Working in the shell

Introduction to Bash - a summary

Prerequisites

To be able to follow this lecture, you need to be prepared.

You should have read the Swedish compendium, sections “Filer, kataloger och sökvägar” and “Terminalmiljö”, as well as the Wiki:

http://wiki.juneday.se/mediawiki/index.php/ITIC:Working_in_the_shell_-_Introduction_to_Bash

Terminal, command line and shell

Some concepts:

- Command line interface - commands are entered via keyboard, line-by-line
- Shell, a command line interpreter e.g. Bash
- Terminal - A small application which emulates an old computer terminal
    - The terminal runs a shell and lets you interact with it

File system

- Abstraction of a single-rooted tree hierarchy
- The root directory is called /
- Any directory can contain files and more directories
- Any directory has only one “parent directory”
    - Except / which is the top dog
- We use paths to express locations of files and directories
- Two kinds of paths:
    - absolute
    - relative

Absolute paths

Start with the root - / and all the way “down”

Example

/home/rikard/Documents/Schedule.csv

Are canonical and uniquely identify the location of a file.

Work from any directory (or in any script or program)

Relative paths

- Can be shorter
- Never start with /
- Describes a location from the current directory
- Only “*work*” from the current directory
- Can be a problem in a script, if the script is run from a different directory
- Uses .. to signify “up to parent directory” and “.” to signify “current directory”
- ~ (tilde) signifies your home directory (typically /home/*username*)

The prompt

- Shows who you are, where you are and what shell you are running (kind of)
- In Bash for normal users it end with a $ and a space
- Example:

```
rikard@newdelli:~$ echo "Hello Bash"
Hello Bash
rikard@newdelli:~$
```

`user@computer:current_directory$`

`(~ means home directory)`

Secondary prompt

- Tells you to please complete the command line

```
rikard@newdelli:~$ echo "Hello⏎                  #user hits Enter
> many lines"
Hello
many lines
rikard@newdelli:~$
```

Current directory

- An abstraction for “being in a directory”
- Is shown as part of the prompt
- Is stored in $PWD and can be printed by pwd
- Has nickname . (a dot)

  cp /etc/hosts .   ← copy that file *here*
- Processes you start from the shell will be aware of this directory
- Is used as the starting point for relative paths

Home directory

- Every user has one
- Typically in /home/*username* (on Debian/Ubuntu-like systems)
- Is your current directory when starting a shell
- Let’s users keep their files to themselves
- Is stored in $HOME
- Using `cd` without argument takes you here
- Has nickname ~ (tilde)

Issuing commands

- Commands are small programs to be used in the shell
- Commands take flags/options and arguments
- Some commands require arguments to make sense and work:

  ```
  $ cp
  cp: missing file operand
  Try 'cp --help' for more information.
  ```
- Some common/basic commands:

```
echo ls cd cp mv pwd head tail cat file
```

Options/flags

- Tell a command *how to do its job*
- Typically starts with a dash
- Examples:

```
rikard@newdelli:~/opt/progund/datorkunskap-kompendium$ ls -1
globbing
README.md
school
script
text
```

Arguments

- Tell a command *what to do*
- Most commands require one or more arguments

```
cp ../file.txt Documents/

cd Documents/

ls /tmp/

mv old_filname new_filename

mpg321 Warlords.mp3
```

Moving around and knowing where you are

- use cd to change directory
- use pwd to print absolute path of current directory
- use `cd -` (cd then dash) to move to last directory
- Use `cd ..` to go up a directory (relative path)

```
rikard@newdelli:~$ cd ..
rikard@newdelli:/home$ cd ..
rikard@newdelli:/$ cd home
rikard@newdelli:/home$ cd rikard/
rikard@newdelli:~$ cd ../..
rikard@newdelli:/$ cd home/rikard/
rikard@newdelli:~$
```

Creating directories

- Use `mkdir` to create one or more directories
- Use `mkdir -p` to create a whole tree

```
mkdir -p video/tv-series/battlestar_galactica/
```

- Create directories for you work and files - keep things organized
- Use `rmdir` to remove (empty) directories
- Use `rm -r` to remove whole trees (and all their contents)
  - Use with care, there’s no undelete

Some standard directories created for you

In your home directory, you will probably find some standard directories created for you:

- Desktop
- Documents
- Downloads
- Music

Text files

- Can be created by editor, downloaded or from output from command
- Plain text means that the file is encoded as a series of binary numbers, where each number corresponds to a character (from the ASCII table)
- Often used for settings - examples found in /etc
- Used for simple texts without formatting and style
- Source code is stored in plain text
  - Bash scripts, Java programs, C programs, HTML files, ...
- Data can be stored in plain text

Small script example

```
#!/bin/bash

echo -n "Time in Tehran is now "
TZ='Asia/Tehran' date +%T
```

```
# if the script is stored in tehran_time.sh then:

$ chmod u+x tehran_time.sh
$ ./tehran_time.sh
Time in Tehran is now 14:37:42
$
```

The PATH environment variable

```
rikard@newdelli:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
:/usr/games:/usr/local/games:/snap/bin:/home/rikard/bin/

rikard@newdelli:~$ which date
/bin/date
rikard@newdelli:~$ which bash
/bin/bash
rikard@newdelli:~$ which which
/usr/bin/which
rikard@newdelli:~$
```

Running a script in a directory not in PATH

- Use ./script_name.sh
- Let’s you run a script that is not in a directory from path
- You can also tell Bash to run the script:
  $ bash script_name.sh
- Starts a subshell (a shell running in a shell) which executes the lines in the script and then dies

File permissions

```
rikard@newdelli:~/bash-intro/text-files$ ls -l
tehran_time.sh
-rw-rw-r-- 1 rikard rikard 72 jul 29 15:39 tehran_time.sh
rikard@newdelli:~/bash-intro/text-files$ chmod u+x
tehran_time.sh
rikard@newdelli:~/bash-intro/text-files$ ls -l
tehran_time.sh
-rwxrw-r-- 1 rikard rikard 72 jul 29 15:39 tehran_time.sh
rikard@newdelli:~/bash-intro/text-files$
```

File permissions

```
-rwxrw-r-- 1 rikard rikard 72 jul 29 15:39 tehran_time.sh
|||||||||`- others have no execute perm
||||||||`-- others have no write perm
|||||||`--- others have read perm
||||||`---- group members have no exectute perm
|||||`----- group members have write perm
||||`------ group members have read perm
|||`------- user (owner) has execute perm
||`-------- user (owner) has write perm
|`--------- user (owner) has read perm
`---------- this is a normal file (d means directory)
```

File permissions

```
-rwxrw-r-- 1 rikard rikard 72 jul 29 15:39 tehran_time.sh
           |   |      |     |  |   |   |   `-- filename
           |   |      |     |  |   |   `------ time of mod.
           |   |      |     |  |   `---------- date of mod.
           |   |      |     |  `-------------- --””--
           |   |      |     `----------------- bytes
           |   |      `----------------------- group name
           |   `------------------------------ username
           `---------------------------------- link count
```

She-bang!

- first line of scripts contain #! followed by the path to the interpreter
- Instructs Bash what interpreter to use
- A new process with the interpreter will be created when running

Process

- Created by the operating system
- Can be listed with the `ps` command
- Is an environment for a program when it is running
- Keeps executing programs isolated from each other
- A process has an ID (an enumerated number)
- Can run in the foreground or background in the shell
- Can be paused, killed, put to background or foreground

Using an editor

- To edit a plain text file, we use an editor
- Many editors can run inside the terminal (without its own window)
- You must know the basics of a few editors if you are studying or working with computers - they are as important as the terminal and shell
- A lot of files are text files - that’s why we need to learn to use editors
- We recommend starting the editor from the command line, with the file to be edited as the argument
    - Helps avoid common mistakes

Nano

- Runs in the terminal
- Very basic
- You edit text and when you want to save you type Ctrl-O and answer the questions (what filename to use)
- When you want to quit, you type Ctrl-X and answer any questions

Gedit

- Gedit is a simple editor with a graphical user interface (GUI) and it runs in its own window.
- Can be started from the command line
- If you want to run it in the background, put an & at the end of the command line

Other editors

- Emacs (preferred by the authors of the wiki)
- Vim (preferred by others)
- Atom (for programming)
- Sublime (for programming)
- Visual Studio Code
- Notepad++

Tab completion

- Bash can fill out the rest of paths if you press TAB
- If it can’t find a unique “rest”, press TAB again to see the options
- Demonstration
- Helps avoiding confusion of what file you are editing
- Again, don’t start the editor first and then go look for files

Job control

- Processes are sometimes called *jobs*
- Use `jobs` to list jobs running from current shell
- Use & at the end of command line to start process in the background
- Use Ctrl-Z to pause current foreground job
- Use `fg` to send job to foreground and `bg` to send to the background
- You can use % to specify which job if there are more

Example of job control

```
rikard@newdelli:~/bash-intro/text-files$ nano tehran_time.sh
Use "fg" to return to nano.

[1]+  Stopped                 nano tehran_time.sh
rikard@newdelli:~/bash-intro/text-files$ ./tehran_time.sh
Time in Tehran is now 13:20:52
rikard@newdelli:~/bash-intro/text-files$ fg
```

Downloading files from the web

- If you use your browser, files probably end up in ~/Download and have to be moved to where you want them
- Easier to download directly from the shell to the directory where you want them
- You can use `wget` or `curl` to download files *over HTTP*

wget

```
$ wget "http://some.url/some/file"

$ wget "http://some.url/some/file" -O desired.name

# Use quotes around the URL to avoid shell expansion and
# problems with spaces and ampersands etc
```

curl

```
curl -JO "url-to-the-file"
 ^    ^   ^
 |    |   |
 |    |   +--> The URL goes here
 |    +------> flags telling curl how it should work
 |              (download only, use suggested filename)
 +-----------> the command "curl"
```

Anatomy of a URL

https://raw.githubusercontent.com/progund/datorkunskap-kompendium/master/school/courses/command_line/meals.txt

```
https - Use the protocol HTTPS (secure HTTP)

:// - comes after the protocol

raw.githubusercontent.com - the domain name - identifies the computer

/progund/datorkunskap-kompendium/master/school/courses/command_line/ - path

meals.txt - the resource (filename)
```

Server - confusion

- Can mean a powerful computer connected to a network
- Can mean an application running on a computer
- “The web server is running on our server in the server room”

HTTP crash course

- Protocol used on the web
- A *client* requests some resource from a *server*
- The client uses an HTTP *method* like GET
  GET /pictures/henrik.png HTTP/1.1
- The client sends along some *headers* with information about the client
- The server responds with headers and then the requested resource

Example response

```
200 OK
Cache-Control: max-age=300
Connection: close
Date: Wed, 24 Jul 2019 13:10:34 GMT
Via: 1.1 varnish
Accept-Ranges: bytes
Vary: Authorization,Accept-Encoding
Content-Length: 274
Content-Type: text/plain; charset=utf-8
Expires: Wed, 24 Jul 2019 13:15:34 GMT
Client-Date: Wed, 24 Jul 2019 13:10:34 GMT
Client-Peer: 151.101.84.133:443
```

here comes the actual file....

Typical clients

- Browsers
- wget
- curl
- Android (your apps use some library to fetch data and files over HTTP)
- iOS (same)

Creating files from command output

- In Bash, there are three streams for data flowing to and from commands
- Standard In(put) - default stream for data consumed by the command
- Standard Out(put) - Expected printouts from commands (like `ls` etc)
- Standard Err(or) - Dedicated for error messages
- All of the above can be *redirected* to (and for Std In, from) files

Interactive commands, line-based commands

- Many commands for text processing can be run interactively
- Bash is an example
- These commands are line-based, they operate on lines
- That’s why you need to press Enter to issue a command

Running cat interactively

- `cat` reads lines from standard in and prints each line out
- It can read and print lines from files, or from standard in

```
rikard@newdelli:~/tempo$ cat
one
one         <- output from cat
two
two         <- output from cat
three
three       <- output from cat
rikard@newdelli:~/tempo$
```

Running grep interactively

```
rikard@newdelli:~/tempo$ grep bingo
lingo
dingo
zingo
bingo
bingo        <- output from grep
rikard@newdelli:~/tempo$
```

Redirecting standard in

- You can use Bash to redirect standard in to come from a file instead of the keyboard

```
rikard@newdelli:~/tempo$ cat < meals.txt
Breakfast: Egg and tea
Lunch: Fish and chips
Snack: Sandwich and juice
Dinner: Stake and sallad
Breakfast: Egg and coffee
Lunch: Hamburger and coke
Snack: Peanuts and beer
Dinner: Pizza
Breakfast: Sandwich and milk
Lunch: Fish and potato
Snack: Apple
Dinner: Pasta and wine
rikard@newdelli:~/tempo$
```

Redirecting standard in

- You can use Bash to redirect standard in to come from a file instead of the keyboard

```
rikard@newdelli:~/tempo$ grep ch < meals.txt
Lunch: Fish and chips
Snack: Sandwich and juice
Lunch: Hamburger and coke
Breakfast: Sandwich and milk
Lunch: Fish and potato
rikard@newdelli:~/tempo$
```

Redirecting standard out

- You can use Bash to redirect standard out to go to a file, rather than the terminal
- Doing so will create or overwrite the file

```
rikard@newdelli:~/tempo$ grep Lunch meals.txt > lunches.txt

rikard@newdelli:~/tempo$ cat lunches.txt
Lunch: Fish and chips
Lunch: Hamburger and coke
Lunch: Fish and potato
rikard@newdelli:~/tempo$
```

Copying text from the terminal

- Two clipboards
- Things you highlight with the mouse is copied and can be pasted via the middle button
- You can also highlight and copy with Ctrl-Ins and paste with Shift-Ins
- If you are used to Ctrl-C and Ctrl-V, you need to reprogram your brain
- Ctrl-C is used to make the terminal send the Terminate signal to the program in the foreground

```
rikard@newdelli:~/bash-intro/text-files$ cat
^C
rikard@newdelli:~/bash-intro/text-files$
```

Redirecting standard out with append

- Use >> to redirect and append (first time the file will be created)

```
rikard@newdelli:~/tempo $ grep Lunch meals.txt > lunches_and_dinners.txt

rikard@newdelli:~/tempo $ grep Dinner meals.txt >> lunches_and_dinners.txt

rikard@newdelli:~/tempo $ cat lunches_and_dinners.txt
Lunch: Fish and chips
Lunch: Hamburger and coke
Lunch: Fish and potato
Dinner: Stake and sallad
Dinner: Pizza
Dinner: Pasta and wine
```

Error messages go to Standard err

```
rikard@newdelli:~/tempo$ grep Breakfast meels.txt > breakfasts.txt
grep: meels.txt: No such file or directory
rikard@newdelli:~/tempo$ ls
breakfasts.txt  lunches_and_dinners.txt  lunches.txt  meals.txt
rikard@newdelli:~/tempo$
```

```
# What is inside breakfasts.txt ?
# The file to redirect to is created as an empty file!
# It’s a good thing that error messages aren’t redirected
# unless we explicitly ask for it...
```

Redirecting standard err and standard out

```
$ ls pictures/ 2>> errors.txt >> picture-and-movies.txt
$ ls movies/ 2>> errors.txt >> picture-and-movies.txt
$ ls images/ 2>> errors.txt >> picture-and-movies.txt
$ ls films/ 2>> errors.txt >> picture-and-movies.txt

$ cat errors.txt
ls: cannot access 'images/': No such file or directory
ls: cannot access 'films/': No such file or directory
$ cat picture-and-movies.txt
img1.png
img2.png
img3.png
terror-on-elm-st1.avi
terror-on-elm-st2.avi
terror-on-elm-st3.avi
```

The streams have numbers

- 0 - standard in
- 1 - standard out
- 2 - standard err

File type versus file name

- In decent operating systems, the file name has no meaning
- You still use suffices to make it clear for humans the type of file
- The computer couldn’t care less
- Use `file` to investigate filetype

```
rikard@newdelli:~/bash-intro/text-files$ cat todo-list.txt
* Buy beer
* Throw party
* Clean apartment
rikard@newdelli:~/bash-intro/text-files$ mv todo-list.txt todo-list.mp3
rikard@newdelli:~/bash-intro/text-files$ cat todo-list.mp3
* Buy beer
* Throw party
* Clean apartment
rikard@newdelli:~/bash-intro/text-files$ file todo-list.mp3
todo-list.mp3: ASCII text
```

Structure of plain text files

- Text consists of characters, words, lines and paragraphs
- A word is printable characters separated by *whitespace* (blanks, tabs, newlines)
- A line is characters or words with a newline character at the end
- A paragraph is lines separated by an extra newline
- Everything is encoded in binary using e.g. the ASCII table for the numbers

US ASCII table

0	NUL	16	DLE	**32**		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	**HT**	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	**LF**	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Expressing whitespace in Bash etc

- A blank is written as is (press the space bar)
- A tab is *escaped* as \t
- A newline (linefeed) is escaped as \n

```
$ echo -e "Number\tName\tPrice\n9\tUrquel\t12.90"
Number  Name    Price
9       Urquel  12.90
$

# -e is the flag to echo for using escaped characters
```

How many characters?

```
These are four words.
```

How many characters?

```
These are four words.
```

- Depends! Is there a newline at the end?
- It is common to end a text with a newline (but not required)

ASCII-encoded

```
 84 (T)
101 (e)
115 (s)
101 (e)
 32 (blank)
 97 (a)
114 (r)
101 (e)
 32 (blank)
102 (f)
111 (o)
117 (u)
114 (r)
 32 (blank)
119 (w)
111 (o)
114 (r)
100 (d)
115 (s)
 46 (.)
 10 (LF - newline)      ← Optional
```

Using wc to count text

- `wc -l` number of lines (actually, number of newlines)
- `wc -c` number of bytes
- `wc -m` number of characters
- `wc -w` number of words

Swedish characters take two bytes (are encoded using utf-8)

English characters take one byte (encoded as plain text ASCII)

Textwrapping

- Plain text is just lines (where empty lines are just a single newline)
- Lines have a width (the number of characters before the next newline)
- Applications displaying text will automatically wrap lines, so that the window displays the maximum number of words per line
- Changing the width of the application window will make the text look different
- Think about where you put newlines
- Either, newlines only occur between sections/paragraphs
- or, you will use a fixed linewidth

Using fold to create a fixed linewidth

```
rikard@newdelli:~/bash-intro/text-files$ fold -s --width=80 lorem.txt > lorem80.txt

## -s means "break at spaces" (don't cut in the middle of a word)

## using wc to find lenght of the widest line

rikard@newdelli:~/bash-intro/text-files$ wc -L lorem80.txt
80 lorem80.txt
rikard@newdelli:~/bash-intro/text-files$
```

Fixed width helps when investigating text

- Text commands are line-based
- If lines are very long, results may be overwhelming
- Using, e.g. `grep` to print lines with some pattern you search for works not so well if your file has only one very long line - the whole file will be printed it there’s a match

grep

- `grep pattern file` - search for lines containing pattern (even substrings)
- `grep -i pattern file` - ignore case
- `grep -w pattern file` - search for lines with words matching pattern
- `grep -v pattern file` - search for lines that don’t have pattern
- `grep -iwv pattern file` - ignore case, lines without word matching pattern
- `grep -r pattern directory` - search whole directory tree for files with lines matching pattern

Regular expressions intro

- `.` - any one character
- `\.` - an actual dot
- `pattern$` - line ends with pattern
- `^pattern` - line starts with pattern
- `[a-z]` - any one character between a and z in the ascii table
- `[^a-z]` - any one character not among a-z
- `pattern?` - one or zero occurrences of the character expressed by pattern
- `pattern*` - zero or more occurences ...

Printing parts of files, etc

- `head` - print the ten first lines
  - `head -5` - print the five first lines
- `tail` - print the ten last lines
  - `tail -7` - print the seven last lines
- `cat` - print all lines from file or std in
- `tac` - print all lines in reverse order from file or std in
- `rev` - print all lines backwards
- `sort` - print all lines sorted by the ASCII table or as specified

Cutting up URLs

```
rikard@newdelli:~/bash-intro/text-files$ cat a_few_urls.txt
http://www.gu.se/bazinga
https://ait.gu.se/forskning/journalister
http://www.elsewhere.org/pomo/
http://snarxiv.org/vs-arxiv/
http://www.physics.nyu.edu/faculty/sokal/afterword_v1a/after
word_v1a_singlefile.html

# How can we get rid of the protocols?
```

Cutting up lines using cut

```
# Cut lines using / as delimiter, show fields 3 and up
rikard@newdelli:~/bash-intro/text-files$ cut -d '/' -f3-
a_few_urls.txt
www.gu.se/bazinga
ait.gu.se/forskning/journalister
www.elsewhere.org/pomo/
snarxiv.org/vs-arxiv/
www.physics.nyu.edu/faculty/sokal/afterword_v1a/afterword_v1
a_singlefile.html
```

Cutting up lines using cut

```
# Cut lines using / as delimiter, show field 3 only
rikard@newdelli:~/bash-intro/text-files$ cut -d '/' -f3
a_few_urls.txt
www.gu.se
ait.gu.se
www.elsewhere.org
snarxiv.org
www.physics.nyu.edu
```

Translating characters

- tr 'a' 'e' - translate all ‘a’ to ‘e’ from standard in
- tr -d 's' - remove all ‘s’ from standard in
- tr 'ng' 'ss' - translate all ‘ng’ to ‘ss’
- tr '[a-z]' '[A-Z]' - translate any lowercase to its uppercase version

```
rikard@newdelli:~/bash-intro/text-files$ tr 'a-zA-Z' 'n-za-mN-ZA-M'
Top secret message!
Gbc frperg zrffntr!       <- tr replies
Gbc frperg zrffntr!
Top secret message!       <- tr replies
(user presses Ctrl-D)
rikard@newdelli:~/bash-intro/text-files$
```

Uniquify sorted consecutive lines

- `uniq some_sorted_text.txt resulting_file.txt` - remove duplicate lines and save the result in resulting_file.txt
- `uniq -c sorted.txt result.txt` - remove duplicate lines and report how many duplicate each line had in the resulting `result.txt` file

More about sort

- `sort` - sort lexicographically
- `sort -r` - sort descending (reverse)
- `sort -n` - sort numerically
- `sort -k1` - sort using column 1

Using pipes

- A pipe | makes the output from one command the input to the next
- Makes the Bash text commands a veritable toolbox for doing almost anything with just one command line
- Saves a lot of time (no need to write a program)
- Is the one thing you should learn to get the most from your shell

Getting the word frequency from a text file

```
$ tr ' ' '\n' < lorem80.txt | tr -d '[.,:;!?]'| grep -v '^$'| sort -i | uniq -ic | sort -rnk1 | head -11
     11 sed
      8 in
      8 dolor
      8 at
      6 vel
      6 id
      6 elit
      6 ac
      5 tortor
      5 quis
      5 Donec
```

Getting the word frequency from a text file

```
$ tr ' ' '\n' < lorem80.txt |
# make all spaces into newlines (so that we get one word per line)
  tr -d '[.,:;!?]' |
# delete all punctuation listed inside the square brackets
  grep -v '^$' |
# find all lines that are not empty
  sort -i |
# sort the result caseinsensitively
  uniq -ic |
# remove duplicate lines and report the number of dupes
  sort -rnk1 |
# sort the result descendingly, using first column, treated numerically
  head -11
# keep only the eleven first lines (the top 11 frequencies)
```

Editing the command line - get efficient

- Arrow up goes up the history of commands, arrow down goes back
- One single arrow up gives you the previous command
- Escape and then a dot (or Alt-.) gives you the previous command’s last argument
- Ctrl-A moves the cursor to the beginning of the line, Ctrl-E to the end
- Ctrl-arrow left skips one word to the left, Ctrl-arrow right, the opposite
- Ctrl-W erases the word to the left, Ctrl-K erases all words to the right
- Ctrl-Y pastes whatever was erased last

More nifty tricks

- !! issues the last command again
- Ctrl-R - searches the history interactively

Demonstration

Exit status

- All commands report their exit status to the shell
- Numeric value, where 0 means expected good result
- Other numbers mean some kind of failure, see the manual to learn what
- Is stored in the variable $?
- Allows you to use && for commands that depend on success of previous commands and || for commands to issue if previous failed

```
rikard@newdelli:~/bash-intro/text-files$ find lorem80.txt && wc -l lorem80.txt
lorem80.txt
38 lorem80.txt
rikard@newdelli:~/bash-intro/text-files$ find lorem90.txt && wc -l lorem90.txt
find: ‘lorem90.txt’: No such file or directory
rikard@newdelli:~/bash-intro/text-files$ find lorem80.txt > /dev/null && wc -l lorem80.txt
38 lorem80.txt
rikard@newdelli:~/bash-intro/text-files$ find lorem90.txt &> /dev/null && wc -l lorem90.txt
```

Some words on the if-statement

- `if` takes one or more commands as arguments
- if the command (or last command) has exit status 0, then the `then`-branch is executed
- otherwise the `elif`- or `else`-branch is executed
- ends with `fi`

Example IF

```
$ if date | grep Mon
> then
>   echo "Week starts now"
> elif date | grep Tue
> then
>   echo "Only four more days"
> else
>   echo "It's not Monday or Tuesday"
> fi
Tue Jul 30 14:38:59 CEST 2019
Only four more days
```

Example conditional

```
$ date | grep -q Tue && echo "Tuesdays rock" || echo "It's not Tuesday"
Tuesdays rock
```

Globbing

- Used to expand filenames (and directory names)
- * means “Anything”
    - *.txt - all files ending with .txt
- [0-9] means any one character between 0 and 9
- [A-H] means any one character between A and H
- ? means any one character

Brace expansion

- Curly braces allow us to expand combinations

```
$ echo SVT{1,2,24}
SVT1 SVT2 SVT24
$
```

Brace expansion

- Can be nested and very powerful

```
music/
├── classical
│   ├── classicism
│   ├── modernism
│   ├── modernist
│   └── renaissance
├── jazz
│   ├── bebop
│   ├── free_jazz
│   └── fusion
└── rock
    ├── hard_rock
    ├── metal
    └── rockabilly
```

Brace expansion

```
# the directory tree was created with one single command line:

$ mkdir -p
music/{classical/{modernist,renaissance,classicism,modernism},rock/{hard_rock,
metal,rockabilly},jazz/{bebop,free_jazz,fusion}}

# all on one line
```

Variables

- A named memory location
- Use $variable to expand the value
- Environment variables are shared between shells and initialized when the shell starts
- Variable you create are local to the shell where they were created

Arguments to scripts end up in special variables

```
rikard@newdelli:~/bash-intro/text-files$ cat arguments.sh
#!/bin/bash

echo "Script name: $0"
echo "Number of arguments: $#"
echo "All arguments $*"
echo "First argument: $1"
echo "Second argument: $2"

rikard@newdelli:~/bash-intro/text-files$ ./arguments.sh one two
Script name: ./arguments.sh
Number of arguments: 2
All arguments one two
First argument: one
Second argument: two
rikard@newdelli:~/bash-intro/text-files$
```

Use quotes around variables

- If a variable contains spaces, Bash will treat the value as many words if used unquoted
- Using double quotes around a variable when used, will tell Bash to treat it as one single string (which may or may not contain spaces)

Forgetting to use quotes

```
rikard@newdelli:~/bash-intro/text-files$ name="Rikard Fröberg"
rikard@newdelli:~/bash-intro/text-files$ mkdir $name  ← oops! should have used quotes
rikard@newdelli:~/bash-intro/text-files$ ls
a_few_urls.txt      group_genre.txt                   lorem.txt
apa                 latin_uniq_frequencies.txt        replaceme.txt
four.txt            latin_words_sorted_lower_case.txt  Rikard
frequency_table.txt  latin_words_sorted.txt            small_text.txt
Fröberg             latin_words.txt                   swe.txt
group_album.txt     lorem80.txt
rikard@newdelli:~/bash-intro/text-files$ ls -ltr
total 68
... (cut to fit the slide)...
-rw-rw-r-- 1 rikard rikard 2073 jul 25 14:42 frequency_table.txt
-rw-rw-r-- 1 rikard rikard  131 jul 25 14:53 group_album.txt
-rw-rw-r-- 1 rikard rikard   55 jul 25 14:53 group_genre.txt
drwxrwxr-x 2 rikard rikard 4096 jul 26 11:37 apa
drwxrwxr-x 2 rikard rikard 4096 jul 29 10:02 Rikard
drwxrwxr-x 2 rikard rikard 4096 jul 29 10:02 Fröberg
```