Chapter 1

Regular Expressions

`[A-Z][a-z]+\d{4}`

Chatbot+9000

Regular expressions is something that is programming language independent. Since I started teaching this course, we have changed what language we used when going over this topic. Thus, this chapter will start off language independent before going into the language dependent sections. While you can skip the sections of the languages we are not using, it may be helpful to see regular expressions are used in other languages.
Additionally regex comes from the theory related very closely to finite automata which is a later chapter so for now we will do just the basics and a surface level of the topic here.

1.1 Introduction

For those that do not know, programs live in RAM on the machine. Since RAM is wiped everytime you power down your machine[1], programs and program memory is designed to live short term. For long term storage, we need to use hard drives, since what is written to a hard drive is saved for a much longer period of time. When we write to the hard drive we typically do this by writing a file and saving it.

One defining feature of the UNIX family is the idea that everything is treated as a file, and for the most part, this works fine since everything that is stored, is stored as a file. We have various file types and file descriptors but ultimately whenever we need to save data for long-term storage, we save it to a file (which ultimately is just a segment of bytes in memory). This also means that whenever we need to load something from storage, we need to open up a file and read from it.

Opening a file and getting the data from it is the easy part. The hard part is to try and parse and make use of the data. Typically from a coding perspective, you open a file, read from it and then have a string of data. Being able to properly and efficiently parse this string is what loading typically is. Examples of this is reading a save file for a videogame, loading an image from a `.`png file, loading settings from a config file. Regardless reading and parsing strings is important, but can be hard to do.

Many would think that if you are trying to read certain data from a string, that methods that deal with strings is the solution. If I loaded a configuration file and wanted to know if I had to set a value to **`'true'`** then I would probably want to check if the string has a subtring of **`'true'`**. This is certainly one way to solve this problem, but it soon gets very inefficient with large amounts of data. The solution to some of our problems here is Regular Expressions, or more commonly known as RegEx.

1.2 Regular Expression as a Tool

At a basic level, a **Regular Expression** is a pattern that describes a set of strings. At a deeper level, a regular expression defines a regular language, a language which can be created from a finite state machine. More on regular languages in a bit.

[1]for the most part. There is always the Cold Boot Attack

A finite state machine will be covered in a later chapter as well. For now however, we can think of regex as a tool (or library) used to search (and extract!) text.

When we wish to define a pattern to describe a set of strings, there are a few things that we must consider.

- An Alphabet - An **Alphabet** is the set of symbols or characters allowed in the string. If our set of strings are for English words, we would have a different alphabet than one that describes a set of mandarin strings.
- Concatenation - Since most strings are longer than a single character, we will need some way to demonstrate the concatenation of single characters to create longer strings.
- Alternation - Being able to say one thing or another. We could say that any non-empty set is a union of 2 other sets. So I want to say that a string in set S is in either S_1 or S_2 where $\{S_1, S_2\}$ is a partition of S.
- Quantification - The thing about patterns is that repeat. So if I want to have a pattern, then I need to allow for repetition.
- Precedence - Ultimately not something that is the basis of regex, but is helpful making sure we know what exactly is being described.

We will talk more about all of this in a future chapter.

1.3 Regular Expression Creation

Now that we have an idea of what regular expressions are, let us see how we can create a regular expression. To reiterate, a regular expression describes a set of strings. This means we need some way to describe a set. In mathematics, there are universally accepted ways to describe a set. For example: $\{x \in \mathbb{N} | \exists y \in \mathbb{Z} \wedge y = 2x\}$. This is the very common **set builder** notation. The mathematical community has agreed that this is valid syntax to describe a set. Analogously, the CS community has agreed there should be a special syntax to describe a set of strings called regular expressions.

In my experience, most languages use the POSIX-EXT standard for the syntax of a regular expression. However different languages may have different support.

To begin, let's make our very first regular expression. I will surround a regular expression with the '' (backtick) characters. This is just my notation, so refer to the future sections to see how to write a regular expression in specific languages.

```
‘a‘
```

That's it. That's our very first pattern. What does it mean?
This is a very boring regular expression, it just describes a set with a single element of the "a" string: $\{"a"\}$.

What about something longer? Maybe my name?

```
‘cliff‘
```

This is again, quite boring. It's just the small set of the string "cliff": $\{"cliff"\}$. What if we wanted to describe a set of greetings?

```
‘hello|hi|good morning|sup|salutations‘
```

Now we are getting somewhere. We now have something that describes the set: {"hello", "hi", "good morning", "sup", "salutations"}. Additionally, we have just seen our first regex operator: the `or` operation, denoted with a pipebar (|)[2]. We can use this to describe the set of digits.

```
‘0|1|2|3|4|5|6|7|8|9‘
```

Now for the first challenge: describe a set of strings that look like "I am x years old", where x is 0 through 9, inclusive.

To approach this problem, we could brute force it:

[2]One thing to note about this and all other operators: if you wanted to describe a set that includes the pipebar ({"|"}), then you would have to escape this operator: `‘(true|false)\|\|(true|false)‘`

```
'I am 0 years old|I am 1 years old|I am 2 years old|...'
```

While this does work, notice there is a lot of repetition and this can get annoying should we want more than just 10 ages. This would be equivalent of describing a set of the first 100 naturals by just listing them out: $\{0, 1, 2, 3, 4, 5, \ldots 98, 99\}$. This is cringe and so instead we rather use set builder notation: $\{x \in \mathbb{N} | x < 100\}$. In regular expressions case, we also have a shortcut to help us out here.

```
'I am (0|1|2|3|4|5|6|7|8|9) years old'
```

Notice that in our previous patterns when we used the `or` operator, the scope of the operator extended until the start or end of the pattern, or until another `or` operator. So something like `'there's one|two|three dogs'` describes the set {"there's one", "two", "three dogs"}.

To get around this issue, we can put parenthesis to change the scope of the operator. So we could describe the set {"hi cliff", "bye cliff", "hi clyff", "bye clyff"} using the following regular expression:

```
'(hi|bye) cl(i|y)ff'
```

Now while this shortcut is helpful, it's still a little annoying. What if I wanted to include more numbers? I don't really want to do `'0|1|2|3|4|5|...|99'` because this is just the previous problem of repetition I just mentioned. Thus, let me introduce the next shortcut: quantification.

If I want to repeat a string multiple times, we can do this by adding a quantifier to a pattern (or sub-pattern). There is only 1 true quantifier, called the `Kleene operator`: *. This operator will repeat said pattern 0 or more times. Let's see what that means

```
'(ha)*'
```

This pattern describes the set: {"", "ha", "haha", "hahaha", "hahahaha", ...}. It is important to note that zero or more times includes the empty string. So if I wanted to describe any number, I can use the following pattern:

```
'(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*'
```

This regular expression describes a digit followed by 0 or more digits. This itself has a shortcut because this can also get long:

```
'(0|1|2|3|4|5|6|7|8|9)+'
```

The + operator is used to denote 1 or more repetitions. Other quantifiers include the following:

```
'(0|1|2|3|4|5|6|7|8|9)?' - one or zero times
'(0|1|2|3|4|5|6|7|8|9){x}' - x times
'(0|1|2|3|4|5|6|7|8|9){x,}' - at least x times
'(0|1|2|3|4|5|6|7|8|9){,y}' - at most y times
'(0|1|2|3|4|5|6|7|8|9){x,y}' - between x and y times (inclusive)
```

Now these shortcuts are helpful, and cut down the length of the pattern, but imagine if we wanted to describe any lowercase string of size 3.

```
'(a|b|c|d|...|x|y|z){3}'
```

This would be a very long pattern leading to the last few shortcuts: bracket expressions. Bracket expressions allow us to describe ASCII ranges,or characters together,or even denote anything but a series of characters.

```
'[a-z]{3}' - any ASCII character between a-z (inclusive) repeated thrice
'[0-9]+' - any ASCII character between 0-9 (inclusive) repeated at least once
'[aeiou]' - any vowel. This is equivalent to '(a|e|i|o|u)'
'[a-zA-Z]' - any upper or lowercase letter. Equivalent to '[a-z]|[A-Z]'
'[^abc]' - anything EXCEPT an "a","b", or "c"'
```

Now brackets expressions can help us describe large ranges of characters, but what if we wanted to describe any character? Maybe something like any string of size 3? This can be solved with the dot character ("."). The dot character to represent any character.

```
'.{3}' - any sting of length 3
'.at' - any string of length 3 that ends with 'at'.
```

It is important to note that the dot, the or, and the brackets only represent a single character. It is only when we add a quantifier, do we then repeat. To also be very clear, these all describe strings. '[0-9]' describes a string that represents a one digit number, but the set described consists of only strings.

1.4 Regular Expression Examples

Consider the following regular expressions and think about what types of strings they describe.

- '-?[0-9]+' - strings of integers
- '-?[0-9]+$\dot{[}$0-9]+' - strings of floats with or without leading and trailing zeros
- '[A-Z][a-z]*' - strings that begin with a capital followed by zero or more lowercase characters
- '[AEIOUaeiou]{5,10}' - strings consisting of only vowels of length 5 - 10
- 'I am [1-9]?[0-9] years old' - I am x years old where $0 \leq x \leq 99$
- 'I am ([2-4][0-9]|50) years old' - I am x years old where $20 \leq x \leq 50$

1.5 Regular Expression Matching

Now that we know how to describe our set, we need some way to check if a string (or part of a string) is in the set. This is where we get into the usage of regular expressions for practical use. So we will now start talking about how languages use regular expressions to solve problems.

If a string, *s*, is in the set described by the regular expression *r*, then we say that *r* **accepts** s. If *s* is not in the set, then we say that *r* rejects *s*. Checking if a regular expression accepts or rejects a string can be done in O(n) time where *n* is the length of the string. We will talk more about how to implement this later, but the great thing is because it's linear, we can actually check if any part of a string is in the set described by the regular expression.

Since we can check if any part of a string is in the set, we may need to denote we want an **exact match** over a **partial match**. An exact match is one where the entire string matches the regular expression. Most languages I have worked with perform a partial match, although I am aware that other languages default to an exact match.

To demonstrate this, let's assume that there is some function called `match(str,re)` that has two parameters, a string `str` and a regular expression `re`. It returns `true` if `re` accepts `str` and `false` otherwise.

```
If using an exact match:
match("unknowing",'know') returns false since "unknowing" is not in {"know"}
If using a partial match:
match("unknowing",'know') returns true since "know" can be found in "knowing"
```

If the language's default is a partial match, we have two other symbols to help us achieve something similar to an exact match.

```
If performing a partial match
'^ch' - any string that begins with 'ch'
'at$' - any string that ends with 'at'
```

The carot symbol (ˆ) is something we already saw as something akin to a **not** operator in a bracket expression. When used outside of a bracket environment, it is used to denote the start of the string.

```
If performing a partial match:
'^(this|that)' - any string that begins with 'this or that'
'^this|that' - any string that begins with 'this' or contains 'that'
```

The dollar sign ($) is the symbol for denoting the end of the string.

```
If performing a partial match:
'(this|that)$' - any string that ends with 'this or that'
'this|that$' - any string that ends with 'that' or contains 'this'
```

Putting these two together allow us to obtain exact match functionality.

```
Regardless of performing a partial or exact match:
'^(this|that)$' - either the string "this" or the string "that"
```

Now this process is very helpful for checking if a string matches a certain pattern, but this is only so useful. Sometimes it is useful to actually extract parts of a string based on a pattern.

1.6 Regular expression Grouping

While we can check for acceptance of a string, regex is more useful to actually extract data from a string. For example, you have a file and it has a 1000 lines of phone numbers (eg. `"John Smith:123-4567890"`). Your job is to sum up all the area codes.

To do this, we need some way to mark certain parts of the strings to be stored somewhere. Most implementations of regular expressions just use the parenthesis to do this. So if I wanted to mark the area code, I could do the following:

```
if doing a partial match:
'([0-9]{3})-[0-9]{7}' - put () around the area code part: ([0-9]{3})
If doing an exact match:
'[A-Z][a-z]+ [A-Z][a-z]+:([0-9]{3})-[0-9]{7} - this makes assumptions about what a valid name is
```

Now that we marked what we wanted to extract, we need some way of obtaining the extracted parts. This is language dependent, but the convention of what order the groups are is not. The previous example is simple in the sense there is only 1 set of parenthesis. What happens if we use a more complicated pattern?

```
'Cl(i|y)ff's phone number is (([0-9]{3})-([0-9]{7}))'
```

Here we have marked four (4) parts (groups). The first group if apart of the `'Cl(i|y)ff'` segment. This was more to allow an more alternate spelling of my name, but regex will still extract either the 'i' or 'y' and store it in group 1.

Continuing from that, we read the regex left to right, and whenever we see a left (open) parenthesis, that is the next group. Thus, group 2 is the entire phone number, group 3 is the area code, and group 4 is the rest of the phone number.

1.7 Regular Expressions in Programming Languages

1.7.1 Regular Expression In Python

Let's start out on making our first pattern. To begin, we need to include the regular expression module which we can do with the `import re` command. Once we have done that, we can start using regular expression methods.

```python
# regex.py
m = re.compile("I am \d+ years old")
matched = m.match("I am 23 years old")
if matched:
  print("Successfully matched")
else:
  print("Did not match")
```

The return value of `m.match()` is a `matched_data` type which has information regarding what you are trying to extract.

```python
# regex.py
m = re.compile("I am (\d+) years old")
matched = m.match("I am 23 years old")
```

```
if matched:
  print("I am " + m.group(1)))
else:
  print("Did not match")
```

Calling `match()` on a compiled regular expression in Python will check if the string matches from the beginning of the input string. Thus, `match()` performs a semi-partial match. Other methods exist that do a partial match anywhere in the string, or an exact match (`search()`,`fullmatch()`).

You can actually test and see what is accepted and captured using this fun online tool called `https://pythex.org/`. That or you can play around with the code segments found in this chapter and see what happens.

1.7.2 Regular Expression In Ruby

Let's start out on making our first pattern. Much like `Strings` denoted with single or double quotes, and arrays with the `[]` symbols, and hashes using `{}`, patterns are surrounded by the forward-slash: `/`.

```
# regex.rb
p = /pattern/
puts p.class #Regexp
```

This pattern is the string literal **'pattern'**. That is, this pattern describes the set of strings that contain the substring **'pattern'**. Not a very fun pattern, but a pattern nonetheless.

But Great! We have a pattern. Now how do we use it? There are 2 main ways that we can match this pattern that I know of, one of which I like and the other I do not. We will go over the latter because it is easier version and my reason for not liking it it petty.

```
# regex-1.rb
p = /pattern/
if p =~ "pattern" then
  puts "Matched"
else
  puts "Not matched"
end
```

If everything goes as planned, running this file will have `"Matched"` printed. Why does this happen? = is a method associated with Regexp objects which take in a string and returns an integer or nil depending on if the pattern can be found *anywhere* in the string. Thus Ruby performs partial matching be default. If the pattern is found, it will return the index of the first character of the first instance of the pattern.

Since using the = method will search the entire string for a match, we will have to use your start and end of string specifiers.

```
# regular_expressions.rb
if /^where/ =~ "anywhere in the string" then
  puts "matched at the begining"
end
if /where$/ =~ "anywhere in the string" then
  puts "matched at the end"
end
if /where/ =~ "anywhere in the string" then
  puts "matched somewhere in the string"
end
```

Here the only thing printed is `Matched somewhere in the string`.

Now that you can check to see if a string matches a pattern, we can move onto grouping. Ruby uses the global variables `$1, $2, ...,$x` to refer to the first, second, ..., xth capture group.

```
# capture.rb
pattern = /([0-9]{3})-[0-9]{7}/
```

```
if pattern =~ "111-1111111" then
    puts $1 #111
end

pattern = /(([0-9])[0-9]{2}))-[0-9]{7}/
if pattern =~ "123-4567890" then
    puts $1 #123
    puts $2 #1
end
```

One other important thing to note is that whenever the = method is called, then these top level variables $1, $2, etc, are all reset. This is the main reson as to why I do not like this method of matching patterns despite how easy it is.

The way that I prefer to match patterns is by using the `match` method. This instead returns an array of all matched groups (if any) which means I can store the results to a variable and refer to them later and not worry about data being wiped. That's it, the only reason why I do not like the previously described method.

You can actually test and see what is accepted and captured using this fun online tool called `http://rubular.com`. That or you can play around with the following code segments and see what happens.

```
# capture-2.rb
pattern = /[A-Z][A-Z]*/
strs = ["a","A","abcD","ABDC"]
for test_string in strs
  if pattern =~ test_string
    puts "matched"
  else
    puts "not matched"
  end
end

#what if the pattern and test strings are as follows:
pattern = /a[A-Za-z]?/
strs = ["a","abd","bad"]

pattern = /^a[A-Za-z]?$/
strs = ["a","abd","bad"]

pattern = /a*b*c*d*/ # how is this different from /[a-d]/ ?
strs = ["abcd", "bad", "cad", "aaaaaacd", "bbbdddd"]

pattern = /^(..)$/
strs = ["even","odd","four","three","five"]

# an even number of vowels
pattern = /^([^aeiou]*[aeiou][^aeiou]*[aeiou][^aeiou])*$/
```

1.7.3 Regular Expression In OCaml

//TODO