Introduction to Regular Expressions

Learning Goals

Be able to answer the following questions:
- What is a regular expression? - How can I write a regular expression to match on a specific pattern? - How can I use regular expressions to speed up my development workflow?

Introduction

Regular expressions are a great tool for working with text. Using regular expressions (AKA Regex) you can **identify and process patterns** of text. Many people find regular expressions difficult to understand and use, but they can make a variety of tasks much easier, like validating that a phone number or zip code is in the right format.

You will likely only use a limited number of expressions during your time at Ada, but understanding Regex can simplify your code, and Regex is useful across multiple languages.

Regular Expression Basics

Like `String` and `Integer`, a regular expression is a data type in Ruby, defining a pattern of characters.

You can form a Regex variable like this:

```
pattern = /ada/
```

`/ada/` is a RegEx literal representing a pattern matching any String with the letters “ada” inside it. The two forward slashes indicate a regular expression, or a pattern of text. Anything put between the slashes forms the pattern of text we can match strings against.

You can think of it like a String, but instead of specifying a specific list of characters by enclosing them with quotes (“”), instead it defines a **pattern** of characters by enclosing them with forward slashes.

You can test a String against the regular expression with the Regex’s `match` method. The `match` method compares the string to the pattern, character-by-character and will return a `MatchData` object upon a match and `nil` if the String does not match the pattern. It’s important to note that `match` will return a

`MatchData` object if any substring matches the pattern, not necessarily the entire String.

For example:

```
pattern = /ada/
if pattern.match("ada lovelace")
  puts "The String has ada in it!"
else
  puts "It doesn't match"
end
```

The above snippet will print out “The String has ada in it!”

Regular Expressions can also be compared using the `=~` operator. The `=~` operator returns the index of the first match in the string For example: `pattern =~ 'ada'` will return 0, while `pattern =~ "learn at ada academy."` will return 9.

Both `match` and `=~` will return a truthy result if any substring matches the pattern. That’s an important issue to remember. If you want to match a pattern exactly, the regular expression needs to be more specific using special characters to indicate the start and end of the string.

On the other hand what if you wanted to match either “Ada” or “ada.” To handle both lower and upper case “Ada,” we need to provide our pattern options to match against. To provide a list of possible characters we can use Character sets.

Character Sets

A **character set**, also called a **character class** is a way to tell the regex engine to match only one out of several characters. We define a character set with square brackets. For example `/[Ss]/` will match both capital and lowercase S. Combining the character set with the previous larger literal, `[Aa]da` will match both “Ada” and “ada”.

You can also adjust the character set to accept a range of characters. For example: `/[A-Z]/` will accept a single character in the range A to Z (must be capitalized), while `/[0-9]/` will accept a single digit. If you wanted to accept any alphabetic characters you could use `/[A-Za-z]/`.

Practice

How could you match any alphanumeric digit like “a”, “W”, or “0”?

Check your answer here

The Wildcard and Quantifiers

Sometimes you will want to accept any character. For that purpose you can use the *wildcard* character, a period (`.`).

Another common need is for characters that are optional or can repeat. For this there are several helpful symbols called *quantifiers*: `*` , `+` and `?` .

These characters are summarized in the following table.

Character	Meaning	Example
`.`	Any one character	`/a.a/` matches `ada` , `ava` , and `a!a`
`*`	Preceding token may occur **zero** or more times	`/ad*a/` matches `aa` , `ada` , and `addda` `/[0-9]*/` matches `1` , `345` , and the empty string `.*` matches any string
`+`	Preceding token may occur **one** or more times.	`/ad+a/` matches `ada` and `addda` but **not** `aa` `/[0-9]+/` matches `1` and `345` but **not** the empty string `.+` matches any string **except for** the empty string

`?`	Preceding token is optional (may occur zero or one times)	`/ad?a/` matches `aa` and `ada`
		
		`/[0-9]/` matches `2` and the empty string but not `27` or `356`
		
		`.?` matches any one character or the empty string

Practice

Write a regular expression to match a valid email of form `name@domain.tld` - Matches `dee@adadev.org`, `adalovelace@gmail.com`, `magictavern@puppies.supplies` - Rejects `dan@adadev.`, `charles.com`, `@adadev.org`, `sarah@.org` - Use `\.` for a literal period (more on this later)

Check your answer here

The NOT `^` Character

Sometimes you want to exclude a certain group of characters, or sometimes it's easier to exclude a type of character rather than list all the valid possibilities. In that case you need the `^` character and the square brackets.

For example: `/[^abc]/` excludes a, b and c.

Another example would be `/[^0-9]/` which would exclude any digit or `/Ada is number [^2-9^a-z^A-Z^0]` which would exclude any letter or digit, except `1`.

Practice

How can you write a regex which would accept `dog`, `sog`, and `hog`, but exclude `bog`?

Check your answer here

Escape characters

There are lots of characters that have special meanings in a Regex (such as the `+` or `*` characters). Just like Strings you can use the backslash character to select the exact character in the text. It can also be used as a shortcut for common classes of characters.

Some examples include:

Escape Character	
`\n`	newline character
`\s`	Any whitespace character (space, tab, newline)
`\S`	Any non-whitespace character
`\d`	Any digit
`\D`	Any non-digit
`\.`, `\+`, `\*`, etc.	The literal character following the backslash, for example `\\` searches the String for a backslash, while `\.` looks for a period.

Practical Example:

- If we wanted a Regex to validate a US phone number in the format (ddd) ddd-dddd.
  - `/\(\d\d\d\) \d\d\d\-\d\d\d\d/`

 - This regular expression takes 3 digits inside parentheses followed by a space, then three digits a dash and then 4 digits. We will see how to simplify this a bit later.

Practice

Write a regex for any amount of US currency, for example it should match `$3.25`, `$102.73`, and `$0.25`.

Check your answer here

Start and End of a String

By default a regex will match a string if any part of the string matches. Sometimes you want your regex to be at the very beginning or very end of a string, or to match the whole string with nothing left over. In this case you can use the special characters `^` and `$`.

`^`, when placed at the beginning of a regex, will match the beginning of the string. If the string has characters before the match begins, it's not a match - `/^ada/` matches `ada` and `ada end` but not `start ada`

`$`, when placed at the end of a regex, will match the end of the regex. If there are characters in the string after the match ends, it's not a match. - `/ada$/` matches `ada` and `start ada` but not `ada end`

It is common to combine `^` and `$` in order to match an entire string.

Practice

Write a regex that will match only strings without any leading whitespace. - `"ada"`, `"ada academy"` and `"ada "` all match - `" ada"`, `" ada "` and `" "` do not match

Check your answer here

Repetitions

The `*` and `+` characters allow a token to be repeated, but often, such as for a zip code, you will want to limit a token to a specific number of repetitions. For that you can use the curly braces (`{}`). A number placed in the curly braces will indicate how many times the preceding token can be repeated. So for example `/[abc]{3}/` will allow the letters `a`, `b` or `c` to be repeated three times, so `"aaa"` would match, as would `"abc"` and `"cab"`.

A range of repetitions can also be repeated by using two parameters in the curly braces. For example `/[abc]{3, 5}/` would allow the characters to repeat between 3 and 5 times.

An example using repetitions in our phone number example would include: `/\(\d{3}\) \d{3}\-\d{4}/`

If you want to match a token a variable number of times you can place a comma inside the `{ }`. So `/a{2, 3}/` would match 2 to 3 `"a"`'s.

Conclusion

Regular expressions are a powerful tool that works in almost all languages. The same syntax with minimal changes can work in JavaScript, Ruby, Python, Java, C++, shell scripts… you get the idea.

Because Regex is almost universal there are a **lot** of tools available to compose them and a variety of pre-made Regular Expressions. Some tools are listed below.

It is also common to find pre-made regular expressions online, for example on Stack Overflow. Having a strong understanding of regex fundamentals will allow you to combine these and tweak them to your needs.

Regex Tools

There a number of tools you can use to compose regular expressions.

- Regexpal - A useful tool for composing Regular Expressions

- Rubular - similar to Regexpal, a site you can use to compose a regular expression with a handy reference table on the page.
- Regexper - a fantastic tool to generate a visual diagram of a regular expression.

List of Regex Special Characters

Character	Name	Description
\	Backslash	The backslash gives special meaning to the character following it. For example, the combination “\n” stands for the newline, one of the control characters.
^	Caret	The caret is the start of line anchor or the negate symbol. Example: “^a” matches “a” at the start of a line. Example: “[^0-9]” matches any non digit.
$	Dollar	$ the dollar is the end of line anchor.
\A	Beginning of String	\A indicates the beginning of the String, not the beginning of a line.
\Z	End of String	\Z matches the end of the String, not line.
{}	Curly Braces	{ } the open and close curly bracket are used as range quantifiers.
[]	Square Brackets	Open and close square bracket define a character class to match a single token inside the brackets.
()	Parentheses	The open and close parenthesis are used for grouping characters
.	Dot	the dot matches any character except the newline.
*	Star	The star is the match-zero-or-more quantifier.
+	Plus	The plus is the match-one-or-more quantifier.
?	Question Mark	The question mark is the match-one-or-more quantifier.
|	Pipe	The vertical pipe separates a series of alternatives.
< >	Anchors	The smaller and greater signs are anchors that specify a left or right word boundary.
-	Minus	the minus indicates a range in a character class (when it is not at the first position after the “[” opening bracket or the last position before the “]” closing bracket. For example “[A-Z]” matches any uppercase character.
&	Ampersand	The and is the “substitute complete match” symbol.

Resources

- Regex in Wikipedia
- Regular Expressions in rubylearning.com