Data Manipulation & Bash Scripting

Sevilay Güleşen

September 12th, 2021

Redirection Input - Output Symbols

> : To output Linux-command result into the file (If file already exists, it'll be overwritten instead of creating new file.)

```
ls > myfiles.txt
```

>> : To output Linux-command result to end of the file (If file already exists, it'll be opened and results of command will be written at the end of the file.)

```
ls >> myfiles.txt
```

< : To take input to Linux-command from the file.

```
cat <  myfiles.txt
```

| : (Pipe operator) To connect the output of one command to the input of next command

```
ls -lr | head
```

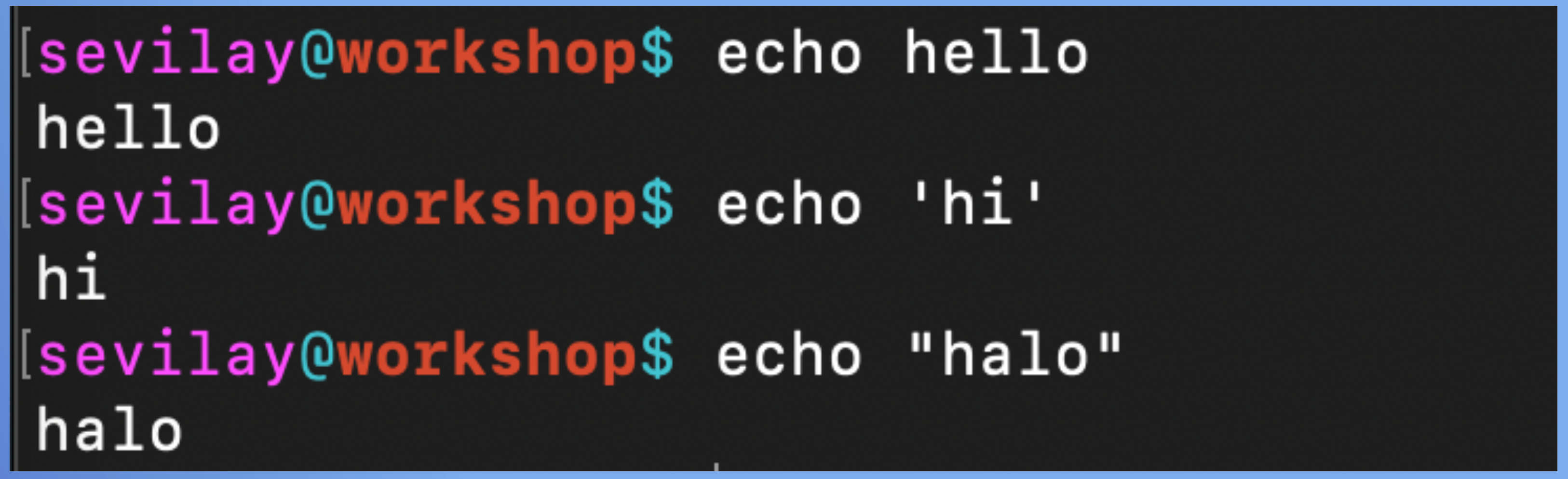
$: To get named variable

```
echo $SHELL
```

User Defined Variables

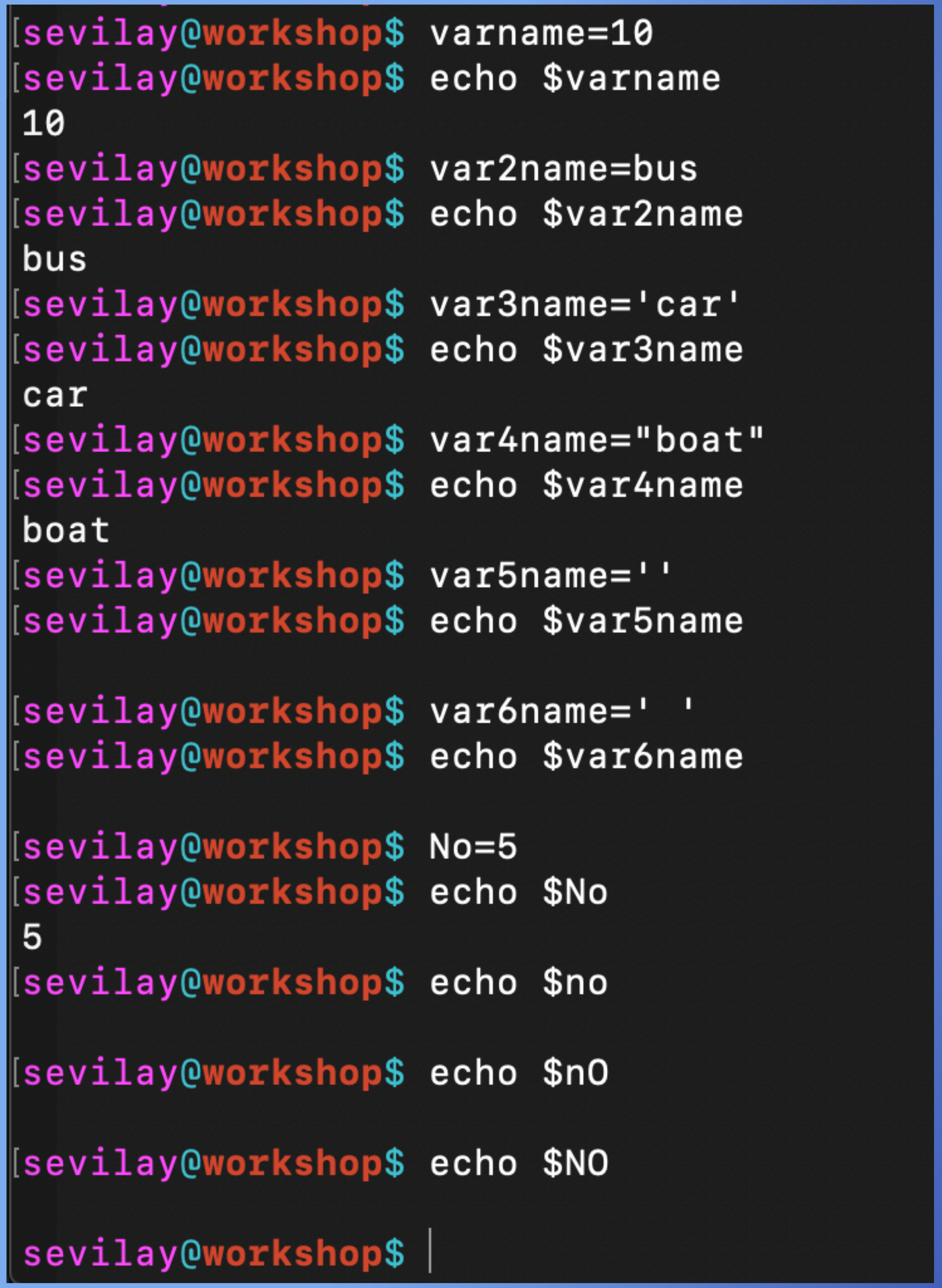
- **echo** is used to print something or user defined variable in bash screen

```
echo hello
echo 'hi'
echo "halo"
```



```
sevilay@workshop$ echo hello
hello
sevilay@workshop$ echo 'hi'
hi
sevilay@workshop$ echo "halo"
halo
```

```
varname=variable
echo $varname
```

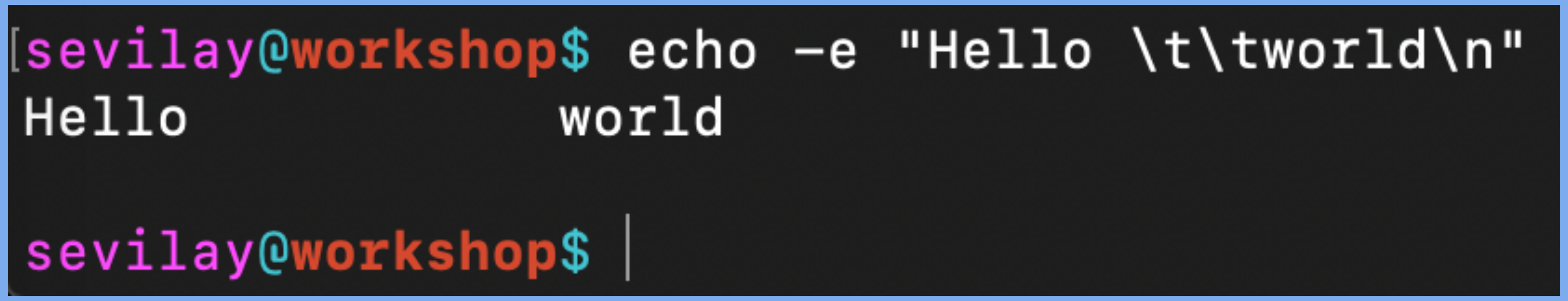


```
sevilay@workshop$ varname=10
sevilay@workshop$ echo $varname
10
sevilay@workshop$ var2name=bus
sevilay@workshop$ echo $var2name
bus
sevilay@workshop$ var3name='car'
sevilay@workshop$ echo $var3name
car
sevilay@workshop$ var4name="boat"
sevilay@workshop$ echo $var4name
boat
sevilay@workshop$ var5name=''
sevilay@workshop$ echo $var5name

sevilay@workshop$ var6name=' '
sevilay@workshop$ echo $var6name

sevilay@workshop$ No=5
sevilay@workshop$ echo $No
5
sevilay@workshop$ echo $no

sevilay@workshop$ echo $nO

sevilay@workshop$ echo $NO

sevilay@workshop$
```

User Defined Variables

- Some of **echo** options
  - **-n** : do not output the trailing new line
  - **-e** : enable interpretation of the following backslash escaped characters in the string:
    - **\b :** backspace
    - **\c :** suppress trailing new line
    - **\n :** add new line
    - **\t :** add TAB

```
echo -e "Hello \t\tworld\n"
```



```
[sevilay@workshop$ echo -e "Hello \t\tworld\n"
Hello           world

sevilay@workshop$ |
```

User Defined Variables

- **expr** is used to do calculation

```
expr 5 + 2
expr 5 \* 3
expr 20 % 3
```

+ addition

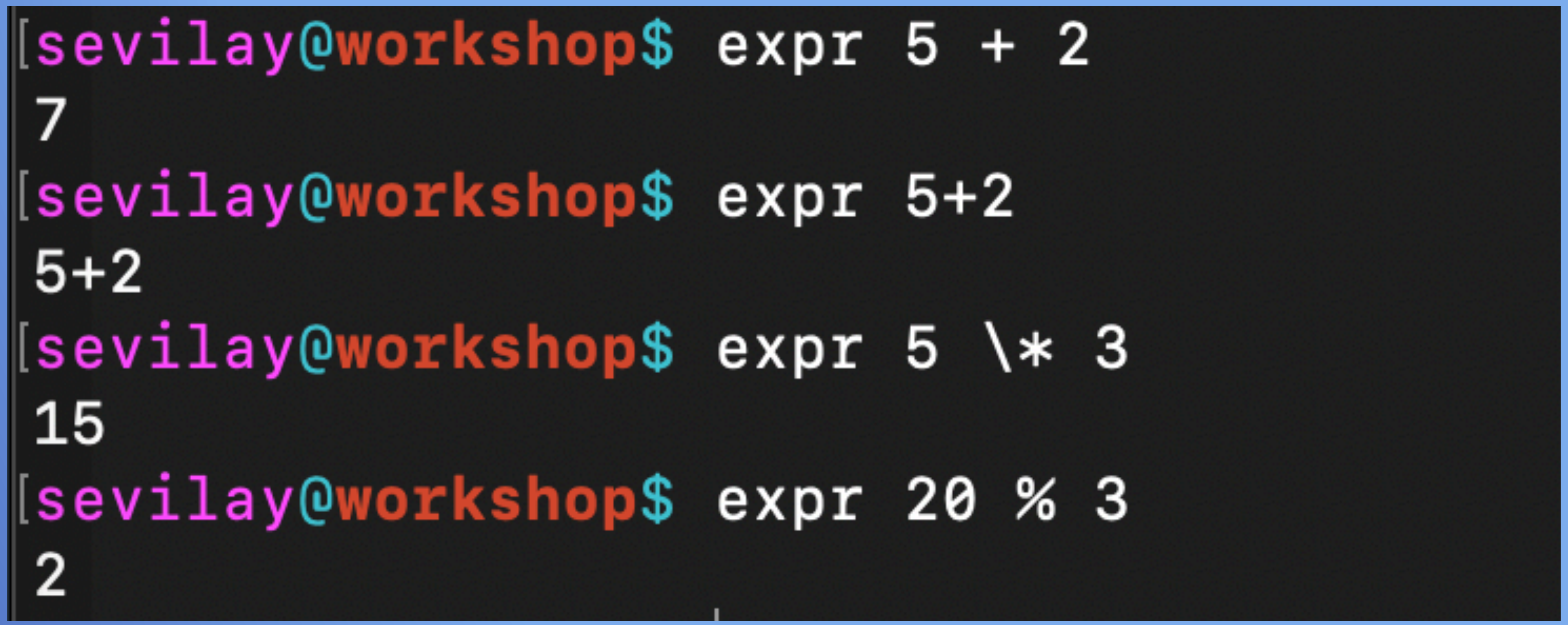
- subtraction

/ division

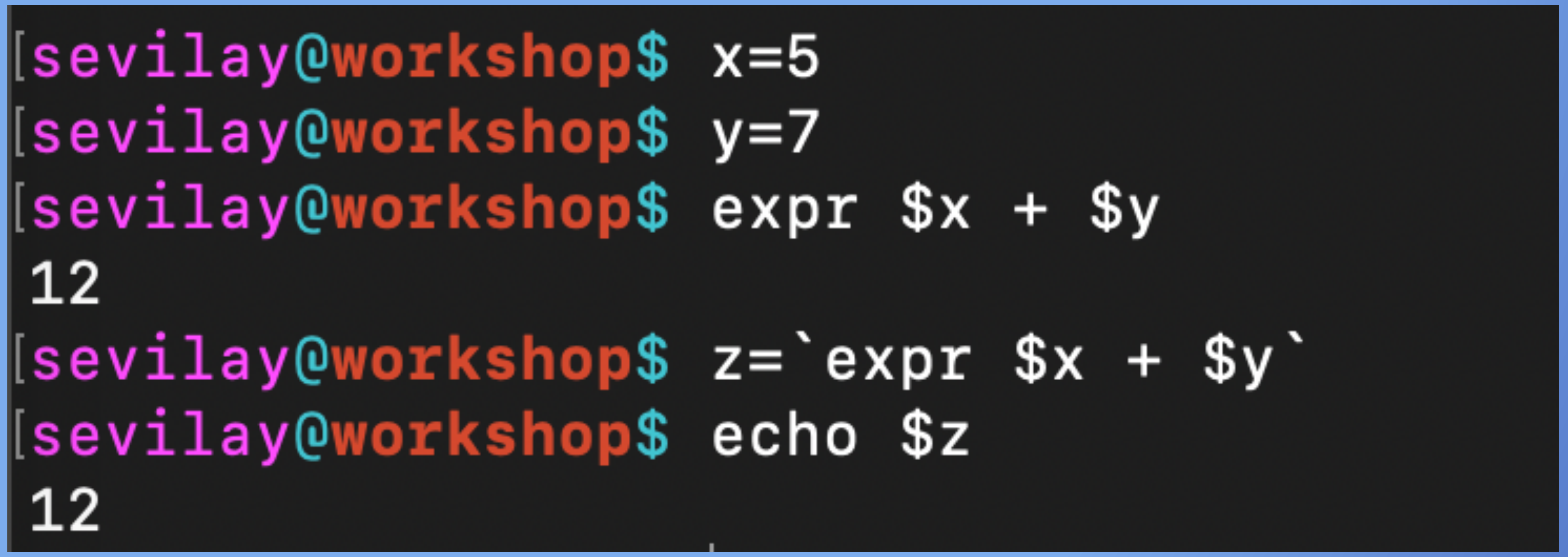
% modular (to find reminder)

***** multiplication

```
x=5
y=7
expr $x + $y
z= `expr $x + $y`
```



```
[sevilay@workshop$ expr 5 + 2
7
[sevilay@workshop$ expr 5+2
5+2
[sevilay@workshop$ expr 5 \* 3
15
[sevilay@workshop$ expr 20 % 3
2
```



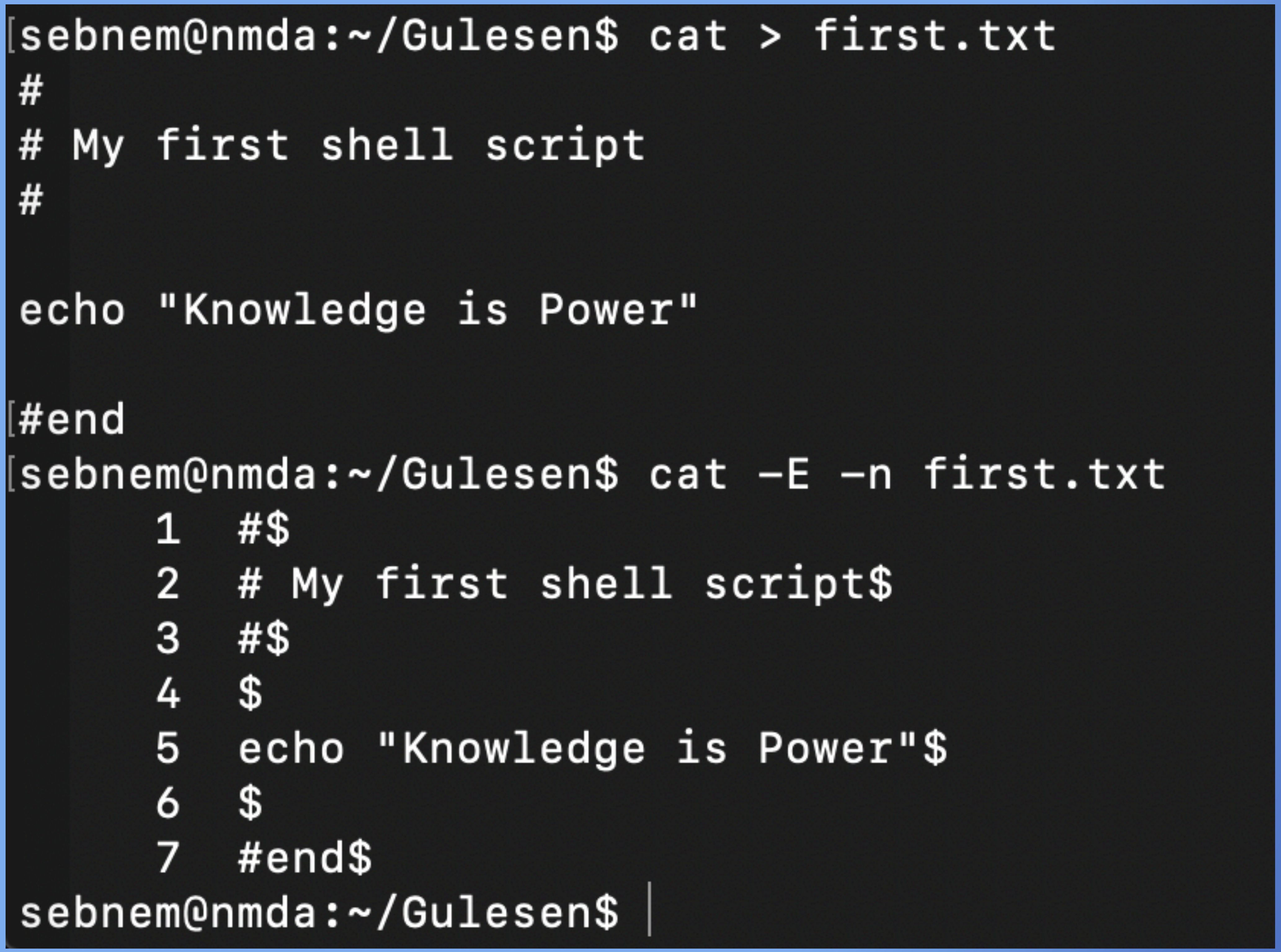
```
[sevilay@workshop$ x=5
[sevilay@workshop$ y=7
[sevilay@workshop$ expr $x + $y
12
[sevilay@workshop$ z=`expr $x + $y`
[sevilay@workshop$ echo $z
12
```

Command: cat

- **cat** can be used:
  - to display the contents of a file
  - to display the contents of multiple files
  - to create new file instead of text editors (exit writing with ctrl+d)
  - to combine multiple files
  - to append data to a file

- Some options of **cat**:
  - **-b** : number nonempty output lines
  - **-E** : display **$** at end of each line
  - **-n** : number all output lines
  - **-s** : suppress repeated empty output lines

```
cat first.txt
cat first.txt second.txt
cat second.txt  >> first.txt
cat first.txt second.txt  > third.txt
```



```
sebnem@nmda:~/Gulesen$ cat > first.txt
#
# My first shell script
#

echo "Knowledge is Power"

#end
sebnem@nmda:~/Gulesen$ cat -E -n first.txt
     1	#$
     2	# My first shell script$
     3	#$
     4	$
     5	echo "Knowledge is Power"$
     6	$
     7	#end$
sebnem@nmda:~/Gulesen$
```

Command: cat

- Create a file named as **first.txt**

```
cat > first.txt
```

- Write following to **first.txt** file and exit with **ctrl+D** buttons

```
Welcome to our workshop
Welcome to our workshop

Hope you'll like the workshop
Hope you'll like the workshop
Hope you'll like the workshop

Thanks!
```

- Create a file named as **second.txt**

```
cat > second.txt
```

- Write following to **second.txt** file and exit with **ctrl+D** buttons

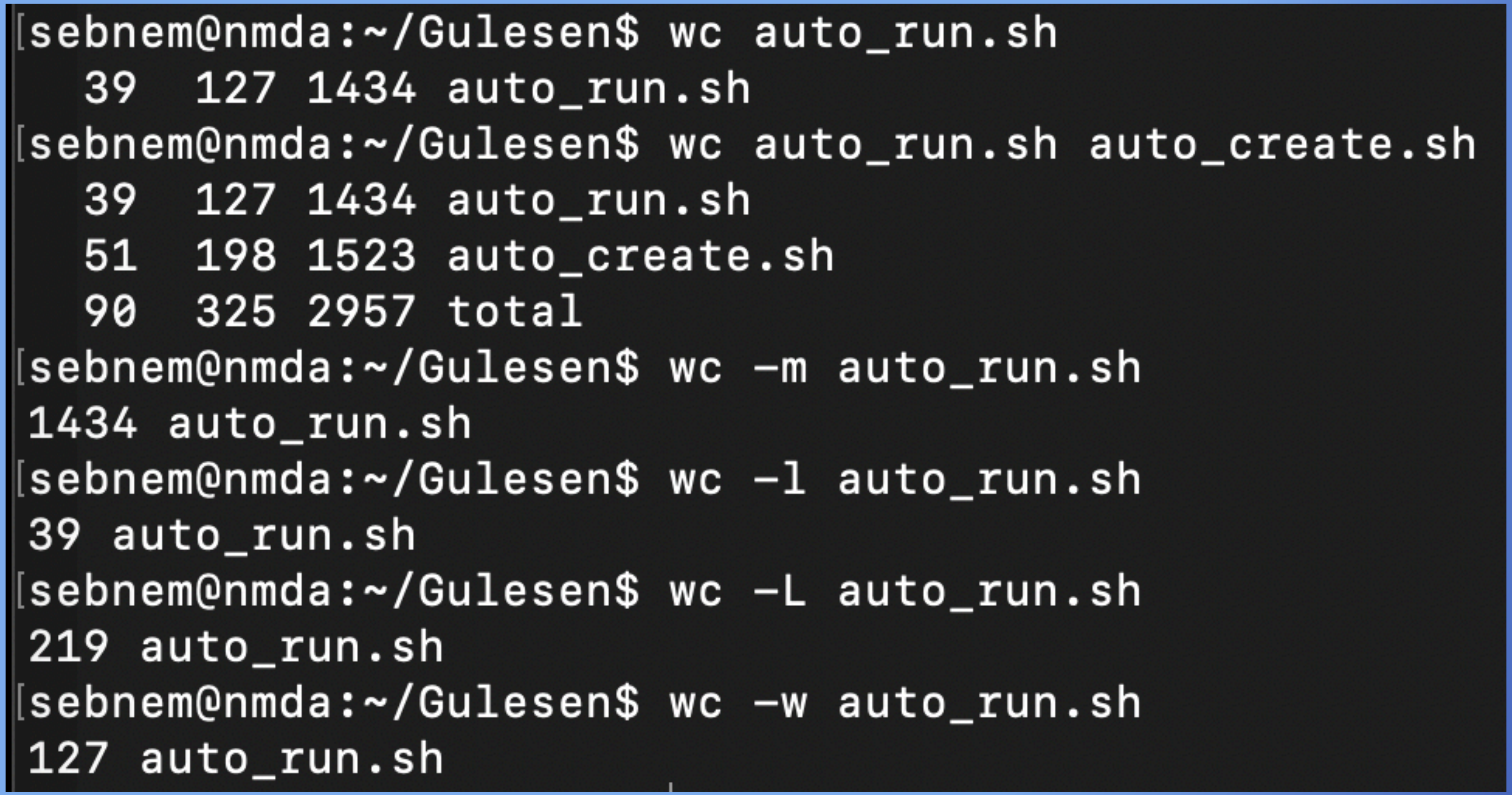
```
Welcome to our workshop
Welcome to our workshop
This is BASH workshop
Hope you'll like the workshop
Hope you'll like the workshop
Hope you'll like the workshop

Thanks!
```

Command: wc

- **wc** can be used:
  - to display the number of lines, words, characters and the name of a file
  - to display the number of lines, words, characters and the name of multiple files and total of the each type count of all files

- Some options of **wc**:
  - **-m** : print the character counts
  - **-l** : print the line counts
  - **-L** : print the length of the longest line
  - **-w** : print the word counts

```
wc first.txt
wc first.txt second.txt
```



```
[sebnem@nmda:~/Gulesen$ wc auto_run.sh
   39   127 1434 auto_run.sh
[sebnem@nmda:~/Gulesen$ wc auto_run.sh auto_create.sh
   39   127 1434 auto_run.sh
   51   198 1523 auto_create.sh
   90   325 2957 total
[sebnem@nmda:~/Gulesen$ wc -m auto_run.sh
1434 auto_run.sh
[sebnem@nmda:~/Gulesen$ wc -l auto_run.sh
39 auto_run.sh
[sebnem@nmda:~/Gulesen$ wc -L auto_run.sh
219 auto_run.sh
[sebnem@nmda:~/Gulesen$ wc -w auto_run.sh
127 auto_run.sh
```

Command: uniq

- **uniq** can be used to report and filter out the repeated lines in a file

- Some options of **uniq**:
  - **-c** : print how many time a line is repeated at the beginning of the line as a prefix
  - **-d** : print only the repeated lines
  - **-D** :print all repeated lines together
  - **-u** : print only unique lines

```
uniq first.txt
uniq -c first.txt
uniq -u first.txt
uniq -d second.txt
```

Command: diff

- **diff** can be used to find differences between two files by comparing line by lines. It reports which lines in one file has to be changed to make the two identical files (default is case sensitive)
- a : means added line

  d : means deleted lines

  c : means changed lines —> c=d+a : mainly, deleted and added different lines
- Commonly used option of **diff:**
  - **-i** : print the differences by ignoring case differences

```
diff first.txt second.txt
diff -i first.txt second.txt
```

Commands: head and tail

- **head** can be used:
  - to print the first 10 lines of the input file (by default)
  - to print the first 10 lines of multiple input file (by default)
- **tail** can be used:
  - to print the last 10 lines of the input file (by default)
  - to print the last 10 lines of multiple input file (by default)

```
head pdbInfo.txt
head - 12 pdbInfo.txt
head -v pdbInfo.txt
```

```
tail pdbInfo.txt
tail - 9 pdbInfo.txt
tail -v pdbInfo.txt
```

- Some options of **head** and **tail**:
  - **-n** ——lines=***K*** : print the first/last K lines
  - **-n** ——lines=**-*K*** : print the all lines up to the last(for head)/first(for tail) K lines
  - **-q** : never print the file name as the header
  - **-v** : always print the file name as the header

Command: sort

- **sort** can be used to sort lines numerically and alphabetically according to first character of the first word in the line
- Default order of **sort**:
  - Line starting with number is printed first before a letter
  - Line starting with a lowercase letter is printed first before an uppercase letter
- Some options of **sort**:
  - **-o** : write the output to a new file
  - **-r** : print the file in reverse order
  - **-n** :sort and print the file numerically (not only first character of the first word in the line. It is for whole first word)
  - **-nr** : sort the file with numeric data in reverse order(not only first character of the first content in the line. It is for whole first number)
  - **-k —option** :sort a table based on any (option, ex:2n) column number
  - **-c** : check the file whether it's already sorted
  - **-u** : print the sorted file without duplicates

Command: sort

- sort the content of **second.txt**

```
sort second.txt
```

- sort the content of **second.txt** in reverse order

```
sort -r second.txt
```

- sort the unrepeated content of **second.txt**

```
sort -u second.txt
```

- sort the unrepeated content of **second.txt** in reverse order

```
sort -u -r second.txt
```

Command: grep

- **grep** is used to filter a file based on searches as a specific pattern of characters, and it displays all lines that contain this patterns.

- Some options of **grep**:
  - **-c** : print only a count of the lines which match
  - **-i** : print the line that contains the pattern with ignoring upper/lower case
  - **-l** :print a list of filenames only
  - **-n** : print lines that contains the pattern with the line number of each
  - **-v** : print all lines that don't match with the pattern
  - **-e** exp : print the specific expression (can be used multiple times)
  - **-w** : search and print the pattern matches as the whole word
  - **-o** : print only the matched part of the pattern in the matched line
  - **-A**n : print searched line and n lines after the matched
  - **-B**n : print searched line and n lines before the matched
  - **-C**n : print searched line and n lines after and before the matched

```
grep "you" first.txt
grep -i "yOu"  first.txt
grep -B2 "our" first.txt
grep "^will" first.txt
grep "op$"  first.txt
```

Command: grep

- find “helix” in **pdbInfo.txt**

```
grep "helix" pdbInfo.txt
```

- find “HeLiX” in **pdbInfo.txt**

```
grep -i "HeLiX" pdbInfo.txt
```

- Print count number of “sheet” in **pdbInfo.txt**

```
grep -c "sheet" pdbInfo.txt
```

- print “hel” with the line number of each in **pdbInfo.txt**

```
grep -n "hel" pdbInfo.txt
```

- print 3 lines after “CRISPR” for **pdbInfo.txt**

```
grep -A3 "CRISPR" pdbInfo.txt
```

Command: sed

- **sed** is used to edit the files without opening it. It works as powerful text stream editor. It can be used to do insertion, deletion, search and substitution. As a default, it does changes as printing in the terminal, not write in the file. It prints changed form of whole file to terminal.

- Replacing and addition string with **sed:**
  - **sed** replaces the first occurrence of the given pattern in each line, not the second or third occurrence in the line
  - **-i** : do changes in file and it isn't print on the terminal screen
  - **s** : represent substitution operation
  - **/** : is delimiters
  - "the" : is the search pattern
  - "our" : is the replacement string

```
sed 's/the/our/' first.txt
```

```
sed 's/the/our/2' first.txt
```

- Replacing the "n"th occurrence of the pattern in a line with **sed:**
  - **/2** : replace the second occurrence of the pattern in the line

Command: sed

- Replacing all the occurrence of the pattern in a line with **sed:**
  - **/g** : replace all occurrence of the pattern in the line

```
sed 's/the/our/g' first.txt
```

- Replacing string on the specific line number or range of line numbers with **sed:**
  - **3** : specify the line number (that should be before **s** with space)
  - **1,3** : specify the range of line numbers

```
sed '3 s/the/our/' first.txt
sed '1,3 s/the/our/' first.txt
```

- Printing only the replaced lines with **sed:**
  - **/p** : means print

```
sed -n 's/the/our/p' first.txt
```

- Deleting the line with **sed:**
  - **5d** : delete the 5th line
  - **$d** : delete last the line
  - **3,6d** : delete all the line from range 3 to 6
  - **5,$d** : delete all from the 5th line to last line
  - **/d** : delete all lines that have occurrence of the pattern in the line

```
sed '5d' first.txt
sed '$d' first.txt
sed '3,6d' first.txt
sed '5,$d' first.txt
sed '/our/d' first.txt
```

Command: sed

- Change all “sheet” filed with “SHEET”

```
sed 's/sheet/SHEET/' pdbInfo.txt
```

- Change “sheet” filed in 5th line with “SHEET”

```
sed '5 s/sheet/SHEET/' pdbInfo.txt
```

- Change the 2th occurrence of “CS” filed with “cs”

```
sed 's/CS/cs/2' pdbInfo.txt
```

- Change the all occurrence of “CS” filed with “cs”

```
sed 's/CS/cs/2' pdbInfo.txt
```

- Change all “sheet” filed with “SHEET” and print twice

```
sed 's/sheet/SHEET/p' pdbInfo.txt
```

- Change all “sheet” filed with “SHEET” and print only changed ones

```
sed -n 's/sheet/SHEET/p' pdbInfo.txt
```

- delete the 5th line
- delete last the line
- delete all the line from range 3 to 6
- delete all from the 5th line to last line
- delete all lines that have occurrence of the “sheet” in the line

```
sed '5d' first.txt
sed '$d' first.txt
sed '3,6d' first.txt
sed '5,$d' first.txt
sed '/sheet/d' first.txt
```

Command: awk

- **awk** is a scripting language and used to manipulate data. variables, numeric functions, string functions and logical operators can be used with **awk.**

- With **awk:**
  - read a file line by line
  - split the line into fields
  - compare given input with the pattern
  - make changes on matched pattern
  - format output lines
  - write arithmetic and string operation
  - create conditionals and loops

- Example roles of some options of **awk**:
  - **NF** **:** gives the number of the fields in the line
  - **NR :** enumerates each line
  - **FS :** specify the separator between each field in the line(default whitespace)
  - **-F :** specify the separator between each field in the line

Command: awk

- Print 3rd column of the **pdbInfo.txt**

```
awk '{ print $3 }' pdbInfo.txt
```

- Print column 1 of each data which has "SH3"

```
awk '/SH3/ {print $1}' pdbInfo.txt
```

- Print column 1 and 2 of each data (with space between them) whose 2nd field has "DHPS"

```
awk '$2 ~ /SH3/ {print $1,$2}' pdbInfo.txt
```

- Print column 1 and 2 of each data (with space between them) whose 2nd field has **NOT** "CSP"

```
awk '$2 !~ /CSP/ {print $1,$2}' pdbInfo.txt
```

- Print column 1 of all data whose 4nd field is greater than "100"

```
awk '$4 > 100 {print $1}' pdbInfo.txt
```

- Print column 1 of all data whose 4nd field is equal to "67"

```
awk '$4==67 {print $1}' pdbInfo.txt
```

Command: awk

- Print column 1 of all data whose 4nd field is equal to “67” **OR** “291"

```
awk '$4==67 || $4==291 {print $1}' pdbInfo.txt
```

- Print column 1 of all data whose 4nd field is equal to “67” **AND** 3rd feeld is”helix"

```
awk '$4==67 && $3 ~ /helix/ {print $1}' pdbInfo.txt
```

- Print column 1,2,3 of all data whose 2nd field is between “SH3” and “DHPS”

```
awk '/SH3/ , /DHPS/ {print $1}' pdbInfo.txt
```

- Print number of rows in the pdbInfo.txt with “NR”. (END is to get the last enumerated line)

```
awk 'END {print "we have", NR, "rows"}' pdbInfo.txt
```

- Print 4th column and sort it numerically (not fort first character of number)

```
awk '{print $4}' pdbInfo.txt | sort -n
```

Command: if-then-fi

- **If-then-fi** statement is used to run different parts of your shell program based on whether certain condition(s) is true. If the given conditional expression is true, then it enters and executes the statements between “**then**” and “**fi**”.

```
If  [ conditional expression ];
then
      Statement 1
      Statement 2
         .
         .
fi
```

```
count=99
if [ $count -eq 100 ];
then
      echo "Count is 100"
fi
```

Command: if-then-else-fi

```
If  [ conditional expression ];
then
      Statement 1
      Statement 2
         .
else
      Statement 3
      Statement 4
         .
fi
```

```
count=99
if [ $count -eq 100 ];
then
      echo "Count is 100"
else
      echo "Count is not 100"
fi
```

Command: if-then-else-fi

```
If  [ conditional expression 1 ];
then
      Statement 1
      Statement 2
         .
elif [ conditional expression 2 ];
then
      Statement 3
      Statement 4
else
      Statement 5
         .
fi
```

```
count=99
if [ $count -eq 100 ];
then
      echo "Count is 100"
elif [ $count -gt 100 ];
then
      echo "Count is greater than 100"
else
      echo "Count is less than 100"
fi
```

Command: for loop

- **for loop** statement is used to run repeatedly until the last item in the list. It is used when the number of iterations is already known.

```
for  var in [ LIST ]
do
      Statement 1
      Statement 2
         .
         .
done
```

```
for i in  {0..3}
do
      echo "Number: $i"
done
```

Command: for loop

```
for  var in [START..END..INCREMENT]
do
      Statement 1
      Statement 2
         .
done
```

```
for i in  {0..20..5}
do
      echo "Number: $i"
done
```

```
for  ((i=0 ; i<=1000 ; i++))
do
      Statement 1
      Statement 2
         .
done
```

```
for ((i=0 ; i<=1000 ; i++))
do
      echo "Number: $i"
done
```

Command: for loop

```
for  field in "${arrayName[@]}"
do
      Statement 1
      Statement 2
         .
done
```

```
arrayCitys=('Istanbul' 'New York' 'Amsterdam' 'Berlin')

for x in  "${arrayCitys[@]}"
do
      echo "City: $x"
done
```

Command: while loop

- **while loop** statement is used to run given commands repeatedly based on given condition. It is used when the number of iterations is not exactly known.

```
while [ condition ]
do
      Statement 1
      Statement 2
          .
          .
done
```

```
i=1
while [[ $i -le 3 ]]
do
      echo "$i"
      (( i++ ))
done
```

Bash Scripting

- It is important that to mention which SHELL type is used for the script at the beginning of the file

```
#!/bin/bash
```

- **#** : is used at the beginning of the line and represents that this line is comment line. Simply, it deactivates the line.

```
# My first shell script
```

Bash Scripting

- Write a script that only lists the items in the current folder and finds “helix” in the pdbInfo.txt file

```
#!/bin/bash
ls
echo " "
grep "helix" pdbInfo.txt
```

- Write a basic script that includes **if-then-else-fi statement** with following condition:

Check 99 is grater than 100

```
#!/bin/bash
#basic if statement
count=99
if [ $count -eq 100 ];
then
      echo "Count is 100"
else
      echo "Count is 99"
fi
```

Bash Scripting

- Write a basic script that includes **if-then-fi statement** with following condition:

  Check given variable is grater than 100

```
#!/bin/bash
#basic if statement that takes variable from user

if [ $1 -gt 100 ];
then
	echo $1 "is larger than 100"
fi
```

Bash Scripting

- Write a basic script that includes **for loop statement** that print the numbers 0 to 20 in increment of 5

```
#!/bin/bash
#basic for loop statement

for i in  {0..20..5}
do
      echo "Number: $i"
done
```

Thank you for joining the workshop!

any question?

by Sevilay Güleşen