Overview

- Basic regex's.
- `String` methods.
- `RegExp` methods.
- Crawler

Regular Expressions Introduction

- Regex's provide special syntax for string matching.
- Allows matching, sub-string extraction and substitution.
- Initially popularized by early Unix tools.
- Indispensable tool in every programmer's toolbox.

Basic Regex Syntax

A regular expression literal can be written within / delimiters.

`/hello/` Matches the string `"hello"`.

`/hello|world/` Matches either the string `"hello"` or the string `"world"`. `|` is the **alternation operator**.

`/[hH]ello/` Matches either the string `"hello"` or the string `"Hello"`. `[hH]` is a character class equivalent to `/(h|H)/`.

`/[0-9]/` Matches a digit. `[0-9]` is a **range character class**.

`/[^0-9]/` Matches a non-digit. `[^0-9]` is a **negated character class**.

`/worlds?/` Matches either the string `"worlds"` or the string `"world"`. `?` is a suffix operator indicating that the preceeding sub-regex is **optional**.

Basic Regex Syntax: Continued

/[0-9]+/ Matches one-or-more digits. + is a suffix operator indicating **one-or-more** matches of the preceeding regex.

/[a-zA-Z]*/ Matches zero-or-more lowercase or uppercase alphabetic characters. $*$ is a suffix operator indicating **zero-or-more** matches of the preceeding regex.

- So the basic regular expression operators in order of decreasing precedence are:

  Quantifiers optional ?, one-or-more +, zero-or-more $*$ Indicated by unary suffix operators.

  Concatenation Indicated by juxtaposition.

  Alternation Indicated by infix binary operator |.

- Use parentheses ( ) to override default precedence.

Escaping Special Characters

- Characters like +, ∗ |, ?, (,) are part of the regular expression notation and are referred to as special characters or **meta-characters**.
- Special characters can be escaped by preceeding them with a \ character. For example, `/hello\*/` is a regex matching the string `"hello*"`.
- The \ character can itself be escaped using a preceeding \. Hence `/hello\\/` is a regex matching the string `"hello"` followed by a \ character.
- Within a character class the usual meta-characters loose their meaning and do not need to be escaped; the only characters which are special are the range operator - (though not at the start), and the negated character class operator ^ (only at the start) and \ for escaping.

More Regex Examples

`/[1-9][0-9]*|0/` Matches an integer with no non-significant leading zeros.

`/[-+]?[0-9]+/` Matches an integer with an optional sign (no restriction on leading zeros).

`/[0-9]+[lL]?/` Matches an integer with an optional suffix of `l` of `L` (no restriction on leading zeros).

`/[a-zA-Z_][a-zA-Z0-9_]*/` Matches the definition of an *identifier* in many programming languages. An identifier consists of one or more alphanumeric characters or underscores with the restriction that the first character cannot be a digit.

String search() Method

Invoking `search(`*regex*`)` on a string will return the index of the start of the first match of *regex* in the string; -1 if not found.

```
> "abcd123".search(/[a-z]+[0-9]+/)
0
> "+-abcd123".search(/[a-z]+[0-9]+/)
2
> "+-abcd".search(/[a-z]+[0-9]+/)
-1
```

String `match()` Method

Invoking `match(`*regex*`)` on a string results in an "array":

- Element 0 contains the entire match.
- Elements n for $n > 0$ contains the substring matched by the n'th **capturing parentheses** group.
- An `index` property contains the index of the start of the match in the input string.
- An `input` property contains the input string.

String match() Examples

```
> "abc123".match(/[a-z]+[0-9]+/)
[ ’abc123’, index: 0, input: ’abc123’ ]
> "abc123".match(/([a-z]+)([0-9]+)/)
[ ’abc123’, ’abc’, ’123’, index: 0, input: ’abc123’ ]
> "+-/abc123".match(/([a-z]+)([0-9]+)/)
[ ’abc123’, ’abc’, ’123’, index: 3, input: ’+-/abc123’
]
> "+-abc123".match(/(([\-\+])[a-z]+)([0-9]+)/)
[ ’-abc123’, ’-abc’, ’-’, ’123’, index: 1, input:
’+-abc123’ ]
> [, a, b] = //destructuring
   ’abc3-123’.match(/(\w+)\-(\d+)/)
[ ’abc3-123’, ’abc3’, ’123’, index: 0,
  input: ’abc3-123’ ]
> [a, b]
[ ’abc3’, ’123’ ]
```

String split() Method

Can split a string on a regex:

```
> "ab, x12, de , f".split(/ +/)
[ ’ab,’, ’x12,’, ’de’, ’,’, ’f’ ]
> "ab, x12, de , f".split(/ *, */)
[ ’ab’, ’x12’, ’de’, ’f’ ]
```

More Regex Syntax

`[a-z]` not portable as it depends on the character codes for lowercase characters being adjacent in the underlying character set. True in ASCII, but for example, in EBCDIC alphabetic character codes are not contiguous.

Digit Char `/\d/` A regular expression which matches any digit.

Non-Digit Char `/\D/` A regular expression which matches any non-digit.

Word Char `/\w/` A regular expression which matches any word-char (alphanumeric or _).

Non-Word Char `/\W/` A regular expression which matches any non-word-char.

Space Char `/\s/` A regular expression which matches any whitespace character (blank, tab, newline etc.).

Non-Space Char `/\S/` A regular expression which matches any non-whitespace character.

Any char other than newline `/./` Matches any character other than newline.

Fixed Count or Range Count Quantifiers

`/\d{5}/` Matches exactly 5 digits. Suffix {*n*} means match exactly *n* occurrences of preceeding regex.

`/\d{5,}/` Matches at least 5 digits. Suffix {*n*,} means match at least *n* occurrences of preceeding regex.

`/\d{2,5}/` Matches 2 - 5 digits. Suffix {*n*,*m*} means match *n* through *m* occurrences of preceeding regex.

`/[a-zA-Z_]\w{0,7}/` Matches an identifer containing upto 8 characters.

Context Regex Syntax

Does not actually match anything, just provides a **context** for other regex's to match. Often known as **anchor**'s

`/^\d/` Matches a digit but only at the start of the input string. Can be set to match at start of a line if `m` flag set.

`/\d$/` Matches a digit but only at the end of the input string. Can be set to match at end of a line if `m` flag set.

`/\bm/` Matches a `m` but only on a word boundary. Hence it will match only the first `m` in the word `"mommy"`.

`/\Bm/` Matches a `m` which is **not** on a word boundary. Hence it will match only the internal `m`'s in the word `"mommy"`.

More on Capturing Groups

Non-capturing Group `/(?:\s|\w)\d+/` Just like `/(\s|\w)\d+/` but `(?: )`parentheses are non-capturing.

Named Group `/(?:\s|\w)(?<num>\d+)/` Captures `\d+` into `groups.num`.

```
> 'abc123'.match(/(?:\s|\w)(?<num>\d+)/).groups
[Object: null prototype] { num: '123' }
>
```

Non-Greedy Quantifiers

Normal quantifiers are greedy and attempt to match maximal text. Make quantifier **non-greedy** or **lazy** by suffixing quantifier with a ?.

```
> 'abc123'.match(/(\w*)(\d+)/).slice(1)
[ 'abc12', '3' ]
> 'abc123'.match(/(\w*?)(\d+)/).slice(1)
[ 'abc', '123' ]
>
> '<a> <b>'.match(/\<.*\>/)[0]
'<a> <b>'
> '<a> <b>'.match(/\<.*?\>/)[0]
'<a>'
>
```

RegExp() Constructor

- Constructor constructs a regex from a string.
- Can be used to build dynamic regex's depending on a variable.
- Suffers from *backslashitis*.

```
> new RegExp('[abc]')
/[abc]/
> someVar = 'x'
'x'
> new RegExp(`${someVar}\\d`)
/x\d/
> new RegExp(`${someVar}\\\\\\d`)
/x\\\d/
>
```

More Regex Examples

`/\d{1,3}(?:\.\d{1,3}){3}/` An IPv4 internet address. Consists of 4 decimal numbers each containing up to 3 digits, separated by `.` characters.

`/\/\/.*/` A JavaScript single-line comment. 2 forward slashes followed by zero-or-more characters other than newline.

`/[-+]?\d+(?:\.\d+)?(?:[eE][-+]?\d+)?/` An optionally signed integer or floating point literal. The integer part must be present. It may be optionally followed by a fraction part which consists of a decimal point followed by one-or-more digits. That too may optionally be followed by an exponent part which consists of either an `e` or an `E` followed by an optional sign, followed by one-or-more digits.

Regex Flags

Specify flags after closing slash when using literal syntax or as a string second argument to `RegExp()` constructor:

`/hello/i` Case-insensitive. Equivalent to `[hH][eE][lL][lL][oO]`.

`/hello/g` Global search. Start search from index of last match of this regex in string.

`/^\#.*$/m` Multiline flag changes ^ and $ to match only at the start/end of a **line**.
Example gives a single-line comment starting with a #-character in column 1 at the start of a line and continuing until the end of the line.

Unfortunately, no way to include ignored formatting whitespace within regex like the **verbose** `/x` flag available in other languages like Python or Ruby.

String matchAll() Method

Like match(), but returns iterator over **all** matches. Must use global g flag.

```
> regex = /(?<x>\w+?)(?<y>\d+)/g
> m = ' prj1, hw2 ## #quiz1 '.matchAll(regex)
Object [RegExp String Iterator] {}
> for (const a of m) {
    console.log(a.groups.x, a.groups.y);
  }
prj 1
hw 2
quiz 1
undefined
>
```

matchAll() Iterator to Array

Can spread iterator returned from `matchAll()` into an array to get access to array methods.

```
> regex = /(?<x>\w+?)(?<y>\d+)/g
/(?<x>\w+?)(?<y>\d+)/g
> m = ’ prj1, hw2 ## #quiz1 ’.matchAll(regex)
Object [RegExp String Iterator] {}
> m.map(x => x.groups)
Uncaught TypeError: m.map is not a function
> [...m].map(x => x.groups)
[
  [Object: null prototype] { x: ’prj’, y: ’1’ },
  [Object: null prototype] { x: ’hw’, y: ’2’ },
  [Object: null prototype] { x: ’quiz’, y: ’1’ }
]
>
```

String `replace()` Method

`string.replace(regex|substr, replacement|function)`

`replacement` can contain:

- `$$` Inserts a `$`.
- `$&` Inserts match.
- `$‘` Inserts portion of `string` before match.
- `$’` Inserts portion of `string` after match.
- `$`*n* Inserts text matched by *n*'th capturing parentheses in `regex`.

String replace() Examples

```
> ’the dog’.replace(’dog’, ’cat’)
’the cat’
> ’the Dog’.replace(/dog/i, ’cat’)
’the cat’
> ’the dog’.replace(/[aeiou]/, ’’)
’th dog’
> ’the dog’.replace(/[aeiou]/g, ’’)
’th dg’
> ’the dog123 fido; cat99 eve’.
     replace(/([^\d\s]+)(\d+)/g, ’$1-$2’)
’the dog-123 fido; cat-99 eve’
```

String replace() Examples Continued

```
> '0 cats, 1 cat, 7 cats'.replace(/\d+/,
     function(match) {
       const n = Number(match);
       if (n === 0) {
         return 'zero';
       }
       else if (n === 1) {
         return 'one';
       }
       else {
         return 'many';
       }
   })
'zero cats, 1 cat, 7 cats'
>
```

String replace() Examples Continued

```
> '0 cats, 1 cat, 7 cats'.replace(/\d+/g,
     function(match) {
       const n = Number(match);
       if (n === 0) {
         return 'zero';
       }
       else if (n === 1) {
         return 'one';
       }
       else {
         return 'many';
       }
   })
'zero cats, one cat, many cats'
>
```

RegExp Methods

`exec(str)` Searches for match of this in String `str`. Return value similar to `String.prototype.match()`. Traditionally used with a global regexp with flag /g to match repeatedly in a loop:

```
const regex = /.../g;
let m;
while ( (m = regex.exec(str)) ) { ... }
```

`test(str)` Searches for match of this in String `str`. Returns `true` if search successful, `false` otherwise.

RegExp Limitations

- Standard regular expressions cannot be used to recognize nested constructs for an arbitrary nesting depth.
- For example, it is not possible to write a regex for strings of balanced parentheses like ’((()))’ for an arbitrary depth of nesting.
- Colloquially, regex’s cannot **count**.
- It is always possible to write a regex for a nested construct when the maximum depth of nessing is bounded. For example, all strings of balanced parentheses of depth upto 2 can be written by simply enumerating all possibilities:

  ```
      /( \( ( \(\) )* \) )*/
  ```

  where the whitespace is used only for formatting and should be ignored.

RegEx Denial of Service

Wikipedia

```
> function time(fn) {
    t0 = Date.now(); fn(); return Date.now() - t0;
  }
> re = /^(a|aa)+$/
/^(a|aa)+$/
> time(() => ('a'.repeat(30) + 'c').match(re))
18
> time(() => ('a'.repeat(40) + 'c').match(re))
1835
> time(() => ('a'.repeat(46) + 'c').match(re))
32408
>
```

Crawler

- Hardcoded to crawl only cs544 website.
- Look for `href=` links. Does so in a `matchAll()` loop using a global regex specified using the g flag (could also use RegExp `exec()`).
- Also uses regex's to extract a page title if possible.
- For each referenced page we track its url, title and referers.
- Uses a BFS `toDos` agenda. URL put onto agenda only if it is new, extends the start cs544 start URL and looks like a HTML url.
- *Running App*.

References

- MDN
- *Regular-Expressions.info*
- Jeffrey E. F. Friedl, *Mastering Regular Expressions*, 3rd Edition, O'Reilly, 2006.