Regular Expressions

Jonathan Geisler

April 28, 2008

Taylor University
CSS

What are regular expressions?

- Patterns

What are regular expressions?

- Patterns
- Strings

What are regular expressions?

- Patterns
- Strings
- Powerful

What are regular expressions?

- Patterns
- Strings
- Powerful
- Heavily copied

What are regular expressions?

- Patterns
- Strings
- Powerful
- Heavily copied
  - PCRE library

What are regular expressions?

- Patterns
- Strings
- Powerful
- Heavily copied
  - PCRE library
  - Java, Python, Ruby, C#, etc.

How do they work?

- `"Hello World!" =~ /Hello/`

How do they work?

- `"Hello World!" =~ /Hello/`
- Uses match operator (=~)

How do they work?

- `"Hello World!" =~ /Hello/`
- Uses match operator (=~)
- Checks whether the string `"Hello"` exists anywhere inside the string `"Hello World!"`

How do they work?

- `"Hello World!" =~ /Hello/`
- Uses match operator (=~)
- Checks whether the string `"Hello"` exists anywhere inside the string `"Hello World!"`
- Returns boolean value in scalar context (we'll talk about list context later)

How do they work?

- `"Hello World!" =~ /Hello/`
- Uses match operator (=~)
- Checks whether the string `"Hello"` exists anywhere inside the string `"Hello World!"`
- Returns boolean value in scalar context (we'll talk about list context later)
- Opposite is !~

When are they used?

- Matching (`m//`)

When are they used?

- Matching (`m//`)
- Substitution (`s///`)

When are they used?

- Matching (`m//`)
- Substitution (`s///`)
- Translation (`tr///`)

When are they used?

- Matching (`m//`)
- Substitution (`s///`)
- Translation (`tr///`)
- `split()`

Where is the match?

Perl always matches the longest possible match that comes earliest in the target string. This means that

```
"This is his bat" =~ /his/
```

matches the his in This because it is the first successful match.

Placeholders

- The special character $ matches the end of a string

Placeholders

- The special character $ matches the end of a string
- The special character ^ matches the beginning of a string

Placeholders

- The special character $ matches the end of a string
- The special character ^ matches the beginning of a string
- If no matching string is listing, the $_ is used to match against (e.g., `if (m/string/)`)

Search Modifiers

- `g` = global
- `i` = ignore case
- `m` = multiple lines
- `s` = treat newline as a whitespace
- `x` = ignore whitespace
- Can be combined (e.g., `m/test/ig`)

Character Classes

- To list a group of characters that could match, include all possibilites within []

Character Classes

- To list a group of characters that could match, include all possibilites within []
- The group represents a single character in the matching string

Character Classes

- To list a group of characters that could match, include all possibilites within `[]`
- The group represents a single character in the matching string
- For example, `/[Yy][Ee][Ss]/` matches yes in a case insensitive manner

Character Classes

- To list a group of characters that could match, include all possibilites within `[]`
- The group represents a single character in the matching string
- For example, `/[Yy][Ee][Ss]/` matches yes in a case insensitive manner
- To represent a range, use a dash to separate beginning and ending characters (e.g., `/[a-z]/` matches lower case characters)

Character Classes

- To list a group of characters that could match, include all possibilites within `[]`
- The group represents a single character in the matching string
- For example, `/[Yy][Ee][Ss]/` matches yes in a case insensitive manner
- To represent a range, use a dash to separate beginning and ending characters (e.g., `/[a-z]/` matches lower case characters)
- For any character outside a character class, start with a carat (e.g., `/[^N]/`)

Special Character Classes

- . = anything except newline

Special Character Classes

- `.` = anything except newline
- `\d` = `[0-9]`

Special Character Classes

- `.` = anything except newline
- `\d` = `[0-9]`
- `\w` = `[a-zA-Z_]`

Special Character Classes

- `.` = anything except newline
- `\d` = `[0-9]`
- `\w` = `[a-zA-Z_]`
- `\s` = `[\ \n\r\t\f]`

Special Character Classes

- `.` = anything except newline
- `\d` = `[0-9]`
- `\w` = `[a-zA-Z_]`
- `\s` = `[\ \n\r\t\f]`
- `\D` = `[^\d]`

Special Character Classes

- `.` = anything except newline
- `\d` = `[0-9]`
- `\w` = `[a-zA-Z_]`
- `\s` = `[\ \n\r\t\f]`
- `\D` = `[^\d]`
- `\W` = `[^\w]`

Special Character Classes

- `.` = anything except newline
- `\d` = `[0-9]`
- `\w` = `[a-zA-Z_]`
- `\s` = `[\ \n\r\t\f]`
- `\D` = `[^\d]`
- `\W` = `[^\w]`
- `\S` = `[^\s]`

Alternation

- If you want options for more than one word, use vertical bar to separate options (e.g., `/pole|bean/`)

Alternation

- If you want options for more than one word, use vertical bar to separate options (e.g., `/pole|bean/`)
- More powerful than character classes, because above example would be impossible to express one character at a time

Repetition

- $\{x\}$ = exactly x times

Repetition

- $\{x\}$ = exactly x times
- $\{m,\ n\}$ = between m and n times, inclusive

Repetition

- $\{x\}$ = exactly x times
- $\{m,\ n\}$ = between m and n times, inclusive
- $\{p,\ \}$ = at least p times

Repetition

- $\{x\}$ = exactly x times
- $\{m,\ n\}$ = between m and n times, inclusive
- $\{p,\ \}$ = at least p times
- + = $\{1,\ \}$

Repetition

- $\{x\}$ = exactly x times
- $\{m,\ n\}$ = between m and n times, inclusive
- $\{p,\ \}$ = at least p times
- $+ = \{1,\ \}$
- $* = \{0,\ \}$

Repetition

- $\{x\}$ = exactly x times
- $\{m,\ n\}$ = between m and n times, inclusive
- $\{p,\ \}$ = at least p times
- `+` = $\{1,\ \}$
- `*` = $\{0,\ \}$
- `.*` is not usually what you want (why?)

Grouping

- Use parentheses to group things together (e.g., `/(dog|cat)(house|abode)/`)

Grouping

- Use parentheses to group things together (e.g., `/(dog|cat)(house|abode)/`)
- Allows alternations to be part of a regex instead of covering the entire thing

Grouping

- Use parentheses to group things together (e.g., `/(dog|cat)(house|abode)/`)
- Allows alternations to be part of a regex instead of covering the entire thing
- Also allows capturing for later use . . .

Capturing

- If you're interested in portions of the match, you can access them in two ways:

Capturing

- If you're interested in portions of the match, you can access them in two ways:
  1. Using $1, $2, $3, etc.

Capturing

- If you're interested in portions of the match, you can access them in two ways:
  1. Using $1, $2, $3, etc.
  2. Using the returned values from the match

Capturing

- If you're interested in portions of the match, you can access them in two ways:
  1. Using $1, $2, $3, etc.
  2. Using the returned values from the match
- Each grouping is saved for later use (e.g., `m/(\w+)\s+(\w+)/` returns two values)

Using numbered variables

You may use $1, $2, $3, etc. as soon as the match is complete until you perform another match. For the previous example:

```
m/(\w+)\s+(\w+)/;
if ($1 eq "Hello" && $2 eq "World") {
    print "Got it!\n";
}
```

Using returned values

In list context, the match returns each match as a separate element in a list. For the previous example:

```
my ($a, $b) = /(\w+)\s+(\w+)/;
if ($a eq "Hello" && $b eq "World") {
    print "Got it!\n";
}
```

Remember to use list context!!!

Let's make a regex for a URL

We need to match:

- The protocol

Let's save each part for later use.

Let's make a regex for a URL

We need to match:

- The protocol
- The funky `://`

Let's save each part for later use.

Let's make a regex for a URL

We need to match:

- The protocol
- The funky `://`
- The server

Let's save each part for later use.

Let's make a regex for a URL

We need to match:

- The protocol
- The funky `://`
- The server
- The port (optionally)

Let's save each part for later use.

Let's make a regex for a URL

We need to match:

- The protocol
- The funky `://`
- The server
- The port (optionally)
- The path

Let's save each part for later use.

Let's make a regex for a URL

We need to match:

- The protocol
- The funky `://`
- The server
- The port (optionally)
- The path
- The parameters (optionally)

Let's save each part for later use.

Let's get each parameter

Now that we've captured the parameters, let's break them down to key/value pairs. The list consists of:

- Repeating *key*=*value* pairs

Let's get each parameter

Now that we've captured the parameters, let's break them down to key/value pairs. The list consists of:

- Repeating *key*=*value* pairs
- Separated by &

Making * work for you

- By default, Perl wishes to match the largest possible value

Making * work for you

- By default, Perl wishes to match the largest possible value
- The * and + operators assist in this

Making * work for you

- By default, Perl wishes to match the largest possible value
- The * and + operators assist in this
- For example, `"Hello" =~ /.*/` matches the entire word `Hello`

Making * work for you

- By default, Perl wishes to match the largest possible value
- The * and + operators assist in this
- For example, `"Hello" =~ /.*/` matches the entire word `Hello`
- This is called greedy matching because the * is greedy and slurps up everything it can

Making * work for you

- By default, Perl wishes to match the largest possible value
- The * and + operators assist in this
- For example, `"Hello" =~ /.*/` matches the entire word `Hello`
- This is called greedy matching because the * is greedy and slurps up everything it can
- To make it non-greedy, use ?