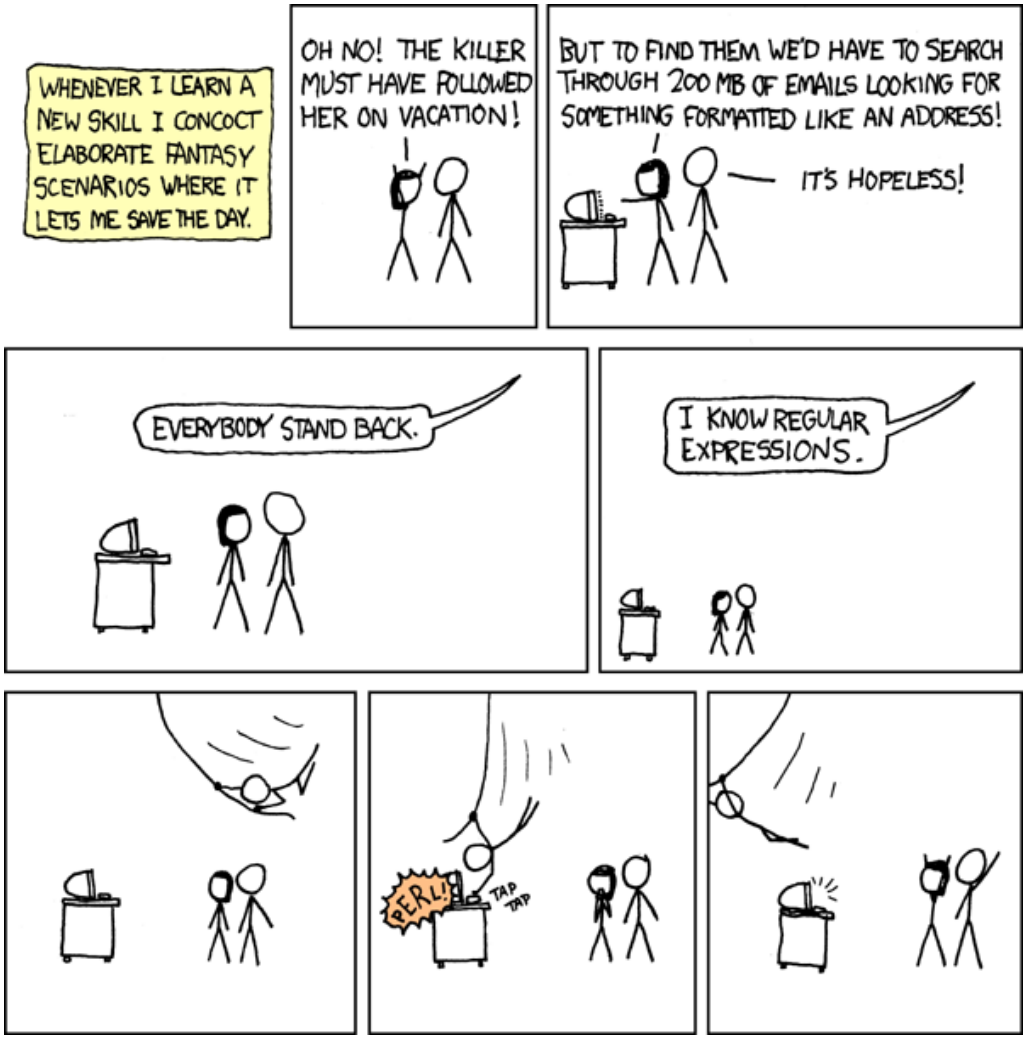
Regular Expressions Julia Style



https://xkcd.com/208/

WWWWWW

What

A *regular expression* or *regex* or *regexp* is a way of expressing a pattern.

Why

Regular expressions are used primarily as a means to find or match patterns of characters in strings. They can be used to validate text, or extract a part, or replace text.

As such, they are one of the most powerful tools available for working with text and data in general.

When

The concept arose in the 50s, but came into wider use with the rise of Unix tools such as `sed`.

The programming language Perl brought in a alternative way of supporting regexes compared to the unix standard (in the late 80s), and the power of Perl's version (particularly in Perl 5 in the 90s) lead to it becoming dominant.

Who

Kleene is responsible for much of the idea of formalising the core concept. It arose in the context of *regular languages* as a fundamental concept in computer science, but it is noteworthy that modern regular expressions are not formally regular for reasons I won't delve into here.

Larry Wall has probably had the most influence on the modern Perl-like syntax for regular expressions.

N.B. PERL = *Practical Extraction and Report Language* or *Pathologically Eclectic Rubbish Lister*.

Where

Regexes (in their pure sense) are ideal for implementation in interpreted languages. A regex engine should be able to process the regex with one pass and finite memory.

Real regexes now use lots of tricks that break their pure nature, but also they have been ported into most modern high-level languages in some form, including some compiled languages like Julia. So perhaps their exact nature has become moot.

How

That's what we are going to talk about now.

Perl Compatible Regular Expressions (PCRE)

Perl is a much hated language. What can you do? Haters r' gonna hate.

But Perl was (and is) very useful for certain tasks, in particular working with text. One of the features that made it useful was a sophisticated regular expression engine and syntactic sugar to go with it.

Perl still exists. It is less favoured, but still 18th on Tiobe's list.

However, one of its most lasting legacies is the PCRE. Many modern languages, Julia included, that have included regular expressions have done so through the PCRE libraries (written in C) or compatible code. The PCRE libraries are not 100% the same as Perl regexes. The detailed differences probably won't matter for anyone except old-hand Perl programmers (e.g. PCRE has a hard limit on recusions and Perl doesn't -- not sure why I should have expected infinite recursion though).

Hence, what follows is a quick explanation of the PCRE syntax for regular expressions, which will apply for many modern programming languages We will follow that with some examples, and then an explanation of how to use these in Julia. Note that many tutorials on regexs are in the context of a particular language, and so have constructions based on that language, but I want to start with just the regex, and we will get into language specifics later.

A PCRE is a pattern match criteria consisting itself of a sequence of characters, but in the regex the characters can have special meanings. Some just mean themselves, e.g., standard ASCII characters like `'a'` mean themselves (by default). Others match special groups of characters, and others combine together to have special meanings. The Cheat Sheets here and here are helpful, and cover more cases than I can here. I will just go over the basics, which are actually the things I use 95% of the time.

Regex	Meaning
.	Any character except newline
a	The character a
ab	The string ab
a|b	a or b
a*	0 or more a's
a+	1 or more a's
a?	0 or 1 a's
a{2}	exactly 2 a's and there are many variants like a{2,5} or a{2,}
\	*Escape* in combination with other characters (see next table)
[abc]	Any character from the set {a,b,c}
[a-c]	Any character from the range a to c
[^abc]	Any character except {a,b,c}
[[:xxx:]]	Special groups, e.g. xxx=alnum,alpha,ascii,lower,punt,space,word,...
^	Start of a string
$	End of a string
(...)	Group together and 'capture' (more on this later)
(?:...)	Group without capturing
(?'name':)	Named capturing group

Roughly these are divided into

- literals, e.g., 'a'
- alternation `|`
- quantifiers, e.g., `+, *, ?, {2,4}`
- character classes, e.g., `[abc],` `[[:xxx]]` and some of the escape codes below.
- assertions (anchors), e.g., `^` or `$`
- capture groups and backreferences (see below)
- escape codes (see below).

There are many more, but this is enough to get started in conjunction with the escape codes below.

Regex	Meaning
\d	One digit = [0-9]
\D	One non-digit = [^\s]
\s	One whitespace (space, tab, endline) = [\f\t\n\r]
\S	One non-whitespace = [^\s]
\w	One word character (a letter or digit or underscore) = [a-zA-Z0-9_]
\W	One non-word character = [^\w]
\n	Newline
\t	Tab
\b	Word boundary (an 'anchor' not a character match)
\B	Not a word boundary
\\	Backslash
\(	A left bracket, and similarly \ is used to escape any other character like or \^
\1	A backreference -- related to grouping and capturing (more later)

There are others. Note in particular the role of backslash in "escaping" other special characters such as (,), {,},\,*,.,?,+,[,], ^, $, ...

Some regular expressions *capture* part of the regex. This is because when you match you often want to (i) check that a string has a format, and simultaneously (ii) extract a particular piece of the match. Round brackets let you do such captures. They are particularly useful when you want to find-and-replace some text in a single line of code. They can also be used to match something that has already appeared in your expression. For instance, to find double words.

There is much more to know here, but one extra topic I need to mention is that many regex engines (including Julia's) are, by default, *greedy*. That means they will match the biggest chunk they can. There are ways to specigfy alternatives (reluctant and so on).

A note on language compatibility: many languages handle strings in different ways. So although they use the PCRE2 format, they may require you to do extra things, e.g., \ is often used a special character, so you might need to double escape it, e.g., type something like `'\\d'` when inputting a regex. Julia is not one of those languages.

One final note. You may hear of PCRE2. This is the current version (PCRE is considered obsolete). There are subversions (series) inside these, but again, you are unlikely to care about most of the details as they are largely about performance, and consistency for Unicode, and you are unlikely to be using PCRE1 in most of your day-to-day.

Examples

I will present a few examples to make this clearer. The examples have three parts

1. The regex (excluding the syntax that defines it as a regex, e.g., some form of quotation around the regex)
2. A string
3. The matching part of the string is indicated in bold if there is one.

Regex	String	Notes
World	Hello **World**	
world	Hello World	no match because of the case
.*World	**Hello World**	greedy match
hat	T**hat** was my red **hat**	Two possible matches
^house	**house**keeper	
house$	housekeeper	no match because of the end-of-string
[ba]+	**ab**out	
[\d]*	The **300** Spartans	
\bcat\b	The **cat** in the hat	
\bcat\b	The catalogue	no match because of no word boundary
cat|cats	**cat**s	matches the earliest alternative
cat[s]?	**cats**	does the greedy match
(\d\d):(\d{2}):(\d{2})	**17:21:00**	matches a time, capturing hh,mm,s[s]
gr[ae]y	**gray**	could also match **grey**
abc{2}	abcabc	No match as it matches **abcc**
a(bc){2}	**abcbc**	
a(b|c)	**ab**	Also matches **ac**

Some bigger examples:

- Finding double words in text:

```
(\b(\w+)\b\s+\b\2\b)
```

Explanation: the outer brackets are to capture the match (to use it). The inner brackets `(\w+)` capture a word. The later `\2` refers to this first word. So this regex is looking for a word `\b\w+\b`, then a set of at least one space, then a repeat of this word with appropripriate word boundaries.

- Match an email:

```
\b[\w-\.]+@\b([\w-]+\.){1,4}+[\w]\b
```

Explanation: An email like `matthew.roughan@adelaide.edu.au` has

1. a name `[\w-\.]+`, which is a list of one or more word characters, but might also include some limited punctuation such as - or a full stop.
2. the `@`
3. a set of 1-4 dot-separated pieces, `([\w-]+\.)`, e.g., adelaide.
4. a final `com` or `edu` or other top-level domain.

N.B. Actually emails can be more complicated than this, but this will find 99% of them. The name can actually contain other things like its own escaped characters

- Match a floating point number

```
[-+]?([0-9]*\.[0-9]+|[0-9]+)
```

Explanation: Obv. As we are matching numbers like -0.42345, we are mainly matching digits, which we could do with `\d` or the range `[0-9]`. There is an arbitrary number of digits before the decimal, with an optional plus or minus at the start. Then there are digits after the decimal. Alternatively, we just allow an integer like 1234, with no decimal at all.

Once again, the real version of this is more complicated as you may have to take into account scientific notation and other variations. See this for a more complete version.

Other common uses: trim excessive whitespace, find HTML tags, verifying credit card numbers,

Julia Regex

The previous stuff was about the generic idea, but you also need a wayu to work with them in any given language. How do you do this in Julia?

Construction of regexes

Julia has two main approaches to constructing regular expressions. The first uses a *string literal* modifier. Its very easy -- you just put a `r` before the first float of string literal, e.g.,

```
julia> regex = r"match this"
```

That is the most common approach for me, but it has a key limitation in that sometimes you want to construct a regular expression programmatically using information that is only available at run time. For example you might want to test a name against a user input. Then you need to construct the regex using the constructor, e.g., the example above is

```
julia> regex = Regex("match this")
```

However, the input to the constructor is a string, and this can be in turn created, for instance, by string interpolation or some other construction. For example:

```
julia> x = "this"
julia> regex = Regex("match $x")
```

This approach is a little more flexible.

Triple quoted regex definitions are also supported, e.g., `r"""..."""` but I haven't used these much.

Use of regexes

Search and Match

There is a function called `match`, e.g.,

```
julia> m = match(r"(cat|dog)s?", "my mice are too big")
julia> println(m)
nothing
```

Here there was no match, and so the function returns `nothing`. If instead there is a match, you get something like this:

```
julia> m = match(r"(cat|dog)s?", "my cats are dogs")
RegexMatch("cats", 1="cat")
```

You can see that the return type is a `RegexMatch`, which is a little weird. A `RegexMatch` is a composite type that has fields:

- match -- a `SubString` containing the matched text;
- captures -- a vector of capture groups;
- offset -- the offset of the match;
- offsets -- the offsets of the capture group; and
- regex -- the regex used.

So

```
julia> m = match(r"(cat|dog)s?", "my cats are dogs")
julia> m.match
"cats"
julia> m.captures
1-element Vector{Union{Nothing, SubString{String}}}:
 "cat"
julia> m.regex
r"(cat|dog)s?"
```

Note the differences between the captured group and the overall match.

There is an `eachmatch` which is an iterator, so you can do things like

```
julia> rx = r"a.a"
julia> m = eachmatch(rx, "a1a2a3a")
julia> for m in  eachmatch(rx, "a1a2a3a")
          println(m)
       end
RegexMatch("a1a")
RegexMatch("a3a")
```

There is also a `occursin` to just check if the thing you are looking for exists.

Underlying all of these is a `needle, haystack` philosophy, i.e., the input of the "needle" (the thing you are looking for) precedes the "haystack" (the thing you are looking through).

There are some optional extras in some of these functions, but you can look them up yourself. Named capture groups and so on can all be used in cute ways.

N.B. Names of some of these functions have changed over time. These seem to be stable now.

Replacement and substitution

The other main use for regexes, historically at least, is in directly replacing parts of a string. You can use regexs very simply in replacements to replace the regex with a string as follows:

```
julia> str = replace("catsss are cool", r"cats+" => "dogs")
```

However, often you don't want to replace the whole match. You want to replace, but reuse. You need to know an extra thing in order to do such replacement: to do something more sophisticated than just replace a string with a string, you will need a second type of object called a `SubstitutionString` which is specified by using `s"..."`. For example:

```
julia> str = replace("cats", r"(\w+)s" => s"\1")
"cat"
```

Mind you this example isn't the best way to remove an 's' from a string, but it might give you the idea.

Capture groups are incredibly powerful, but also not trivial to use well.

Also note that `replace` is heavily overloaded, so pay attention to which version you are using.

More Tips and Tricks

Performance

There are untold ways to create regexes for any occasion. Unfortunately, it is highly opaque to understand which approaches will be fast, and which slow.

Some approaches that can help:

- Be more specific (avoid generic matches that then somehow get refined). Often a longer regex is better, because it will be more specific. The faster you throw out non-matches the better.
- On a related point, showing something can't match is usually more work than finding a match. Try to do the latter.
- Cut down on backtracking -- that occurs when a matcher must go back through a string to find out if something match. Matches that just look for a pattern, without checking back can be much faster. Backrefences are the obvious case, but it isn't always obvious how to avoid backtracking altogether.
- Learn about consumption (see below).
- Anchors like ^ and $ and \b can help a lot because they reduce the number of possible places to look for a match.
- Order matters. As with `if/else/then` control flow, with alternatives specified by `|` you want the most common case first so that when you hit that case (often) you avoid the work of the other cases.

See here for more details.

Consumption

A regex match **consumes** the characters it matches. How it does this is important as well as the fact.

The fact of consumption makes some matching and replacement harder because you might want to match on a long pattern, but only replace a small amount. Capture groups help, but it ain't alway easy.

The type of consumption is also very important. The types are

- **Greedy:** [Julia's default] matches (consumes) as much as it can.

```
julia> m = match(r"^(.*)ab", "bcdabdcbabcd")
RegexMatch("bcdabdcbab", 1="bcdabdcb")
```

- **Reluctant:** by putting an extra ? after an quantifier (?,*,+) you make it reluctant. This matches as little as possible.

```
julia> m = match(r"^(.*?)ab", "bcdabdcbabcd")
RegexMatch("bcdab", 1="bcd")
```

- **Possessive:** by putting an extra + after a quantifier (?,*,+) you make it possessive. This matches as much as possible in a more efficient manner because it won't give back once it has found some match.

```
julia> m = match(r"^(.*+)ab", "bcdabdcbabcd")
nothing
```

  The last one is strange and to be honest I don't think I really understand it. It is meant to be greedy without backtracking, so it should be matching the same, but ???

https://www.computerworld.com/article/2786107/regular-expression-tutorial-part-5--greedy-and-non-greedy-quantification.html

https://stackoverflow.com/questions/5319840/greedy-vs-reluctant-vs-possessive-qualifiers

Operator precedence

Precedence determines the order in which operations are interpreted, similar to BODMATH (Brackets, Orders (square roots and powers), Division/Multiplication, Addition/Subtraction) in conventional arithmetic. Hence `2 + 4 * 5 = 2 + 20 = 22`.

The precedence of operators doesn't seem to be be described often. A very brief desciption is here. It states that "The operator precedence, from weakest to strongest binding, is first alternation, then concatenation, and finally the repetition operators. Explicit parentheses can be used to force different meanings, just as in arithmetic expressions. Some examples: `ab|cd` is equivalent to `(ab)|(cd)`; `ab*` is equivalent to `a(b*)`." But this doesn't note that escapes seem to have the highest precedence, or that the precedence of brackets (round, square and curly) is also high, but I don't know which is highest.

Some experimentation might be needed here.

Modifier flags

Most regular expression engines also allow some extras. The most common one I use is "case independence", i.e., instead of having to include 'a' and 'A' in my regex, I can tell it to ignore the case and include both.

Julia has 4 flags:

- i -- do case insenstive match;
- m -- treat strings as multiline;
- s -- treat strings as single line; and
- x -- allows whitespace in the regex definition, so that you can make this more human readable.

The most common one for me is 'i', so that you can do things like

```
julia> match(r"cruel"i, "Goodbye, Oh Cruel World!\n")
RegexMatch("Cruel")
```

Theory

The theory of languages is actually very mathsy. The classic example is the Chomsky Hierachy, which have regular languages or grammars at the base. Regexes started as regular langauges, but feature creep added components that broke that, so they are no longer.

https://sodocumentation.net/regex

String macros

Ever wonder what `r"stuff"` or `s"other stuff"` actually do? They're using a *string macro*, turning

```
julia> mymacro"string"
```

into

```
julia> @mymacro_str "string"
```

There are a whole bunch of such string macros:

- b"hello" converts into a byte array (a traditional string);
- doc"string" is used in constructing the docs for Julia from markdown;
- L"stuff" as part of `LaTeXSAtrings` creates a LaTeX string.

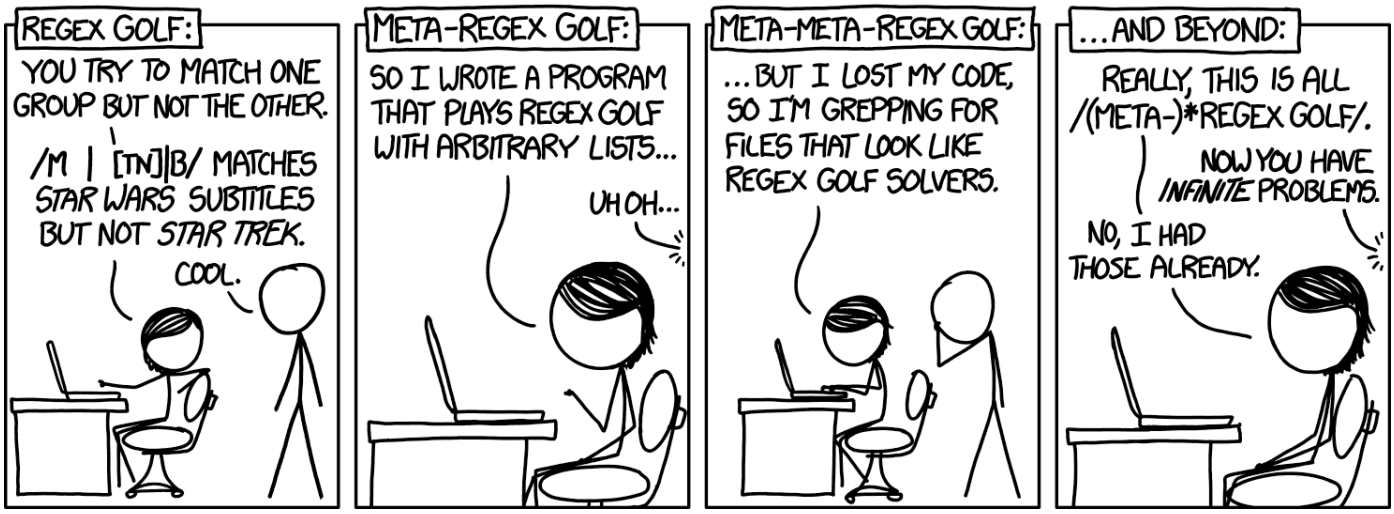
and more.

https://sodocumentation.net/julia-lang/topic/5817/string-macros

Summary

Regular expressions are sdcary at first but most of the basic things you might want to do with them aren't too hard and they are very powerful.

But in the end they are still scary ;)



https://xkcd.com/1313/

Links

https://en.wikipedia.org/wiki/Regular_expression

Perl docs and tutes

- https://perldoc.perl.org/perlre
- https://perldoc.perl.org/perlrequick
- https://www.tutorialspoint.com/perl/perl_regular_expressions.htm
- https://www.perltutorial.org/perl-regular-expression/
- https://perldoc.perl.org/perlretut

PCRE

- https://www.pcre.org/
- http://pcre.org/pcre.txt
- https://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions

Cheat Sheets (Perl focussed)

- https://perlmaven.com/regex-cheat-sheet
- https://www.debuggex.com/cheatsheet/regex/pcre

Regular expression implementation and programming language theory

- https://www.npopov.com/2012/06/15/The-true-power-of-regular-expressions.html
- https://www.perlmonks.org/?node_id=809842
- https://cs.stackexchange.com/questions/4839/which-languages-do-perl-compatible-regular-expressions-recognize
- https://swtch.com/~rsc/regexp/regexp1.html
- https://swtch.com/~rsc/regexp/

Julia Regexes

- https://riptutorial.com/julia-lang/example/20707/regex-literals
- https://sodocumentation.net/julia-lang/topic/5890/regexes
- https://docs.julialang.org/en/v1/manual/strings/#Regular-Expressions
- https://www.geeksforgeeks.org/regular-expressions-in-julia/

Cross-language comparison

- https://cs.lmu.edu/~ray/notes/regex/
- https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285