RegexOne

Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 1: An Introduction, and the ABCs

Regular expressions are extremely useful in extracting information from text such as code, log files, spreadsheets, or even documents. And while there is a lot of theory behind formal languages, the following lessons and examples will explore the more practical uses of regular expressions so that you can use them as quickly as possible.

The first thing to recognize when using regular expressions is that **everything is essentially a character**, and we are writing patterns to match a specific sequence of characters (also known as a string). Most patterns use normal ASCII, which includes letters, digits, punctuation and other symbols on your keyboard like %#$@!, but unicode characters can also be used to match any type of international text.

Below are a couple lines of text, notice how the text changes to highlight the matching characters on each line as you type in the input field below. To continue to the next lesson, you will need to use the new syntax and concept introduced in each lesson to write a pattern that matches all the lines provided.

Go ahead and try writing a pattern that matches all three rows, **it may be as simple as the common letters on each line**.

Exercise 1: Matching Characters

Task	Text
Match	abcdefg
Match	abcde
Match	abc

abc

Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

Lesson 1½: The 123s

Characters include normal letters, but digits as well. In fact, numbers 0-9 are also just characters and if you look at an ASCII table, they are listed sequentially.

Over the various lessons, you will be introduced to a number of special metacharacters used in regular expressions that can be used to match a specific type of character. In this case, the character **\d** can be used in place of **any digit from 0 to 9**. The preceding slash distinguishes it from the simple **d** character and indicates that it is a metacharacter.

Below are a few more lines of text containing digits. Try writing a pattern that matches all the digits in the strings below, and notice how your pattern matches **anywhere within the string**, not just starting at the first character. We will learn how to control this in a later lesson.

Exercise 1½: Matching Digits

Task	Text	
Match	abc123xyz	✓
Match	define "123"	✓
Match	var g = 123;	✓

d	Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 3: Matching specific characters

The dot metacharacter from the last lesson is pretty powerful, but sometimes **too** powerful. If we are matching phone numbers for example, we don't want to validate the letters "(abc) def-ghij" as being a valid number!

There is a method for **matching specific characters** using regular expressions, by defining them inside **square brackets**. For example, the pattern **[abc]** will only match a **single** a, b, or c letter and nothing else.

Below are a couple lines, where we only want to match the first three strings, but not the last three strings. Notice how we can't avoid matching the last three strings if we use the dot, but have to specifically define what letters to match using the notation above.

Exercise 3: Matching Characters

Task	Text	
Match	can	✓
Match	man	✓
Match	fan	✓
Skip	dan	
Skip	ran	
Skip	pan	

[cmf]an

Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 4: Excluding specific characters

In some cases, we might know that there are specific characters that we don't want to match too, for example, we might only want to match phone numbers that are **not** from the area code 650.

To represent this, we use a similar expression that **excludes specific characters** using the **square brackets** and the ^ (**hat**). For example, the pattern **[^abc]** will match any **single** character **except for** the letters a, b, or c.

With the strings below, try writing a pattern that matches only the live animals (hog, dog, but not bog). Notice how most patterns of this type can also be written using the technique from the last lesson as they are really two sides of the same coin. By having both choices, you can decide which one is easier to write and understand when composing your own patterns.

Exercise 4: Excluding Characters

Task	Text	
Match	hog	✓
Match	dog	✓
Skip	bog	

[^b]og

Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

Next - Lesson 5: Character ranges
Previous - Lesson 3: Matching specific characters

Lesson 5: Character ranges

We just learned how to create a pattern that matches or excludes specific characters -- but what if we want to match a character that can be in a sequential range characters? Do we have no choice but to list them all out?

Luckily, when using the square bracket notation, there is a shorthand for matching a character in list of **sequential characters** by using the **dash** to indicate a character range. For example, the pattern **[0-6]** will only match any single digit character from zero to six, and nothing else. And likewise, **[^n-p]** will only match any single character **except** for letters n to p.

Multiple character ranges can also be used in the same set of brackets, along with individual characters. An example of this is the alphanumeric \w metacharacter which is equivalent to the character range **[A-Za-z0-9_]** and often used to match characters in English text.

In the exercise below, notice how all the match and skip lines have a pattern, and use the bracket notation to match or skip each character from each line. Be aware that patterns are **case sensitive** and **a-z differs** from **A-Z** in terms of the characters it matches (lower vs upper case).

Exercise 5: Matching Character Ranges

Task	Text	
Match	Ana	✓
Match	Bob	✓
Match	Cpc	✓
Skip	aax	
Skip	bby	
Skip	ccz	

[A-C][a-z][a-z]

Continue ›

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 6: Catching some zzz's

Note: Some parts of the repetition syntax below isn't supported in all regular expression implementations.

We've so far learned how to specify the range of characters we want to match, but how about the number of **repetitions** of characters that we want to match? One way that we can do this is to explicitly spell out exactly how many characters we want, eg. **\d\d\d** which would match exactly three digits.

A more convenient way is to specify how many repetitions of each character we want using the **curly braces** notation. For example, **a{3}** will match the a character exactly three times. Certain regular expression engines will even allow you to specify a range for this repetition such that **a{1,3}** will match the a character no more than 3 times, but no less than once for example.

This quantifier can be used with any character, or special metacharacters, for example **w{3}** (three w's), **[wxy]{5}** (five characters, each of which can be a w, x, or y) and **.{2,6}** (between two and six of **any** character).

In the lines below, the last string with only one z isn't what we would consider a proper spelling of the slang "wazzup?". Try writing a pattern that matches only the first two spellings by using the curly brace notation above.

Exercise 6: Matching Repeated Characters

Task	Text	
Match	wazzzzzup	✓
Match	wazzzup	✓
Skip	wazup	

waz{3,}up

Continue ›

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 7: Mr. Kleene, Mr. Kleene

A powerful concept in regular expressions is the ability to match an arbitrary number of characters. For example, imagine that you wrote a form that has a donation field that takes a numerical value in dollars. A wealthy user may drop by and want to donate $25,000, while a normal user may want to donate $25.

One way to express such a pattern would be to use what is known as the **Kleene Star** and the **Kleene Plus**, which essentially represents either **0 or more** or **1 or more** of the character that it follows (it always follows a character or group). For example, to match the donations above, we can use the pattern **\d*** to match any number of digits, but a tighter regular expression would be **\d+** which ensures that the input string has at least one digit.

These quantifiers can be used with any character or special metacharacters, for example **a+** (one or more a's), **[abc]+** (one or more of any a, b, or c character) and **.*** (zero or more of **any** character).

Below are a few simple strings that you can match using both the star and plus metacharacters.

Exercise 7: Matching Repeated Characters

Task	Text	
Match	aaaabcc	✓
Match	aabbbbc	✓
Match	aacc	✓
Skip	a	

a+a*b*c+ Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 8: Characters optional

As you saw in the previous lesson, the Kleene star and plus allow us to match repeated characters in a line.

Another quantifier that is really common when matching and extracting text is the **?** (question mark) metacharacter which denotes **optionality**. This metacharacter allows you to match either zero or one of the preceding character or group. For example, the pattern **ab?c** will match either the strings "abc" or "ac" because the b is considered optional.

Similar to the dot metacharacter, the question mark is a special character and you will have to escape it using a slash **\?** to match a plain question mark character in a string.

In the strings below, notice how the the plurality of the word "file" depends on the number of files found. Try writing a pattern that uses the optionality metacharacter to match only the lines where one or more files were found.

Exercise 8: Matching Optional Characters

Task	Text	
Match	1 file found?	✓
Match	2 files found?	✓
Match	24 files found?	✓
Skip	No files found.	

`files? found\?` Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 9: All this whitespace

When dealing with real-world input, such as log files and even user input, it's difficult not to encounter whitespace. We use it to format pieces of information to make it easier to read and scan visually, and a single space can put a wrench into the simplest regular expression.

The most common forms of whitespace you will use with regular expressions are the **space** (␣), the **tab** (\t), the **new line** (\n) and the carriage return (\r) (useful in Windows environments), and these special characters match each of their respective whitespaces. In addition, a **whitespace** special character \s will match **any** of the specific whitespaces above and is extremely useful when dealing with raw input text.

In the strings below, you'll find that the content of each line is indented by some whitespace from the index of the line (**the number is a part of the text to match**). Try writing a pattern that can match each line containing whitespace characters between the number and the content. Notice that the whitespace characters are just like any other character and the special metacharacters like the star and the plus can be used as well.

Exercise 9: Matching Whitespaces

Task	Text
Match	1. abc
Match	2. abc
Match	3. abc
Skip	4.abc

\.\s+abc

Continue ›

Solve the above task to continue on to the next problem, or read the Solution

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 10: Starting and ending

So far, we've been writing regular expressions that partially match pieces across all the text. Sometimes this isn't desirable, imagine for example we wanted to match the word "success" in a log file. We certainly don't want that pattern to match a line that says "Error: unsuccessful operation"! That is why it is often **best practice to write as specific regular expressions as possible** to ensure that we don't get false positives when matching against real world text.

One way to tighten our patterns is to define a pattern that describes both the **start and the end of the line** using the special **^** (**hat**) and **$** (**dollar sign**) metacharacters. In the example above, we can use the pattern **^success** to match **only** a line that begins with the word "success", but not the line "Error: unsuccessful operation". And if you combine both the hat and the dollar sign, you create a pattern that matches the whole line completely at the beginning and end.

Note that this is different than the hat used inside a set of bracket **[^...]** for excluding characters, which can be confusing when reading regular expressions.

Try to match each of the strings below using these new special characters.

Exercise 10: Matching Lines

Task	Text
Match	Mission: successful
Skip	Last Mission: unsuccessful
Skip	Next Mission: successful upon capture of target

^Mission: successful$

Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 11: Match groups

Regular expressions allow us to not just match text but also to **extract information for further processing**. This is done by defining **groups of characters** and capturing them using the special parentheses **(** and **)** metacharacters. Any subpattern inside a pair of parentheses will be **captured** as a group. In practice, this can be used to extract information like phone numbers or emails from all sorts of data.

Imagine for example that you had a command line tool to list all the image files you have in the cloud. You could then use a pattern such as **^(IMG\d+\.png)$** to capture and extract the full filename, but if you only wanted to capture the filename without the extension, you could use the pattern **^(IMG\d+)\.png$** which only captures the part before the period.

Go ahead and try to use this to write a regular expression that matches only the filenames (not including extension) of the PDF files below.

Exercise 11: Matching Groups

Task	Text	Capture Groups
Capture	file_record_transcript.pdf	file_record_transcript
Capture	file_07241999.pdf	file_07241999
Skip	testfile_fake.pdf.tmp	

^(.*)\.pdf$

Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

Next - Lesson 12: Nested groups
Previous - Lesson 10: Starting and ending

Lesson 12: Nested groups

When you are working with complex data, you can easily find yourself having to extract multiple layers of information, which can result in nested groups. Generally, the results of the captured groups are in the order in which they are defined (in order by open parenthesis).

Take the example from the previous lesson, of capturing the filenames of all the image files you have in a list. If each of these image files had a sequential picture number in the filename, you could extract both the filename and the picture number using the same pattern by writing an expression like **^(IMG(\d+))\.png$** (using a nested parenthesis to capture the digits).

The nested groups are read from left to right in the pattern, with the first capture group being the contents of the first parentheses group, etc.

For the following strings, write an expression that matches **and captures** both the full date, as well as the year of the date.

Exercise 12: Matching Nested Groups

Task	Text	Capture Groups
Capture	Jan 1987	Jan 1987 | 1987
Capture	May 1969	May 1969 | 1969
Capture	Aug 2011	Aug 2011 | 2011

`([A-Za-z]{3} (\d{4}))`

Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 13: More group work

As you saw in the previous lessons, all the quantifiers including the star *****, plus **+**, repetition **{m,n}** and the question mark **?** can all be used within the capture group patterns. This is the only way to apply quantifiers on sequences of characters instead of the individual characters themselves.

For example, if I knew that a phone number may or may not contain an area code, the right pattern would test for the existence of the whole group of digits **(\d{3})?** and not the individual characters themselves (which would be wrong).

Depending on the regular expression engine you are using, you can also use non-capturing groups which will allow you to match the group but not have it show up in the results.

Below are a couple different common display resolutions, try to capture the width and height of each display.

Exercise 13: Matching Nested Groups

Task	Text	Capture Groups
Capture	1280x720	1280 720
Capture	1920x1600	1920 1600
Capture	1024x768	1024 768

```
(\d+)x(\d+)
```

Continue ›

Solve the above task to continue on to the next problem, or read the Solution.

Next - Lesson 14: It's all conditional
Previous - Lesson 12: Nested groups

RegexOne
Learn Regular Expressions with simple, interactive exercises.

Interacti

Lesson 14: It's all conditional

As we mentioned before, it's always good to be precise, and that applies to coding, talking, and even regular expressions. For example, you wouldn't write a grocery list for someone to **Buy more .*** because you would have no idea what you could get back. Instead you would write **Buy more milk** or **Buy more bread**, and in regular expressions, we can actually define these conditionals explicitly.

Specifically when using groups, you can use the | (**logical OR, aka. the pipe**) to denote **different possible sets of characters**. In the above example, I can write the pattern "Buy more (milk|bread|juice)" to match only the strings Buy more milk, Buy more bread, or Buy more juice.

Like normal groups, you can use any sequence of characters or metacharacters in a condition, for example, ([cb]ats*|[dh]ogs?) would match either cats or bats, or, dogs or hogs. Writing patterns with many conditions can be hard to read, so you should consider making them separate patterns if they get too complex.

Go ahead and try writing a conditional pattern that matches only the lines with small fuzzy creatures below.

Exercise 14: Matching Conditional Text

Task	Text	
Match	I love cats	✓
Match	I love dogs	✓
Skip	I love logs	
Skip	I love cogs	

I love (cats|dogs)

Continue ›