacu{2}m`** only matches **`vacuum`**, because we specified there had to be two **`u`**'s there. This also means you could rewrite the phone number pattern from earlier (in "Groups") as **`(\d{3})-(\d{3})-(\d{4})`** and mean exactly the same thing.
        - The braces also can contain a range of numbers; for example, **`\d{7,10}`** will match any number that's got seven, eight, nine or 10 digits.
        - Omitting the number before the comma is the same as putting a zero there; for example, **`[A-Z]{,3}`** matches any sequence of *up to three* capital letters, such as **`X`**, **`AA`** and **`BLS`**.
        - Omitting the number after the comma is like putting an infinity there; for example, **`Go{2,}gle`** would match **`Google`**, **`Goooooogle`** and any other form of the word (like at the bottom of Google's search results), as long as it has *at least two* consecutive **`o`**'s.
- Anchors
    - A caret (**`^`**) matches the beginning of a line of text, which can be useful if you're searching through an entire file, like if you wanted to match the first field in each row of a CSV.
    - A dollar sign (**`$`**) similarly matches the end of a line of text.
    - **`\A`** matches the beginning of the entire string you're searching against, regardless of whether there are multiple lines in it or not.
    - **`\Z`** does the same thing, but for the end of the entire string.

Putting it all together

So, once you have some of those building blocks, you might see a regular expression like this:

```
^(IX|IV|V?I{,3})$
```

This regex is searching for any Roman numeral from one through nine, on a line of text all by itself. How does that work?

First, we know each match must be on its own line. The pattern starts with `^`, which means whatever comes after it *must* be at the beginning of a line. It also ends with `$`, which means whatever comes before it *must* be at the end of a line.

Between the start and end of the line, we have a group with three alternatives in it, separated by pipes:

- The first alternative is **`IX`**, the Roman numeral for nine.
- The second alternative is **`IV`**, for four.
- The third alternative covers all of the remaining numerals:
    - It starts with a **`V`** and makes that optional with a question mark.
    - Whether that **`V`** is present or not, we then may have up to three **`I`**'s. (So without the **`V`**, this alternative covers the numbers one through three; with the **`V`**, we also get five through eight.)

Note this expression also can match a completely empty line. We could be in the third alternative there, which would match a string with zero **`V`**'s and zero **`I`**'s—which would be nothing!

Common differences

Different programming languages, text editors and other tools handle regular expressions in their own ways. Some of the things that differ most often:

Character classes

The shorthand for various character classes (all digits, all letters, all white space, etc.) can vary between environments. Some don't support any, so you'd need to define them manually, such as with brackets.

Others support more than just the basic classes like **`\d`** mentioned earlier. Look into "POSIX character classes" for some examples; they tend to look like **`[[:alpha:]]`**, which matches any letter.

Escaping

Some environments have different rules about when certain characters do and don't need to be escaped. For example, Vim requires you to use escaped parentheses—**`\(`** and **`\)`**—to define groups; unescaped parentheses are treated instead as literal characters.

Group references

You might need to refer to groups by using dollar signs rather than backslashes; for example, the first group might be represented by **`$1`** instead of **`\1`**.

Sometimes there are ways to assign names to your groups, so you don't have to use position-based numbers. (Django's URL definitions are an example of this.) The syntax is environment-specific, so look it up first.

Flags

Some environments support the use of "flags", which modify the rules of the regular expressions you use. Some common ones:

- "Global" means your expression can match more than one occurrence of the same pattern in the same line or document. This often is implied in text editors, but you might need to specify it when programming.
- "Multiline" expands the meaning of `.` to include line breaks, so you can match pieces of text that span multiple lines.
- "Verbose" ignores the literal meanings of most whitespace characters that are in your expression, so you can break it up and include comments in order to make it easier to read and debug.

How you enable these depends on your environment, so read up on the language or editor you'll be using.

Environment-specific resources

Here are some places to find details about the regex implementation in your own project.

Editors

- Emacs: https://www.emacswiki.org/emacs/RegularExpression
- Notepad++: http://docs.notepad-plus-plus.org/index.php/Regular_Expressions
- Sublime Text: http://bit.ly/st-regexp
- TextWrangler: https://gist.github.com/ccstone/5385334
- Vim: http://vim.wikia.com/wiki/Search_patterns

Languages

- Bash: http://tldp.org/LDP/abs/html/regexp.html
- JavaScript: https://bit.ly/js-regexp
- PHP: http://php.net/manual/en/book.pcre.php
- Python: https://docs.python.org/3.6/library/re.html
- Ruby: http://ruby-doc.org/core-2.4.0/Regexp.html

Other resources

- RegExr: http://regexr.com/
  - Web page that lets you test out different regular expressions
  - Annotates each part of your expression, and provides helpful info about each match
- Regex Crossword: https://regexcrossword.com/
  - Game to help you practice interpreting regular expressions