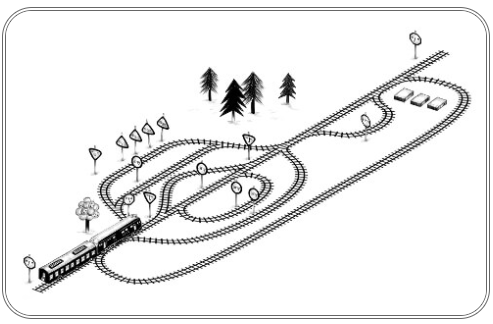
**Regular Expressions:-**

Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.’ Now they have two problems.

Jamie Zawinski

Yuan-Ma said, ‘When you cut against the grain of the wood, much strength is needed. When you program against the grain of the problem, much code is needed.’

Master Yuan-Ma, *The Book of Programming*



## Creating a regular expression

A regular expression is a type of object. It can be either constructed with the RegExp constructor or written as a literal value by enclosing a pattern in forward slash (/) characters.

let re1 = new RegExp("abc");

let re2 = /abc/;

Both of those regular expression objects represent the same pattern: an acharacter followed by a b followed by a c.

When using the RegExp constructor, the pattern is written as a normal string, so the usual rules apply for backslashes.

The second notation, where the pattern appears between slash characters, treats backslashes somewhat differently. First, since a forward slash ends the pattern, we need to put a backslash before any forward slash that we want to be part of the pattern. In addition, backslashes that aren’t part of special character codes (like \n) will be preserved, rather than ignored as they are in strings, and change the meaning of the pattern. Some characters, such as question marks and plus signs, have special meanings in regular expressions and must be preceded by a backslash if they are meant to represent the character itself.

let eighteenPlus = /eighteen\+/;

## Testing for matches:-

Regular expression objects have a number of methods. The simplest one is test. If you pass it a string, it will return a Boolean telling you whether the string contains a match of the pattern in the expression.



console.log(/abc/.test("abcde"));

// → true

console.log(/abc/.test("abxde"));

// → false

A regular expression consisting of only nonspecial characters simply represents that sequence of characters. If *abc* occurs anywhere in the string we are testing against (not just at the start), test will return true.

## Sets of characters:-

Finding out whether a string contains abc could just as well be done with a call to indexOf. Regular expressions allow us to express more complicated patterns.

Say we want to match any number. In a regular expression, putting a set of characters between square brackets makes that part of the expression match any of the characters between the brackets.

console.log(/[0123456789]/.test("in 1992"));

// → true

console.log(/[0-9]/.test("in 1992"));

// → true

A number of common character groups have their own built-in shortcuts. Digits are one of them: \d means the same thing as [0-9].

|  |  |
| --- | --- |
| \d | Any digit character |
| \w | An alphanumeric character (“word character”) |
| \s | Any whitespace character (space, tab, newline, and similar) |
| \D | A character that is *not* a digit |
| \W | A nonalphanumeric character |
| \S | A nonwhitespace character |
| . | Any character except for newline |

let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;

console.log(dateTime.test("01-30-2003 15:20"));

// → true

console.log(dateTime.test("30-jan-2003 15:20"));

// → false

These backslash codes can also be used inside square brackets. For example, [\d.] means any digit or a period character. But the period itself, between square brackets, loses its special meaning. The same goes for other special characters, such as +.

To invert a set of characters—that is, to express that you want to match any character except the ones in the set—you can write a caret (^) character after the opening bracket.

let notBinary = /[^01]/;

console.log(notBinary.test("1100100010100110"));

// → false

console.log(notBinary.test("1100100010200110"));

// → true

## Repeating parts of a pattern:-

When you put a plus sign (+) after something in a regular expression, it indicates that the element may be repeated more than once. Thus, /\d+/matches one or more digit characters.

console.log(/'\d+'/.test("'123'"));

// → true

console.log(/'\d+'/.test("''"));

// → false

console.log(/'\d\*'/.test("'123'"));

// → true

console.log(/'\d\*'/.test("''"));

// → true

A question mark makes a part of a pattern optional, meaning it may occur zero times or one time. In the following example, the u character is allowed to occur, but the pattern also matches when it is missing.

let neighbor = /neighbou?r/;

console.log(neighbor.test("neighbour"));

// → true

console.log(neighbor.test("neighbor"));

// → true

Here is another version of the date and time pattern that allows both single- and double-digit days, months, and hours. It is also slightly easier to decipher.

let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;

console.log(dateTime.test("1-30-2003 8:45"));

// → true

## Grouping subexpressions:-

To use an operator like \* or + on more than one element at a time, you have to use parentheses. A part of a regular expression that is enclosed in parentheses counts as a single element as far as the operators following it are concerned.

let cartoonCrying = /boo+(hoo+)+/i;

console.log(cartoonCrying.test("Boohoooohoohooo"));

// → true

The first and second + characters apply only to the second o in boo and hoo, respectively. The third + applies to the whole group (hoo+), matching one or more sequences like that.

## Matches and groups:-

Regular expressions also have an exec (execute) method that will return null if no match was found and return an object with information about the match otherwise.

let match = /\d+/.exec("one two 100");

console.log(match);

// → ["100"]

console.log(match.index);

// → 8

An object returned from exec has an index property that tells us where in the string the successful match begins. Other than that, the object looks like (and in fact is) an array of strings, whose first element is the string that was matched.

String values have a match method that behaves similarly.

console.log("one two 100".match(/\d+/));

// → ["100"]

The next element is the part matched by the first group (the one whose opening parenthesis comes first in the expression), then the second group, and so on.

let quotedText = /'([^']\*)'/;

console.log(quotedText.exec("she said 'hello'"));

// → ["'hello'", "hello"]

Similarly, when a group is matched multiple times, only the last match ends up in the array.

console.log(/bad(ly)?/.exec("bad"));

// → ["bad", undefined]

console.log(/(\d)+/.exec("123"));

// → ["123", "3"]

Groups can be useful for extracting parts of a string. If we don’t just want to verify whether a string contains a date but also extract it and construct an object that represents it, we can wrap parentheses around the digit patterns and directly pick the date out of the result of exec.

## The Date class:-

JavaScript has a standard class for representing dates—or, rather, points in time. It is called Date. If you simply create a date object using new, you get the current date and time.

console.log(new Date());

// → Mon Nov 13 2017 16:19:11 GMT+0100 (CET)

You can also create an object for a specific time.

console.log(new Date(2009, 11, 9));

// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)

console.log(new Date(2009, 11, 9, 12, 59, 59, 999));

// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)

JavaScript uses a convention where month numbers start at zero (so December is 11), yet day numbers start at one.

The getTime method on a date object returns this number. It is big, as you can imagine.

console.log(new Date(2013, 11, 19).getTime());

// → 1387407600000

console.log(new Date(1387407600000));

// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)

Date objects provide methods such as getFullYear, getMonth, getDate, getHours, getMinutes, and getSeconds to extract their components. Besides getFullYear there’s also getYear, which gives you the year minus 1900 (98or 119) and is mostly useless.

Putting parentheses around the parts of the expression that we are interested in, we can now create a date object from a string.

function getDate(string) {

let [\_, month, day, year] =

/(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);

return new Date(year, month - 1, day);

}

console.log(getDate("1-30-2003"));

// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)

The \_ (underscore) binding is ignored and used only to skip the full match element in the array returned by exec.

## Word and string boundaries:-

we can add the markers ^ and $. The caret matches the start of the input string, whereas the dollar sign matches the end. So, /^\d+$/ matches a string consisting entirely of one or more digits, /^!/ matches any string that starts with an exclamation mark, and /x^/ does not match any string (there cannot be an xbefore the start of the string).

If, on the other hand, we just want to make sure the date starts and ends on a word boundary, we can use the marker \b. The string that has a word character (as in \w) on one side and a nonword character on the other.

console.log(/cat/.test("concatenate"));

// → true

console.log(/\bcat\b/.test("concatenate"));

// → false

## Choice patterns:-

We could write three regular expressions and test them in turn, but there is a nicer way. The pipe character (|) denotes a choice between the pattern to its left and the pattern to its right. So I can say this:

let animalCount = /\b\d+ (pig|cow|chicken)s?\b/;

console.log(animalCount.test("15 pigs"));

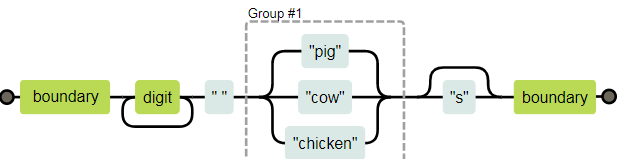
// → true

console.log(animalCount.test("15 pigchickens"));

// → false

Parentheses can be used to limit the part of the pattern that the pipe operator applies to, and you can put multiple such operators next to each other to express .

## The mechanics of matching:-



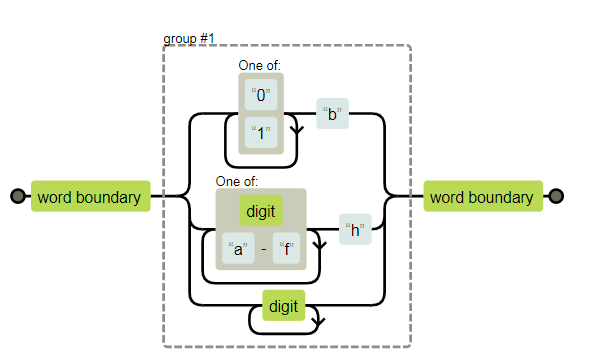
Our expression matches if we can find a path from the left side of the diagram to the right side. We keep a current position in the string, and every time we move through a box, we verify that the part of the string after our current position matches that box.

So if we try to match "the 3 pigs" from position 4, our progress through the flow chart would look like this:

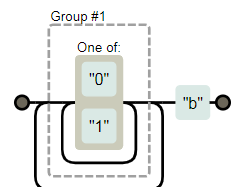
* At position 4, there is a word boundary, so we can move past the first box.
* Still at position 4, we find a digit, so we can also move past the second box.
* At position 5, one path loops back to before the second (digit) box, while the other moves forward through the box that holds a single space character. There is a space here, not a digit, so we must take the second path.
* We are now at position 6 (the start of pigs) and at the three-way branch in the diagram. We don’t see cow or chicken here, but we do see pig, so we take that branch.
* At position 9, after the three-way branch, one path skips the s box and goes straight to the final word boundary, while the other path matches an s. There is an s character here, not a word boundary, so we go through the s box.
* We’re at position 10 (the end of the string) and can match only a word boundary. The end of a string counts as a word boundary, so we go through the last box and have successfully matched this string.

## Backtracking:-

The regular expression /\b([01]+b|[\da-f]+h|\d+)\b/ matches either a binary number followed by a b, a hexadecimal number (that is, base 16, with the letters a to f standing for the digits 10 to 15) followed by an h, or a regular decimal number with no suffix character. This is the corresponding diagram:



Backtracking also happens for repetition operators like + and \*. If you match /^.\*x/ against "abcxe", the .\* part will first try to consume the whole string. The engine will then realize that it needs an x to match the pattern. Since there is no x past the end of the string, the star operator tries to match one character less. But the matcher doesn’t find an x after abcx either, so it backtracks again, matching the star operator to just abc



## The replace method:-

String values have a replace method that can be used to replace part of the string with another string.

console.log("papa".replace("p", "m"));

// → mapa

The first argument can also be a regular expression, in which case the first match of the regular expression is replaced. When a g option (for *global*) is added to the regular expression, *all* matches in the string will be replaced, not just the first.

console.log("Borobudur".replace(/[ou]/, "a"));

// → Barobudur

console.log("Borobudur".replace(/[ou]/g, "a"));

// → Barabadar

For example, say we have a big string containing the names of people, one name per line, in the format Lastname, Firstname. If we want to swap these names and remove the comma to get a Firstname Lastname format, we can use the following code:

console.log(

"Liskov, Barbara\nMcCarthy, John\nWadler, Philip"

.replace(/(\w+), (\w+)/g, "$2 $1"));

// → Barbara Liskov

// John McCarthy

// Philip Wadler

The $1 and $2 in the replacement string refer to the parenthesized groups in the pattern. $1 is replaced by the text that matched against the first group, $2by the second, and so on, up to $9. The whole match can be referred to with $&.

For each replacement, the function will be called with the matched groups (as well as the whole match) as arguments, and its return value will be inserted into the new string.

Here’s a small example:

let s = "the cia and fbi";

console.log(s.replace(/\b(fbi|cia)\b/g,

str => str.toUpperCase()));

// → the CIA and FBI

Here’s a more interesting one:

let stock = "1 lemon, 2 cabbages, and 101 eggs";

function minusOne(match, amount, unit) {

amount = Number(amount) - 1;

if (amount == 1) { // only one left, remove the 's'

unit = unit.slice(0, unit.length - 1);

} else if (amount == 0) {

amount = "no";

}

return amount + " " + unit;

}

console.log(stock.replace(/(\d+) (\w+)/g, minusOne));

// → no lemon, 1 cabbage, and 100 eggs

## Greed:-

It is possible to use replace to write a function that removes all comments from a piece of JavaScript code. Here is a first attempt:

function stripComments(code) {

return code.replace(/\/\/.\*|\/\\*[^]\*\\*\//g, "");

}

console.log(stripComments("1 + /\* 2 \*/3"));

// → 1 + 3

console.log(stripComments("x = 10;// ten!"));

// → x = 10;

console.log(stripComments("1 /\* a \*/+/\* b \*/ 1"));

// → 1 1

The part before the or operator matches two slash characters followed by any number of non-newline characters. The part for multiline comments is more involved. We use [^] (any character that is not in the empty set of characters) as a way to match any character.

And that is exactly what we want in this case. By having the star match the smallest stretch of characters that brings us to a \*/, we consume one block comment and nothing more.

function stripComments(code) {

return code.replace(/\/\/.\*|\/\\*[^]\*?\\*\//g, "");

}

console.log(stripComments("1 /\* a \*/+/\* b \*/ 1"));

// → 1 + 1

## Dynamically creating RegExp objects:-

There are cases where you might not know the exact pattern you need to match against when you are writing your code. Say you want to look for the user’s name in a piece of text and enclose it in underscore characters to make it stand out. Since you will know the name only once the program is actually running, you can’t use the slash-based notation.

But you can build up a string and use the RegExp constructor on that. Here’s an example:

let name = "harry";

let text = "Harry is a suspicious character.";

let regexp = new RegExp("\\b(" + name + ")\\b", "gi");

console.log(text.replace(regexp, "\_$1\_"));

// → \_Harry\_ is a suspicious character.

When creating the \b boundary markers, we have to use two backslashes because we are writing them in a normal string, not a slash-enclosed regular expression.

To work around this, we can add backslashes before any character that has a special meaning.

let name = "dea+hl[]rd";

let text = "This dea+hl[]rd guy is super annoying.";

let escaped = name.replace(/[\\[.+\*?(){|^$]/g, "\\$&");

let regexp = new RegExp("\\b" + escaped + "\\b", "gi");

console.log(text.replace(regexp, "\_$&\_"));

// → This \_dea+hl[]rd\_ guy is super annoying.

## The search method:-

The indexOf method on strings cannot be called with a regular expression. But there is another method, search, that does expect a regular expression. Like indexOf, it returns the first index on which the expression was found, or -1 when it wasn’t found.

console.log(" word".search(/\S/));

// → 2

console.log(" ".search(/\S/));

// → -1

Unfortunately, there is no way to indicate that the match should start at a given offset (like we can with the second argument to indexOf), which would often be useful.

## The lastIndex property:-

Regular expression objects have properties. One such property is source, which contains the string that expression was created from. Another property is lastIndex, which controls, in some limited circumstances, where the next match will start.

Those circumstances are that the regular expression must have the global (g) or sticky (y) option enabled, and the match must happen through the execmethod. Again, a less confusing solution would have been to just allow an extra argument to be passed to exec, but confusion is an essential feature of JavaScript’s regular expression interface.

let pattern = /y/g;

pattern.lastIndex = 3;

let match = pattern.exec("xyzzy");

console.log(match.index);

// → 4

console.log(pattern.lastIndex);

// → 5

The difference between the global and the sticky options is that, when sticky is enabled, the match will succeed only if it starts directly at lastIndex, whereas with global, it will search ahead for a position where a match can start.

let global = /abc/g;

console.log(global.exec("xyz abc"));

// → ["abc"]

let sticky = /abc/y;

console.log(sticky.exec("xyz abc"));

// → null

Your regular expression might be accidentally starting at an index that was left over from a previous call.

let digit = /\d/g;

console.log(digit.exec("here it is: 1"));

// → ["1"]

console.log(digit.exec("and now: 1"));

// → null

When called with a global expression, instead of returning an array similar to that returned by exec, match will find *all*matches of the pattern in the string and return an array containing the matched strings.

console.log("Banana".match(/an/g));

// → ["an", "an"]

### Looping over matches:-

A common thing to do is to scan through all occurrences of a pattern in a string, in a way that gives us access to the match object in the loop body. We can do this by using lastIndex and exec.

let input = "A string with 3 numbers in it... 42 and 88.";

let number = /\b\d+\b/g;

let match;

while (match = number.exec(input)) {

console.log("Found", match[0], "at", match.index);

}

// → Found 3 at 14

// Found 42 at 33

// Found 88 at 40

## Parsing an INI file:-

To conclude the chapter, we’ll look at a problem that calls for regular expressions. Imagine we are writing a program to automatically collect information about our enemies from the Internet. (We will not actually write that program here, just the part that reads the configuration file. Sorry.) The configuration file looks like this:

searchengine=https://duckduckgo.com/?q=$1

spitefulness=9.7

; comments are preceded by a semicolon...

; each section concerns an individual enemy

[larry]

fullname=Larry Doe

type=kindergarten bully

website=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]

fullname=Davaeorn

type=evil wizard

outputdir=/home/marijn/enemies/davaeorn

The exact rules for this format (which is a widely used format, usually called an INI file) are as follows:

* Blank lines and lines starting with semicolons are ignored.
* Lines wrapped in [ and ] start a new section.
* Lines containing an alphanumeric identifier followed by an = character add a setting to the current section.
* Anything else is invalid.

Our task is to convert a string like this into an object whose properties hold strings for settings written before the first section header and subobjects for sections, with those subobjects holding the section’s settings.

Since the format has to be processed line by line, splitting up the file into separate lines is a good start. We saw the split method in [Chapter 4](https://eloquentjavascript.net/04_data.html#split). Some operating systems, however, use not just a newline character to separate lines but a carriage return character followed by a newline ("\r\n"). Given that the split method also allows a regular expression as its argument, we can use a regular expression like /\r?\n/ to split in a way that allows both "\n" and "\r\n" between lines.

function parseINI(string) {

// Start with an object to hold the top-level fields

let result = {};

let section = result;

string.split(/\r?\n/).forEach(line => {

let match;

if (match = line.match(/^(\w+)=(.\*)$/)) {

section[match[1]] = match[2];

} else if (match = line.match(/^\[(.\*)\]$/)) {

section = result[match[1]] = {};

} else if (!/^\s\*(;.\*)?$/.test(line)) {

throw new Error("Line '" + line + "' is not valid.");

}

});

return result;

}

console.log(parseINI(`

name=Vasilis

[address]

city=Tessaloniki`));

// → {name: "Vasilis", address: {city: "Tessaloniki"}}

The code goes over the file’s lines and builds up an object. Properties at the top are stored directly into that object, whereas properties found in sections are stored in a separate section object. The section binding points at the object for the current section.

There are two kinds of significant lines—section headers or property lines. When a line is a regular property, it is stored in the current section. When it is a section header, a new section object is created, and section is set to point at it.

Note the recurring use of ^ and $ to make sure the expression matches the whole line, not just part of it. Leaving these out results in code that mostly works but behaves strangely for some input, which can be a difficult bug to track down.

## International characters:-

Because of JavaScript’s initial simplistic implementation and the fact that this simplistic approach was later set in stone as standard behavior, JavaScript’s regular expressions are rather dumb about characters that do not appear in the English language. For example, as far as JavaScript’s regular expressions are concerned, a “word character” is only one of the 26 characters in the Latin alphabet (uppercase or lowercase), decimal digits, and, for some reason, the underscore character. Things like é or β, which most definitely are word characters, will not match \w (and will match uppercase \W, the nonword category).

By a strange historical accident, \s (whitespace) does not have this problem and matches all characters that the Unicode standard considers whitespace, including things like the nonbreaking space and the Mongolian vowel separator.

Another problem is that, by default, regular expressions work on code units, as discussed in [Chapter 5](https://eloquentjavascript.net/05_higher_order.html#code_units), not actual characters. This means characters that are composed of two code units behave strangely.

console.log(/🍎{3}/.test("🍎🍎🍎"));

// → false

console.log(/<.>/.test("<🌹>"));

// → false

console.log(/<.>/u.test("<🌹>"));

// → true

The problem is that the 🍎 in the first line is treated as two code units, and the {3} part is applied only to the second one. Similarly, the dot matches a single code unit, not the two that make up the rose emoji.

You must add a u option (for Unicode) to your regular expression to make it treat such characters properly. The wrong behavior remains the default, unfortunately, because changing that might cause problems for existing code that depends on it.

Though this was only just standardized and is, at the time of writing, not widely supported yet, it is possible to use \p in a regular expression (that must have the Unicode option enabled) to match all characters to which the Unicode standard assigns a given property.

console.log(/\p{Script=Greek}/u.test("α"));

// → true

console.log(/\p{Script=Arabic}/u.test("α"));

// → false

console.log(/\p{Alphabetic}/u.test("α"));

// → true

console.log(/\p{Alphabetic}/u.test("!"));

// → false

Unicode defines a number of useful properties, though finding the one that you need may not always be trivial. You can use the \p{Property=Value} notation to match any character that has the given value for that property. If the property name is left off, as in \p{Name}, the name is assumed to be either a binary property such as Alphabetic or a category such as Number.

## Summary:-

Regular expressions are objects that represent patterns in strings. They use their own language to express these patterns.

|  |  |
| --- | --- |
| /abc/ | A sequence of characters |
| /[abc]/ | Any character from a set of characters |
| /[^abc]/ | Any character not in a set of characters |
| /[0-9]/ | Any character in a range of characters |
| /x+/ | One or more occurrences of the pattern x |
| /x+?/ | One or more occurrences, nongreedy |
| /x\*/ | Zero or more occurrences |
| /x?/ | Zero or one occurrence |
| /x{2,4}/ | Two to four occurrences |
| /(abc)/ | A group |
| /a|b|c/ | Any one of several patterns |
| /\d/ | Any digit character |
| /\w/ | An alphanumeric character (“word character”) |
| /\s/ | Any whitespace character |
| /./ | Any character except newlines |
| /\b/ | A word boundary |
| /^/ | Start of input |
| /$/ | End of input |

A regular expression has a method test to test whether a given string matches it. It also has a method exec that, when a match is found, returns an array containing all matched groups. Such an array has an index property that indicates where the match started.

Strings have a match method to match them against a regular expression and a search method to search for one, returning only the starting position of the match. Their replace method can replace matches of a pattern with a replacement string or function.

### Regexp golf:-

Code golf is a term used for the game of trying to express a particular program in as few characters as possible. Similarly, regexp golf is the practice of writing as tiny a regular expression as possible to match a given pattern, and only that pattern.

For each of the following items, write a regular expression to test whether any of the given substrings occur in a string. The regular expression should match only strings containing one of the substrings described. Do not worry about word boundaries unless explicitly mentioned. When your expression works, see whether you can make it any smaller.

1. car and cat
2. pop and prop
3. ferret, ferry, and ferrari
4. Any word ending in ious
5. A whitespace character followed by a period, comma, colon, or semicolon
6. A word longer than six letters
7. A word without the letter e (or E)

Refer to the table in the [chapter summary](https://eloquentjavascript.net/09_regexp.html#summary_regexp) for help. Test each solution with a few test strings.

// Fill in the regular expressions

verify(/.../,

["my car", "bad cats"],

["camper", "high art"]);

verify(/.../,

["pop culture", "mad props"],

["plop", "prrrop"]);

verify(/.../,

["ferret", "ferry", "ferrari"],

["ferrum", "transfer A"]);

verify(/.../,

["how delicious", "spacious room"],

["ruinous", "consciousness"]);

verify(/.../,

["bad punctuation ."],

["escape the period"]);

verify(/.../,

["hottentottententen"],

["no", "hotten totten tenten"]);

verify(/.../,

["red platypus", "wobbling nest"],

["earth bed", "learning ape", "BEET"]);

function verify(regexp, yes, no) {

// Ignore unfinished exercises

if (regexp.source == "...") return;

for (let str of yes) if (!regexp.test(str)) {

console.log(`Failure to match '${str}'`);

}

for (let str of no) if (regexp.test(str)) {

console.log(`Unexpected match for '${str}'`);

}

}

### Quoting style:-

Imagine you have written a story and used single quotation marks throughout to mark pieces of dialogue. Now you want to replace all the dialogue quotes with double quotes, while keeping the single quotes used in contractions like aren’t.

Think of a pattern that distinguishes these two kinds of quote usage and craft a call to the replace method that does the proper replacement.

let text = "'I'm the cook,' he said, 'it's my job.'";

// Change this call.

console.log(text.replace(/A/g, "B"));

// → "I'm the cook," he said, "it's my job."

### Numbers again

Write an expression that matches only JavaScript-style numbers. It must support an optional minus or plus sign in front of the number, the decimal dot, and exponent notation—5e-3 or 1E10—again with an optional sign in front of the exponent. Also note that it is not necessary for there to be digits in front of or after the dot, but the number cannot be a dot alone. That is, .5 and 5. are valid JavaScript numbers, but a lone dot isn’t.

// Fill in this regular expression.

let number = /^...$/;

// Tests:

for (let str of ["1", "-1", "+15", "1.55", ".5", "5.",

"1.3e2", "1E-4", "1e+12"]) {

if (!number.test(str)) {

console.log(`Failed to match '${str}'`);

}

}

for (let str of ["1a", "+-1", "1.2.3", "1+1", "1e4.5",

".5.", "1f5", "."]) {

if (number.test(str)) {

console.log(`Incorrectly accepted '${str}'`);

}

}