`grep` and Regular Expressions

4.3 `grep` and `fgrep`

`grep`, the *generalized regular expressions* command is an incredibly powerful tool to have in your arsenal. `grep` will search through text, either from a file or standard in and attempt to match on your provided patterns. The patterns you provide to `grep` are regular expressions (we've dipped our toes into regular expressions in earlier chapters), but we'll cover them in significantly greater detail in the next section. The command for `grep` looks like:

```
grep [options] [patterns] [files]
```

As we saw briefly in chapter 3's I/O, a simple, yet extremely powerful use of `grep` is to find old commands through your `history` command. Suppose I want to see previously exported variables:

```
history | grep export
```

> 499 export COOLSTUFF="Pizza, my favorite food" 756 export COOLSTUFF='Pizza, my favorite food' 1112 history | grep export

If I wanted to see the count of how many times I had exported, I could use the `-c` flag:

```
history | grep -c export
```

> 4

Note the above shows one more than the above command's output because our executed command had the word export in it as well.

Suppose we wished to extract the data for a specific person from `employeeData.csv`. We'll use the `-i` flag to ensure case insensitivity because we may not be sure how we entered the name:

```
grep -i gwen employeeData.csv
```

> `8,Gwendolen,McIlwrick,gmcilwrick7@sciencedaily.com,Design Engineer,$127821.14`

So far, we've only been searching on strings. If we wanted to only search on strings, we could use the `-F` flag for `fixed strings`. This uses faster algorithms to search through the files / text. You could equivantly just use the **fgrep** command to do the exact same thing.

```
fgrep -i gwen employeeData.csv
```

4.4 / 4.5 Regular Expressions and Use with `grep`

A regular expression, also called *regex*, is a way of writing an abstracted pattern matching string that is used ubiquitously throughout the world of software from bash commands to filters in other languages. So far, we've been mostly using fixed strings to match on, but it's possible to spruce these patterns up to match on so much more!

We've already seen the usage of a *regex* by giving the most basic pattern, that is, a fixed string, to grep, where `export` was the fixed string:

```
history | grep export
```

Regex matching for fixed strings can contain any character (e.g. `a`, `b`, `c`, `1`, `2`, `3`, and even special characters, like `å`, `é`, `ç`, `∫`, and so on), however, if that character is a special character used by the shell, you will need to escape it (`\$`, `\^`, etc.). Depending on the language, however, you may need to do some extra work to find (we won't be going that far into regexes, but regexing unicode characters is easily searchable).

Regex with Fixed Strings:

Regexes with fixed strings are something we've already seen, however, let's take a deeper look at their complexity. Suppose you had a specific substring you wished to search for, like `abc`. Suppose we had a file, `fileWithWords.txt`:

```
A B C
here is a string with abc in it
hereIsAWordWithabcInTheMiddleOfIt
abcIn the front of a word.
This sentence ends withabc

X Y Z
here is x string with xyz in it
hereIsAWordWithxyzInTheMiddleOfIt
xyzIn the front of x word.
This sentence ends withxyz

1 2 3
here is 1 string with 123 in it
hereIsAWordWith123InTheMiddleOfIt
123In the front of 1 word.
This sentence ends with123
```

If this were an incredibly long file and we wished to only retrieve the `abc` lines out of it, we could use the fixed string to search:

```
grep abc fileWithWords.txt
```

> here is a string with abc in it hereIsAWordWithabcInTheMiddleOfIt abcIn the front of a word. This sentence ends withabc

This has retrieved all of the sentences with the string `abc`. Notice that it doesn't matter that `abc` is standing alone, starting a string, ending a string, or nestled inside of a string for it to be returned. Let's try the same thing for `xyz`, using our history expansion from chapter 3:

```
^abc^xyz
```

> grep xyz fileWithWords.txt here is x string with xyz in it hereIsAWordWithxyzInTheMiddleOfIt xyzIn the front of x word. This sentence ends withxyz

And again, with:

```
^xyz^123
```

> grep 123 fileWithWords.txt here is 1 string with 123 in it hereIsAWordWith123InTheMiddleOfIt 123In the front of 1 word. This sentence ends with123

The Wild Cards `.`:

Regexes, however, can be a lot more generic than just typing a fixed string with letters or numbers. We can use periods `.` to act as wild cards to search for things in a file. These do act on a per character basis. Suppose we wished to grab a line from a file which has the string `XXXIn` where the `X`s could be anything. To do that, we could use:

```
grep ...In fileWithWords.txt
```

> hereIsAWordWithabcInTheMiddleOfIt abcIn the front of a word.
> hereIsAWordWithxyzInTheMiddleOfIt xyzIn the front of x word.
> hereIsAWordWith123InTheMiddleOfIt 123In the front of 1 word.

Curiously, though, suppose we wished to grab only the strings that end with a `.`. Given that `.` is a reserved character, we'll need to escape it like `\.`:

```
grep word\. fileWithWords.txt
```

> abcIn the front of a word. xyzIn the front of x word. 123In the front of 1 word.

Begins / Ends With

Often times you'll wish to find a line that starts with a specific pattern, or possibly one that ends with a given pattern. To do that, you'll need to use the `^` and `$` characters respectively:

Begins With

To match a pattern that begins with something, you need to prefix your string or pattern with a carrot:

```
1 grep ^here fileWithWords.txt
```

> here is a string with abc in it hereIsAWordWithabcInTheMiddleOfIt here is x string with xyz in it hereIsAWordWithxyzInTheMiddleOfIt here is 1 string with 123 in it hereIsAWordWith123InTheMiddleOfIt

It doesn't matter if the chosen pattern is part of a word. So long as it begins with the same pattern defined in the command, it will be matched.

Ends With When defining a command to end with a pattern, the `$` character needs to be added as a suffix (not as a prefix!):

```
1 grep word\.$ fileWithWords.txt
```

> abcIn the front of a word. xyzIn the front of x word. 123In the front of 1 word.

Try adding the `$` suffix for a word or pattern that is not at the end of a line, and see what happens!

Specific Characters

Often times files contain many similar lines, but with a handful of differing characters. We can use the same patterns we used back in the expansions to do exactly that with the pattern `[ax1]` where any of the characters within the brackets are patterns to match to:

```
1 grep [cz3]In\ the fileWithWords.txt
```

> abcIn the front of a word. xyzIn the front of x word. 123In the front of 1 word.

Note: Notice that after `In` we have an escaped space? By escaping the space and adding `the` we can more specifically match to things!

Exclude Specific Characters

Just like with the expansions, we can also specify which characters not to match on within the brackets by preceding with a carrot:

```
grep [^z3]In\ the fileWithWords.txt
```

> abcIn the front of a word.

Note: Just like with the other brackets, we can specify ranges to search through as well with `[a-z]`

Digits

Similar to the range of digits for letters, we can also specify ranges of digits to search through, and just like the other uses of brackets, this only works for one specific character:

```
grep [0-9][0-9][0-9] fileWithWords.txt
```

> here is 1 string with 123 in it hereIsAWordWith123InTheMiddleOfIt 123In the front of 1 word.
> This sentence ends with123

While we could have been safe enough by matching on one of the digits, it must be noted that each individual [0-1] is specifically referring to only one character, such that `[0-9][0-9][0-9]` matches to `123`

Repetitions

Note: This is useful to know, however, delmar does not support this syntax:

You might come to a point where you'll need to find a number of characters based on how many times a specific character is repeating. In a real world example, it may be that you're attempting to find a website, prefixed with `www`. Let's quickly add a few lines with repeating characters to our file:

```
A B C
here is a string with abc in it
hereIsAWordWithabcInTheMiddleOfIt
abcIn the front of a word.
This sentence ends withabc
aaa

X Y Z
here is x string with xyz in it
hereIsAWordWithxyzInTheMiddleOfIt
xyzIn the front of x word.
This sentence ends withxyz
aaaaa

```

```
1 2 3
here is 1 string with 123 in it
hereIsAWordWith123InTheMiddleOfIt
123In the front of 1 word.
This sentence ends with123
aaaaaaa
```

Now, it's unlikely that you'll run into this specific scenario, but repeating characters are a possibility. Suppose you wished to match any repeating `a` characters that had 4 or more repeating `a`s:

```
egrep a{4} fileWithWords.txt
```

> aaaaa aaaaaaa

Where the above finds all lines with 4 or more `a`s strung together.

You can additionally define how many repetitions at most there may be, where the final number is the stopping point (inclusive).

```
egrep a{2,5} fileWithWords.txt
```

> aaa aaaaa

You can also define zero or more repetitions with the wild card:

```
grep a* fileWithWords
```

Or One or more repetitions:

```
egrep a+ fileWithWords
```

Note: Delmar does allow for the use of the `*` notation, however, because it matches on 0 to any instances of that character, it will output the entire file with the desired characters highlighted (in this instance).

Optional Characters

Note: As above, the following does not work in delmar, but is useful to know it exists.

You'll find it to be the case that you'll need to match on a things that have potentially optional characters, such as a numerical value being anywhere from 0-999. There are a possible two extra digits that we'll have to search on.

Given a file `money.txt`:

```
CREAM
$23 gas
$2000 new computer
$3 coffee
$475 rent
```

If we wished to get all of the purchases under $1000 from our list, we could use optionals:

```
grep [0-9][0-9]?[0-9]? money.txt
```

> `$23 gas $3 coffee $475 rent`

By placing the `?` as a suffix to the number ranges, we're able to specify that, yes if characters exist after the original `[0-9]` we would like for them to match on `[0-9]`, however, if they don't exist, we'd still like to match on the original `[0-9]`.