**INTRODUCTION TO REGULAR EXPRESSIONS**

**Literals**

The simplest text we can match with regular expressions are ***literals***. This is where our regular expression contains the exact text that we want to match. The regex a, for example, will match the text a, and the regex bananas will match the text bananas.

We can additionally match just part of a piece of text. Perhaps we are searching a document to see if the word monkey occurs, since we love monkeys. We could use the regex monkey to match monkey in the piece of text The monkeys like to eat bananas..

Not only are we able to match alphabetical characters — digits work as well! The regex 3 will match the 3 in the piece of text 34, and the regex 5 gibbons will completely match the text 5 gibbons!

Regular expressions operate by moving character by character, from left to right, through a piece of text. When the regular expression finds a character that matches the first piece of the expression, it looks to find a continuous sequence of matching characters.

**Alternation**

Do you love baboons and gorillas? You can find either of them with the same regular expression using ***alternation!*** Alternation, performed in regular expressions with the pipe symbol, |, allows us to match either the characters preceding the | OR the characters after the |. The regex baboons|gorillas will match baboons in the text I love baboons, but will also match gorillas in the text I love gorillas.

Are you thinking about how to match the whole piece of text I love baboons or I love gorillas? We will get to that later on!

cat|dog matches

✅cat

✅dog

Doesn’t match

bat

rat

**Character Sets**

Spelling tests may seem like a distant memory from grade school, but we ultimately take them every day while typing. It’s easy to make mistakes on commonly misspelled words like consensus, and on top of that, there are sometimes alternate spellings for the same word.

***Character sets***, denoted by a pair of brackets [], let us match one character from a series of characters, allowing for matches with incorrect or different spellings.

The regex con[sc]en[sc]us will match consensus, the correct spelling of the word, but also match the following three incorrect spellings: concensus, consencus, and concencus. The letters inside the first brackets, s and c, are the different possibilities for the character that comes after con and before en. Similarly for the second brackets, s and c are the different character possibilities to come after en and before us.

Thus the regex [cat] will match the characters c, a, *or* t, but not the text cat.

The beauty of character sets (and alternation) is that they allow our regular expressions to become more flexible and less rigid than by just matching with literals!

We can make our character sets even more powerful with the help of the caret ^ symbol. Placed at the front of a character set, the ^ negates the set, matching any character that is *not* stated. These are called *negated character sets*. Thus the regex [^cat] will match any character that is not c, a, *or* t, and would completely match each character d, o *or* g.

Do we have a consensus that regular expressions are pretty cool?

[chr]at matches

✅cat

✅hat

✅rat

Doesn’t match

🚫eat

🚫mat

🚫sat

**Wild for Wildcards**

Sometimes we don’t care exactly WHAT characters are in a text, just that there are SOME characters. Enter the wildcard .! ***Wildcards*** will match any single character (letter, number, symbol or whitespace) in a piece of text. They are useful when we do not care about the specific value of a character, but only that a character exists!

Let’s say we want to match any 9-character piece of text. The regex ......... will completely match orangutan and marsupial! Similarly, the regex I ate . bananas will completely match both I ate 3 bananas and I ate 8 bananas!

What happens if we want to match an actual period, .? We can use the escape character, \, to escape the wildcard functionality of the . and match an actual period. The regex Howler monkeys are really lazy\. will completely match the text Howler monkeys are really lazy..

....\. matches

✅bear.

✅lion.

✅orca.

Doesn’t match

🚫mouse

🚫koala

🚫snail

**Ranges**

Character sets are great, but their true power isn’t realized without ranges. ***Ranges*** allow us to specify a range of characters in which we can make a match without having to type out each individual character. The regex [abc], which would match any character a, b, *or* c, is equivalent to regex range [a-c]. The - character allows us to specify that we are interested in matching a range of characters.

The regex I adopted [2-9] [b-h]ats will match the text I adopted 4 bats as well as I adopted 8 cats and even I adopted 5 hats.

With ranges we can match any single capital letter with the regex [A-Z], lowercase letter with the regex [a-z], any digit with the regex [0-9]. We can even have multiple ranges in the same character set! To match any single capital *or* lowercase alphabetical character, we can use the regex [A-Za-z].

Remember, within any character set [] we only match *one* character.

[c-e][l-u][[b-k] matches

✅cub

✅dog

✅elk

Doesn’t match

🚫ape

🚫cow

🚫ewe

**Shorthand Character Classes**

While character ranges are extremely useful, they can be cumbersome to write out every single time you want to match common ranges such as those that designate alphabetical characters or digits. To alleviate this pain, there are ***shorthand character classes*** that represent common ranges, and they make writing regular expressions much simpler. These shorthand classes include:

* \w: the “word character” class represents the regex range [A-Za-z0-9\_], and it matches a single uppercase character, lowercase character, digit or underscore
* \d: the “digit character” class represents the regex range [0-9], and it matches a single digit character
* \s: the “whitespace character” class represents the regex range [ \t\r\n\f\v], matching a single space, tab, carriage return, line break, form feed, or vertical tab

For example, the regex \d\s\w\w\w\w\w\w\w matches a digit character, followed by a whitespace character, followed by 7 word characters. Thus the regex completely matches the text 3 monkeys.

In addition to the shorthand character classes \w, \d, and \s, we also have access to the *negated shorthand character classes*! These shorthands will match any character that is NOT in the regular shorthand classes. These negated shorthand classes include:

* \W: the “non-word character” class represents the regex range [^A-Za-z0-9\_], matching any character that is not included in the range represented by \w
* \D: the “non-digit character” class represents the regex range [^0-9], matching any character that is not included in the range represented by \d
* \S: the “non-whitespace character” class represents the regex range [^ \t\r\n\f\v], matching any character that is not included in the range represented by \s

\d\s\w\w\w\w\w\w matches

✅5 sloths

✅8 llamas

✅7 hyenas

doesn’t match

🚫one bird

🚫two owls

**Grouping**

Remember when we were in love with baboons and gorillas a few exercises ago? We were able to match either baboons or gorillas using the regex baboons|gorillas, taking advantage of the | symbol.

But what if we want to match the whole piece of text I love baboons and I love gorillas with the same regex? Your first guess might be to use the regex I love baboons|gorillas. This regex, while it would completely match the string I love baboons, would not match I love gorillas, and would instead match gorillas. This is because the | symbol matches the *entire* expression before or after itself.

Grouping to the rescue! ***Grouping***, denoted with the open parenthesis ( and the closing parenthesis ), lets us group parts of a regular expression together, and allows us to limit alternation to part of the regex.

The regex I love (baboons|gorillas) will match the text I love and *then* match either baboons or gorillas, as the grouping limits the reach of the | to the text within the parentheses.

These groups are also called *capture groups*, as they have the power to select, or capture, a substring from our matched text.

(puppies|kitty cats) are my favorite! Matches

✅puppies are my favorite!

✅kitty cats are my favorite!

Doesn’t match

🚫deer are my favorite!

🚫otters are my favorite!

🚫hedgehogs are my favorite!

**Quantifiers - Fixed**

Here’s where things start to get really interesting. So far we have only matched text on a character by character basis. But instead of writing the regex \w\w\w\w\w\w\s\w\w\w\w\w\w, which would match 6 word characters, followed by a whitespace character, and then followed by more 6 word characters, such as in the text rhesus monkey, is there a better way to denote the quantity of characters we want to match?

The answer is yes, with the help of quantifiers! ***Fixed quantifiers***, denoted with curly braces {}, let us indicate the exact quantity of a character we wish to match, or allow us to provide a quantity range to match on.

* \w{3} will match *exactly* 3 word characters
* \w{4,7} will match *at minimum* 4 word characters and *at maximum* 7 word characters

The regex roa{3}r will match the characters ro followed by 3 as, and then the character r, such as in the text roaaar. The regex roa{3,7}r will match the characters ro followed by *at least* 3 as and *at most* 7 as, followed by an r, matching the strings roaaar, roaaaaar and roaaaaaaar.

An important note is that quantifiers are considered to be *greedy*. This means that they will match the greatest quantity of characters they possibly can. For example, the regex mo{2,4} will match the text moooo in the string moooo, and not return a match of moo, or mooo. This is because the fixed quantifier wants to match the largest number of os as possible, which is 4 in the string moooo.

Both cases match

sque\w{3,5}k

squea{3,5}k

✅squeaaak

✅squeaaaak

✅squeaaaaak