Regular Expressions

Anyone working with text files will often find regular expressions to be very helpful. In most digital humanities projects, you'll spend as much time cleaning data as you'll spend actually analyzing it. Unless you want to clean data entirely by hand, you'll want to use some basic regular expressions to parse through them.

If you're working on a website, too, knowing your way around regular expressions can frequently save you enormous amounts of time; rather than tediously replace the same pattern over and over again, you can

Regular expressions (or "regexes") are, to put it generally, a vocabulary for abstractly describing text. Any reader knows that "1785-1914" is a range of dates, or that "bwilliams@nbc.com" is an e-mail address. If you have a document full of date ranges, or e-mail addresses, or any other sort of text, you probably have some structured entities just like this. But a computer needs to be told what a "date range" or an "e-mail address" is. Regular expressions offer a way to define them.

A year range might be defined, say, as `[0-9]+-[0-9]+`.

Valid e-mail addresses are more complicated: you might search for them using the expression: `^([A-Z0-9._%+-]+)@([A-Z0-9.-]+)`
Obviously that's longer than any single e-mail address. But you can understand what's going on inside it, and why you might want to.

Where to use regexes:

Regexes are embedded in all sorts of computer software. The easiest place to use them is inside a text area. Many text editors contain them, but Microsoft Word does not include the full range. If you have a Mac, the program TextWrangler offers one easy-to-use environment with regexes built in. For Windows, Notepad-plus-plus does much the same thing. These are programs well worth installing on your computer–many tedious editing tasks can be sidestepped by reprogramming them as a regular expression. On either, Atom is a good general purpose choice.

If you want to unlock the full power of regular expressions, you can find them in most modern computer languages.

`sed` has the classical set of regular expressions, and is easily invoked through the command line on a Mac or Linux machine:

```
echo "hi" | sed 's/h/i/g;'
```

If you ever use the command line, the perl one-liner syntax is well worth knowing as well: try to figure out what the following will do before pasting it into a terminal.

```
echo "Some letters look like numbers" | perl -pe 's/o/0/g; s/l/1/g; s/e/3/g'
```

Basic search-replace operations

Basic Operators:

`|`:

- `cat|dog` matches the words "cat" and the words dog: it also matches "doggerel" and "scattered."

`*`, `?` and `+`

- `*` matches the preceding character **any number of times,* including no times at all.
- `+` matches the preceding expression **at least one time.**
- `?` matches the preceding expression exactly **zero or one times.**

`[]` You can use brackets to indicate a **range** of characters. Suppose you are searching through the Schmidt family records, but learn that 18th century families often spelled the name "Schmitt." The regular expression `Schmi[td]t` would match either spelling.

`()` Parenthesis let you group a set of characters together. That is useful with replacements, described below: but it also lets you apply the operators above to **groups** of words.

Suppose you have a document full of references to John Quincy Adams, but that it sometimes calls him "John Q. Adams" and sometimes "John Quincy Adams." If you want to standardize, you want to make the whole "uincy" field optional. You can do this by searching for the following regex:

`John Q(uincy)?.? Adams`

Note that you need the period too, or else it won't match for `John Q. Adams`.

You can combine groups with vertical brackets to match complicated expressions.

(Mac|Mc)Donald would match the names "MacDonald" or "McDonald." This is just the same as "Ma?cDonald": either one is fine, but the brackets can be more readable.

`.` One last special character is the period, which matches *any single character*. The previous regex, for John Q. Adams,

The most capacious regex of all is `.*` which tells the parser to match "any character any number of times." There are situations where this can be useful, particularly inside another regex.

`{}` For most cases, `*`, `+`, or `?` will work to capture an expression. But if you want to specify a particular number of times, you can use angle brackets. So to find Santa Claus, you could type `(Ho){3}`.

Replacements

The syntax for replacing a regex will change from language to language, but the easiest substitution is to replace a regex by a string. I'll use here perl syntax, which gives the name of the operation (`s/` for substitute, `m/` for "match") separated by forward slashes. More recent languages or text editors may have a different syntax, but the important thing is that any substituting regex has two primary parts; the field to be matched, and its substitution.

Escape characters.

Escaping special characters

Sometimes, of course, you'll actually want to search for a bracket, parenthesis, or other special character. To describe a literal bracket in a regex, you use the so-called "escape character": the backslash, `\`. "Escaping" a character means putting a backslash in front of it, so that it takes a special meaning. To represent a literal period, for example, you'd have to specify the regex `\.`. The backslash is hardly ever used in normal writing, so it makes a safe choice for this: but you can always "escape" even the backslash itself, by prefacing it with another backslash: `\\`

Group matches

In addition to escaping those special characters, regexes also allow you to create *other* special characters.

The most powerful ones, and the ones best worth knowing, take their meaning from the context of the regular expression.

When you use parentheses in a regex, it doesn't only create a group for matching: it also sets aside that group for future reference. Those can be accessed by escaping a digit from one to ten.

That means that you can replace a string contextually.

If you wanted to replace every occurrence of "ba" in a text with "ab," say, you could simply run the following substitution:

`s/ba/ab/`

But what if you actually want to swap any two letters?

`s/(b)(a)/\2\1/` does the same thing, but more generally. You could put anything into the parentheses.

Say you wanted to reformat a list of names from Firstname Lastname format to `Lastname, Firstname`.

The regex `s/(.*) (.*)/\2, \1/` matches any characters, followed by a space, followed by any characters, and replaces them with the second group and the first group.

Creating other special characters.

Other important special characters come from prefacing letters.

- `\n`: a "newline"
- `\t`: a **tab**

In addition, other special characters will match a whole **range** of letters. Usually, there would be a way to write these as a regular expression on their own: but it can be very helpful to have a more succinct version. Some of the most useful are:

- `\w`: Any **word** character. (The same as `[A-Za-z]`).
- `\W`: Any **non-word** character. (The same as [^A-Z-a-z])
- `\d`: Any **numeric** (digit) character.
- `\D`: Any **non-numeric** (digit) character.

(If you are working in non-English languages, there are unicode extensions that work off the special character `\p` (or `\P` to designate the inverse of a selection). `\p{L}` matches any unicode letter, for example. See the unicode web site for more on this.)