Regular Expressions

From CSE330 Wiki

Is this string a phone number? Is it an e-mail address? Is it a URL? How do I parse the time of day out of this date string? How do I find all words in the string that contain the letter "a"?

The tool we use to answer these questions is **regular expressions**. Regular expressions are a powerhouse for string matching.

This guide serves as an introduction to regular expressions. However, there is enough to discuss about them that regular expressions could be their *own class*. We will only be scratching the surface so that you can get up and running with regular expressions and start to use them in your projects.

Contents

Testing Regular Expressions

There are several tools freely available online to assist you when writing regular expressions. In this wiki, we will be using two:

- **RegExPal (http://regexpal.com/) :** Performs a global search on an input string. Best for writing regular expressions used for parsing text.
- **Debuggex (http://www.debuggex.com/) :** Matches a regular expression against the input string, but does not support searches. Shows a schematic to help you understand what your regular expression is doing. Best for fine-tuning regular expressions that act on known input formats.

You can view all regular expressions on this page in RegExPal or Debuggex by pressing "View in _____" above each example.

Your First Regular Expression

Here we go. If you wish to follow along, click "View in RegExPal" below; Debuggex does not work well for this example.

View in Debuggex (http://www.debuggex.com/?re=%5Cb%5Cware%5Cb&str=The+tortoise+and+the+hare+took+on+a+dare+to+find+the+rare+Aare.) | View in RegExPal (http://regexpal.com/?regex=%5Cb%5Cware%5Cb&input=The+tortoise+and+the+hare+took+on+a+dare+to+find+the+rare+Aare.)

```
\b\ware\b
```

This regular expression finds all words of the form "_are", where _ is an alphanumeric character, and matches that first letter. Let's go through the parts of this regular expression.

- **\b** means "word boundary". If we didn't have the **\b**'s, then this regular expression would also match words like "day**care**" or "app**are**l" or "**are**st".
- **\w** means "any alphanumeric character or underscore". Thus, this regular expression will match **dare**, **care**, **zare**, **5are**, **_are**, and so on.
- **are** are literal characters in the regular expression.

Here is a little animation to help you visualize how this regular expression is assembled:

Regular Expression Syntax

Character classes and modifiers enable you to fine-tune your matches..

Character Classes

A **character class** enables you to match any one of a set of characters. Surround your characters of interest in square brackets.

The following regular expression matches all occurrences of a vowel:

View in Debuggex (http://www.debuggex.com/?re=%5Baeiou%5D&str=The+quick+brown+fox+jumps+over+the+lazy+dog.) | View in RegExPal (http://regexpal.com/?regex=%5Baeiou%5D&input=The+quick+brown+fox+jumps+over+the+lazy+dog.)

```
[aeiou]
```

Character classes can be *negated* by starting the character set with a caret `^`. For example, the following regular expression matches all occurrences of a consonant:

View in Debuggex (http://www.debuggex.com/?re=%5B%5Eaeiou%5CW%5D&str=The+quick+brown+fox+jumps+over+the+lazy+dog.) | View in RegExPal (http://regexpal.com/?regex=%5B%5Eaeiou%5CW%5D&input=The+quick+brown+fox+jumps+over+the+lazy+dog.)

```
[^aeiou\W]
```

The **\W** is a shorthand character class that prevents the above regular expression from also matching non-word characters.

Shorthand Character Classes

Some character classes are provided for you by default.

Description	Literal Character Class	Shorthand
Digit	[0-9]	\d
Word Character	[A-Za-z0-9_]	\w
Whitespace	[\t\r\n]	\s
Non-Digit	[^0-9]	\D
Non-Word Character	[^A-Za-z0-9_]	\W
Non-Whitespace	[^ \t\r\n]	\S
Wildcard character except for line breaks	[^\r\n]	.

Quantifiers

For a character, character class, or group, you can specify a required number of occurrences.

Description	Syntax	Example
Zero or more of	*	Match any string, even the empty string: `.*`
One or more of	+	Match any string, except for the empty string: `.+`
Zero or one of	?	Match either "color" or "colour": `colou?r`
N of	{N}	Match all three-digit numbers: `\b\d{3}\b`
Between *M* and *N* of	{M,N}	Match all two- or three-digit numbers: `\b\d{2,3}\b`

Greedy and Lazy Quantifiers

By default, the ***** and **+** quantifiers are **greedy**: this means that they will continue searching until the end of the string if they can. More often, however, you want them to stop as soon as something matching the next part of your regular expression is found. This is where **lazy** modifiers come into play.

To make ***** or **+** lazy, simply add a **?**, as in ***?** and **+?**.

For example, you might write the following regular expression to match all HTML tags (view in RegExPal to follow along):

View in Debuggex (http://www.debuggex.com/?re=%3C.%2B%3E&str=Hello+%3Cp%3E%3Cstrong%3ERoses%3C%2Fstrong%3E+are+%3Cem+class%3D%22red%22%3ERed%3C%2Fem%3E% | View in RegExPal (http://regexpal.com/?regex=%3C.%2B%3E&input=Hello+%3Cp%3E%3Cstrong%3ERoses%3C%2Fstrong%3E+are+%3Cem+class%3D%22red%22%3ERed%3C%2Fem

```
<.+>
```

However, this will not separately match the HTML tags; instead, it will start from the first occurrence of **<** and continue until the *last* occurrence of **>**. This is the *greedy* behavior. The following regular expression will perform what you want:

View in Debuggex (http://www.debuggex.com/?re=%3C.%2B%3F%3E&str=Hello+%3Cp%3E%3Cstrong%3ERoses%3C%2Fstrong%3E+are+%3Cem+class%3D%22red%22%3ERed%3C%2Fem% | View in RegExPal (http://regexpal.com/?regex=%3C.%2B%3F%3E&input=Hello+%3Cp%3E%3Cstrong%3ERoses%3C%2Fstrong%3E+are+%3Cem+class%3D%22red%22%3ERed%3C%2

```
<.+?>
```

Now, the match will stop as soon as a **>** is encountered.

Anchors

If your regular expression begins with a cared **^**, you will generate only one match, which must be at the beginning of the string. If your regular expression ends with a dollar sign **$**, you will generate only one match, which must be at the end of the string. If your regular expression starts with a **^** *and* ends with a **$**, you will match if and only if the entire string matches the entire regular expression.

For example, the following regular expression generates a match on all strings that start with a capital letter:

View in Debuggex (http://www.debuggex.com/?re=%5E%5BA-Z%5D&str=Hello+World) | View in RegExPal (http://regexpal.com/?regex=%5E%5BA-Z%5D&input=Hello+World)

```
^[A-Z]
```

The following regular expression tests whether the entire string is a valid US phone number:

View in Debuggex (http://www.debuggex.com/?re=%5E%5Cd%7B3%7D-%5Cd%7B3%7D-%5Cd%7B4%7D%24&str=314-935-5555) | View in RegExPal (http://regexpal.com/?regex=%5E%5Cd%7B3%7D-%5Cd%7B3%7D-%5Cd%7B4%7D%24&input=314-935-5555)

```
^\d{3}-\d{3}-\d{4}$
```

Groups

Groups enable you to perform operations on multi-character strings. To specify a group, surround it with parentheses **()**.

For example, the following regular expression crudely matches a domain name:

View in Debuggex (http://www.debuggex.com/?re=%5B%5Cw%5C-%5D%2B%28%5C.%5B%5Cw%5C-%5D%2B%29%2B&str=example.co.uk) | View in RegExPal (http://regexpal.com/?regex=%5B%5Cw%5C-%5D%2B%28%5C.%5B%5Cw%5C-%5D%2B%29%2B&input=example.co.uk)

```
[\w\-]+(\.[\w\-]+)+
```

First, we match one or more word character or hyphen, and then we match one or more suffixes, each suffix consisting of a period followed by one or more word characters.

Regular Expression Examples

You know know enough to write some pretty sophisticated regular expressions.

Can you figure out what the following regular expression matches?

View in Debuggex (http://www.debuggex.com/?re=%5E%5B%5Cw%21%23%24%25%26%27%2A%2B%2F%3D%3F%5E_%60%7B%7D%7E-%5D%2B%40%5B%5Cw%5C-%5D%2B%28%5C.%5B%5Cw%5C-%5D%2B%29%2B%24&str=me%40example.com) | View in RegExPal (http://regexpal.com/?regex=%5E%5B%5Cw%21%23%24%25%26%27%2A%2B%2F%3D%3F%5E_%60%7B%7D%7E-%5D%2B%40%5B%5Cw%5C-%5D%2B%28%5C.%5B%5Cw%5C-%5D%2B%29%2B%24&input=me%40example.com)

```
^[\w!#$%&'*+/=?^_`{}~-]+@[\w\-]+(\.[\w\-]+)+$
```

If you guessed "e-mail address", you are correct! You can recognize that our domain-name regular expression has been copied after the "@" sign, and before the "@" sign is simply one or more of an alphanumeric or a handful of other characters. We also surrounded the regex with **^** and **$** to ensure that we match the entire string.

Can you guess this one?

View in Debuggex (http://www.debuggex.com/?re=%5E%5Cd%7B1%2C3%7D%28%5C.%5Cd%7B1%2C3%7D%29%7B3%7D%24&str=128.252.202.246) | View in RegExPal (http://regexpal.com/?regex=%5E%5Cd%7B1%2C3%7D%28%5C.%5Cd%7B1%2C3%7D%29%7B3%7D%24&input=128.252.202.246)

```
^\d{1,3}(\.\d{1,3}){3}$
```

This will match an IP address (IPv3). We use the quantifier-on-group trick like we did earlier when we matched valid domain names.

Note: Both of these examples are crude. You should use formal regular expressions issued by organizations like the IETF when matching e-mail addresses and the like.

Groups

One use of regular expressions is when you want to extract information of a known format out of a string. This is when *groups* come into play.

For example, consider the phone number expression from before. Suppose we test it against the phone number "123-456-7890":

View in Debuggex (http://www.debuggex.com/?re=%5Cd%7B3%7D-%28%5Cd%7B3%7D%29-%5Cd%7B4%7D&str=123-456-

7890) | View in RegExPal (http://regexpal.com/?regex=%5Cd%7B3%7D-%28%5Cd%7B3%7D%29-%5Cd%7B4%7D&input=123-456-7890)

```
\d{3}-(\d{3})-\d{4}
```

Notice how we put parentheses around the second set of digits. This makes the second set of digits a *group*. View the regular expressin in Debuggex, and you can see how the second set of numbers in the phone number is matched as the group.

What happens if we put parentheses around all three?

View in Debuggex (http://www.debuggex.com/?re=%28%5Cd%7B3%7D%29-%28%5Cd%7B3%7D%29-%28%5Cd%7B4%7D%29&str=123-456-7890) | View in RegExPal (http://regexpal.com/?regex=%28%5Cd%7B3%7D%29-%28%5Cd%7B3%7D%29-%28%5Cd%7B4%7D%29&input=123-456-7890)

```
(\d{3})-(\d{3})-(\d{4})
```

We now have *three* groups matched:

1. **123**
2. **456**
3. **7890**

We can also have groups within other groups. For example, we could match the area code (the first set), the office code (the second set), the station code (the third set), and the subscriber code (the second and third sets combined):

View in Debuggex (http://www.debuggex.com/?re=%28%5Cd%7B3%7D%29-%28%28%5Cd%7B3%7D%29-%28%5Cd%7B4%7D%29%29&str=123-456-7890) | View in RegExPal (http://regexpal.com/?regex=%28%5Cd%7B3%7D%29-%28%28%5Cd%7B3%7D%29-%28%5Cd%7B4%7D%29%29&input=123-456-7890)

```
(\d{3})-((\d{3})-(\d{4}))
```

Open that in Debuggex to visualize how the groups are formed.

Different programming languages all have different ways to extract groups from a match. These different methods are documented in the next section.

Non-Capturing Groups

You can ignore certain groups from being matched by appending **?:** to the start of the parentheses. For example, if you wanted to ignore matches from the group in the URL regex we made above, you would add **?:** in the correct spot:

View in Debuggex (http://www.debuggex.com/?re=%5B%5Cw%5C-%5D%2B%28%3F%3A%5C.%5B%5Cw%5C-%5D%2B%29%2B&str=bbc.co.uk) | View in RegExPal (http://regexpal.com/?regex=%5B%5Cw%5C-%5D%2B%28%3F%3A%5C.%5B%5Cw%5C-%5D%2B%29%2B&input=bbc.co.uk)

```
[\w\-]+(?:\.[\w\-]+)+
```

Using Regular Expressions in Programming Languages

You are learning three new programming languages in CSE 330: PHP (from Module 2), Python (from Module 4), and JavaScript (from Module 6). Below, you may find how to implement regular expressions in each of these three languages.

The example is a function that tests whether a string matches a regular expression representing an e-mail address, and if it does, the function returns the domain name from the e-mail.

PHP

The magic function in PHP is **preg_match($regex, $str, $matches)**, where *$regex* is the regular expression, *$str* is the string to test, and *$matches* is an extra argument which will be modified to contain the matches array. By convention, the regular expression itself must be surrounded in forward slashes.

```
function domain_from_email($str){
    $email_regex = "/^[\w!#$%&'*+\/=?^_`{|}~-]+@([\w\-]+(?:\.[\w\-]+)+)$/";
    if(preg_match($email_regex, $str, $matches)){
        return $matches[1];
    } else return false;
}

echo domain_from_email("sam@bbc.co.uk"); // bbc.co.uk
```

If you want to perform a global search for *all* matches of a regular expression, use **preg_match_all($regex, $str, $matches)** instead.

```
function find_all_email_domains($str){
    $email_regex = "/\b[\w!#$%&'*+\/=?^_`{|}~-]+@([\w\-]+(?:\.[\w\-]+)+)\b/";
    preg_match_all($email_regex, $str, $matches);
    return $matches[1];
}

print_r(find_all_email_domains("sam@bbc.co.uk, julie@amazon.com, rick@wustl.edu"));
// Array ( [0] => bbc.co.uk [1] => amazon.com [2] => wustl.edu )
```

Python

To use regular expressions in Python, you need to import the **re** module. From there, you can compile your regular expression and perform searches with it.

```
import re

email_regex = re.compile(r"^[\w!#$%&'*+/=?^_`{|}~-]+@([\w\-]+(?:\.[\w\-]+)+)$")
```

```
def domain_from_email(test):
    match = email_regex.match(test)
    if match is not None:
        return match.group(1)
    else:
        return False

print(domain_from_email("sam@bbc.co.uk")) # bbc.co.uk
```

Note that we prefix our regex string with an "r", for a "raw string". This is to prevent Python from trying to parse special characters in our regular expression.

To perform global searches, use the regex **findall** method:

```
import re

email_regex = re.compile(r"\b[\w!#$%&'*+/=?^_`{|}~-]+@([\w\-]+(?:\.[\w\-]+)+)\b")

def find_all_email_domains(test):
    return email_regex.findall(test)

print(find_all_email_domains("sam@bbc.co.uk, julie@amazon.com, rick@wustl.edu"))
# ['bbc.co.uk', 'amazon.com', 'wustl.edu']
```

In Python if you want to use the \b in a regex string (without declaring the string as raw) you need to use the \\b instead of \b . Consider the examples below, which provide the same output, the first uses the raw input parameter, the second does not.

```
import re

re.match(r"\bhello world\b","hello world").group()

re.match("\\bhello world\\b","hello world").group()
```

JavaScript

Regular expressions in JavaScript are their own native data type, so native that they are expressed using literals surrounded with forward slashes and not even strings.

To perform a simple match, call the **test** method on the regular expression:

```
alert(/^\d+$/.test("123")); // true
```

To find a match, call the **exec** method on the regular expression:

```
var emailRegex = /^[\w!#$%&'*+\/=?^_`{|}~-]+@([\w\-]+(?:\.
[\w\-]+)+)$/

function domainFromEmail(str){
    if(match=emailRegex.exec(str)){
        return match[1];
    }
}

alert(domainFromEmail("sam@bbc.co.uk")); // bbc.co.uk
```

Watch on DOCTYPE (http://www.youtube.com/watch?v=vpqV8n56J6w&t=5m34s)

Regular Expressions in JavaScript Part 2.

Continue calling the **exec** method to produce more matches:

```
var emailRegex = /\b[\w!#$%&'*+\/=?^_`{|}~-]+@([\w\-]+(?:\.[\w\-]+)+)\b/g

function findAllEmailDomains(str){
    var matches = []
    while(match=emailRegex.exec(str)){
        matches.push(match[1]);
    }
    return matches;
}

alert(findAllEmailDomains("sam@bbc.co.uk, julie@amazon.com, rick@wustl.edu"));
// bbc.co.uk,amazon.com,wustl.edu
```

Watch on DOCTYPE (http://www.youtube.com/watch?v=7dVfzg22ono&t=3n

Basics of Regular Expressions. Uses JavaScript as the language of focus.

Watch on DOCTYPE (http://www.youtube.com/watch?v=WwsNPSO7J-I&t=1m48s)

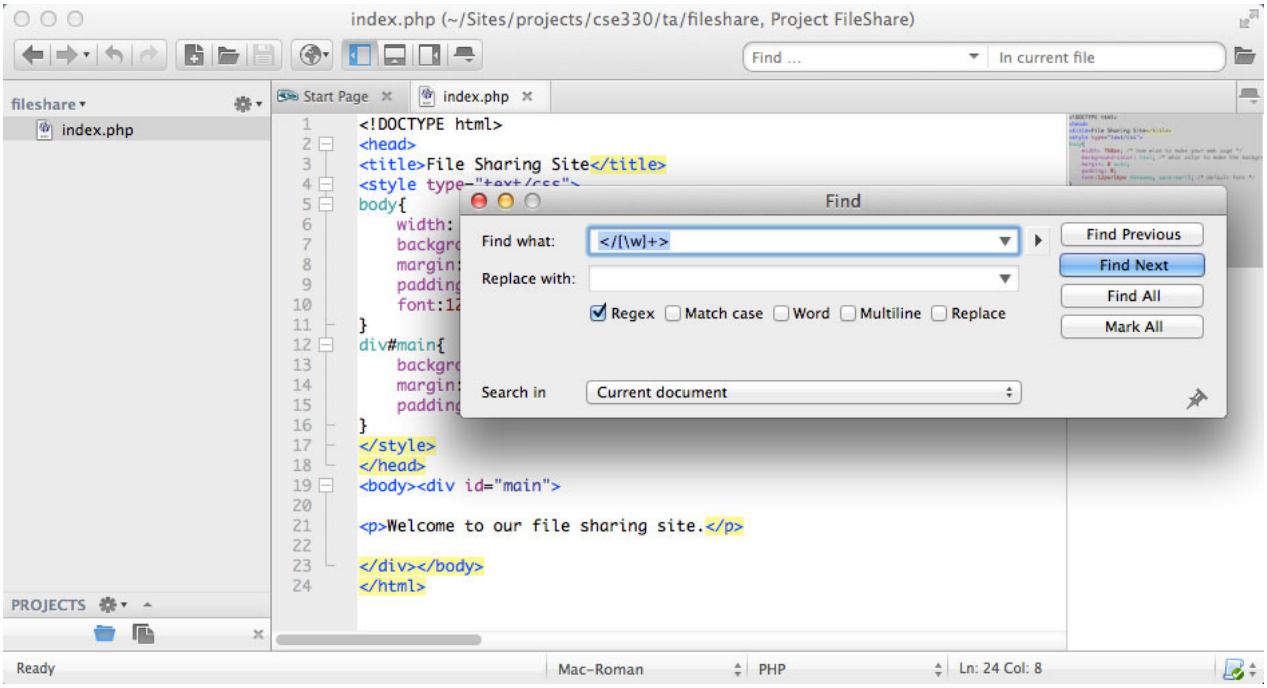
Regular Expressions in JavaScript Part 3.

Note: In order to make the regular expression "searchable", we need to add the **g** flag in its definition. The flag goes after the closing forward slash.

Using Regular Expressions in Your Workflow

Regular expressions aren't limited to use in programming languages. In fact, you can use them *right now* in your text editor!

In Komodo, go to **File->Find**, and in the dialog, select "Regex". Bam: whatever you type into the search box will be evaluated as a regular expression! Here is an example where I find all end tags in my document:



You can also do regular expression **find and replace** in Komodo. You can use groups, too: in the "Replace with" box, a **\1** will insert your first group, a **\2** will insert your second group, and so on. Nifty!

Categories: Exclude in print | Module 4