8.3.9 Write Regular Expressions

You knew this day would come: the day when you conquered regular expressions. While they may sound a bit underwhelming, regular expressions are actually quite powerful. (You could almost say the outsized impact they will allow you to have on your dataset is *irregular*.) Puns aside, you know a big reason Britta trusted you with this dataset is because you wouldn't shy away from learning how to use and apply regular expressions.

Regular expressions are just strings of characters that are used as a search pattern. They are used to test if strings are in a specific format or contain a substring in a specific format, to extract pertinent information from strings while discarding unnecessary information, and to perform complicated replacements of substrings.

Regular expressions are used in almost all general-purpose languages like Python. For example, they are also used in JavaScript, C#, and Java. Sometimes they are the only viable solution to a problem.

Each character in a regular expression serves a purpose, based on what kind of character it is. We'll break down the different character types and the purposes they serve.

Literal Characters

Literal characters are the simplest character class. A regular expression made of literal characters will match any string that contains the expression as a substring. For example, if we made a regular expression of the string "cat" and used it to search for any matches in another string —"The tomcat was placated with the catch of the day"—it would match three times: "The tomcat was placated with the catch of the day."

However, regular expressions also have special characters that make it much more powerful than just finding a substring. We've already used the special character `\d` to find any digit from 0 to 9.

NOTE

> You might be thinking, "But `\d` is two characters!" You're not wrong. It is written with two characters. However, backslashes in strings have a long history of being used to write special characters that would be difficult to enter directly.
>
> For example, there is a character in ASCII and Unicode for creating a new line (it's aptly named "newline character"). However, adding it to a string is difficult to do, so we write `\n` inside the string, which Python converts to the newline character. Regular expressions also treat character combinations that start with a backslash as one character, so it's fine to refer to them as a single character.

Character Types: \d, \w, \s (and \D, \W, \S)

As we've seen, `\d` is a special character that matches any digit from 0 to 9. There are other special characters like `\d`. The full list is:

- `\d` will match any digit from 0 to 9.
- `\D` will match any non-digit character.
- `\w` matches a word character (a letter, digit, or underscore).

- `\W` matches any non-word character (anything other than a letter, digit, or underscore, such as spaces and punctuation).
- `\s` will match any whitespace character (including spaces, tabs, and newlines).
- `\S` will match any non-whitespace characters.

Character Sets: [ ]

If we need to be more specific than digits, alphanumeric characters, or whitespace characters, we can use the square brackets to define a character set. For example, "[ceh]at" would match '"cat", "eat", "hat", and "that", but not "rat."

We can also specify ranges of characters inside a character set. For example:

- `"[a-z]"` matches any lowercase letter.
- `"[A-Z]"` matches any uppercase character.
- `"[0-9]"` matches any digit.

We can include multiple ranges, so:

- `"[a-zA-Z]"` matches any lowercase or uppercase letter.
- `"[a-zA-Z0-9]"` matches any alphanumeric character.

But, we can also have smaller ranges, such as:

- `"[A-E]"`would match "A", "B", "C", "D", or "E".
- `"[1-3]"` would match the digits "1", "2", or "3".

We can also include character types inside a character set, so `"[a-zA-Z\d]"` and `"[a-zA-Z0-9]"` are equivalent expressions that would match any alphanumeric character.

Inside of a character set, we can specify a character that we do NOT want to include by prefacing it with a caret: `^`.

Match (Almost) Everything: .

The period, or dot (`.`), is a **wildcard** in regular expressions, which means it will match any single character whether it is a digit, a letter, whitespace, or punctuation. The only thing that a dot won't match is a line break (remember, line breaks are also stored as characters). In Python's regular expression module, there's an option to make the dot match every character, including line breaks.

Escaping: \

The dot and square brackets are examples of **metacharacters** in regular expressions. Metacharacters are like the superheroes in a regular expression because they have powers like "match everything" or "create a character set."

But what if we need metacharacters to act like ordinary literal characters? For example, if we use the regular expression "ca.," the dot will match any character, so "cat", "car", "cab", "ca!", "ca?", and "ca." would all be matches.

What if we want to specifically search for only "ca." with an actual period? The period by itself in a regular expression is being a superhero, matching everything. So when we want it to act like just a literal character, we need to give it a secret identity. We use the backslash "`\`" to do this. The backslash tells the parser to treat the upcoming metacharacter like a literal character. So, "ca\." will only match "ca.". Strings such as "cat", "car", and "cab", and the others from above won't be matched.

The backslash in a regular expression is called the **escape character**. It says that the next character gets to escape its duties as a special character in the regular expression and act like a plain old literal character. We'll meet more superpowered metacharacters, such as curly brackets, parentheses, and plus signs. If we want to match text that has curly brackets, parentheses, or plus signs, we'll use the backslash to treat these superpowered metacharacters like literal characters.

Special Counting Characters: *, +, {}, ?

There are also special **counting characters** that specify how many times a character can show up.

The first counting character is the asterisk: *. In regular expressions, the asterisk says the previous character can repeat any number of times, including zero. So, "ca*t" would match "cat" and "caaat" but also "fiction." If we want to specify that the character has to show up at least once, we use the plus sign: **+**. So, "ca+t" would match "cat" and "caaat" but not "fiction."

If we want to search for a character that shows up an exact number of times, we use the curly brackets: **{ }**. When we extracted the IMDb IDs, we wanted IDs that had exactly seven digits, or `"\d{7}"`. We can also put two numbers in curly brackets, and that would match for any number of digits within that range. So, "ca{3}t" wouldn't match "cat" or "fiction" but would match "caaat." "ca{3,5}t" would match "caaat", "caaaat", and "caaaaat" but not "cat" or "caaaaaat."

Finally, the question mark can be considered a counting character as well. The question mark is for optional characters, which means they can show up zero or one time. So, "ca?t" is equivalent to the regular expression pattern "ca{0,1}t".

Alternation: |

If we want to search for a given string or a different string, we use the alternation character, or pipe: **|**. This essentially functions as a logical OR. For example, if we wanted to match "cat" or "mouse" or "dog," we would make a string "cat|mouse|dog."

String Boundaries: ^ and $

If we need to make sure that our expression matches only at the beginning or ending of the string, we use the caret (**^**) to represent the beginning of the string, and the dollar sign (**$**) to represent the end of the string. So, "^cat" would match "cat" and "catatonic," but not "concatenate." "cat$" would also match "cat" and "tomcat," but not "catatonic."

By themselves, the string boundaries represent **zero length matches**; in other words, they don't match any actual characters themselves, just the boundaries of the string being searched.

Capture Groups: ( )

Grouping in regular expressions serves two purposes. First, groups can be used to add structure to a search pattern. For example, "1,000", "1,000,000", and "1,000,000,000" as strings all have a similar structure. There is a comma followed by three zeros that repeats as a group. We can match all of these with one regular expression, using parentheses to create a capture group. One regular expression that matches all three strings would be "1(,000)+."

The second purpose for grouping is hinted at in the name "capture group." **Capture groups** are how regular expressions define what information should be extracted.

For example, when we needed to extract IMDb IDs from the links, we put the entire expression inside brackets. This can also be helpful when you need to make sure a long string matches a certain format, but you only need a substring inside of it.

For example, `"\d{3}-\d{3}-\d{4}"` would match any phone number in the form "333-333-4444," but if you wanted to extract only the digits, you would use `"(\d{3})-(\d{3})-(\d{4})"`, and only the digits would be captured. Specifically, the digits of the phone number would be captured into three groups: area code, prefix, and line number.

Non-Capturing Groups and Negative Lookahead Groups: (?:), (?!)

We can modify the behavior of a group by including a question mark after the opening parenthesis. The first modification is a **non-capturing group**, which uses a colon after the question mark. This specifies that we only want to use the grouping structure, and we do not need to capture the information.

Non-capturing groups can feel superfluous when we're using regular expressions for just matching, but they become very important when we use regular expressions for matching and replacing.

Suppose we're anonymizing a list of phone numbers of the form "333-333-4444," and we want to change the prefix to "555," like the fake phone numbers in movies. We still need to have groups in our regular expression for the area code and the four-digit line number, but we don't want to capture them—we only want to capture the prefix.

The regular expression `"(?:\d{3})-(\d{3})-(?:\d{4})"` will match numbers of the form "123-456-7890," but it will only capture the middle group, the prefix. For example:

- "212-012-9876" matches the regular expression `"(?:\d{3})-(\d{3})-(?:\d{4})"`, but only "012" is captured. So, if we used this regular expression to replace the captured text with "555," it would turn "212-012-9876" to "212-555-9876."
- However, "012-3456" wouldn't match at all, because there's no area code. Even though the area code is in a non-capturing group, the regular expression still needs to see it before it can make a match.

Negative lookahead groups are also non-capturing groups, but they look ahead in the text and make sure a string doesn't exist after the match.

For example, imagine we have text with phone numbers still in the form "333-333-4444," but the text also contains ID numbers that are of the form "333-333-55555." The regular expression we've been using—`"(\d{3})-(\d{3})-(\d{4})"`—will see the first 10 numbers of the ID and recognize a match and return "333-333-5555" as if it were a phone number.

What we need is a regular expression that matches the first 10 numbers, but also checks that there isn't another digit after the phone number. We need a group that looks ahead of the rest of the regular expression, and reports back "negative" if there's some text we don't want to see.

That's a negative lookahead group. Negative lookahead groups start with a question mark and an exclamation mark. So, to make sure there are no extra digits, a negative lookahead group would be `"(?!\d)"`. Our new regular expression is `"(\d{3})-(\d{3})-(\d{4})(?!\d)"`:

- "333-333-4444" will match.
- "333-333-55555" will not.

Here's a cheat sheet for everything related to regular expressions that we've covered so far. The highlighted text in the Example column denotes a match.

Note: For instances of " ", only the whitespace between the quotation marks is a match, not the quotes themselves.

Character	Function	Example
literal characters	Directly matches characters	`"cat"` • "cat" • "dog" (no match)
`\d`	Matches a digit from 0 to 9	`"\d"` • "1" • "A" (no match) • "_" (no match) • "!" (no match) • " " (no match)
`\D`	Matches a non-digit	`"\D"` • "1" (no match) • "A" • "_" • "!" • " "
`\w`	Matches a word character (letter, digit, or underscore)	`"\w"` • "1" • "A"

		• "_" • "!" (no match) • " " (no match)
`\W`	Matches any non-word character	`"\W"` • "1" (no match) • "A" (no match) • "_" (no match) • "!" • " "
`\s`	Matches any whitespace character, such as spaces and tabs	`"\s"` • "1" (no match) • "A" (no match) • "_" (no match) • "!" (no match) • " "
`\S`	Matches any non-whitespace character	`\S` • "1" • "A" • "_" • "!" • " " (no match)
`[ … ]`	**Character Set** Matches any characters inside the brackets. Can	`"[A-C]"` • "A" • "B"

	specify ranges of characters as well.	• “C” • “D” (no match) • “E” (no match)
`[^ … ]`	**Negative Character Set** Matches anything *not* inside the brackets	`"[^C-E]"` • “A” • “B” • “C” (no match) • “D” (no match) • “E” (no match)
`.`	**Wildcard** Matches any character (except a newline)	`"."` • “1” • “A” • “_” • “!” • “ ”
`*`	Matches 0 or more times	`"ca*t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`+`	Matches 1 or more times	`"ca+t"` • “ct” • “cat”

		• “caat” • “caaat” • “caaaat”
`?`	Matches 0 or 1 time	`"ca?t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`{#}`	Matches a specific number of times	`"ca{2}t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`{#,}`	Matches at least a specific number of times	`"ca{2,}t"` • “ct” • “cat” • “caat” • “caaat” • “caaaat”
`{#,#}`	Matches within a specific range of times	`"ca{2,3}t"` • “ct” • “cat”

		• “caat” • “caaat” • “caaaat”
`|`	**Alternation** Matches either the expression before or the expression after	`"cat|dog"` • “cat” • “dog” • “bird”
`^`	Start of the string	`"^cat"` • “cat” • “catsup” • “concatenate” (no match) • “kitty-cat” (no match)
`$`	End of the string	`"cat$"` • “cat” • “catsup” (no match) • “concatenate” (no match) • “kitty-cat”
`</code>`	**Escape Character** Escapes the next character to be treated as a literal character	`"\$"` • “$”
`( … )`	**Capture Group**	`"c(at)"`

	Identifies matches that should be extracted	• “cat” (“at” is captured) • “bat” (no match)
`(?: … )`	**Non-Capturing Group** Identifies matches that should not be extracted	`"c(?:at)"` • “cat” ("c" is captured) • “bat” (no match)
`(?! … )`	**Negative Lookahead Group** Identifies expressions that negate earlier matches	`"cat(?! burglar)"` • “cat” • “cats” • “cat burglar” (no match)

It might feel as if we've taken a deep dive into regular expressions, but there's even more they can do. It starts getting wild quickly, with different languages having slightly different implementations. It's extremely helpful to use a regular expression tester like **RegExr** (https://regexr.com/) or **RegEx101** (https://regex101.com/) when building more complicated regular expressions.

Even though there's still more to regular expressions, we now have enough for us to parse the information in our dataset, so let's get back to it.