CLI 3

Multiplexers

`screen`

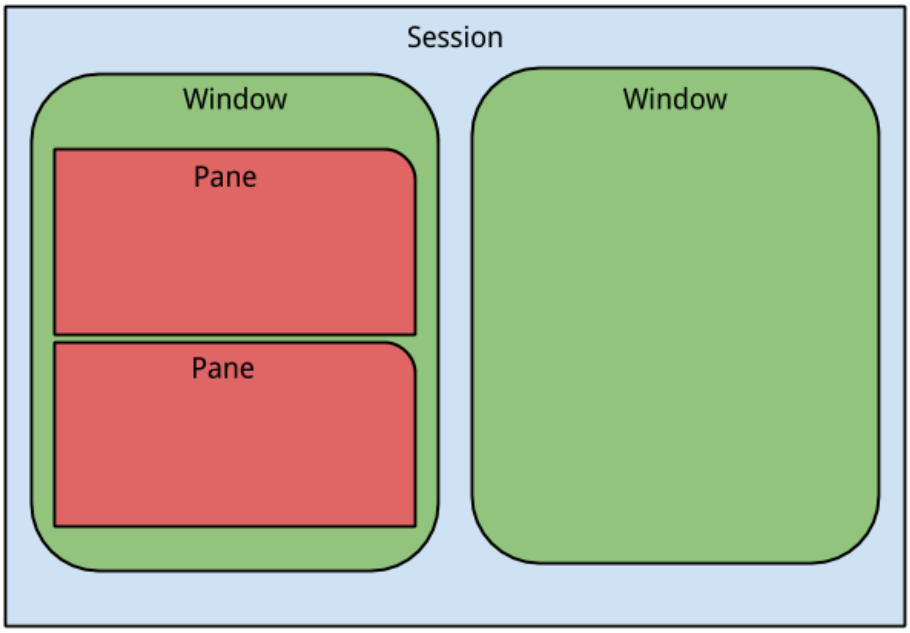
`screen` is a very useful tool that is primarily used to keep things running in the background, and to be able to regain control over these things when desired.

It can do alot more than that, but for more fancy use-cases, people usually use `tmux`. Which we will handle shortly

Running `screen` will start a new session with a shell in it.

- `Ctrl-a` followed by `d` to detach
- `screen -r` to reattach
- Multiple sessions? `screen -r [name]`. The name is reported by `screen` in this case. It doesn't have to be the whole name, the first few (unique) numbers of the PID is enough.
- `screen -list` prints all sessions. Yes, just one `-`, because standards.
- `screen -S [name]` creates a new session with the given name. So you can do `screen -r [name]` with the same name.

`tmux`



- **panes**: the different splits in your window
- **window**: like tabs in your browser or terminal
- **session**: the tmux process

Used to run multiple shells/applications in a single terminal/ssh session. You can attach to and detach a `tmux` session like `screen` and keep them running in the background.

Default prefix = `Ctrl+b` for ordering `tmux` around, can be changed.

You can *split* the current **pane** vertically with `Ctrl+b %`, and horizontally with `Ctrl+b "` These are organised hierarchically.

Switching **panes** is done with `Ctrl+b [arrowkey]`.

If you want to *fullscreen* the selected **pane**, do `Ctrl-b z`. And the same to zoom back out.

Like your favourite terminal, `tmux` has something like tabs called **windows**. To *create* a new one, to `Ctrl-b c`. You can see all the **windows** of the current **session** in the bar at the bottom. Next to their corresponding **window number**.

Switching **window**: `Ctrl-b [window number]`

Closing everything(including the shell) in the **window/page** will close it aswell.

Resizing the current pane is with `Ctrl+b Ctrl+[arrowkey]`. You can hold `Ctrl-[arrowkey]` to keep resizing in that direction.

Managing sessions

`tmux new` will create a new session (the `new` is usually optional)

`tmux new -s [session-name]` to give the new session a name. Or you can rename the current session with `Ctrl-b $`.

`tmux a(ttach)` will attach to the newest session. Even if it is already attached! Which is very cool.

`tmux a -t [session-name]` to attach to the session with the given name.

`tmux detach` or `Ctrl-b d` from within the session will make you detach from it and keep it running in the background.

`Ctrl-b s` will give you a list with sessions, `q` will close this view, using the arrow keys and enter, you can switch session easily.

List of `tmux`commands:

These are to be used inside `tmux`, and need the prefix.

Basics

- `?` get help

Session management

- `s` list sessions
- `$` rename the current session
- `d` detach from the current session

Windows

- `c` create a new window
- `,` rename the current window
- `w` list windows
- `%` split horizontally
- `"` split vertically
- `n` change to the next window
- `p` change to the previous window
- `0-9` select windows 0 through 9

Panes

- `%` create a horizontal pane
- `"` create a vertical pane
- `[ARROWKEY]` move to other pane

- `q` show pane numbers
- `o` toggle between panes
- `}` swap with next pane
- `{` swap with previous pane
- `!` break the pane out of the window
- `x` kill the current pane

Miscellaneous

- `t` show the time in current pane

Bash Scripting

It all started with a sha-bang (#!)

The first line of a bash script should be `#!/bin/bash`. This is usually not required just a list of commands.

It tells the system that this a script, to be executed with `bash`, python scripts usually have something like this: `#!/usr/bin/python` for example.

Don't forget to `chmod +x [script-file]` your script.

Arguments

If you run a script like this: `./script.sh arg1 arg2 arg3...`, the arguments can be accessed from inside the script with `$1`, `$2`, `$2`,.. And `$@` contains all of them, seperated with a whitespace.

```
#!/bin/bash
# content of script.sh
echo $1 $3 # three, not two
echo "$@"
```

```
$ ./script.sh a    b c # note the extra spaces
a c
a b c
```

Functions

Functions are like tiny scripts defined inside another script or file.

```
function functionname {
  commands
}
```

```
functionname() {
  commands
}
```

Both ways are correct, the second one has better compatibility across shells. In most programming languages, parameters are put between the round brackets, in `bash`, this is not the case.

Parameters are accessed the same way as arguments. And functions are called the same way as any other command.

```
#!/bin/bash
# content of script.sh
f() {
  echo $2
}
f first second third
```

```
$ ./script.sh
second
```

Asynchronous functions

Like any other command, functions can be called asynchronously.

Let's do something stupid.

```
#!/bin/bash
# content of sleepsort.sh
f() {
  sleep "$1"
  echo "$1"
}
for n in $@; do
  if [[ -n $n ]]; then
    f $n &
  fi
done
wait
```

```
$ ./sleepsort.sh 3 5 2 1
1
2
3
5
```

BTW `wait [pid]` blocks untill the process with the given PID has terminated. If no PID is given, it waits for all commands started by the shell to be terminated.

Local variables

The variables used by functions are shared globally, by the entire script. If you want variables that only mean something inside the function, you use `local`.

```
#!/bin/bash
# content of script.sh
f() {
  a="bye"
}
f; echo $a
a="hello"
echo $a
f; echo $a
```

```
$ ./script.sh
bye
hello
bye
```

```
#!/bin/bash
# content of script.sh
f() {
  local a="hello" # the global a will be ignored
  echo $a
}
a="bye"
f
echo $a
```

```
$ ./script.sh
hello
bye
```

`source` and `.bashrc`

`source [script]` will not execute the script, but instantiate the functions and variables from the script in the current shell(script).

```
#!/bin/bash
# content of library.sh
s() {
  sleep "$1"
  echo "$1"
}
sleepsort () {
  for n in $@; do
```

```
    if [[ -n $n ]]; then
      f $n &
    fi
  done
  wait
}
greeting="sorting..."
```

```
#!/bin/bash
# content of script.sh
source library.sh
echo $greeting
sleepsort 3 2
```

```
$ ./script.sh
sorting...
2
3
```

.bashrc and aliases

This file will automatically get `source`'d when starting your shell. It propably already exists and is located inside your home folder.

Here, you can define your own functions/variables so that they are always available in your shell.

Often, people put `alias`es in here. An alias is simpler than a function. When invoked, the alias is just replaced by it's content.

```
alias erc='vim $HOME/.bashrc'
```

They are often used to to create shorter version of long command you often use. Or change the default behaviour or others.

```
alias ls='ls -lh'
```

This wouldn't work with functions, because they will be called recursively.