Regular Expressions Crash Course

Regular expressions (often shortened to "regex") allow you to define *patterns* of text to match. This handout doesn't cover everything there is to know about regular expressions by a long shot—just some of the most commonly used methods. There can be slight differences in the ways that different languages handle regular expressions, but the tips here should work in most situations. For a fuller guide, see: http://www.regular-expressions.info/reference.html. For a terser version of these tips, see this cheat sheet at MIT: http://web.mit.edu/hackl/www/lab/turkshop/slides/regex-cheatsheet.pdf. (I've borrowed examples from both of these resources, as well as from the user manual for Bare Bones Software's TextWrangler.)

You can find a helpful, interactive tester for building your regular expressions at http://www.regexr.com. (Note that this site will only test for regular expression assertions supported by JavaScript. That will serve for many, many cases, but other languages may support other patterns.)

Literals vs. Special characters

Most characters in a regular expression will match themselves—that is, an "m" in a regular expression will match an "m" in the text you're searching—and are known as "literals." But there are some special characters that have different functions in constructing a regular expression (we'll see those functions as we proceed). The special (or "meta") characters are: `[](){}.*^$\|?+` If you actually need to match one of those characters, you need to "escape" it with a backslash. To match a "+" in your text, for example, your regular expression would need to read `\+`. (You can even escape the backslash, itself: `\\`).

Character Sets

To search for any of several characters, place the characters in square brackets.

`[ae]`	Matches any instance of either "a" or "e". Examples: "All of these letters would be matched." `gr[ae]y` would match both "gray" and "grey"
`[0-9]`	You can use a hyphen to indicate a range, in this case, any digit from 0 to 9
`[0-9a-zA-Z]`	You can combine character ranges in a character set, in this case 0 through 9, lower-case a through lower-case z, and upper-case A through upper-case Z.
`[^aeiou]`	Use the caret to *exclude* a character set. This expression would match any character except the vowels. (*N.b.*, the caret plays a different role outside of a character set. See below, under "Position.")

Regular Expressions Crash Course

Special characters in character sets

You can include most special characters in a character set without incident, but a few require special handling:

`[aeiou^]`	Because the caret ordinarily excludes a character set, if you want to include the caret, itself, you need to place it at the end of your set.
`[aeiou-]`	Because the hyphen ordinarily indicates a range of characters, if you want to include the hyphen, itself, you need to place it at the end of your set. (*N.b.* Because the hyphen represents a range, you need to make sure it's the very last character in the set, after any other special characters.)
`[aeiou\]]`	Because the closed bracket ordinarily indicates the end of the character set, if you want to include the closed bracket, itself, you need to escape it.
`[aeiou\\]`	Because the backslash can be used to represent classes of characters (see below), if you want to include the backslash, itself, you need to escape it.

Character Classes

There are several shortcuts for representing entire classes of characters without having to specify them.

`.`	Anything (character or whitespace).
`\d`	Digits (0-9)
`\D`	Non-digits (includes letters, special characters, and whitespace: anything that's not a digit).
`\w`	Any "word" character (letters, digits, and hyphens).
`\W`	Any "non-word" character (punctuation other than hyphens, spaces).
`\s`	Whitespace (spaces, tabs, newlines)
`\S`	Non-whitespace characters
`\t`	Tab
`\n`	Newline

Regular Expressions Crash Course

Position

Sometimes you only want to match characters in certain places.

`^`	At the beginning of a line. (N.b. - This use of the caret is different from the use of the caret inside the square brackets that define a character set)
`$`	At the end of a line
`\b`	Word boundary (e.g., a whitespace or punctuation mark). `ite\b` would match "socialite" and "vegemite", but *not* "item", "iteration", or "politeness". `\bite` would match "item" and "iteration", but *not* "socialite", "vegemite", or "politeness"
`\B`	Not a word boundary `ite\B` would match "item", "iteration", and "politeness", but *not* "socialite" or "vegemite" `\Bite` would match "politeness", "socialite", and "vegemite", but *not* "item" or "iteration" `\Bite\B` would match "politeness" and "whitest", but not "item", "iteration", "socialite", or "vegemite".

Quantifiers

Specify how many occurrences of a character or class of characters to match

`*`	0 or more `ful*` looks for "fu" and zero or more "l"s. It would match "fulsome", and "fully", but also "futile".
`+`	1 or more `ful+` looks for "fu" and one or more "l"s. It would match "fulsome" and "fully", but *not* "futile".
`?`	0 or 1 `ful?` Looks for "fu" and zero or one (and only one) "l". It would match "futile" and "fulsome", but *not* "fully".

Regular Expressions Crash Course

Quantifiers (continued)

`{x}`	Looks for precisely *x* occurrences of a character or character class. `\d{5}` would match a string of five digits (like a US ZIP code).
`{x,}`	Looks for at least *x* occurrences of a character or character class. `fu[\w]{3,}\b` looks for "fu" and *at least* three more word characters that end on a word boundary. It would match "futile", "fulsome", "fully", and "beautifullest", but *not* "full". `\bfu[\w]{3,}\b` looks for a word boundary followed by "fu" (i.e., a word beginning "fu"), followed by at least three more word characters that end on a word boundary. It would match "futile", "fulsome", and "fully", but *not* "full" or "beautifullest".
`{x,y}`	Looks for between *x* and *y* occurrences of a character or character class. `\bfu[\w]{3,5}\b` looks for a word boundary followed by "fu", followed by between three and five more word characters that end in a word boundary (i.e., any five- to seven-letter word beginning in "fu"). It would match "futile", "fulsome", and "fully", but *not* "full" or "fulmination".

Non-greedy or "Lazy" Quantifiers

By default, all of the above quantifiers will return the longest possible match for a pattern, which isn't always what you want to do. Imagine you wanted to remove the HTML tags from the source of a web page you had downloaded.

Given a section of text like this:

```
<span class="quotation">Outside of a dog, a book is a man's best friend. Inside of a dog, it's too dark to read.</span>
```

The regular expression `<.+>` (an opening angle bracket, followed by one or more characters of any kind, followed by a closing angle bracket) will match *everything* between the first opening angle bracket and the last closing angle bracket. To remove just the tags and their contents, we need to make the quantifier "non-greedy" by adding a question mark: `<.+?>` will match `<span class="quotation">` and will also match `</span>`, but will leave the text in the middle alone.

All of the quantifiers in the table above can be made "non-greedy" by adding a question mark in this way.

Regular Expressions Crash Course

Grouping and Backreferences

Often, you'll want not simply to find a pattern of text, but also to save and reuse it. You identify a group of characters whose value you want to save by placing it in parentheses. You can refer to that group later by number.

The method of referring to captured groups can vary depending on the software you're using. Often, you precede the number of the saved group with a backslash, as in the examples below—`\1` for the first saved reference, `\2` for the second, and so on.

> Given the text "Jane Smith"
> And the regular expression `([A-Z][a-z]+)\s([A-Z][a-z]+)`
> `\2, \1` would yield "Smith, Jane"
>
> The regular expression is looking for: a) an upper-case letter followed by one or more lower-case letters; b) a whitespace character; and c) an upper-case letter followed by one or more lower-case letters. Since a) and c) are enclosed in parentheses, those patterns are captured for reuse later. `\2, \1` gives us the second group ("Smith"), followed by a comma and a space, and then the first group ("Jane").

Alternation

The vertical pipe character indicates a choice between two patterns.

> `(dog|cat)s` would match either "dogs" or "cats"

Regular Expressions Crash Course

Non-grouping Parentheses (Advanced)

The fact that the parentheses normally group a pattern for reuse can present problems when you're trying to match on a range of conditions. Imagine we wanted to catch references to any of three London parishes—St. Botolph Aldgate, St. Clement Danes, or St. Dionis Backchurch—but suspected that there might be some irregularities, including:

1. "Saint" might sometimes be spelled out, sometimes abbreviated "St.", and sometimes be abbreviated "St" without a period.
2. We might encounter "Botolph", "Botolph's", or "Botolphs", and it might or might not be followed by a comma.
3. "Backchurch" might sometimes be rendered "Blackchurch"

```
((S(ain)?t\.?) (Botolph('?s)?,? Aldgate|Clement Danes|Dionis
Bl?ackchurch)
```

would match any of our cases, but we would end up with more matches than we wanted. If we count our open parentheses, we see we'd have *five* saved groups:

\1 St. Botolph's, Aldgate \2 St. \3 \4 Botolph's, Aldgate \5 's	\1 Saint Clement Danes \2 Saint \3 ain \4 Clement Danes \5

We can make those nested parentheses around our alternation choices "non-grouping" by adding a question mark and a colon after the opening parenthesis for each sub-group:
`((?:S(?:ain)?t\.?) (?:Botolph(?:'?s)?,? Aldgate|Clement Danes|Dionis Bl?ackchurch))` matches:

```
\1 St. Botolph's, Aldgate
\1 Saint Clement Danes
```