10th October 2019

Sets and ranges [...]

Several characters or character classes inside square brackets `[…]` mean to “search for any character among given”.

Sets

For instance, `[eao]` means any of the 3 characters: `'a'`, `'e'`, or `'o'`.

That’s called a *set*. Sets can be used in a regexp along with regular characters:

```
1  // find [t or m], and then "op"
2  alert( "Mop top".match(/[tm]op/gi) ); // "Mop", "top"
```

Please note that although there are multiple characters in the set, they correspond to exactly one character in the match.

So the example below gives no matches:

```
1  // find "V", then [o or i], then "la"
2  alert( "Voila".match(/V[oi]la/) ); // null, no matches
```

The pattern searches for:

- `V`,
- then *one* of the letters `[oi]`,
- then `la`.

So there would be a match for `Vola` or `Vila`.

Ranges

Square brackets may also contain *character ranges*.

For instance, `[a-z]` is a character in range from `a` to `z`, and `[0-5]` is a digit from `0` to `5`.

In the example below we’re searching for `"x"` followed by two digits or letters from `A` to `F`:

```
1  alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF
```

Here `[0-9A-F]` has two ranges: it searches for a character that is either a digit from `0` to `9` or a letter from `A` to `F`.

If we'd like to look for lowercase letters as well, we can add the range `a-f`: `[0-9A-Fa-f]`. Or add the flag `i`.

We can also use character classes inside `[…]`.

For instance, if we'd like to look for a wordly character `\w` or a hyphen `-`, then the set is `[\w-]`.

Combining multiple classes is also possible, e.g. `[\s\d]` means "a space character or a digit".

Character classes are shorthands for certain character sets

For instance:

- **\d** – is the same as `[0-9]`,
- **\w** – is the same as `[a-zA-Z0-9_]`,
- **\s** – is the same as `[\t\n\v\f\r ]`, plus few other rare unicode space characters.

Example: multi-language \w

As the character class `\w` is a shorthand for `[a-zA-Z0-9_]`, it can't find Chinese hieroglyphs, Cyrillic letters, etc.

We can write a more universal pattern, that looks for wordly characters in any language. That's easy with unicode properties: `[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]`.

Let's decipher it. Similar to `\w`, we're making a set of our own that includes characters with following unicode properties:

- `Alphabetic` (`Alpha`) – for letters,
- `Mark` (`M`) – for accents,
- `Decimal_Number` (`Nd`) – for digits,
- `Connector_Punctuation` (`Pc`) – for the underscore `'_'` and similar characters,
- `Join_Control` (`Join_C`) – two special codes `200c` and `200d`, used in ligatures, e.g. in Arabic.

An example of use:

```
1  let regexp = /[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]/gu;
2
3  let str = `Hi 你好 12`;
4
5  // finds all letters and digits:
6  alert( str.match(regexp) ); // H,i,你,好,1,2
```

Of course, we can edit this pattern: add unicode properties or remove them. Unicode properties are covered in more details in the article Unicode: flag "u" and class \p{...}.

Unicode properties aren't supported in Edge and Firefox

Unicode properties `p{…}` are not yet implemented in Edge and Firefox. If we really need them, we can use library XRegExp.

Or just use ranges of characters in a language that interests us, e.g. `[а-я]` for Cyrillic letters.

Excluding ranges

Besides normal ranges, there are "excluding" ranges that look like `[^…]`.

They are denoted by a caret character `^` at the start and match any character *except the given ones*.

For instance:

- `[^aeyo]` – any character except `'a'`, `'e'`, `'y'` or `'o'`.
- `[^0-9]` – any character except a digit, the same as `\D`.
- `[^\s]` – any non-space character, same as `\S`.

The example below looks for any characters except letters, digits and spaces:

```
alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // @ and .
```

Escaping in [...]

Usually when we want to find exactly a special character, we need to escape it like `\.`. And if we need a backslash, then we use `\\`, and so on.

In square brackets we can use the vast majority of special characters without escaping:

- Symbols `. + ( )` never need escaping.
- A hyphen `-` is not escaped in the beginning or the end (where it does not define a range).
- A caret `^` is only escaped in the beginning (where it means exclusion).
- The closing square bracket `]` is always escaped (if we need to look for that symbol).

In other words, all special characters are allowed without escaping, except when they mean something for square brackets.

A dot `.` inside square brackets means just a dot. The pattern `[.,]` would look for one of characters: either a dot or a comma.

In the example below the regexp `[-().^+]` looks for one of the characters `-().^+`:

```
// No need to escape
let regexp = /[-().^+]/g;

alert( "1 + 2 - 3".match(regexp) ); // Matches +, -
```

…But if you decide to escape them "just in case", then there would be no harm:

```
// Escaped everything
let regexp = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(regexp) ); // also works: +, -
```

Ranges and flag “u”

If there are surrogate pairs in the set, flag `u` is required for them to work correctly.

For instance, let’s look for `[𝒳𝒴]` in the string `𝒳`:

```
alert( '𝒳'.match(/[𝒳𝒴]/) ); // shows a strange character, like [?]
// (the search was performed incorrectly, half-character returned)
```

The result is incorrect, because by default regular expressions “don’t know” about surrogate pairs.

The regular expression engine thinks that `[𝒳𝒴]` – are not two, but four characters:

1. left half of `𝒳` `(1)`,
2. right half of `𝒳` `(2)`,
3. left half of `𝒴` `(3)`,
4. right half of `𝒴` `(4)`.

We can see their codes like this:

```
for(let i=0; i<'𝒳𝒴'.length; i++) {
  alert('𝒳𝒴'.charCodeAt(i)); // 55349, 56499, 55349, 56500
};
```

So, the example above finds and shows the left half of `𝒳`.

If we add flag `u`, then the behavior will be correct:

```
alert( '𝒳'.match(/[𝒳𝒴]/u) ); // 𝒳
```

The similar situation occurs when looking for a range, such as `[𝒳-𝒴]`.

If we forget to add flag `u`, there will be an error:

```
'𝒳'.match(/[𝒳-𝒴]/); // Error: Invalid regular expression
```

The reason is that without flag `u` surrogate pairs are perceived as two characters, so `[𝒳-𝒴]` is interpreted as `[<55349><56499>-<55349><56500>]` (every surrogate pair is replaced with its codes). Now it’s easy to see that the range `56499-55349` is invalid: its starting code `56499` is greater than the end `55349`. That’s the formal reason for the error.

With the flag `u` the pattern works correctly:

```
1  // look for characters from 𝒳 to 𝒵
2  alert( '𝒴'.match(/[𝒳-𝒵]/u) ); // 𝒴
```

Tasks

Java[^script]

We have a regexp `/Java[^script]/` .

Does it match anything in the string `Java` ? In the string `JavaScript` ?

solution

Find the time as hh:mm or hh-mm

The time can be in the format `hours:minutes` or `hours-minutes` . Both hours and minutes have 2 digits: `09:00` or `21-30` .

Write a regexp to find time:

```
1  let regexp = /your regexp/g;
2  alert( "Breakfast at 09:00. Dinner at 21-30".match(regexp) ); // 09:00, 21-30
```

P.S. In this task we assume that the time is always correct, there's no need to filter out bad strings like "45:67". Later we'll deal with that too.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.

- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)