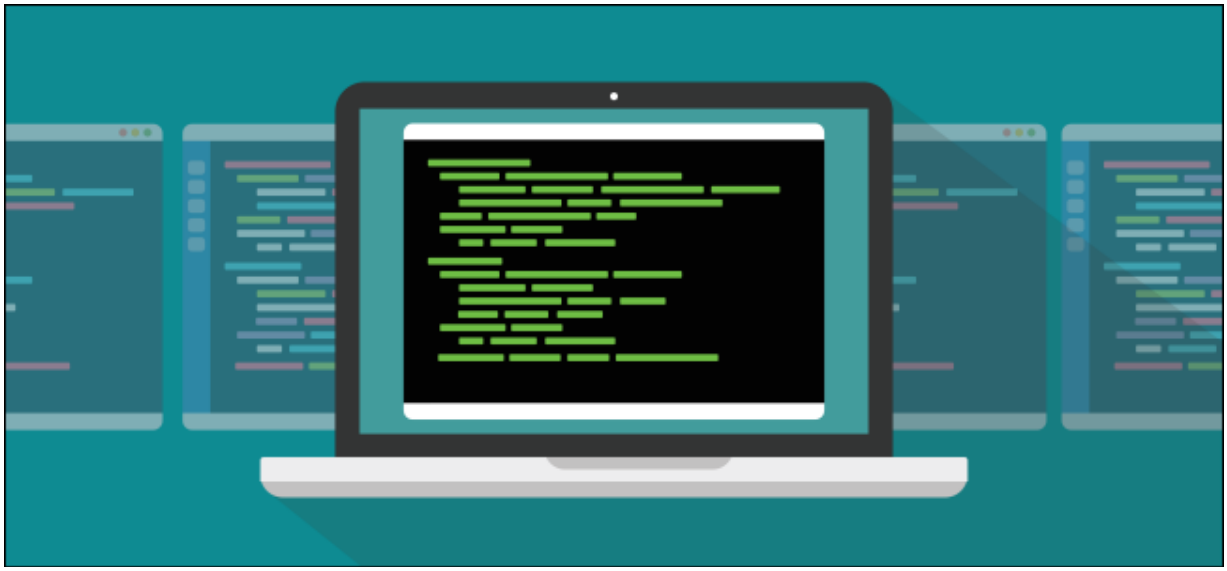
How to Use Regular Expressions (regexes) on Linux



Fatmawati Achmad Zaenuri/Shutterstock

Wondering what those weird strings of symbols do on Linux? They give you command-line magic! We'll teach you how to cast regular expression spells and level up your command-line skills.

What Are Regular Expressions?

Regular expressions (regexes) are a way to find matching character sequences. They use letters and symbols to define a pattern that's searched for in a file or stream. There are several different flavors off regex. We're going to look at the version used in common Linux utilities and commands, like `grep`, the command that prints lines that match a search pattern.

Entire books have been written about regexes, so this tutorial is merely an introduction. There are basic and extended regexes, and we'll use the extended here.

To use the extended regular expressions with `grep`, you have to use the `-E` (extended) option. Because this gets tiresome very quickly, the `egrep` command was created. The `egrep` command is the same as the `grep -E` combination, you just don't have to use the `-E` option every time.

If you find it more convenient to use `egrep`, you can. However, just be aware it's officially deprecated. It's still present in all the distributions we checked, but it might go away in the future.

Of course, you can always make your own aliases, so your favored options are always included for you.

RELATED: ***How to Create Aliases and Shell Functions on Linux***

From Small Beginnings

For our examples, we'll use a plain text file containing a list of Geeks. Remember that you can use regexes with many Linux commands. We're just using `grep` as a convenient way to demonstrate them.

Here are the contents of the file:

```
less geek.txt
```

```
dave@howtogeek:~$ less geeks.txt
```

The first part of the file is displayed.

```
Tim Brooks
Catherine Chang
Joel Cornell
Michael Crider
Justin Duino
Benj Edwards
Jason Fitzpatrick
Amanda Gambill
Walter Glenn
Harry Guinness
Marshall Gunnell
Lowell Heddings
Andrew Heinzman
Josh Hendrickson
Chris Hoffman
Akemi Iwaya
Dave Johnson
Dave McKay
Alan Murray
Khamosh Pathak
geeks.txt
```

Let's start with a simple search pattern and search the file for occurrences of the letter "o." Again, because we're using the `-E` (extended regex) option in all of our examples, we type the following:

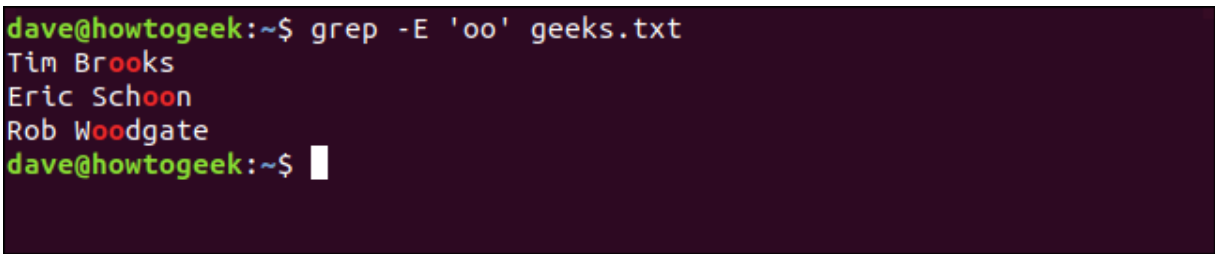
```
grep -E 'o' geeks.txt
```

```
dave@howtogeek:~$ grep -E 'o' geeks.txt
Tim Brooks
Joel Cornell
Justin Duino
Jason Fitzpatrick
Lowell Heddings
Josh Hendrickson
Chris Hoffman
Dave Johnson
Khamosh Pathak
Eric Schoon
Cameron Summerson
Tom Westrick
Rob Woodgate
No more.
dave@howtogeek:~$
```

Each line that contains the search pattern is displayed, and the matching letter is highlighted. We've performed a simple search, with no constraints. It doesn't matter if the letter appears more than once, at the end of the string, twice in the same word, or even next to itself.

A couple of names had double O's; we type the following to list only those:

```
grep -E 'oo' geeks.txt
```



Our result set, as expected, is much smaller, and our search term is interpreted literally. It doesn't mean anything other than what we typed: double "o" characters.

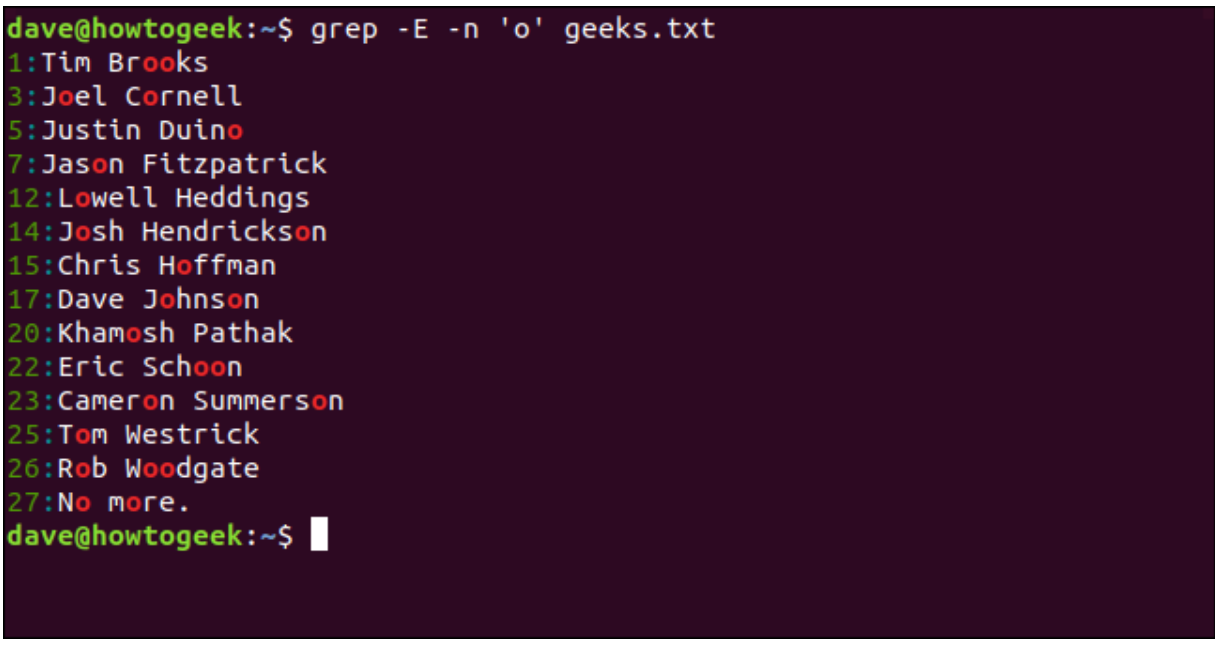
We'll see more functionality with our search patterns as we move forward.

Line Numbers and Other grep Tricks

If you want `grep` to list the line number of the matching entries, you can use the `-n` (line number) option. This is a `grep` trick—it's not part of the regex functionality. However, sometimes, you might want to know where in a file the matching entries are located.

We type the following:

```
grep -E -n 'o' geeks.txt
```



Another handy `grep` trick you can use is the `-o` (only matching) option. It only displays the matching character sequence, not the surrounding text. This can be useful if you need to quickly scan a list for duplicate matches on any of the lines.

To do so, we type the following:

```
grep -E -n -o 'o' geeks.txt
```

```
dave@howtogeek:~$ grep -E -n -o 'll' geeks.txt
3:ll
8:ll
11:ll
11:ll
12:ll
dave@howtogeek:~$
```

If you want to reduce the output to the bare minimum, you can use the `-c` (count) option.

We type the following to see the number of lines in the file that contain matches:

```
grep -E -c 'o' geeks.txt
```

```
dave@howtogeek:~$ grep -E -c 'o' geeks.txt
14
dave@howtogeek:~$
```

The Alternation Operator

If you want to search for occurrences of both double “l” and double “o,” you can use the pipe (`|`) character, which is the alternation operator. It looks for matches for either the search pattern to its left or right.

We type the following:

```
grep -E -n -o 'll|oo' geeks.txt
```

```
dave@howtogeek:~$ grep -E -n -o 'll|oo' geeks.txt
1:oo
3:ll
8:ll
11:ll
11:ll
12:ll
22:oo
26:oo
dave@howtogeek:~$
```

Any line containing a double “l,” “o,” or both, appears in the results.

Case Sensitivity

You can also use the alternation operator to create search patterns, like this:

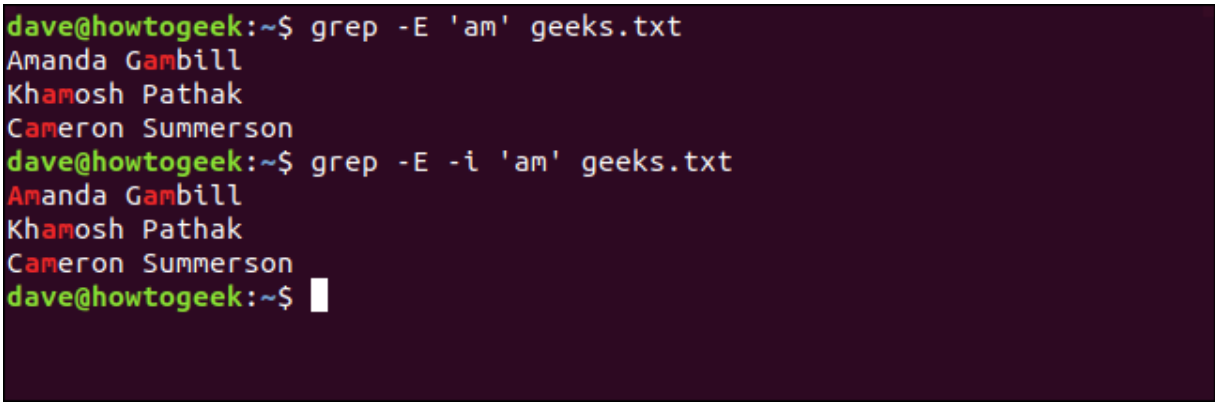
```
am|Am
```

This will match both “am” and “Am.” For anything other than trivial examples, this quickly leads to cumbersome search patterns. An easy way around this is to use the `-i` (ignore case) option with `grep`.

To do so, we type the following:

```
grep -E 'am' geeks.txt
```

```
grep -E -i 'am' geeks.txt
```



The first command produces three results with three matches highlighted. The second command produces four results because the “Am” in “Amanda” is also a match.

Anchoring

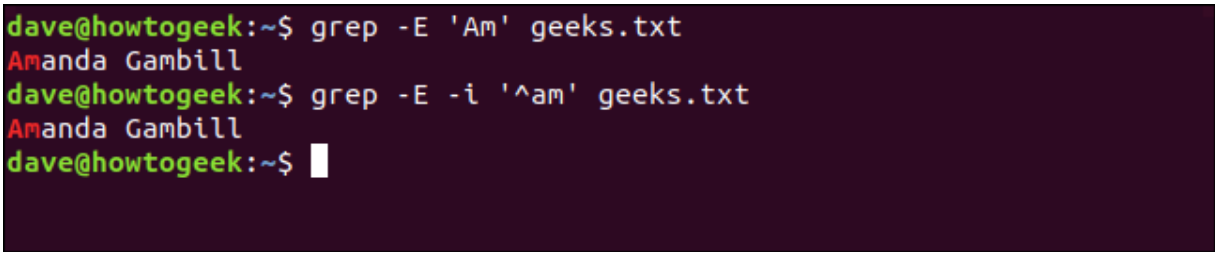
We can match the “Am” sequence in other ways, too. For example, we can search for that pattern specifically or ignore the case, and specify that the sequence must appear at the beginning of a line.

When you match sequences that appear at the specific part of a line of characters or a word, it’s called anchoring. You use the caret (^) symbol to indicate the search pattern should only consider a character sequence a match if it appears at the start of a line.

We type the following (note the caret is inside the single quotes):

grep -E ‘Am’ geeks.txt

```
grep -E -i '^am' geeks.txt
```



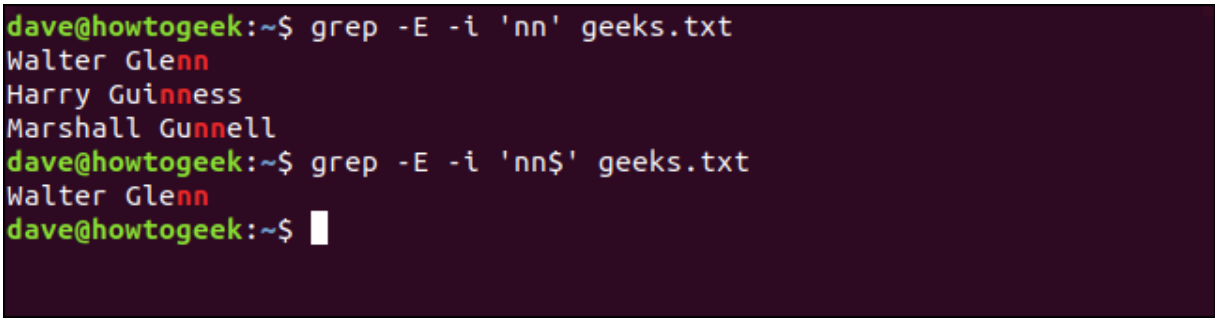
Both of these commands match “Am.”

Now, let’s look for lines that contain a double “n” at the end of a line.

We type the following, using a dollar sign ($) to represent the end of the line:

```
grep -E -i 'nn' geeks.txt
```

```
grep -E -i 'nn$' geeks.txt
```

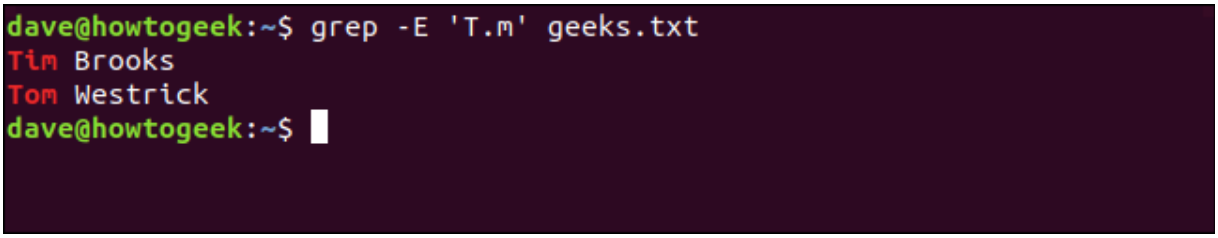


Wildcards

You can use a period (.) to represent any single character.

We type the following to search for patterns that start with “T,” end with “m,” and have a single character between them:

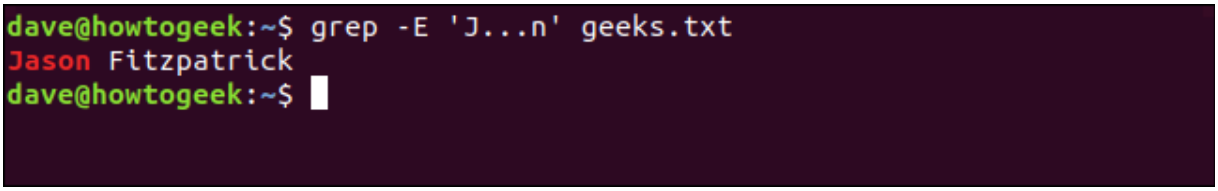
```
grep -E 'T.m' geeks.txt
```



The search pattern matched the sequences “Tim” and “Tom.” You can also repeat the periods to indicate a certain number of characters.

We type the following to indicate we don’t care what the middle three characters are:

```
grep-E 'J...n' geeks.txt
```



The line containing “Jason” is matched and displayed.

Use the asterisk (*) to match zero or more occurrences of the preceding character. In this example, the character that will precede the asterisk is the period (.), which (again) means any character.

This means the asterisk (*) will match any number (including zero) of occurrences of any character.

The asterisk is sometimes confusing to regex newcomers. This is, perhaps, because they usually use it as a wildcard that means “anything.”

In regexes, though, `'c*t'` doesn’t match “cat,” “cot,” “coot,”‘ etc. Rather, it translates to “match zero or more ‘c’ characters, followed by a ‘t’.” So, it matches “t,” “ct,” “cct,” “ccct,” or any number of “c” characters.

Because we know the format of the content in our file, we can add a space as the last character in the search pattern. A space only appears in our file between the first and last names.

So, we type the following to force the search to include only the first names from the file:

```
grep -E 'J.*n ' geeks.txt
```

```
grep -E 'J.*n ' geeks.txt
```

```
dave@howtogeek:~$ grep -E 'J.*n' geeks.txt
Joel Cornell
Justin Duino
Jason Fitzpatrick
Josh Hendrickson
Dave Johnson
dave@howtogeek:~$ grep -E 'J.*n ' geeks.txt
Justin Duino
Jason Fitzpatrick
dave@howtogeek:~$
```

At first glance, the results from the first command seem to include some odd matches. However, they all match the rules of the search pattern we used.

The sequence has to begin with a capital “J,” followed by any number of characters, and then an “n.” Still, although all the matches begin with “J” and end with an “n,” some of them are not what you might expect.

Because we added the space in the second search pattern, we got what we intended: all first names that start with “J” and end in “n.”

Character Classes

Let’s say we want to find all lines that start with a capital “N” or “W.”

If we use the following command, it matches any line with a sequence that starts with either a capital “N” or “W,” no matter where it appears in the line:

```
grep -E 'N|W' geeks.txt
```

That’s not what we want. If we apply the start of line anchor (^) at the beginning of the search pattern, as shown below, we get the same set of results, but for a different reason:

```
grep -E '^N|W' geeks.txt
```

```
dave@howtogeek:~$ grep -E 'N|W' geeks.txt
Walter Glenn
Erik Wang
Tom Westrick
Rob Woodgate
No more.
dave@howtogeek:~$ grep -E '^N|W' geeks.txt
Walter Glenn
Erik Wang
Tom Westrick
Rob Woodgate
No more.
```

The search matches lines that contain a capital “W,” anywhere in the line. It also matches the “No more” line because it starts with a capital “N.” The start of line anchor (^) is only applied to the capital “N.”

We could also add a start of line anchor to capital “W,” but that would soon become inefficient in a search pattern any more complicated than our simple example.

The solution is to enclose part of our search pattern in brackets (`[]`) and apply the anchor operator to the group. The brackets (`[]`) mean "any character from this list." This means we can omit the (`|`) alternation operator because we don't need it.

We can apply the start of line anchor to all the elements in the list within the brackets (`[]`). (Note the start of line anchor is outside of the brackets).

We type the following to search for any line that starts with a capital "N" or "W":

```
grep -E '^[NW]' geeks.txt
```

```
dave@howtogeek:~$ grep -E '^[NW]' geeks.txt
Walter Glenn
No more.
dave@howtogeek:~$
```

We'll use these concepts in the next set of commands, as well.

We type the following to search for anyone named Tom or Tim:

```
grep -E 'T[oi]m' geeks.txt
```

If the caret (`^`) is the first character in the brackets (`[]`), the search pattern looks for any character that doesn't appear in the list.

For example, we type the following to look for any name that starts with "T," ends in "m," and in which the middle letter isn't "o":

```
grep -E 'T[^o]m' geeks.txt
```

We can include any number of characters in the list. We type the following to look for names that start with "T," end in "m," and contain any vowel in the middle:

```
grep -E 'T[aeiou]m' geeks.txt
```

```
dave@howtogeek:~$ grep -E 'T[oi]m' geeks.txt
Tim Brooks
Tom Westrick
dave@howtogeek:~$ grep -E 'T[^o]m' geeks.txt
Tim Brooks
dave@howtogeek:~$ grep -E 'T[aeiou]m' geeks.txt
Tim Brooks
Tom Westrick
dave@howtogeek:~$
```

Interval Expressions

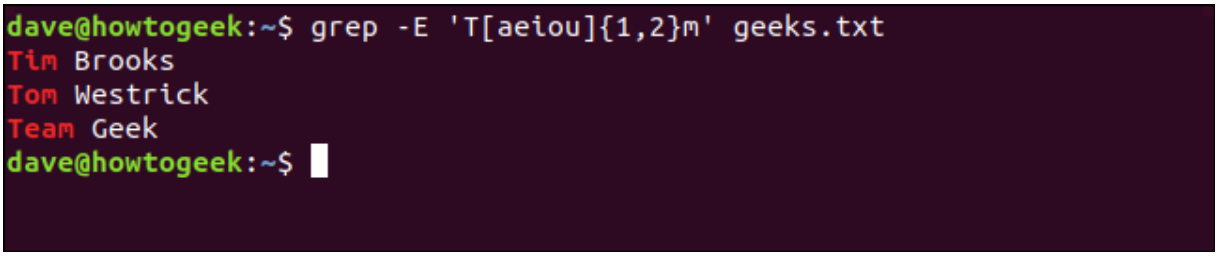
You can use interval expressions to specify the number of times you want the preceding character or group to be found in the matching string. You enclose the number in curly brackets (`{}`).

A number on its own means specifically that number, but if you follow it with a comma (`,`), it means that number or more. If you separate two numbers with a comma (`1,2`), it means the range of numbers from the smallest to largest.

We want to look for names that start with "T," are followed by at least one, but no more than two, consecutive vowels, and end in "m."

So, we type this command:

```
grep -E 'T[aeiou]{1,2}m' geeks.txt
```



This matches “Tim,” “Tom,” and “Team.”

If we want to search for the sequence “el,” we type this:

```
grep -E 'el' geeks.txt
```

We add a second “l” to the search pattern to include only sequences that contain double “l”:

```
grep -E 'ell' geeks.txt
```

This is equivalent to this command:

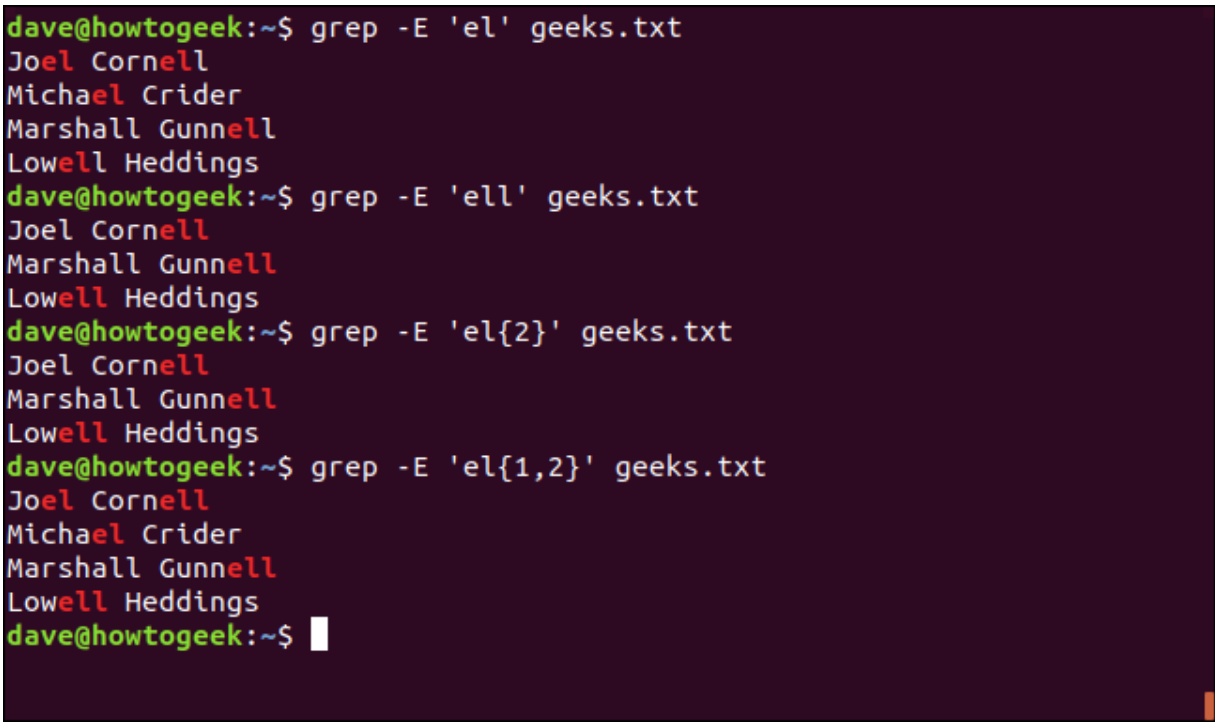
```
grep -E 'el{2}' geeks.txt
```

If we provide a range of “at least one and no more than two” occurrences of “l,” it will match “el” and “ell” sequences.

This is subtly different from the results of the first of these four commands, in which all the matches were for “el” sequences, including those inside the “ell” sequences (and only one “l” is highlighted).

We type the following:

```
grep -E 'el{1,2}' geeks.txt
```



To find all sequences of two or more vowels, we type this command:

```
grep -E '[aeiou]{2,}' geeks.txt
```

```
dave@howtogeek:~$ grep -E "[aeiou]{2,}" geeks.txt
Tim Brooks
Joel Cornell
Michael Crider
Justin Duino
Harry Guinness
Andrew Heinzman
Ian Paul
Eric Schoon
Rob Woodgate
Team Geek 2020
dave@howtogeek:~$
```

Escaping Characters

Let's say we want to find lines in which a period (`.`) is the last character. We know the dollar sign (`$`) is the end of line anchor, so we might type this:

```
grep -E '.$' geeks.txt
```

```
dave@howtogeek:~$ grep -E '.$' geeks.txt
```

However, as shown below, we don't get what we expected.

```
Walter Glenn
Harry Guinness
Marshall Gunnell
Lowell Heddings
Andrew Heinzman
Josh Hendrickson
Chris Hoffman
Akemi Iwaya
Dave Johnson
Dave McKay
Alan Murray
Khamosh Pathak
Ian Paul
Eric Schoon
Cameron Summerson
Erik Wang
Tom Westrick
Rob Woodgate
Team Geek
No more.
dave@howtogeek:~$
```

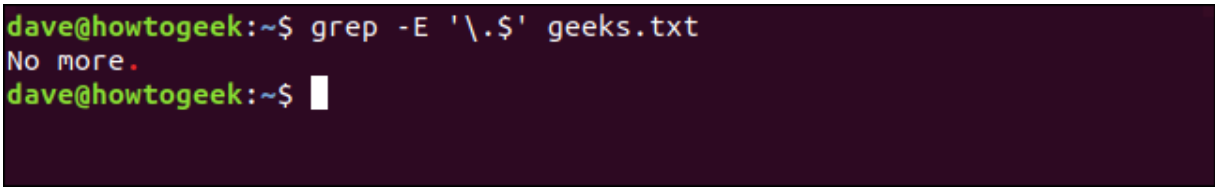
As we covered earlier, the period (`.`) matches any single character. Because every line ends with a character, every line was returned in the results.

So, how do you prevent a special character from performing its regex function when you just want to search for that actual character? To do this, you use a backslash (`\`) to escape the character.

One of the reasons we're using the `-E` (extended) options is because they require a lot less escaping when you use the basic regexes.

We type the following:

```
grep -e '\.$' geeks.txt
```



This matches the actual period character (`.`) at the end of a line.

Anchoring and Words

We covered both the start (`^`) and end of line (`$`) anchors above. However, you can use other anchors to operate on the boundaries of words.

In this context, a word is a sequence of characters bounded by whitespace (the start or end of a line). So, “psy66oh” would count as a word, although you won’t find it in a dictionary.

The start of word anchor is (`\<`); notice it points left, to the start of the word. Let’s say a name was mistakenly typed in all lowercase. We can use the grep `-i` option to perform a case-insensitive search and find names that start with “h.”

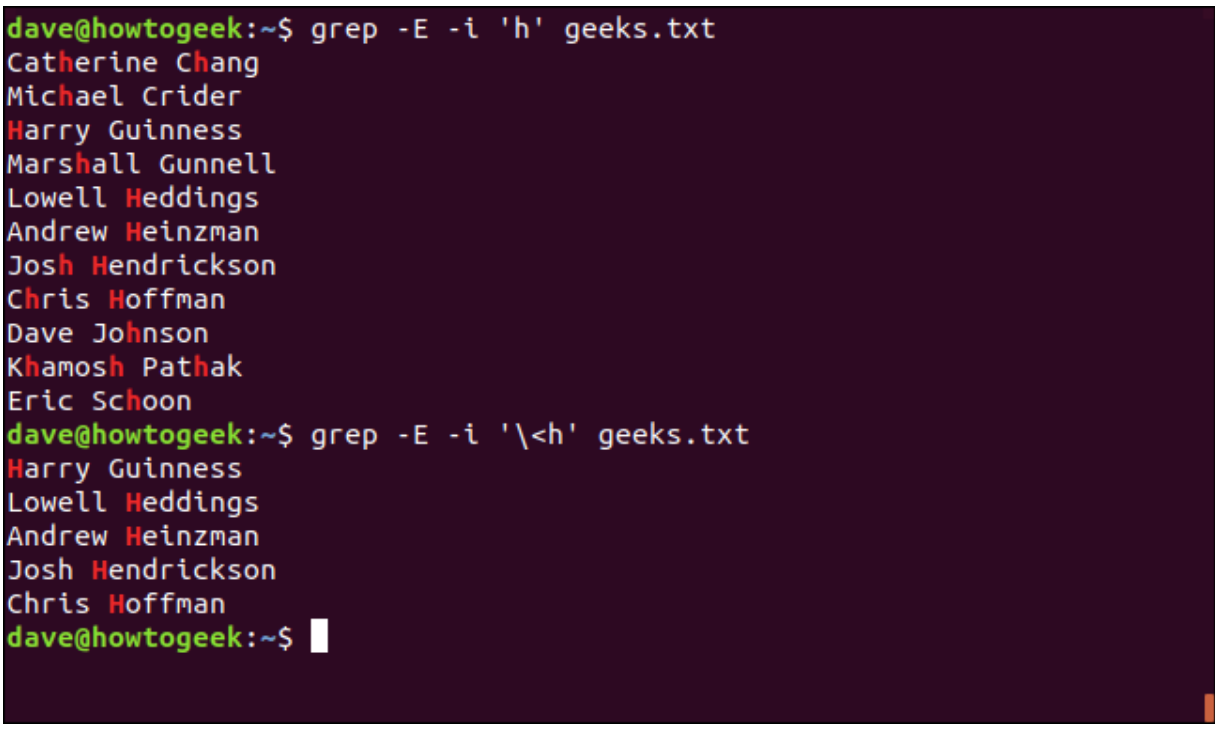
We type the following:

```
grep -E -i 'h' geeks.txt
```

That finds all occurrences of “h”, not just those at the start of words.

```
grep -E -i '\<h' geeks.txt
```

This finds only those at the start of words.



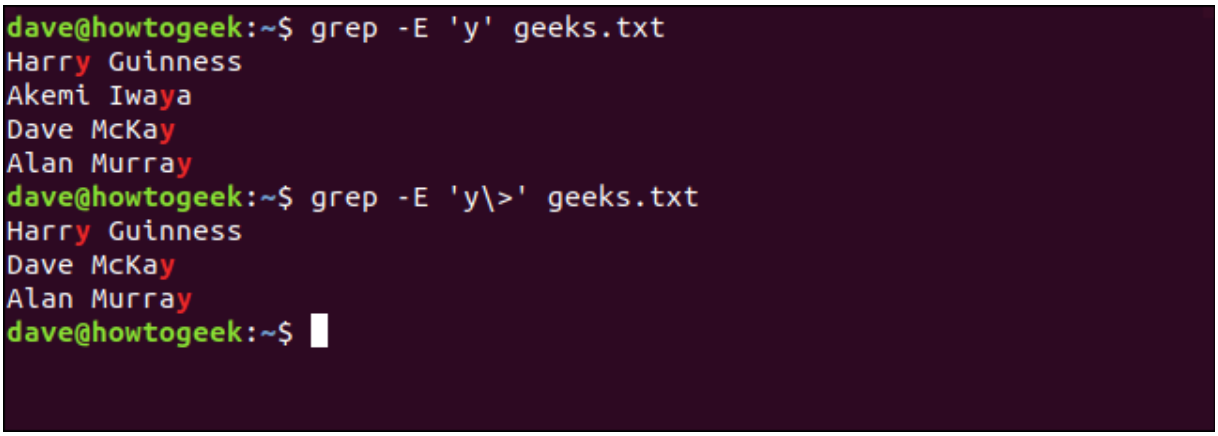
Let’s do something similar with the letter “y”; we only want to see instances in which it’s at the end of a word. We type the following:

```
grep -E 'y' geeks.txt
```

This finds all occurrences of “y,” wherever it appears in the words.

Now, we type the following, using the end of word anchor (`/>`) (which points to the right, or the end of the word):

```
grep -E 'y\>' geeks.txt
```

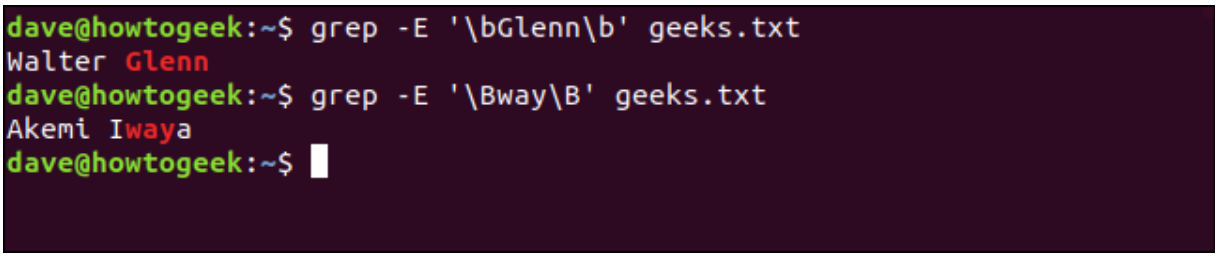


The second command produces the desired result.

To create a search pattern that looks for an entire word, you can use the boundary operator (`\b`). We’ll use the boundary operator (`\B`) at both ends of the search pattern to find a sequence of characters that must be inside a larger word:

```
grep -E '\bGlenn\b' geeks.txt
```

```
grep -E '\Bway\B' geeks.txt
```



More Character Classes

You can use shortcuts to specify the lists in character classes. These range indicators save you from having to type every member of a list in the search pattern.

You can use all the following:

- **A-Z:** All uppercase letters from “A” to “Z.”
- **a-z:** All lowercase letters from “a” to “z.”
- **0-9:** All digits from zero to nine.
- **d-p:** All lowercase letters from “d” to “p.” These free-format styles allow you to define your own range.
- **2-7:** All numbers from two to seven.

You can also use as many character classes as you want in a search pattern. The following search pattern matches sequences that start with “J,” followed by an “o” or “s,” and then either an “e,” “h,” “l,” or “s”:

```
grep -E 'J[os][ehls]' geeks.txt
```

```
dave@howtogeek:~$ grep -E "J[os][ehls]" geeks.txt
Joel Cornell
Josh Hendrickson
Dave Johnson
dave@howtogeek:~$
```

In our next command, we'll use the `a-z` range specifier.

Our search command breaks down this way:

- **H:** The sequence must start with "H."
- **[a-z]:** The next character can be any lowercase letter in this range.
- ***:** The asterisk here represents any number of lowercase letters.
- **man:** The sequence must end with "man."

We put it all together in the following command:

```
grep -E 'H[a-z]*man' geeks.txt
```

```
dave@howtogeek:~$ grep -E "H[a-z]*man" geeks.txt
Andrew Heinzman
Chris Hoffman
dave@howtogeek:~$
```

Nothing's Impenetrable

Some regexes can quickly become difficult to visually parse. When people write complicated regexes, they usually start off small and add more and more sections until it works. They tend to increase in sophistication over time.

When you try to work backward from the final version to see what it does, it's a different challenge altogether.

For example, look at this command:

```
grep -E '^([0-9]{4}[- ]){3}[0-9]{4}|[0-9]{16}' geeks.txt
```

Where would you begin untangling this? We'll start at the beginning and take it one chunk at a time:

- **^:** The start of line anchor. So, our sequence has to be the first thing on a line.
- **([0-9]{4}[-]):** The parentheses gather the search pattern elements into a group. Other operations can be applied to this group as a whole (more on that later). The first element is a character class that contains a range of digits from zero to nine `[0-9]`. Our first character, then, is a digit from zero to nine. Next, we have an interval expression that contains the number four `{4}`. This applies to our first character, which we know will be a digit. Therefore, the first part of the search pattern is now four digits. It can be followed by either a space or a hyphen (`[- ]`) from another character class.
- **{3}:** An interval specifier containing the number three immediately follows the group. It's applied to the whole group, so our search pattern is now four digits, followed by a space or a hyphen, that's repeated three times.
- **[0-9]:** Next, we have another character class that contains a range of digits from zero to nine `[0-9]`. This adds another character to the search pattern, and it can be any digit from zero to nine.
- **{4}:** Another interval expression that contains the number four is applied to the

previous character. This means that character becomes four characters, all of which can be any digit from zero to nine.

- **|:** The alternation operator tells us everything to the left of it is a complete search pattern, and everything to the right is a new search pattern. So, this command is actually searching for either of two search patterns. The first is three groups of four digits, followed by either a space or a hyphen, and then another four digits tacked on.
- **[0-9]:** The second search pattern starts with any digit from zero to nine.
- **{16}:** An interval operator is applied to the first character and converts it to 16 characters, all of which are digits.

So, our search pattern is going to look for either of the following:

- Four groups of four digits, with each group separated by a space or a hyphen (-).
- One group of sixteen digits.

The results are shown below.

```
dave@howtogeek:~$ grep -E '^([0-9]{4}[- ]){3}[0-9]{4}|[0-9]{16}' geeks
.txt
1111-2345-6879-2222
2222 3434 4545 5656
6666333388882222
dave@howtogeek:~$
```

This search pattern is looking for common forms of writing credit card numbers. It's also versatile enough to find different styles, with a single command.

Take It Slow

Complexity is usually just a lot of simplicity bolted together. Once you understand the fundamental building blocks, you can create efficient, powerful utilities, and develop valuable new skills.