Regular expressions

DECEMBER 30, 2021 ~ BOB

This is the first article in a short series on some classic bits of computer science, which are occasionally useful in professional programming:

- Regular expressions
- Finite state machines
- Comparing regular expressions and finite state machines

A useful tool with a bad reputation

Regular expressions are a way to define a set of 0 or more text strings that you think are OK. This is usually because you want to see if some input text matches this set in some way. I don't use them every day, but they are often just the tool for the job if I'm processing text. They are widely supported, but unfortunately you may come across different dialects and/or levels of support. The core principles are usually the same, so I hope that this article is useful enough for you.

Most programming languages have commands that process regular expressions, and they are often taken as inputs by tools that work with text. For instance, text editors such as Notepad++ let you search for text that matches a regular expression. I first came across them in the UNIX tools grep and awk, which let me easily build sophisticated text processing pipelines from simple components.

Regular expressions are concise and flexible. It's this conciseness that is part of their bad reputation – they are often hard to understand because they're so densely packed with information. There are no comments, there's no white space to break things up helpfully, and often no breaking logic up into smaller chunks with descriptive names.

Another part of the bad reputation is the inscrutability that is unfortunately too common in computing. If it goes wrong, it's often hard to know why. I hope I can give you enough of an introduction that they become less opaque, and I will go into some tips at the end for working with them. I won't go into every part of regular expressions – please see e.g. https://www.regular-expressions.info for a more complete explanation.

Also, there seems to be no agreement on how to abbreviate it. Is it *regex* or *regexp*? If it's the first one, how do you pronounce it?

Watch now

@nimay.ndolo

Yeah, I hate how it sounds, but I'm a, "ReGGex," person. #softwared ...See more

POPSTAR (Feat. Drake) - DJ Khaled

Simple stuff

I'll start with the simplest building blocks. To make it obvious what's text and what's regular expression, I will write text like this

"this is some text"

and regular expressions like this

```
/this is a regular expression/
```

Using / to delimit regular expressions is a common but not universal convention.

The simplest non-empty regular expression would be to match a single letter, e.g. "a". This is just `/a/`. Note that by default, regular expressions are case-sensitive, so `/a/` will match "a" but won't match "A". If you want the whole regular expression to be case-insensitive then add i at the very end, e.g. `/a/i`. You can make just some bits case-insensitive if you like, which is a little bit fiddlier.

It's important to realise that `/a/` will also match "apple", "Bananarama" etc. A regular expression will match anywhere that it can. If you want to constrain it in some way, e.g. it must match from the beginning of the string, or at the end, or the regular expression must cover the whole string, then you need to add this constraint yourself – see **Anchors** below.

As you might expect, if you have two bits of text and you want them to be one after the other, e.g. "ab" then the regular expression is `/ab/`. This might seem obvious and trivial, but it's a general principle in regular expressions – the order inside the regular expression matters as things are matched left to right.

If you want one thing **or** another but not both, then you separate the options with |. E.g. to match "apple" or "banana" you could use `/apple|banana/`. You can use () to set the scope of operators such as `|`. So `/p(a|e)n/` will match "pan" or "pen" – you could also use `/pan|pen/` but the first option has factored out the parts that are common to all alternatives.

With just these three things

1. Matching specific characters
2. Building up a sequence of elements into a bigger regular expression
3. Listing alternatives

you can go a long way. However, there are other operations that let you take things even further.

Repeating

If you want to repeat something a fixed number of times you can already do this by simply repeating it as needed e.g. `/pigs pigs pigs pigs pigs pigs pigs/` will match any string that contains "pigs pigs pigs pigs pigs pigs pigs". However, you can abbreviate this to `/(pigs ){6}pigs/`. The `{6}` says that the previous element must be repeated exactly 6 times, and the previous element in this case is everything inside the `()`, which is the word *pigs* followed by a space. Then there should be one instance of *pigs* where we aren't asking for a following space.

This introduces another concept in regular expressions – whitespace is a first class citizen. If you want a space, you have to ask for one. If you want two tabs, you have to ask for two tabs (and not one or three).

You can have up to two numbers inside the `{}`, e.g. `{2,5}`. The first number is the minimum number of repetitions, and the second number is the maximum number. If they're the same, i.e. you're asking for exactly one number of repetitions rather than a range, then you can simplify things to just e.g. `{6}`. If you keep the comma but leave out one of the numbers, that missing number means *any sensible number*. E.g. `{,3}` means *any number up to and including 3*, and `{3,}` means *3 or more*.

There are some cases that happen often enough that they're given special abbreviations:

- `? = {0,1}`
- `* = {0,}`
- `+ = {1,}`

`/colou?r/` will match either the British English or American English spellings of *colour* i.e. "colour" or "color". This is because the `?` makes the previous element optional (i.e. it must occur 0 to 1 time), and the previous element is the letter u.

`/>#*</` will match "><", ">#<", ">##<", ">###<" etc.

`/ba+ng!/` will match "bang!", "baang!", "baaang!" and so on.

Sets of characters

You often want to match any character from a set of characters. For the most used sets there are pre-defined short-cuts, and you can define your own set if these aren't what you would like.

The broadest set is *all characters.* This is represented by a single dot, so `/.{4}/` will match exactly 4 characters, but they can be any characters. I.e. `/.{4}/` will match "   " (4 spaces), "abcd", "ABCD", "1&Jw" etc.

The other pre-defined sets vary from one dialect to another. I will give some examples from one dialect – I suggest that you consult the documentation for the dialect you're working with.

- `\w` = alphanumeric characters plus _ (as you might get in a word in computer text e.g. source code)
- `\W` = anything that isn't an alphanumeric character, e.g. white space, punctuation that isn't _ etc.
- `\d` = digits
- `\D` = anything that isn't a digit
- `\s` = whitespace
- `\S` = anything that isn't whitespace

`/My name is \w+/` will match "My name is " followed by 1 or more letters, numbers and/or _ characters.

`/\S+\s+\S+/` will match a string of one or more non-whitespace characters, then one or more whitespace characters, then one or more non-whitespace characters.

If the pre-defined sets aren't what you want, then you can define your own using `[]`. There are two main options – you can either define the characters in the set, or the characters that aren't in the set (implying that all other characters are in the set). You can also abbreviate obvious contiguous ranges of characters.

- `[aeiou]` means any vowel
- `[^aeiou]` means anything that isn't a vowel
- `[a-g]` means any of the first 7 lower-case letters of the alphabet
- `[^a-g]` means anything that isn't one of the first 7 lower-case letters of the alphabet, e.g. an upper-case letter, a lower case letter later in the alphabet than "g", a number, or punctuation or whitespace
- `[0-9a-fA-F]` means any digit, or any of the first 6 letters of the alphabet in either case i.e. any hexadecimal digit
- `[^0-9a-fA-F]` means anything that isn't a hexadecimal digit e.g. "J" or "@".

`/p[aei]+n/` will match "pan", "pen", "pin", "pain", "piaieeeen" etc.

Anchoring

In the first section I said that you need to be explicit if you want to constrain the regular expression to match from the beginning / at the end / over the whole of the text. You do this using the anchors `^` and `$`.

- `^` matches the beginning of the text – not the first character, but the magic invisible 0-width character that's before the first proper character.
- `$` matches the end of the text – again, not the last character, but the magic character that's after the last proper character

`/^a/` will match any string that starts with "a", e.g. "a", "apple", "aardvark" or "a bit of text" but it won't match e.g. "Bananarama" as none of its "a"s are the first proper character.

`/a$/` will match any string that ends with "a", e.g. "a", "arena", "Bananarama" or "I like ska" but not "apple".

`/^a$/` will match only the string "a"

`/^a.+a$/` will match any string that starts with "a", ends with "a", with one or more characters in between. E.g. "aba", "aaaaaaaaa" etc, but not "aa" as there's no character in the middle.

`/^a[^a]+a$/` will match any string that starts with "a", ends with "a", with one or more characters in between none of which is "a". So it will match "a a", "abcdea" but not "aa", "aaa" or "abaaaba"

Some dialects let you match against a word boundary, which is defined as the magic invisible 0-width character between a normal character that matches `\w` and one that matches `\W` (or vice versa). Where they are supported, they are `\b`.

Remembering and extracting

As well as grouping characters together so that e.g. + will act on all of them, `()` has an extra purpose. It lets you say that the bit of text that matches what's inside is important. The jargon name for this is a *capture group*.

If you are working with code, the regular expression library you're using will usually give you access to the contents of the capture groups – the first group that occurs in the regular expression is labelled 1, the next is 2 etc.

You can also refer to capture groups within a regular expression. This is useful if you want to match e.g. two copies of the same thing in one bit of text (but you're not sure exactly what form this thing will take, so you can't do a hard-coded match against characters). `\1` means *whatever was in capture group 1*, and you can use `\2`, `\3` etc. for later capture groups.

`/(\w+) daughter of \1/` will match "Mary daughter of Mary" but not "Hooda daughter of Uma". If you wanted to match "Hooda daughter of Uma" then you could use e.g. `/\w+ daughter of \w+/`.

If you want to use `()` only for grouping and not as a capture group, you can add `?:` to the beginning of the contents of the group. E.g. `/(?:apple|banana) (\w+) at \d+ kg per \1/` will match "apple cake at 3 kg per cake" or "banana pie at 12 kg per pie" etc.

Matching characters with special meanings

You might be wondering how to match things like "?" or "." which have special meanings inside a regular expression. The answer is to prefix the character with `\`. As \ is therefore also a character with a special meaning, you have to prefix it with itself to match a single "\".

Therefore `/^How do you like them apples\?$/` will match "How do you like them apples?". If you left out the \ in the regular expression, the `?` would act as an operator and make the previous thing (the s) optional. I.e. `/^How do you like them apples?$/` would match both "How do you like them apples" and "How do you like them apple" but not "How do you like them apples?", because the "s" must be the last character in the string (if it's present).

Working with regular expressions

I recommend a tool such as https://regex101.com/ to do ad hoc work with a regular expression. It will let you test your regular expression against text, tell you about errors in your regular expression (e.g. a missing `]`), and will describe the different elements of the regular expression.

Whether you have access to a tool like this or not, there is a basic approach that I've found helpful. Instead of trying to work out all of a regular expression, chop out a part of it (that's well-formed, e.g. there's a `)` for each `(`), and test that against some text. Work through all the elements of the regular expression like this until you're happy with them, and then start combining them. If you've worked with SQL before, for instance select queries with complex where clauses, then you might have used this approach to understand or debug a select query. (SQL can be a similarly dense and complicated thing.)

If you use a regular expression in some code then you need to test it *thoroughly*. It might look like only one line of code, but it triggers potentially complex behaviour with many test cases. Think through all the different kinds of string that could match – it could be more than you think. What happens if there's no match? What if the match could occur at more than one point in the string?

Related

Comparing regular expressions and finite state machines
January 1, 2022
In "Computer theory"

Capital cities and US place names
May 10, 2020
In "Data processing"

Solving computer problems with indirection
November 6, 2022
In "Architecture"

‹ PREVIOUS
How permanent is your data?

NEXT ›
Finite state machines

Recent Posts

Multiplying using halving, doubling and summing – part 2 March 31, 2025

Palindromic ancient and modern February 7, 2025

Designing the user experience of Top Trumps February 1, 2025

Analysing the radio alphabet January 28, 2025

Arts and humanities in computing December 31, 2024