Overview

- Basic regex's.
- `String` methods.
- `RegExp` methods.
- Word count `wc` example.

Regular Expressions Introduction

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

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:

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 ˆ (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).

`/[1-9]+[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.

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"`.

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 the `match(`*regex*`)` on a string results in a "array" with element 0 containing the entire match, elements n for $n > 0$ containing the substring matched by the n'th **capturing parentheses** group. A `index` property contains the index of the start of the match in the string and `input` contains the input string.

```
> "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'
```

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

`/\d/`	A regular expression which matches any digit.
`/\D/`	A regular expression which matches any non-digit.
`/\w/`	A regular expression which matches any word-char (alphanumeric or _).
`/\W/`	A regular expression which matches any non-word-char.
`/\s/`	A regular expression which matches any whitespace character (blank, tab, newline etc.).
`/\S/`	A regular expression which matches any non-whitespace character.

More Regex Syntax Continued

`/./`	Matches any character other than newline.
`/(?:\s|\w)\d+/`	Just like `/(\s|\w)\d+/` but `(?: )`parentheses are non-capturing.

More Regex Syntax Continued

`/\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{,5}/`	Matches at most 5 digits. Suffix {,*n*} means match at most *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.

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 number. 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.

`/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 whitespace within regex like the **verbose** `/x` flag available in other languages like Python or Ruby.

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

Word Count Program: Log

Print number of lines, words, chars in specified files; subset of functionality of Unix wc(1) program.
Program shows use of regex as well as async functions.

```
$ wc wc.js
  93  239 1932 wc.js
$ ./wc.js wc.js
   93    239   1932 wc.js
$ wc *
  93  239 1932 wc.js
   4    4   38 wc.js~
  97  243 1970 total
$ ./wc.js *
   93    239   1932 wc.js
    4      4     38 wc.js~
   97    243   1970 total
$
```

Word Count Program: Wc object

In wc.js:

```
const OUT_WIDTH = 5;

function Wc(nLines, nWords, nChars) {
  this.nLines = nLines || 0;
  this.nWords = nWords || 0;
  this.nChars = nChars || 0;
}

Wc.prototype.update = function(wc) {
  this.nLines += wc.nLines;
  this.nWords += wc.nWords;
  this.nChars += wc.nChars;
};
```

Word Count Program: `Wc toString()`

```
Wc.prototype.toString = function() {
  const counts = [this.nLines, this.nWords, this.nChars];
  let str = '';
  for (let count of counts) {
    str += String(count).padStart(OUT_WIDTH) + ' ';
  }
  return str;
}
```

Word Count Program: wc() function

```
function wc(text) {
  let nLines = 0, nWords = 0, nChars = 0;
  const re = /([ \t]+)|(\n+)|(\S+)/g;
  let match = null;
  while ((match = re.exec(text)) !== null) {
    nChars += match[0].length;
    if (match[2]) {
      nLines += match[0].length;
    }
    else if (match[3]) {
      ++nWords;
    }
  } //while
  return new Wc(nLines, nWords, nChars);
}
```

Word Count Program: Asynch `wcFile()` function

```
//return promise for Wc object for fName
function wcFile(fName) {
  return new Promise(function(resolve, reject) {
    fs.readFile(fName, function(err, text) {
      if (err) {
        reject(err);
      }
      else {
        resolve(wc(text));
      }
    });
  });
}
```

Word Count Program: main() function

```
function main(argv) {
  const firstIndex = 2;
  const count = argv.length - firstIndex;
  if (count <= 0) { //could read stdin instead of error
    console.error(`usage: ${argv[1]} FILE...`);
    process.exit(1);
  }
  let promise = Promise.resolve();
  const totals = new Wc();
  for (let i = firstIndex; i < argv.length; i++) {
    const fName = argv[i];
```

Word Count Program: main() function Continued

```
      promise = promise.then(function() { //force serialize
        return wcFile(fName);
      }).then(function(file_wc) {
          console.log(`${file_wc.toString()}${fName}`);
          totals.update(file_wc);
      }).catch(function(err) {
        console.error(err);
        process.exit(1);
      });
    }
    if (count > 1) {
      promise.then(function(_) {
        console.log(`${totals.toString()}total`);
      });
    }
  }
```

References

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