

8.4.2

Writing Regular Expressions

Now that you've received an introduction to regular expressions, Britta wants you to learn how to write them on your own.



We can use regular expressions to extract characters from strings in a wide variety of ways. Specifically, each character in a regular expression serves a purpose, based on what kind of character it is. We'll break down the different kinds of characters, the purposes they serve, and the ways to use them in regular expressions.

Literal Characters

The **literal character** is the simplest kind of character. A regular expression that's composed of literal characters will match any string that exactly contains the expression as a substring. For example, say that we make a regular expression of the "cat" string. And, say that we use it to search for matches in another string—specifically, "The tomcat was placated with the catch of the day". It would match three times as follows:

"The tomcat was placated with the catch of the day."

Special Characters

Regular expressions can also have **special characters**, which each consist of a backslash () followed by another character. Special characters make regular expressions more powerful than those that just find a substring. We've already used the  special character to find any digit from 0 to 9.

NOTE

Wait a minute. You might be thinking that `\d` consists of two characters! You're right—it is written with two characters. But, using a backslash to write a particular character in a string—which would be difficult to write directly—has a long history.

For example, ASCII and Unicode have a character for creating a new line, named the "newline character." But, adding it to a string is difficult. So, we write `\n` inside the string, and Python then converts that to the newline character. Regular expressions also treat a character combination that starts with a backslash as one character. So, it's fine to refer to it as a character.

Here's the full list of special characters:

- Backslash followed by lowercase d (`\d`): Matches any digit from 0 to 9.
- Backslash followed by uppercase D (`\D`): Matches any character that's not a digit.
- Backslash followed by lowercase w (`\w`): Matches a word character (that is, a letter, a digit, or an underscore).
- Backslash followed by uppercase W (`\W`): Matches any character that's not a word character (that is, anything other than a letter, a digit, or an underscore). Examples include a space or a punctuation character.
- Backslash followed by lowercase s (`\s`): Matches any whitespace character (including a space, a tab, or a newline character).
- Backslash followed by uppercase S (`\S`): Matches any character that's not a whitespace character.

Character Sets

If we need to be more specific than the special characters allow, we can use a set of brackets (`[]`) to define a character set. For example, `"\[ceh\]at"` will match any of the following strings:

- "cat"
- "eat"
- "hat"
- "that"

But, it won't match "rat".

We can also specify a range of characters for a character set. Here are some examples:

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

We can even include multiple ranges, like the following ones:

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

We can have smaller ranges, like the following ones:

- `"[A-E]"`: Matches "A", "B", "C", "D", or "E".
- `"[1-3]"`: Matches "1", "2", or "3".

We can include special characters in a character set. So, `"[a-zA-Z\d]"` and `"[a-zA-Z0-9]"` are equivalent expressions that each match any alphanumeric character.

Finally, we can specify a character that we do not want to include in a character set by prefacing it with a caret (`^`).

Before moving on, check your knowledge with the following assessment:

Wildcard

The period, or dot (`.`), is a **wildcard** in regular expressions. This means that it will match any single character—whether that's a digit, a letter, a whitespace character, or a punctuation character. The only character that a dot won't match is a newline character. The Python regular expression module has an option to make the dot match every character, including the newline character.

Escape Character

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

But, what if we need a metacharacter to act like an ordinary literal character? For example, consider the "ca." regular expression. The dot will match any character. So, the following will all match:

- "cat"
- "car"
- "cab"
- "ca!"
- "ca?"
- "ca."

What if we want to specifically search only for "ca.", which contains an actual dot? The dot in a regular expression acts as a superhero, matching every character. So when we want it to act like a literal character, we need to give it a secret identity. To do so, we use the backslash (`\`). The backslash tells the parser to treat the upcoming metacharacter like a literal character. So, "ca\" will match only "ca.". Strings such as "cat", "car", and "cab" won't match.

The backslash in a regular expression is called the **escape character**. It indicates that the next character gets to escape its duties as a metacharacter and act like a literal character. We'll meet more superpowered metacharacters, like braces (`{ }`), parentheses (`()`), and the plus sign (`+`). If we want to match text that contains a metacharacter, we'll use the backslash to treat it like a literal character.

Before moving on, check your knowledge with the following assessment:

Special Counting Characters

Special **counting characters** specify the number of times that a character can appear. They include the asterisk (`*`), the plus sign (`+`), braces (`{ }`), and the question mark (`?`). Let's go over each of these.

The asterisk (`*`) means that the previous character can repeat any number of times, including zero. For example, "ca*t" will match any of the following:

- "cat"
- "caaat"
- "fiction"

To specify that the character must appear at least once, we use the plus sign (`+`). For example, "ca+t" will match any of the following:

- "cat"

- "caaat"

But, it won't match "fiction".

To search for a character that appears an exact number of times, we use braces (`{ }`). Earlier, when we extracted the four-digit contact ID numbers, we wanted numbers with exactly four digits. So, we used `"\d{4}"`. Another example is `"ca{3}t"`. This will match "caaat." but not "cat" or "fiction". To match a number of characters within a range, we place two numbers inside the braces. For example, `"ca{3,5}t"` will match any of the following:

- "caaat"
- "caaaat"
- "caaaaaat"

But, it won't match "cat" or "caaaaaat".

Finally, we use the question mark (`?`) for an optional character. This means that it can appear zero times or one time. For example, `"ca?t"` is equivalent to `"ca{0,1}t"`.

Alternation Character

To search for either a particular string or alternate strings, we use the **alternation character**, which is the pipe (`|`). This functions as a logical OR. For example, to match "cat" or "mouse" or "dog", we use `"cat|mouse|dog"`.

String Boundary Characters

To search for a substring that matches only the beginning or the end of a string, we use one of the string boundary characters. Specifically, we use the caret (`^`) to represent the beginning of a string. And, we use the dollar sign (`$`) to represent the end of a string. For example, `"^cat"` will match "cat" and "catch" but not "concatenate". And, `"cat$"` will match "cat" and "tomcat" but not "catch".

By themselves, string boundary characters represent **zero-length matches**. That is, they don't match any characters themselves. Instead, they match the boundaries of the string being searched.

Capture Groups

Earlier, we used a capture group to extract a four-digit number. Now, we'll cover capture groups in more detail.

Grouping in regular expressions serves two purposes. First, we can use groups 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. A comma followed by three zeros repeats as a group. We can match all of these with one regular expression by using parentheses to create a capture group. This expression is `"1(,000)+"`.

The name "capture group" hints at the second purpose for grouping. A **capture group** is how a regular expression defines the information to extract. We use parentheses (`()`) to define a capture group.

For example, `"\d{3}-\d{3}-\d{4}"` will match any phone number in the form "333-333-4444". But, what if we want to extract only the digits and not the hyphens? The answer is that we use `"(\d{3})-(\d{3})-(\d{4})"`. Specifically, that regular expression pattern will capture the digits of the phone number into three groups: area code, prefix, and line number.

Non-Capturing Groups and Negative Lookaheads

We can modify the behavior of a group in two ways by including a question mark after the opening parenthesis. First, we can use a question mark followed by a colon (`(?:)`) to define a non-capturing group. A **non-capturing group** specifies that we want to use the grouping structure but not capture the information. Second, we can use a question mark followed by an exclamation point (`(?!)`) to define a negative lookahead. A **negative lookahead** is also a non-capturing group, but it looks ahead in the text to make sure that a string doesn't exist after the match. Let's discuss each in more detail.

Non-capturing groups can feel superfluous when we use regular expressions for just matching. But, they become important when we use regular expressions for both matching and replacing.

For example, suppose that we're anonymizing a list of phone numbers in the form "333-333-4444". And, we want to change the prefix to "555", like fictitious phone numbers in movies have. 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 want to capture only the prefix.

The regular expression `"(?:\d{3})-(\d{3})-(?:\d{4})"` will match numbers in the form "123-456-7890". But, it will capture only the middle group—that is, the prefix. Consider the following examples:

- The string "212-012-9876" matches the regular expression `"(?:\d{3})-(\d{3})-(?:\d{4})"` but captures only "012". So if we use this regular expression to replace the captured text with "555", it will turn "212-012-9876" into "212-555-9876".
- The string "012-3456" won't match at all, because it doesn't have an area code. Even though a non-capturing group contains the area code, the regular expression still needs to find it before it can make a match.

Let's now discuss negative lookaheads. As stated earlier, a negative lookahead is also a non-capturing group. But, it looks ahead (or checks further along) in the text to make sure that a string doesn't exist after the match.

For example, suppose that we have text with phone numbers still in the form "333-333-4444". But, the text also contains ID numbers in the form "333-333-55555". The regular expression that we've been using—`"(\d{3})-`

`(\d{3})-(\d{4})"`—will find the first 10 numbers of the ID, 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 another digit doesn't exist after the phone number. That is, we need a group that looks ahead and reports back "negative" if another number exists after the tenth digit.

The group that we need is a negative lookahead group. As mentioned earlier, a negative lookahead starts with a question mark followed by an exclamation point (`?!`). So to make sure that no extra digits exist, we use `"` `(?!\d)"`. Our new regular expression is thus `"(\d{3})-(\d{3})-(\d{4})(?!\d)"`. For this regular expression, consider the following examples:

- The string "333-333-4444" will match.
- The string "333-333-55555" won't match.

Regular Expression Cheat Sheet

The following table contains 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 quotes—and not the quotes themselves—match.

Character	Function	Example
Literal characters	Directly matches characters	<code>"cat"</code> <ul style="list-style-type: none"> • "cat" • "dog" (no match)
<code>\d</code>	Matches a digit from 0 to 9	<code>"\d"</code> <ul style="list-style-type: none"> • "1" • "A" (no match) • "_" (no match) • "!" (no match) • " " (no match)
<code>\D</code>	Matches a nondigit	<code>"\D"</code> <ul style="list-style-type: none"> • "1" (no match)

		<ul style="list-style-type: none"> • "A" • " _" • "!" • " "
<code>\w</code>	Matches a word character (letter, digit, or underscore)	<code>"\w"</code> <ul style="list-style-type: none"> • "1" • "A" • " _" • "!" (no match) • " " (no match)
<code>\W</code>	Matches a nonword character	<code>"\W"</code> <ul style="list-style-type: none"> • "1" (no match) • "A" (no match) • " _" (no match) • "!" • " "
<code>\s</code>	Matches any whitespace character, such as a space or a tab	<code>"\s"</code> <ul style="list-style-type: none"> • "1" (no match) • "A" (no match) • " _" (no match) • "!" (no match) • " "
<code>\S</code>	Matches any non-whitespace character	<code>\S</code> <ul style="list-style-type: none"> • "1" • "A" • " _" • "!" • " " (no match)

<div>[...]</div>	Character Set Matches any characters inside the brackets. Can also specify ranges of characters.	<div>"[A-C]"</div> <ul style="list-style-type: none"> "A" "B" "C" "D" (no match) "E" (no match)
<div>[^ ...]</div>	Negative Character Set Matches anything that's not inside the brackets	<div>"[^C-E]"</div> <ul style="list-style-type: none"> "A" "B" "C" (no match) "D" (no match) "E" (no match)
<div>.</div>	Wildcard Matches any character (except a newline)	<div>"."</div> <ul style="list-style-type: none"> "1" "A" "_" "!" " "
<div>*</div>	Matches 0 or more times	<div>"ca*t"</div> <ul style="list-style-type: none"> "ct" "cat" "caat" "caaat" "caaaat"
<div>+</div>	Matches 1 or more times	<div>"ca+t"</div> <ul style="list-style-type: none"> "ct" "cat" "caat" "caaat"

		<ul style="list-style-type: none"> • "caaaat"
?	Matches 0 or 1 time	<p>"ca?t"</p> <ul style="list-style-type: none"> • "ct" • "cat" • "caat" • "caaat" • "caaaat"
{#}	Matches a specific number of times	<p>"ca{2}t"</p> <ul style="list-style-type: none"> • "ct" • "cat" • "caat" • "caaat" • "caaaat"
{#,}	Matches at least a specific number of times	<p>"ca{2,}t"</p> <ul style="list-style-type: none"> • "ct" • "cat" • "caat" • "caaat" • "caaaat"
{#,#}	Matches within a range of times	<p>"ca{2,3}t"</p> <ul style="list-style-type: none"> • "ct" • "cat" • "caat" • "caaat" • "caaaat"
	<p>Alternation</p> <p>Matches the expression either before or after the pipe</p>	<p>"cat dog"</p> <ul style="list-style-type: none"> • "cat" • "dog"

		<ul style="list-style-type: none"> • "bird"
<div>^</div>	Start of the string	<div>"^cat"</div> <ul style="list-style-type: none"> • "cat" • "catsup" • "concatenate" (no match) • "kitty-cat" (no match)
<div>\$</div>	End of the string	<div>"cat\$"</div> <ul style="list-style-type: none"> • "cat" • "catsup" (no match) • "concatenate" (no match) • "kitty-cat"
<div></code></div>	Escape Character Escapes the next character, which will be treated as a literal character	<div>"\\$"</div> <ul style="list-style-type: none"> • "\$"
<div>(...)</div>	Capture Group Identifies matches that should be extracted	<div>"c(at)"</div> <ul style="list-style-type: none"> • "cat" ("at" is captured) • "bat" (no match)
<div>(?: ...)</div>	Non-Capturing Group Identifies matches that should not be extracted	<div>"c(?:at)"</div> <ul style="list-style-type: none"> • "cat" ("c" is captured) • "bat" (no match)
<div>(?! ...)</div>	Negative Lookahead Identifies expressions that negate earlier matches	<div>"cat(?! burglar)"</div> <ul style="list-style-type: none"> • "cat" • "cats" • "cat burglar" (no match)

DEEP DIVE ▼

Even though regular expressions involve even more, we now know enough to parse the information in our dataset. We'll do that next.

NOTE

Different languages have slightly different implementations for regular expressions. Using a regular expression tester, like [RegExr](https://regexr.com/) ➞ [\(https://regexr.com/\)](https://regexr.com/) or [RegEx101](https://regex101.com/) ➞ [\(https://regex101.com/\)](https://regex101.com/), is extremely helpful when building more-complicated regular expressions.

© 2020 - 2022 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.