In this lecture we will go through several ways in which you can improve your workflow when using the shell. We have been working with the shell for a while now, but we have mainly focused on executing different commands. We will now see how to run several processes at the same time while keeping track of them, how to stop or pause a specific process and how to make a process run in the background.

We will also learn about different ways to improve your shell and other tools, by defining aliases and configuring them using dotfiles. Both of these can help you save time, e.g. by using the same configurations in all your machines without having to type long commands. We will look at how to work with remote machines using SSH.

Job Control

In some cases you will need to interrupt a job while it is executing, for instance if a command is taking too long to complete (such as a `find` with a very large directory structure to search through).
Most of the time, you can do `Ctrl-C` and the command will stop.
But how does this actually work and why does it sometimes fail to stop the process?

Killing a process

Your shell is using a UNIX communication mechanism called a *signal* to communicate information to the process. When a process receives a signal it stops its execution, deals with the signal and potentially changes the flow of execution based on the information that the signal delivered. For this reason, signals are *software interrupts*.

In our case, when typing `Ctrl-C` this prompts the shell to deliver a `SIGINT` signal to the process.

Here's a minimal example of a Python program that captures `SIGINT` and ignores it, no longer stopping. To kill this program we can now use the `SIGQUIT` signal instead, by typing `Ctrl-\`.

```python
#!/usr/bin/env python
import signal, time

def handler(signum, time):
    print("\nI got a SIGINT, but I am not stopping")

signal.signal(signal.SIGINT, handler)
i = 0
while True:
    time.sleep(.1)
    print("\r{}".format(i), end="")
    i += 1
```

Here's what happens if we send `SIGINT` twice to this program, followed by `SIGQUIT`. Note that `^` is how `Ctrl` is displayed when typed in the terminal.

```
$ python sigint.py
24^C
I got a SIGINT, but I am not stopping
26^C
I got a SIGINT, but I am not stopping
30^\[1]    39913 quit       python sigint.py
```

While `SIGINT` and `SIGQUIT` are both usually associated with terminal related requests, a more generic signal for asking a process to exit gracefully is the `SIGTERM` signal.
To send this signal we can use the `kill` command, with the syntax `kill -TERM <PID>`.

Pausing and backgrounding processes

Signals can do other things beyond killing a process. For instance, `SIGSTOP` pauses a process. In the terminal, typing `Ctrl-Z` will prompt the shell to send a `SIGTSTP` signal, short for Terminal Stop (i.e. the terminal's version of `SIGSTOP`).

We can then continue the paused job in the foreground or in the background using `fg` or `bg`, respectively.

The `jobs` command lists the unfinished jobs associated with the current terminal session.
You can refer to those jobs using their pid (you can use `pgrep` to find that out).
More intuitively, you can also refer to a process using the percent symbol followed by its job number (displayed by `jobs`). To refer to the last backgrounded job you can use the `$!` special parameter.

One more thing to know is that the `&` suffix in a command will run the command in the background, giving you the prompt back, although it will still use the shell's STDOUT which can be annoying (use shell redirections in that case).

To background an already running program you can do `Ctrl-Z` followed by `bg`.
Note that backgrounded processes are still children processes of your terminal and will die if you close the terminal (this will send yet another signal, `SIGHUP`).
To prevent that from happening you can run the program with `nohup` (a wrapper to ignore `SIGHUP`), or use `disown` if the process has already been started.
Alternatively, you can use a terminal multiplexer as we will see in the next section.

Below is a sample session to showcase some of these concepts.

```
$ sleep 1000
^Z
[1]  + 18653 suspended  sleep 1000

$ nohup sleep 2000 &
[2] 18745
appending output to nohup.out

$ jobs
[1]  + suspended  sleep 1000
[2]  - running    nohup sleep 2000

$ bg %1
[1]  - 18653 continued  sleep 1000
```

```
$ jobs
[1]  - running    sleep 1000
[2]  + running    nohup sleep 2000

$ kill -STOP %1
[1]  + 18653 suspended (signal)  sleep 1000

$ jobs
[1]  + suspended (signal)  sleep 1000
[2]  - running    nohup sleep 2000

$ kill -SIGHUP %1
[1]  + 18653 hangup     sleep 1000

$ jobs
[2]  + running    nohup sleep 2000

$ kill -SIGHUP %2

$ jobs
[2]  + running    nohup sleep 2000

$ kill %2
[2]  + 18745 terminated  nohup sleep 2000

$ jobs
```

A special signal is `SIGKILL` since it cannot be captured by the process and it will always terminate it immediately. However, it can have bad side effects such as leaving orphaned children processes.

You can learn more about these and other signals [here]() or typing [`man signal`]() or `kill -l`.

Terminal Multiplexers

When using the command line interface you will often want to run more than one thing at once.
For instance, you might want to run your editor and your program side by side.
Although this can be achieved by opening new terminal windows, using a terminal multiplexer is a more versatile solution.

Terminal multiplexers like [`tmux`]() allow you to multiplex terminal windows using panes and tabs so you can interact with multiple shell sessions.
Moreover, terminal multiplexers let you detach a current terminal session and reattach at some point later in time.
This can make your workflow much better when working with remote machines since it avoids the need to use `nohup` and similar tricks.

The most popular terminal multiplexer these days is [`tmux`](). `tmux` is highly configurable and by using the associated keybindings you can create multiple tabs and panes and quickly navigate through them.

`tmux` expects you to know its keybindings, and they all have the form `<C-b> x` where that means (1) press `Ctrl+b`, (2) release `Ctrl+b`, and then (3) press `x`. `tmux` has the following hierarchy of objects:

- **Sessions** - a session is an independent workspace with one or more windows
  - `tmux` starts a new session.
  - `tmux new -s NAME` starts it with that name.
  - `tmux ls` lists the current sessions
  - Within `tmux` typing `<C-b> d` detaches the current session
  - `tmux a` attaches the last session. You can use `-t` flag to specify which
- **Windows** - Equivalent to tabs in editors or browsers, they are visually separate parts of the same session
  - `<C-b> c` Creates a new window. To close it you can just terminate the shells doing `<C-d>`
  - `<C-b> N` Go to the *N* th window. Note they are numbered
  - `<C-b> p` Goes to the previous window
  - `<C-b> n` Goes to the next window
  - `<C-b> ,` Rename the current window
  - `<C-b> w` List current windows
- **Panes** - Like vim splits, panes let you have multiple shells in the same visual display.
  - `<C-b> "` Split the current pane horizontally
  - `<C-b> %` Split the current pane vertically
  - `<C-b> <direction>` Move to the pane in the specified *direction*. Direction here means arrow keys.
  - `<C-b> z` Toggle zoom for the current pane
  - `<C-b> [` Start scrollback. You can then press `<space>` to start a selection and `<enter>` to copy that selection.
  - `<C-b> <space>` Cycle through pane arrangements.

For further reading,
here is a quick tutorial on `tmux` and this has a more detailed explanation that covers the original `screen` command. You might also want to familiarize yourself with `screen`, since it comes installed in most UNIX systems.

Aliases

It can become tiresome typing long commands that involve many flags or verbose options.
For this reason, most shells support *aliasing*.
A shell alias is a short form for another command that your shell will replace automatically for you.
For instance, an alias in bash has the following structure:

```
alias alias_name="command_to_alias arg1 arg2"
```

Note that there is no space around the equal sign `=`, because `alias` is a shell command that takes a single argument.

Aliases have many convenient features:

```
# Make shorthands for common flags
alias ll="ls -lh"

# Save a lot of typing for common commands
alias gs="git status"
alias gc="git commit"
alias v="vim"

# Save you from mistyping
alias sl=ls

# Overwrite existing commands for better defaults
alias mv="mv -i"           # -i prompts before overwrite
alias mkdir="mkdir -p"     # -p make parent dirs as needed
alias df="df -h"           # -h prints human readable format

# Alias can be composed
alias la="ls -A"
alias lla="la -l"

# To ignore an alias run it prepended with \
\ls
# Or disable an alias altogether with unalias
unalias la

# To get an alias definition just call it with alias
alias ll
# Will print ll='ls -lh'
```

Note that aliases do not persist shell sessions by default.
To make an alias persistent you need to include it in shell startup files, like `.bashrc` or `.zshrc`, which we are going to introduce in the next section.

Dotfiles

Many programs are configured using plain-text files known as *dotfiles*
(because the file names begin with a `.`, e.g. `~/.vimrc`, so that they are
hidden in the directory listing `ls` by default).

Shells are one example of programs configured with such files. On startup, your shell will read many files to load its configuration.
Depending on the shell, whether you are starting a login and/or interactive the entire process can be quite complex.
[Here](#) is an excellent resource on the topic.

For `bash`, editing your `.bashrc` or `.bash_profile` will work in most systems.
Here you can include commands that you want to run on startup, like the alias we just described or modifications to your `PATH` environment variable.
In fact, many programs will ask you to include a line like `export PATH="$PATH:/path/to/program/bin"` in

your shell configuration file so their binaries can be found.

Some other examples of tools that can be configured through dotfiles are:

- `bash` - `~/.bashrc`, `~/.bash_profile`
- `git` - `~/.gitconfig`
- `vim` - `~/.vimrc` and the `~/.vim` folder
- `ssh` - `~/.ssh/config`
- `tmux` - `~/.tmux.conf`

How should you organize your dotfiles? They should be in their own folder, under version control, and **symlinked** into place using a script. This has the benefits of:

- **Easy installation**: if you log in to a new machine, applying your customizations will only take a minute.
- **Portability**: your tools will work the same way everywhere.
- **Synchronization**: you can update your dotfiles anywhere and keep them all in sync.
- **Change tracking**: you're probably going to be maintaining your dotfiles for your entire programming career, and version history is nice to have for long-lived projects.

What should you put in your dotfiles?
You can learn about your tool's settings by reading online documentation or man pages. Another great way is to search the internet for blog posts about specific programs, where authors will tell you about their preferred customizations. Yet another way to learn about customizations is to look through other people's dotfiles: you can find tons of dotfiles repositories on GitHub --- see the most popular one here (we advise you not to blindly copy configurations though).
Here is another good resource on the topic.

All of the class instructors have their dotfiles publicly accessible on GitHub: Anish, Jon, Jose.

Portability

A common pain with dotfiles is that the configurations might not work when working with several machines, e.g. if they have different operating systems or shells. Sometimes you also want some configuration to be applied only in a given machine.

There are some tricks for making this easier.
If the configuration file supports it, use the equivalent of if-statements to apply machine specific customizations. For example, your shell could have something like:

```
if [[ "$(uname)" == "Linux" ]]; then {do_something}; fi

# Check before using shell-specific features
if [[ "$SHELL" == "zsh" ]]; then {do_something}; fi

# You can also make it machine-specific
if [[ "$(hostname)" == "myServer" ]]; then {do_something}; fi
```

If the configuration file supports it, make use of includes. For example, a `~/.gitconfig` can have a setting:

```
[include]
    path = ~/.gitconfig_local
```

And then on each machine, `~/.gitconfig_local` can contain machine-specific settings. You could even track these in a separate repository for machine-specific settings.

This idea is also useful if you want different programs to share some configurations. For instance, if you want both `bash` and `zsh` to share the same set of aliases you can write them under `.aliases` and have the following block in both:

```
# Test if ~/.aliases exists and source it
if [ -f ~/.aliases ]; then
    source ~/.aliases
fi
```

Remote Machines

It has become more and more common for programmers to use remote servers in their everyday work. If you need to use remote servers in order to deploy backend software or you need a server with higher computational capabilities, you will end up using a Secure Shell (SSH). As with most tools covered, SSH is highly configurable so it is worth learning about it.

To `ssh` into a server you execute a command as follows

```
ssh foo@bar.mit.edu
```

Here we are trying to ssh as user `foo` in server `bar.mit.edu`.
The server can be specified with a URL (like `bar.mit.edu`) or an IP (something like `foobar@192.168.1.42`).
Later we will see that if we modify ssh config file you can access just using something like `ssh bar`.

Executing commands

An often overlooked feature of `ssh` is the ability to run commands directly.
`ssh foobar@server ls` will execute `ls` in the home folder of foobar.
It works with pipes, so `ssh foobar@server ls | grep PATTERN` will grep locally the remote output of `ls` and `ls | ssh foobar@server grep PATTERN` will grep remotely the local output of `ls`.

SSH Keys

Key-based authentication exploits public-key cryptography to prove to the server that the client owns the secret private key without revealing the key. This way you do not need to reenter your password every time. Nevertheless, the private key (often `~/.ssh/id_rsa` and more recently `~/.ssh/id_ed25519`) is effectively your password, so treat it like so.

Key generation

To generate a pair you can run [`ssh-keygen`].

```
ssh-keygen -a 100 -t ed25519 -f ~/.ssh/id_ed25519
```

You should choose a passphrase, to avoid someone who gets hold of your private key to access authorized servers. Use [`ssh-agent`] or [`gpg-agent`] so you do not have to type your passphrase every time.

If you have ever configured pushing to GitHub using SSH keys, then you have probably done the steps outlined [here] and have a valid key pair already. To check if you have a passphrase and validate it you can run `ssh-keygen -y -f /path/to/key`.

Key based authentication

`ssh` will look into `.ssh/authorized_keys` to determine which clients it should let in. To copy a public key over you can use:

```
cat .ssh/id_ed25519.pub | ssh foobar@remote 'cat >> ~/.ssh/authorized_keys'
```

A simpler solution can be achieved with `ssh-copy-id` where available:

```
ssh-copy-id -i .ssh/id_ed25519 foobar@remote
```

Copying files over SSH

There are many ways to copy files over ssh:

- `ssh+tee`, the simplest is to use `ssh` command execution and STDIN input by doing `cat localfile | ssh remote_server tee serverfile`. Recall that [`tee`] writes the output from STDIN into a file.
- [`scp`] when copying large amounts of files/directories, the secure copy `scp` command is more convenient since it can easily recurse over paths. The syntax is `scp path/to/local_file remote_host:path/to/remote_file`

- `rsync` improves upon `scp` by detecting identical files in local and remote, and preventing copying them again. It also provides more fine grained control over symlinks, permissions and has extra features like the `--partial` flag that can resume from a previously interrupted copy. `rsync` has a similar syntax to `scp`.

Port Forwarding

In many scenarios you will run into software that listens to specific ports in the machine. When this happens in your local machine you can type `localhost:PORT` or `127.0.0.1:PORT`, but what do you do with a remote server that does not have its ports directly available through the network/internet?.

This is called *port forwarding* and it
comes in two flavors: Local Port Forwarding and Remote Port Forwarding (see the pictures for more details, credit of the pictures from this StackOverflow post).

Local Port Forwarding

Remote Port Forwarding

The most common scenario is local port forwarding, where a service in the remote machine listens in a port and you want to link a port in your local machine to forward to the remote port. For example, if we execute `jupyter notebook` in the remote server that listens to the port `8888`. Thus, to forward that to the local port `9999`, we would do `ssh -L 9999:localhost:8888 foobar@remote_server` and then navigate to `localhost:9999` in our local machine.

SSH Configuration

We have covered many many arguments that we can pass. A tempting alternative is to create shell aliases that look like

```
alias my_server="ssh -i ~/.id_ed25519 --port 2222 -L 9999:localhost:8888
foobar@remote_server"
```

However, there is a better alternative using `~/.ssh/config`.

```
Host vm
    User foobar
    HostName 172.16.174.141
    Port 2222
    IdentityFile ~/.ssh/id_ed25519
    LocalForward 9999 localhost:8888

# Configs can also take wildcards
Host *.mit.edu
    User foobaz
```

An additional advantage of using the `~/.ssh/config` file over aliases is that other programs like `scp`, `rsync`, `mosh`, &c are able to read it as well and convert the settings into the corresponding flags.

Note that the `~/.ssh/config` file can be considered a dotfile, and in general it is fine for it to be included with the rest of your dotfiles. However, if you make it public, think about the information that you are potentially providing strangers on the internet: addresses of your servers, users, open ports, &c. This may facilitate some types of attacks so be thoughtful about sharing your SSH configuration.

Server side configuration is usually specified in `/etc/ssh/sshd_config`. Here you can make changes like disabling password authentication, changing ssh ports, enabling X11 forwarding, &c. You can specify config settings on a per user basis.

Miscellaneous

A common pain when connecting to a remote server are disconnections due to your computer shutting down, going to sleep, or changing networks. Moreover if one has a connection with significant lag using ssh can become quite frustrating. Mosh, the mobile shell, improves upon ssh, allowing roaming connections, intermittent connectivity and providing intelligent local echo.

Sometimes it is convenient to mount a remote folder. sshfs can mount a folder on a remote server locally, and then you can use a local editor.

Shells & Frameworks

During shell tool and scripting we covered the `bash` shell because it is by far the most ubiquitous shell and most systems have it as the default option. Nevertheless, it is not the only option.

For example, the `zsh` shell is a superset of `bash` and provides many convenient features out of the box such as:

- Smarter globbing, `**`
- Inline globbing/wildcard expansion
- Spelling correction
- Better tab completion/selection
- Path expansion (`cd /u/lo/b` will expand as `/usr/local/bin`)

Frameworks can improve your shell as well. Some popular general frameworks are prezto or oh-my-zsh, and smaller ones that focus on specific features such as zsh-syntax-highlighting or zsh-history-substring-search. Shells like fish include many of these user-friendly features by default. Some of these features include:

- Right prompt
- Command syntax highlighting
- History substring search
- manpage based flag completions
- Smarter autocompletion
- Prompt themes

One thing to note when using these frameworks is that they may slow down your shell, especially if the code they run is not properly optimized or it is too much code. You can always profile it and disable the features that you do not use often or value over speed.

Terminal Emulators

Along with customizing your shell, it is worth spending some time figuring out your choice of **terminal emulator** and its settings. There are many many terminal emulators out there (here is a comparison).

Since you might be spending hundreds to thousands of hours in your terminal it pays off to look into its settings. Some of the aspects that you may want to modify in your terminal include:

- Font choice
- Color Scheme
- Keyboard shortcuts
- Tab/Pane support
- Scrollback configuration
- Performance (some newer terminals like Alacritty or kitty offer GPU acceleration).

Exercises

Job control

1. From what we have seen, we can use some `ps aux | grep` commands to get our jobs' pids and then kill them, but there are better ways to do it. Start a `sleep 10000` job in a terminal, background it with `Ctrl-Z` and continue its execution with `bg`. Now use `pgrep` to find its pid and `pkill` to kill it without ever typing the pid itself. (Hint: use the `-af` flags).
2. Say you don't want to start a process until another completes. How would you go about it? In this exercise, our limiting process will always be `sleep 60 &`.
   One way to achieve this is to use the `wait` command. Try launching the sleep command and having an `ls` wait until the background process finishes.

   However, this strategy will fail if we start in a different bash session, since `wait` only works for child processes. One feature we did not discuss in the notes is that the `kill` command's exit status will be zero on success and nonzero otherwise. `kill -0` does not send a signal but will give a nonzero exit status if the process does not exist.
   Write a bash function called `pidwait` that takes a pid and waits until the given process completes. You should use `sleep` to avoid wasting CPU unnecessarily.

Terminal multiplexer

1. Follow this `tmux` tutorial and then learn how to do some basic customizations following these steps.

Aliases

1. Create an alias `dc` that resolves to `cd` for when you type it wrongly.
2. Run `history | awk '{$1="";print substr($0,2)}' | sort | uniq -c | sort -n | tail -n 10` to get your top 10 most used commands and consider writing shorter aliases for them. Note: this works for Bash; if you're using ZSH, use `history 1` instead of just `history`.

Dotfiles

Let's get you up to speed with dotfiles.

1. Create a folder for your dotfiles and set up version control.
2. Add a configuration for at least one program, e.g. your shell, with some customization (to start off, it can be something as simple as customizing your shell prompt by setting `$PS1`).
3. Set up a method to install your dotfiles quickly (and without manual effort) on a new machine. This can be as simple as a shell script that calls `ln -s` for each file, or you could use a specialized utility.
4. Test your installation script on a fresh virtual machine.
5. Migrate all of your current tool configurations to your dotfiles repository.
6. Publish your dotfiles on GitHub.

Remote Machines

Install a Linux virtual machine (or use an already existing one) for this exercise. If you are not familiar with virtual machines check out this tutorial for installing one.

1. Go to `~/.ssh/` and check if you have a pair of SSH keys there. If not, generate them with `ssh-keygen -a 100 -t ed25519`. It is recommended that you use a password and use `ssh-agent` , more info here.
2. Edit `.ssh/config` to have an entry as follows

```
Host vm
    User username_goes_here
    HostName ip_goes_here
    IdentityFile ~/.ssh/id_ed25519
    LocalForward 9999 localhost:8888
```

3. Use `ssh-copy-id vm` to copy your ssh key to the server.
4. Start a webserver in your VM by executing `python -m http.server 8888`. Access the VM webserver by navigating to `http://localhost:9999` in your machine.
5. Edit your SSH server config by doing `sudo vim /etc/ssh/sshd_config` and disable password authentication by editing the value of `PasswordAuthentication`. Disable root login by editing the value of `PermitRootLogin`. Restart the `ssh` service with `sudo service sshd restart`. Try sshing in again.
6. (Challenge) Install `mosh` in the VM and establish a connection. Then disconnect the network adapter of the server/VM. Can mosh properly recover from it?
7. (Challenge) Look into what the `-N` and `-f` flags do in `ssh` and figure out a command to achieve background port forwarding.