Text Processing

Part 1

Session 5

Agenda:

- **Intro to Text Processing**
- **Basic Text Manipulation commands:**
  - **“cut” command**
  - **“sort” command**
  - **“uniq” command**
- **Searching with “grep” command**
- **Regex and Pattern matching:**
  - **What is Regex?**
  - **BRE syntax**
  - ERR syntax (how it differs from BRE)
  - PCRE syntax and usage
- “sed”
- “awk”

We’ll stop here for now

Introduction to Text Processing

What is Text processing?

- Text processing is the automated manipulation of text data using tools or scripts, usually to extract, transform, or analyze information.

- It makes managing big files like logs or configs a lot easier, saves time, and avoids mistakes

Why Text Processing is Crucial ?

Efficiency

Efficient text processing helps in extracting valuable information quickly.

Data Analysis

Processing logs and configuration files aids in monitoring troubleshooting, and performance tuning

Manipulation

Modifying, rearranging, or cleaning text to prepare it for further use or analysis

Basic text manipulation commands

There are many commands that help in text manipulation

But we'll only mention some of them

`cut` `sort` `uniq`

“cut” command

“cut” command

- A powerful text processing utility that extracts specific sections from each line of files.
- Perfect for working with delimited text files, log files, and formatted output from other commands.
- Basic syntax :

  `cut [option] [file]` (options are necessary)

Options

`-f`	To extract fields/columns
`-d`	To specify a specific delimiter other than the default (Tab)

```
sheikhwalter ~/OSC/Text-processing
❯❯ cut -f 1,3 grades_tab.txt
Name    Physics
Alice   92
Bob     84
Charlie 91
David   95
Eve     55
```

```
sheikhwalter ~/OSC/Text-processing
❯❯ cut -d ',' -f 1,2,4 grades.csv
Name,Math,Chemistry
Alice,88,86
Bob,81,76
Charlie,61,75
Diana,67,100
Ethan,75,87
```

Options

`-f` ***`n,k`***	Extract the ***n***th and the ***k***th fields
`-f` ***`-n`***	Extract from the start to the ***n***th field
`-f` ***`n-`***	Extract from the ***n***th field to the end
`-f` ***`n-k`***	Extract from the ***n***th field to the ***k***th field

Works with -b and -c in the same way

Options

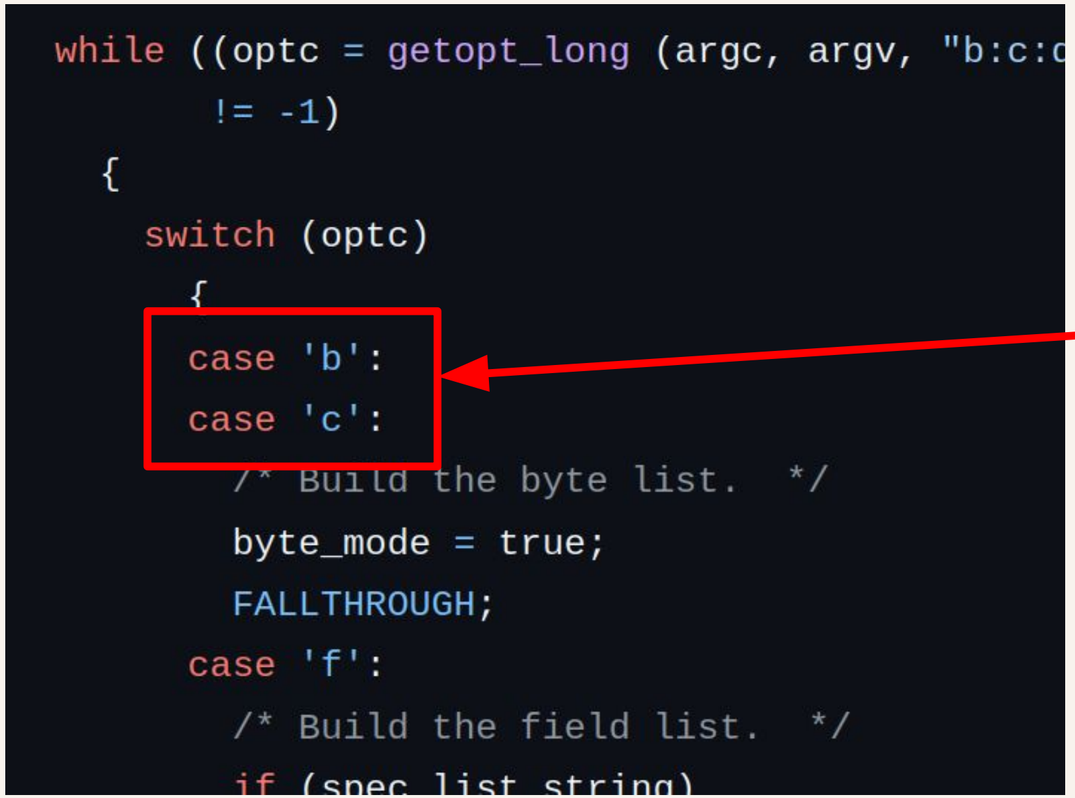
`-b`	To extract specific bytes
`-c`	To extract specific characters

```
sheikhwalter ~/OSC/Text-processing
❯❯ cut -b 1-4 baby
waaa
gogo
gaga
ab32
ab5d
```

```
sheikhwalter ~/OSC/Text-processing
❯❯ cut -c 1-4 baby
waaa
gogo
gaga
ab32
ab5d
```

Options

`-b`	To extract specific bytes
`-c`	To extract specific characters



```
while ((optc = getopt_long (argc, argv, "b:c:d
        != -1)
  {
    switch (optc)
      {
      case 'b':
      case 'c':
        /* Build the byte list.  */
        byte_mode = true;
        FALLTHROUGH;
      case 'f':
        /* Build the field list.  */
        if (spec_list_string)
```

Actually, there's no difference between them (for now at least)

“sort” command

“sort” command

- The **sort** command is a powerful text processing utility that arranges lines of text files in alphabetical or numerical order.
- It supports various options for customizing sort behavior, handling different data types, and processing complex datasets.
- **Basic syntax :**

```
sort [option] [file]
```

By default, the **sort** command:

- Sorts lines alphabetically
- Is case-sensitive (uppercase before lowercase)
- Uses the entire line for sorting

```
$ sort fruits.txt
Banana
Grape
apple
kivi
orange
```

Options

`-n`	Numeric sort (treats multi-digit numbers as numbers not strings)
`-h`	Human-readable numeric sort (2K, 1M)
`-r`	Reverse order
`-f`	Case-insensitive sort
`-u`	Remove duplicates
`-k` ***n***	Sort by the ***n***th column
`-t`	Specify field separator (default is Tab)
`-c`	Check if file is sorted

Examples

Case-Insensitive Sort

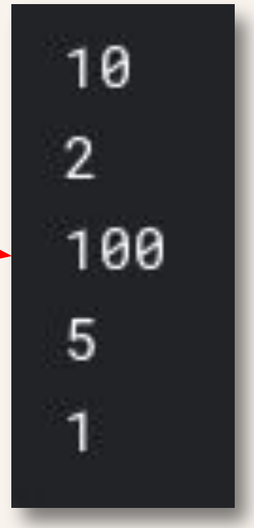
```
$ sort fruits.txt
Banana
Grape
apple
kivi
orange
```

```
$ sort -f fruits.txt
apple
Banana
Grape
kivi
orange
```

Examples

Sort numeric values

numbers.txt



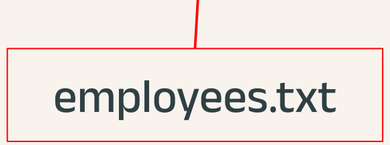

```
$ sort -n numbers.txt
1
2
5
10
100
```

Examples

Sort by a specific column

```
$ sort -k2 -n employees.txt
Alice 28 Designer
Eva 31 Engineer
John 35 Developer
Mike 39 Analyst
Bob 42 Manager
```

```
John 35 Developer
Alice 28 Designer
Eva 31 Engineer
```



Examples

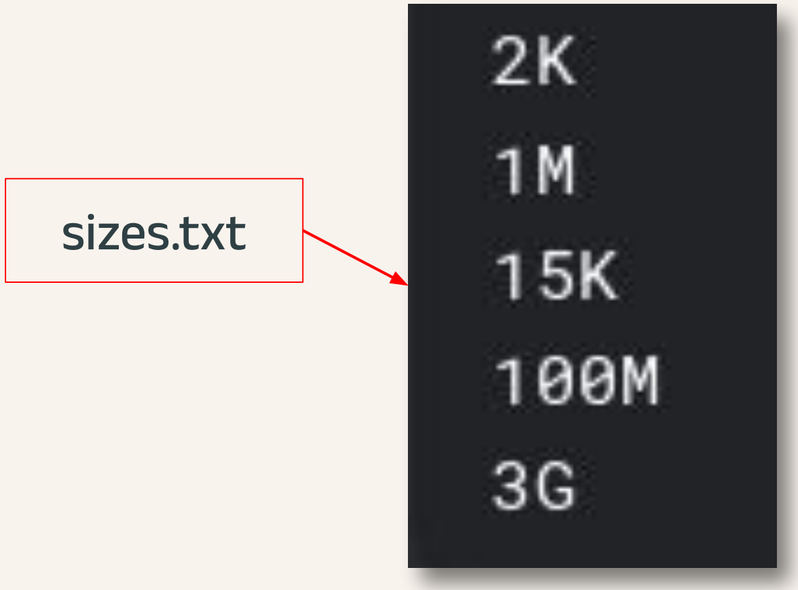
Output sorted values without duplicates

```
$ sort -u colors.txt
blue
green
orange
red
yellow
```

Examples

Human-Readable Sort

```
$ sort -h sizes.txt
2K
15K
1M
100M
3G
```



Examples

Sort with custom delimiter

```
❯❯ sort -t ',' -k 2 grades.csv
Charlie,61,50,75,95,88
Diana,67,60,100,58,50
Ethan,75,74,87,62,57
Bob,81,68,76,88,98
Alice,88,56,86,87,72
```

Check if the file is sorted

```
sort -c file.txt
# Outputs an error message if not sorted
# Nothing is shown if sorted
```

“uniq” command

“uniq” command

- It removes any duplicate lines and sends the results to standard output.
- It is often used in conjunction with sort to clean the output of duplicates.
- **Basic syntax:**

```
uniq [option] [file]
```

- By default , output will be all lines without duplication

Options

`-c`	Print each output line with the number of occurrences
`-d`	Display only duplicate lines
`-u`	Display only unique lines
`-i`	Ignore case sensitive

Searching with “grep” command

“grep” command

- “Grep” stands for ‘Global Regular expression print’.
- Used to search files for the occurrence of a string of characters that matches a specified pattern
- Command syntax:

```
grep [option] [text/pattern] [file]
```

Options

`-n`	Print the number of each line that contain the matching pattern
`-c`	Count matching lines only
`-o`	Print **only** the matched part (not the whole line)
`-i`	Ignore case sensitive
`-v`	Print all lines that doesn't match text
`-l`	Print name of each file that contain match text
`-L`	Print name of each file that **doesn’t** contain match text
`-r`	Recursive search in files inside a directory (prints matches with file paths)

Options for context control

`-A` ***N***	Print ***N*** lines **After** the match.
`-B` ***N***	Print ***N*** lines **Before** the match.
`-C` ***N***	Print ***N*** lines **Before and After** the match.

Hands on

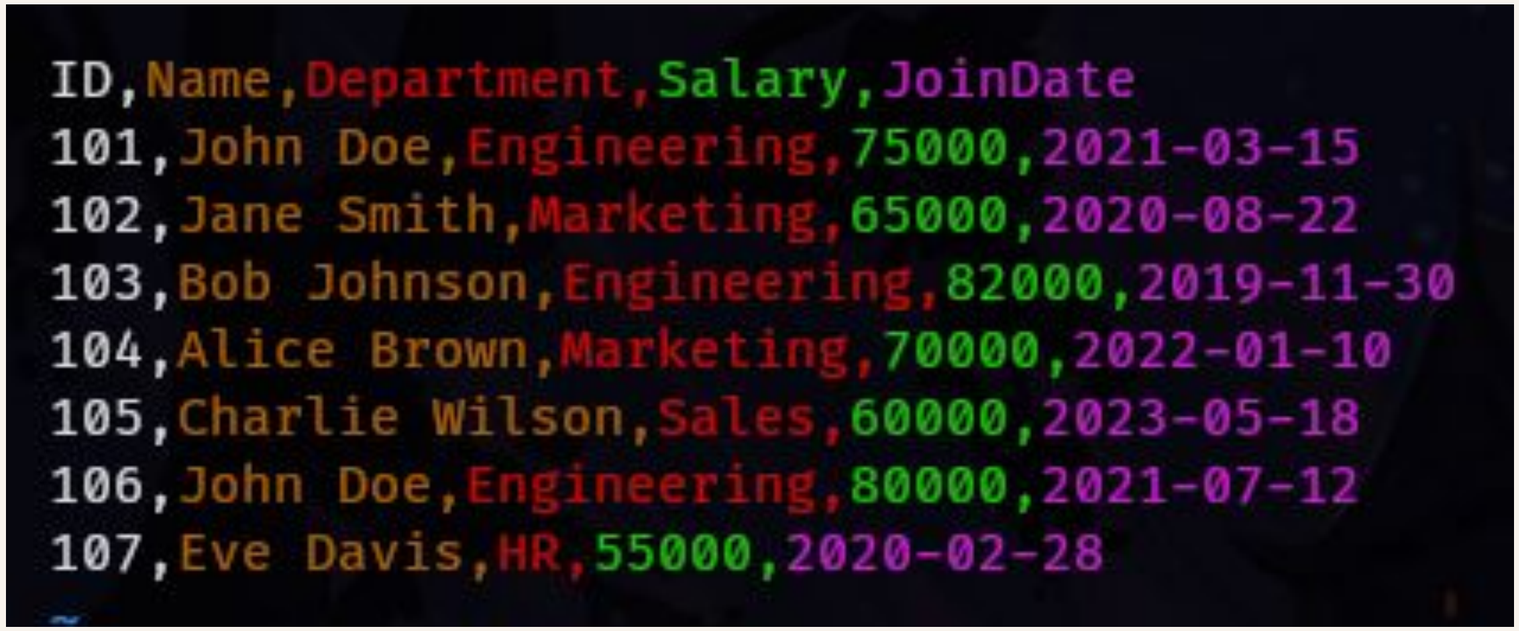
1

Display all unique departments in employees.csv

(Don’t include the first row where the word “Department” lies)

Solution:

```
cut -d ',' -f 3 employees.csv | sort | uniq | grep -v Department
```



Let’s take a Break!

Text can be written in 3 ways:

1. **Text only without quotes**

- Good for simple patterns without special characters or whitespaces
- Lets the shell interpret the special characters then send it to grep
- Words separated by spaces become separate arguments
- Can’t safely reference variables in scripts if they contain spaces or special characters

```
grep text filename
```

Text can be written in 3 ways:

2. **Text between double quotes (“ “)**

- Same as without quotes but keeps the string as one argument, even with spaces
- Allows referencing variables in scripts
- Some backslashes and characters (like \n, \\) are interpreted by the shell
- Can cause issues when using regex

```
grep “text” filename
```

Text can be written in 3 ways:

3. **Text between single quotes (‘ ‘)**

- Handles the whole text as a literal string with no shell interpretation and sends it to grep to do its tricks
- Safest when using regex, backslashes, or special characters
- Can’t use variables in scripts

```
grep ‘text’ filename
```

But wait.

What does Regex even mean?

Regex and Pattern Matching

Regular Expressions

- Regular Expressions (often called **Regex** or **Regexp**) are sequences of characters used to search, find, and manipulate text using patterns.
- They are powerful tools used in Linux programs like `grep`, `bash`, `sed`, and many programming languages.

Regex comes in a few versions, main ones are:

1. **BRE**: Basic Regular Expressions
2. **ERE**: Extended Regular Expressions
3. **PCRE**: Perl-Compatible Regular Expressions

Regular Expressions

The default version in tools like `grep` and `sed` is **BRE**, which is stricter as many characters need to be escaped.

However, `grep` gives you the option to use **ERE** which is easier to write or **PCRE** for more advanced features like lookaheads.

`grep -E`	For Extended Regular Expressions (**ERE**)
`grep -P`	For Perl-Compatible Regular Expressions (**PCRE**)

Basic **R**egular **E**xpression

Types of Special characters

Metacharacters that are special by default:	Characters that become special when escaped by backslash \:
• . • * • ^ • $ • [ ]	• | • \+ • \? • \{ \} • \(\)

Important note:

All characters that are special by default can be made literal by escaping them with backslash \, even the backslash itself.

```
sheikhwalter ~/OSC/session
>> grep 'th\\s' message
Don't screw th\s up

sheikhwalter ~/OSC/session
>> grep '2\*2' message
you know that 2*2 is 4
```

Usages of Special characters

1. Anchors:

It matches according to the position of the pattern in the line, not the characters only

^	Matches lines starting with the pattern
$	Matches lines ending with the pattern

```
sheikhwalter ~/OSC/session
›› grep '^Khayat' message
Khayat went to the college today, Khayat is my friend!

sheikhwalter ~/OSC/session
›› grep 'end$' message
He's my friend to the end of time, I don't want our friendship to end
```

Usages of Special characters

2. Repetitions

.	matches any single character
?	matches zero or one of the previous item
*	matches zero or more of the previous item
+	matches one or more of the previous item

```
sheikhwalter ~/OSC/session
❯❯ grep 'H.ts' words
Hats Hits Huts

sheikhwalter ~/OSC/session
❯❯ grep 'jars\?' words
jar jars jarssss

sheikhwalter ~/OSC/session
❯❯ grep 'jars*' words
jar jars jarssssss

sheikhwalter ~/OSC/session
❯❯ grep 'jars\+' words
jar jars jarssssss
```

Usages of Special characters

2. Repetitions

`\{n\}`	matches exactly **n** times of the previous item
`\{n,\}`	matches at least **n** times of the previous item
`\{,m\}`	matches at most **m** times of the previous item
`\{n,m\}`	matches from **n** to **m** times of the previous item

```
sheikhwalter ~/OSC/session
›› grep 'b\{2\}' words
bbbb
bbb
bb

sheikhwalter ~/OSC/session
›› grep -o 'b\{2\}' words
bb
bb
bb
bb
```

Shows that the pattern was found 4 times not 3 as it seems in the first command

Usages of Special characters

2. Repetitions

At least 2 “b”s

At most 2 “b”s

More than 2 and less than 3 “b”s

```
sheikhwalter ~/OSC/session
>> grep -o 'b\{2,\}' words
bbbb
bbb
bb

sheikhwalter ~/OSC/session
>> grep -o 'b\{,2\}' words
b
bb
bb
bb
b
bb

sheikhwalter ~/OSC/session
>> grep -o 'b\{2,3\}' words
bbb
bbb
bb
```

Usages of Special characters

Alternation \|

Alternation lets you match **one pattern OR another** using | character, which is useful for full patterns

```
sheikhwalter ~/OSC/session
›› grep 'Apple\|Banana' words
Apple
Banana
```

But what if we only want to match **one character out of several**, like one of the 10 first alphabet letters?

```
grep ‘a\|b\|c\|d\|e\|f\|g\|h\|i\|j’ filename
```

???

That would be really tedious ☹

That’s where **character classes** come in

Usages of Special characters

4. Character classes

Character classes let you match any single character from a set without writing multiple full patterns
We write the set we want between **square brackets [ ]**

Type	Example	Syntax
Any of a set	Match a, b, or c	`[abc]`
A range	Match numbers between 1 and 8	`[1-8]`
Negation	Match anything except a, b, or c	`[^abc]`

Usages of Special characters classes

Match any line ending with s or e

Match any line not ending with s or e

Match all numbers from 2 to 4

```
sheikhwalter ~/OSC/session
›› grep '[se]$' words
jar jars jarssss
Apple

sheikhwalter ~/OSC/session
›› grep '[^se]$' words
b
bbbb
bbb
bb
watch
Banana
1 2 3 4 5

sheikhwalter ~/OSC/session
›› grep '[2-4]' words
1 2 3 4 5
```

Usages of Special characters

4. Character classes

POSIX Character Classes: These are special sets that work inside brackets

`[[:digit:]]`	Digits (numbers)
`[[:lower:]]`	Lowercase letters
`[[:upper:]]`	Uppercase letters
`[[:alpha:]]`	All alphabetic letters
`[[:alnum:]]`	Alphanumeric characters (letters and numbers)
`[[:punct:]]`	Punctuation characters
`[[:print:]]`	All printable Characters including alnum, punct and whitespace (doesn't include characters like \n, \t, \b, etc…)
`[[:space:]]`	All space characters including whitespace, \n, \t, \b, etc…

Usages of Special characters classes

Numbers only

Alphabetic characters only

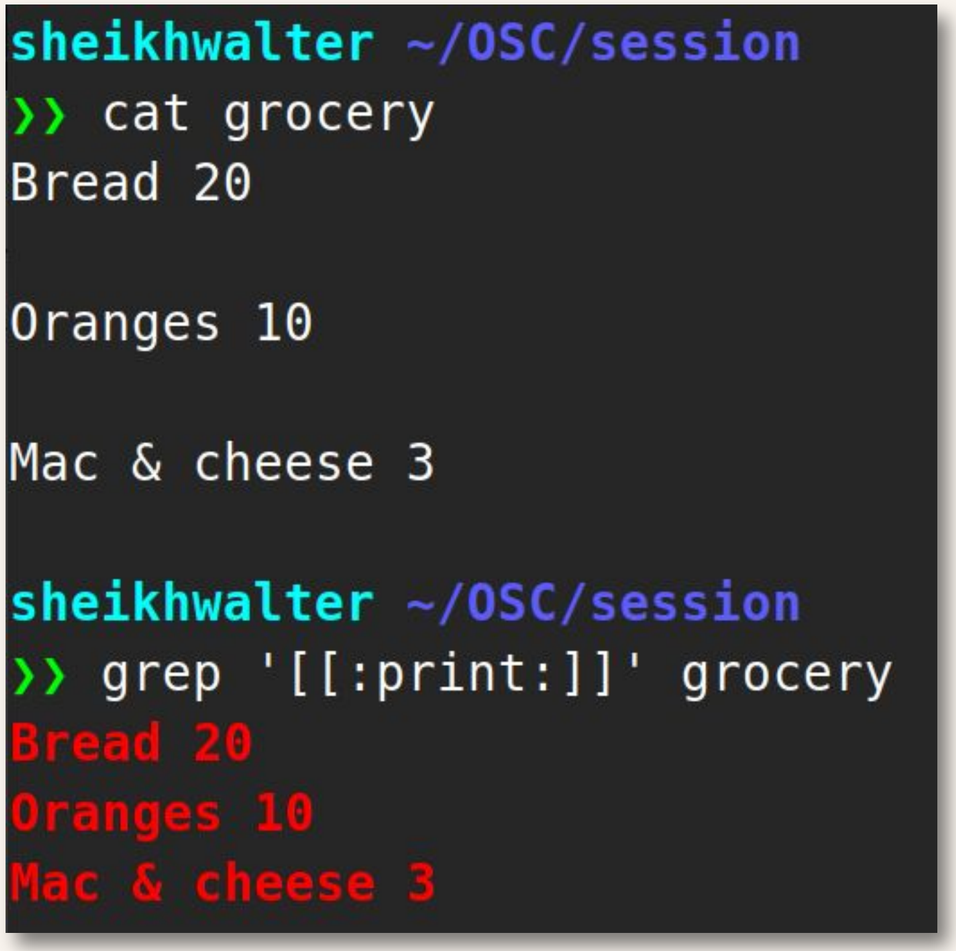
Punctuation characters only

```
sheikhwalter ~/OSC/session
›› grep '[[:digit:]]' grocery
Bread 20
Oranges 10
Mac & cheese 3

sheikhwalter ~/OSC/session
›› grep '[[:alpha:]]' grocery
Bread 20
Oranges 10
Mac & cheese 3

sheikhwalter ~/OSC/session
›› grep '[[:punct:]]' grocery
Mac & cheese 3
```

Usages of Special characters

4. Character classes

It didn't include the new line character (`\n`)



Usages of Special characters

3. Grouping and Capturing

Grouping lets you treat part of a pattern as a single unit which is very useful in many scenarios.
It works by writing the grouped pattern between **escaped braces \(\)**

```
grep '\(pattern\)' filename
```

Usages of Special characters

3. Grouping and Capturing

Grouping allows you to:

1. Apply repetition to a group

   Each group is treated as a single entity which makes applying repetition a lot easier

```
sheikhwalter ~/OSC/session
❯❯ grep '\(waaa\)\{3\}' baby
waaawaaawaaa

sheikhwalter ~/OSC/session
❯❯ grep '\(go\)\{3\}' baby
gogogo

sheikhwalter ~/OSC/session
❯❯ grep '\(ga\)\{3\}' baby
gagaga
```

Usages of Special characters

3. Grouping and Capturing

Grouping allows you to:

2. Use alternation with sub-patterns

```
sheikhwalter ~/OSC/session
›› grep '\(Cat\|Dog\)' animals
CatDog
DogLionCat
```

Matches either Cat or Dog.

Usages of Special characters

3. Grouping and Capturing

Grouping allows you to:

3. Capture part of a match for extraction or replacement

This is useful in grep if the captured group is used in another place in the pattern.

Suppose we have a patten like this: ab.....abab

We can capture the group from the beginning and reuse it later on like this:

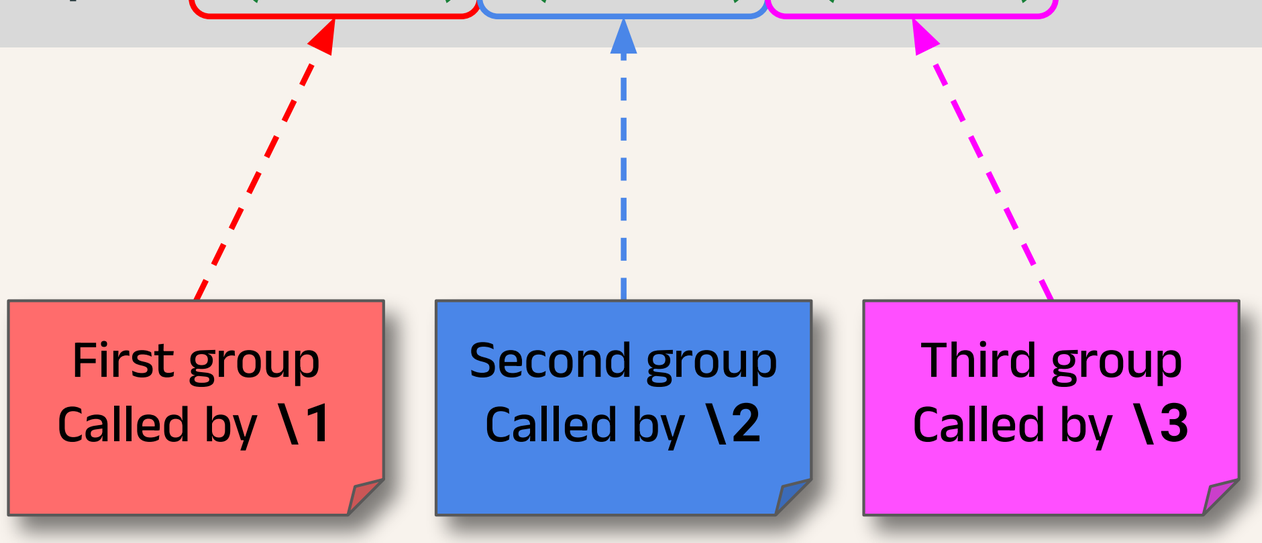
```
sheikhwalter ~/OSC/session
›› grep '\(ab\).....\1\{2\}' baby
ab321f3abab
ab5d8v3abab
```

Usages of Special characters

Grouping and Capturing

How capturing works?

```
grep '\(123\)\(456\)\(abc\)' filename
```



Hands on

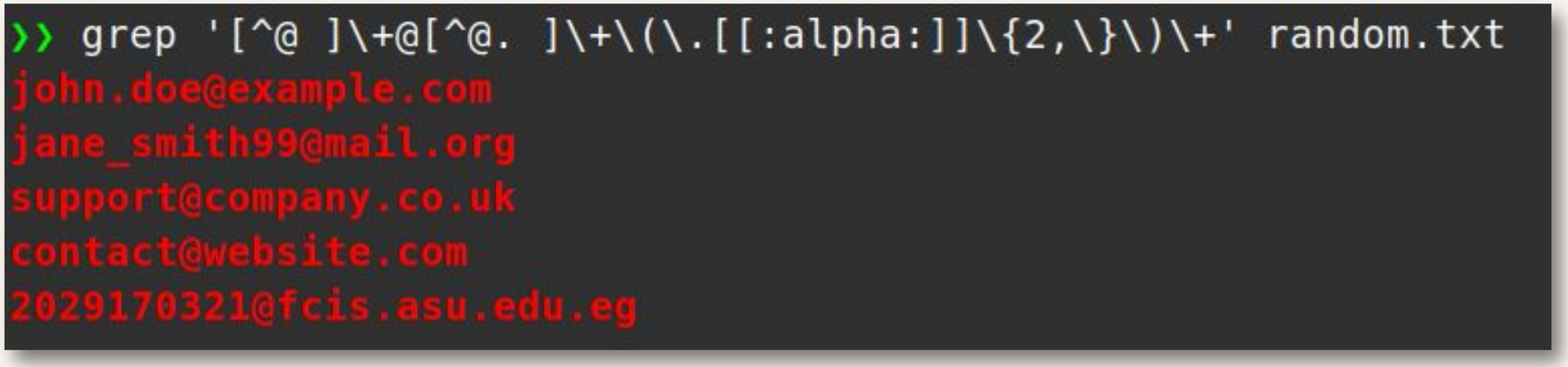
2

Search in “random.txt” for emails

Example emails: john@example.com
corp@dummy.co.uk
200017001@cis.asu.edu.eg

Solution:

```
grep '[^@ ]\+@[^@. ]\+\(\.[[:alpha:]]\{2,\}\)\+' random.txt
```



That’s it

-for now-