Grep command

The third type of command combination is possible through the `|` symbol (which is also known as the "pipe operator" in bash). This is the most frequently used one. As mentioned earlier, this operator redirects the output of one command to the input of another command. Because of this, it's like connecting the two with a pipe (hence the name "pipe operator"). Most often, this operator is used in combination with the `grep` command. It is a very (very, very) useful command, so we will spend a lot of time with it. Essentially, this is what it does: it selects the lines from its input that match a specified pattern. Let's look at some examples! First, let's create a file named `people.csv` with the following content:

```
Name;Birthdate;Phone;Skill
Robert Bob;1997.09.12.;06201975555;IT
Zsuber Driver;1988.10.11.;06304445555;Driving
Hatori Hanso;1966.01.11.;06301234555;Smithing
Rinaldo Orson;2001.05.24.;06709330000;IT
Travis Camel;1970.10.01.;06301717171;Horses
Dagobert McChips;1956.08.31.;06700001111;Cooking
Bumfolt Rupor;1967.09.11.;06201112233;Marketing
```

We will search within this file using the `grep` command.

Example 10
Display the lines where the person's skill is IT:

```
cat people.csv | grep "IT"
```

With the `cat` command, we read the entire `people.csv` file and use the `|` operator to "pipe" it into the `grep` command. Additionally, we specify what we are looking for in the given line: "IT." `grep` will print the lines containing the specified text.

Example 11
Display the lines where the phone number starts with 30:

```
cat people.csv | grep "0630"
```

Example 12
Display the lines where the person was born in or after 2000:

```
cat people.csv | grep ";2"
```

Example 13
Display the lines where the skill ends with "-ing":

```
cat people.csv | grep ";*ing"
```

This is part of the next lesson (regular expressions), but we already know from other studies that the asterisk usually matches everything. Unfortunately, we cannot implement much more complex searches without regular expressions. We will leave these for the next lesson.

Example 14
Create a file named `IT.txt` with the following content:

```
People at IT:
***
```

Where `***` will be replaced by the lines of people working in IT from the `people.csv` file.

```
echo "People at IT:" > IT.txt
cat people.csv | grep "IT" >> IT.txt
```

In the above lines, we combined the stream redirection operator with the pipe operator. We've already seen how they work separately, and combined, they aren't more complicated. You just need to read from left to right to understand what is happening, and then it will be clear.

Tasks
Let's complete the following tasks independently!

1. **Task**
   Collect into a file named `voda.txt` those from `people.csv` who have a phone number starting with 70.
2. **Task**
   Try to enter the folder named `dir_4`, and if successful, create a file named `im_tired.txt` inside it. If unsuccessful, create the folder `dir_4`.
3. **Task**
   Try to list the contents of the folder `dir_5`. If unsuccessful, do not show an error; instead, create the folder `dir_5`.
4. **Task**
   Create the following files in a directory named `pictures`:

```
1.jpg
100.jpg
200.jpg
1.png
10.png
20.png
234.jpg
10.svg
```

List the contents of this folder and write all the `.png` files into a file named `sprites.list`. (Tip: the `cat people.csv` command will now be replaced by the `ls pictures` command.)

Summary
We learned the following commands:

- `cat` prints the content of a file to the standard output.
- `echo` prints the provided text to the standard output.
- `grep` prints the lines matching a specified pattern.

In this lesson, we will take a deeper look at "piping" in Bash and also explore regular expressions.

Let's prepare our work by creating a folder named `lesson4` and placing the `people.csv` file in it, using the content from the previous lesson (the easiest way is to copy it).

```
mkdir szgyak/lesson4
cd szgyak/lesson4
cp ../lesson3/people.csv ./  # copy the file into this directory
```

Chaining pipeline operators

Similar to the `&&` and `||` operators, the `|` (pipeline) operator can also be chained. Generally:

```
command1 | command2 | command3
```

For example, if we want to extract people from the `people.csv` file whose phone numbers start with 70 and who were born in the 1900s, we could do it like this:

```
cat people.csv | grep ";0670" | grep ";19"
```

Illustrated with a diagram, the following happens:

filter_commands

So, the `grep` command "filters" the input before passing it to the output. Commands that behave this way are called "filters" (or filter commands). A filter is a command that reads from input, does something, and then writes the result to the output. Some examples:

- `head`: Outputs the first x lines to the output.
- `tail`: Outputs the last x lines to the output.
- `sort`: Sorts the input and outputs it.
- `grep`: We know this one.
- `wc`: Counts words, characters, lines, and outputs the result.
- `rev`: Reverses the input line by line.
- `uniq`: Checks for uniqueness and frequency in sorted data (e.g., how many times a line appears).
- `cut`: Extracts columns from the data.
- `tr`: Replaces characters in the data.

Let's look at some examples!

1st example

Print the last three lines of the `people.csv` file:

```
cat people.csv | tail --lines=3
# The following command does the same
cat people.csv | tail -3
```

The `tail` command works as a filter, just like `grep`, but performs a different operation. It outputs the number of lines specified after the command name from the end of the file.

2nd example

Print the first three lines of the `people.csv` file, including the header:

```
cat people.csv | head --lines=3
```

The `head` command works similarly to `tail`, but instead of the last lines, it outputs the first lines.

3rd example

Print the last row from the `people.csv` file where the phone number starts with 30:

```
cat people.csv | grep ";0630" | tail -1
# Or:
cat people.csv | grep ";0630" | tail --lines=1
```

The above command is easy to understand if we know that we can chain multiple commands through a pipeline.

4th example

Print all lines from the `people.csv` file except the last five:

```
cat people.csv | head --lines=-4
# Or this command does the same
cat people.csv | head
```

If we provide a negative number after `--lines`, it doesn't print the first n lines, but excludes the last n lines. `tail` behaves similarly, but with a slight twist that we'll see in the next example.

5th example

Print the first three lines from the `people.csv` file without the header:

```
cat people.csv | tail --lines=+2 | head --lines=3
# Or shorter:
cat people.csv | tail +2 | head -3
```

In this case, `tail` starts printing from the nth line, where the number is prefixed by a `+` (instead of a minus as with `head`).

6th example

Print the first 10 files or directories in the `/dev` folder, even if they are hidden! The special directories `.` and `..` should not appear:

```
ls -la /dev | tail +4 | head -10
```

Here, we've cut off the first three lines of the `ls -la` command (which includes the `total number of disk blocks` line and the two special directory lines), and then kept the first 10 of the remaining lines.

This example is closer to what we use Bash for in practice. It's easy to demonstrate how these commands work on a CSV file, but in practice, we often work with the output of commands, not just file contents. These situations arise when we use command line tools that list things for us. Currently, we don't really know such tools, and we won't, until they come up in a real or university project.

Nevertheless, I would mention, for example, the `docker` tool. This manages virtual machines and, with the right switches, prints the currently running virtual machines, their names, and their IDs. In this case, `grep` comes in handy to find the one with the correct name. If we're really command line wizards, we can also extract the ID next to the name using Bash, and we can even shut down the machine. This would involve tedious searching, selecting, copying, and pasting, but if we create a script for it, we can solve it with a single command.

One of the big advantages of Bash is that it allows us to automate frequently used command line operations.

7th example

Print the last 5 disks in the `/dev` folder (for now, let's consider any files whose owner group is `disk` as disks):

```
ls -la /dev | grep " disk " | tail -5
```

8th example

Print only the names of the people from the `people.csv` file (without the header):

```
cat people.csv | tail +2 | cut --field=1 --delimiter=';'
# Or shorter:
cat people.csv | tail +2 | cut -f 1 -d ';'
```

The last element of this command, the `cut` command, needs some explanation. It reads the appropriate columns from a text that consists of columns. A column can be byte-sized or

character-sized, but we need fields now. The `--fields=1` switch specifies that we want to read the first field from the input text. The fields, however, need to be separated by something in the file, known as the delimiter. The `--delimiter=';'` switch tells the command that the semicolon is the field separator in the input text (in our case).

9th example

Print only the phone numbers from the `people.csv` file, and only the ones starting with `0620`:

```
cat people.csv | grep ";0620" | cut -f 3 -d ';' | head -1
```

10th example

Print the names from the `people.csv` file in alphabetical order:

```
cat people.csv | tail +2 | cut -f 1 -d ';' | sort
```

11th example

Print the birth months from the `people.csv` file (only the months):

```
cat people.csv | cut -f 2 -d ';' | cut -f 2 -d '.'
```

At this point, we can start to feel how far we can go by combining these commands. There are many commands, each with its own switches, and they work similarly to the ones we've seen so far. Instead of diving into more commands, let's move on to regular expressions, which will also be useful for us!

Regular Expressions

At university, one of my professors once said: "In computer science, the two hardest things are caching and regular expressions."

Regular expressions are used for text matching tasks. A regular expression is a pattern that, with the help of a "regular expression engine," can unambiguously determine whether a given text matches the expression. It's easier to understand with examples: below are some regular expressions and the kinds of text they match:

- `[0-9]{3}`: Matches any three-digit number.
- `^[A-Z][a-z]+`: Matches any word that starts with a capital letter.

We will learn about some building blocks of regular expressions. I'll show a brief summary in this note, but the topic is much broader, and the internet contains a wealth of information on it. Regular expressions are widely used, for example, for validating the format of form fields on a website (to specify what kind of text a field can contain), or for searches (to specify what kind of text we are looking for). It's a useful but complex topic, and we won't dive too deeply into it.

We will continue to practice on the content of the `people.csv` file, but I suggest that, just this once, we step away from the terminal, because it will only lead to endless frustration.

However, there's one more thing I want to highlight regarding Bash commands: we've seen that during the lesson, I used both single and double quotes. This has practical significance because single quotes contain only text, while double quotes can contain variables that need to be interpreted. We will cover this in more detail in the next lesson, but for now, just remember that it's good practice to write regular expressions between single quotes!

Let's add new content to the `people.csv` file to make the task more complex and justify the use of regular expressions! The new content:

```
Name;Birthdate;Phone;Email;Skill
Robert Bob;1997.09.12.;06201975555;bob2bob@gmail.com;IT
Zsuber Driver;1988.10.11.;06404445555;uber@gmail.com;Driving
Hatori Hanso;1966.01.11.;063012345;jean'dark@gmail.com;Smithing
Rinaldo Orson;2001.05.24.;06709330000;D&D@gmail.com;IT,Weight lifting
Travis Rick Camel;1970.10.01.;06301717171;mummy@citromail.hu;Horses
Dagobert McChips;1956.08.31.;067001111;clap.clapgmail.com;Cooking,Catering
Bumfolt Rupor;1967.09.11.;06201112233;HP123@mg.mh.hu;Marketing
Mad Marx;1965.09.11.;06301442233;money.bad@gmail.com;Motorcycle Riding
Borland Ci Rattleneck;1980.11.02;+36304225555;teller@citromail.hu;Bottleneck
Construction
Baileys Margareti;1999.12.09;06209876543;b1gdr1nk@HOND.hu;Drinking,Catering
Larry Lipstick;1988.07.22;+36206547382;baby@gmail.com;Hairstyling
Inigo Montoya;1977.04.11;+367065682;pr3pare@gmail.to;Swordfighting
```

On the website https://regex101.com, you can try out regular expressions interactively and visually. The following screenshot illustrates what you should see on the webpage. Paste the content of `people.csv` into the largest textbox in the middle.

Regex

In the top-right corner, you'll find English explanations of the expressions you type. At the top center, you can input regular expressions to search within the text. In the bottom-right corner, there is a short reminder of the most common regular expressions. I'll list a few here, and then we'll look at examples!

- `.` matches any single character.
- `?` means the preceding character can appear 1 or 0 times (for example, `blabla?` matches both "blabla" and "blabl").
- `*` means the preceding character can appear any number of times, including 0 (for example, `blabla*` matches "blabla", "blablaaaaaaaa", or "blabl").
- `+` means the preceding character must appear 1 or more times (for example, `blabla+` matches "blabla" or "blablaaaaa", but not "blabl").
- `[]` matches any one character from the set (for example, `[abc]`), or a character range (for example, `[a-z]`).
- `[^]` negates the character set (matches any character except the ones in the set).

- **`{}`** specifies the number of times the preceding character or group must appear. For example, `{2,4}` means at least 2 and at most 4 occurrences. `{N,}` means N or more times, and `{N}` means exactly N times. The special characters `?`, `*`, and `+` can also be expressed with `{}`: `{0,1}` = `?`, `{0,}` = `*`, `{1,}` = `+`.
- **`^`** matches the start of a line.
- **`$`** matches the end of a line.

These are the most commonly used elements in regular expressions, and you won't need anything else for this course.

Using Regular Expressions with `grep`

We will use regular expressions with the `grep` command. To ensure `grep` interprets regular expressions, we need to pass the `-E` flag (for Expression) followed by the regular expression in single quotes. Before running the `grep` command, make sure your regular expression works as expected by testing it on the website mentioned above.

Example 12:

List people from `people.csv` whose first names start with "R":

```
cat people.csv | grep -E '^R'
```

The `^` symbol marks the start of the line, so this expression searches for lines that start with "R."

Example 13:

List people from `people.csv` whose last names start with "R":

```
cat people.csv | grep -E '^[a-zA-Z ]+R[a-z]+;'
```

This expression matches any uppercase "R" at the beginning of a word, preceded by only letters and spaces, before the first semicolon.

Example 14:

List people from `people.csv` who have at least two skills:

```
cat people.csv | grep -E ',[a-zA-Z ]+$'
```

This expression looks for a comma before the last word in the line.

Example 15:

List people from `people.csv` who have three names:

```
cat people.csv | grep -E '^[a-zA-Z]+ [a-zA-Z]+ [a-zA-Z]+;'
```

Example 16:

List people from `people.csv` whose email addresses are valid. (A valid email address is defined here as one where the username contains only letters, numbers, or periods, followed by `@`, and then a group name consisting of lowercase letters, separated by a period from a 2- to 4-character domain. The username should be between 3 and 100 characters long, and the group name between 3 and 20 characters.)

```
cat people.csv | grep -E ';[0-zA-Z0-9.]{3,100}@[a-z]{3,20}\.[a-z]{2,4};'
```

Notice the use of the backslash (`\`) before the period (`.`) in the domain name. Since `.` has a special meaning in regular expressions, we need to escape it with a backslash if we want to use it as a normal character. This is also true for other special characters (`?`, `+`, `{}`, etc.).

Notes:

In previous examples, we used the `cat` command to list the content of a file. This was helpful to learn the command and demonstrate piping, but in the examples above, `cat` can be omitted. The file name can be passed as the last parameter to the `grep` command. The same applies to other filtering commands. For example, the following command is valid:

```
head -1 people.csv
```

However, this method cannot be used to work on the output of commands like `ls`. (Later, we will learn how to write loops to iterate through the contents of a directory.) So the following would not work:

```
head -1 ls
```

Exercises:

Let’s practice with a few independent exercises! Use the updated content of `people.csv` for the tasks below.

1. **Task:** List people who have only one skill, sorted alphabetically by name.
2. **Task:** List the names and phone numbers of people whose name ends with an "o."
3. **Task:** List the first names of people whose phone number starts with `+36` and is valid (consisting of `+36`, followed by 9 digits, and the first two digits are either `20`, `30`, or `70`).
4. **Task:** List the last names of people whose phone number starts with `06` and is valid.
5. **Task:** List the names of people who were born on an even day.