# Missing Semester of CS Notes

Sean Wu

May 5, 2020

## Contents

# 1 The Shell - Bash

## 1.1 Paths

- Cmd line arguments separated by whitespace
- Use quotes `" "` or escape the space `\`

**environment variable:** variable set whenever shell starts (not every run of shell)

- ex. home dir, username, `PATH` variable
- Comments in bash start with `#`

```
echo $PATH # all file paths that bash will search for programs
# OUTPUT: colon-separated list
```

- Whenever name of program (ex. `echo`) is typed, bash will search through this list in `PATH` , looking in each directory for the program matching the command

```
which echo # tells you where file for command is located (ex. echo)
```

**paths:** way to name location of file on computer

- Paths separated by forward slashes `/` for UNIX and backslashes `\` for Windows

`/` root; top of file system

- On UNIX, everything is under the root `/` namespace
- i.e. all absolute paths start with `/`
- On Windows, there is one root for every partition
- ex. `C:\`, `D:\`
- i.e. separate file system path hierarchies for each drive

**absolute path:** fully determines location of file
**relative path:** path relative to your current working directory
`.` current directory
`..` parent directory
`~` home directory
`-` directory you were just in

## 1.2 Flags and Options

- Flags and options specified after the program name
- The short form is usually with single slashes `-<char>` and the long form is usually with double dashes `--<word>`
- ex. `-v` and `--version` tell you the version of the program
- ex. `-h` and `--help` give you a quick help guide for the program
- Running command with `--help` flag gives you the `usage` in the following format

```
usage: ls [OPTION] ... [FILE] ...
# [] means optional
# ... means 1 or more of the previous thing
```

**flag:** doesn't take a value (usually)
**option:** takes a value (usually)

## 1.3 File Permissions

- Get file permissions by running `ls -a`
- Permissions specified in 3 groups of 3 (r, w, x)

1. 1st group of 3 permissions is for owner of file
2. 2nd group of 3 permissions is for the group of people owning the file
3. 3rd group of 3 permissions is for everyone else

- Note: if you have write access on a file but read access on a directory, you cannot directly delete a file (can only empty it)

**For files:**

**-** don't have that permission
**r** read access
**w** write access
**x** execute acess

**For folders:**

**-** don't have that permission
**r** can see files inside directory
**w** can rename, create, remove files

**x** can search this directory (i.e. enter directory with `cd`)

`chmod` : command to change file modes or Access Control Lists (i.e. change permissions)

## 1.4 Deleting things

`rm` : removes a file

- By default, `rm` is **not** recursive on UNIX (i.e. cannot remove a directory)
- Add a `-r` (recursive) flag to delete a directory
- Recursive delete removes everything under the path you give it

`rmdir` : deletes a directory only if it is empty (a safe delete)
`cmd L` : clears terminal output to previous mark
`cmd K` : clears terminal to start

## 1.5 Input and Output Streams

- Each program has 2 primary streams

1. Input stream: terminal by default
2. Output stream: terminal by default

`<` : rewire input of previous program to be the contents of this file on the right
`>` : rewire output of previous program into this file
`>>` : appends to the end of a file instead of overwriting

```
echo hello > hello.txt # writes string "hello" into file hello.txt
```

`|` : a **pipe**; takes the output of program on left and makes it the input of the program on the right. **Input program does not know about output program and vice versa** . The programs just read and write to those spots.

## 1.6 Root User (UNIX)

- Acts like admin user on Windows

- Has user id 0
- Has all permissions (Superuser)

`sudo` : does the following command as superuser (root user)

**kernel:** core of computer

`sysfs` : file system for kernel parameters of computer

- Need to be admin to change kernel params of a computer
- Note: if using `sudo` with pipes and redirects, `sudo` only applies to one portion (because input and output programs don't know about each other)

`$` indicates that you are **not** running as root

`#` indicates that you are running as root

```
sudo echo 500 > brightness
# does not work because brightness doesn't know about sudo
```

`sudo su` gives you a shell as superuser (shell runs as root now)

`exit` allows you to exit out of superuser shell mode

## 1.7 Misc. Helpful Commands

`man` gives you the manual pages for a program

`tail` gives you the last n lines of a file

```
tail -n5 # gives you the last 5 lines of a file
```

`tee` writes to output and to terminal output

```
echo 1000 | sudo tee brightness # changes brightness
# Note: this can be run without using superuser terminal
```

`xdg-open` opens file (Linux)

`open` opens file (macOS)

`source` reads and executes commands from the file specified as its argument in the current shell environment. Useful to load functions, variables and configuration files into shell scripts. It has a synonym in `.` (period).

```
. filename [arguments]
source filename [arguments]
```

```
# Note that ./ and source are not the same
./script
# runs the script as an executable file, launching a new shell to
# run it

source script
# reads and executes commands from filename in the current shell
# environment
# Note: ./script is not . script, but . script == source script
```

### 1.8 Executable and UNIX Shebang

**shebang:** a character sequence involving `#!` at the beginning of a script

- A shebang `#!` indicates that a file is an executable in UNIX

```
#!/bin/sh
curl --head --silent https://missing.csail.mit.edu

# First line indicates that program loader should run the
# program /bin/sh, passing path/to/script (name of this file)
# as the first argument.
```

## 2 Shell Tools and Scripting

### 2.1 Defining Variables

```
foo=bar # make var foo store the value bar
echo $foo # OUTPUT: bar (the value of the foo)
foo = bar # will not work bec of spaces
# interprets as foo being the command with = and bar being args
# Note: spaces reserved in bash for separating CLI args
```

## 2.2 Defining Strings

```
echo "Hello" # OUTPUT: Hello
echo 'World' # OUTPUT: World (literal string for '')
# Note: for literal strings, double "" and single quotes ''
# are equivalent
```

```
echo "value is $foo" # OUTPUT: value is bar
# variable $foo will be expanded in string for double quotes ""
echo 'value is $foo' # OUTPUT: value is $foo
# outputs string characters as displayed for single quotes ''
# doesn't expand $foo
```

## 2.3 Defining Functions

```
# mcd.sh, a command to make a new dir and switch to it
mcd () {
  mkdir -p "$1" # $1 is a special var for 1st CLI arg
  cd "$1"
}
```

```
source mcd.sh # executes the script mcd.sh
# new mcd function has been defined in shell
# can now do
mcd test
```

## 2.4 Special Bash Variables

`$0` : name of script

`$1` : 1st CLI arg

`$2 to $9` : 2nd to 9th arg

`$@` : expands to all args

`$#` : number of args given to current command

`$?` : gets error code from previous command

`$_` : last arg of previous command

`!!` : **bang bang**; Entire last command, including arguments. Usually used when you don't have permission (expands to previous command)

`$$` : Process Identification number for the current script

```
mkdir /mnt/new # Permission denied
sudo !! # becomes equivalent to
sudo mkdir /mnt/new
```

## 2.5 Commands and Exit Codes

- Commands often return output using `STDOUT`, errors through `STDERR` and a Return Code to report errors in a more script friendly manner
- Return code or exit status are used by scripts/commands to communicate how execution went

`0` : no issue; everything went OK

`1` or any number: error or issue with running command

```
echo "Hello" # OUTPUT: Hello
echo $? # OUTPUT: 0
```

```
grep foobar mcd.sh # no output
echo $? # OUTPUT: 1
# bash tried to search for foobar string in mcd script but it
# wasn't there (an error occurred)
```

## 2.6 Boolean Logic

- Note: `true` and `false` always have `0` and `1` error codes

```
true
echo $? # OUTPUT: 0
false
echo $? # OUTPUT: 1
```

## 2.7 Logical Operators

- Exit codes can be used to conditionally execute commands using `&&` and `||`

`||` : **OR operator**; executes 1st commmand and if it fails, it executes the (i.e. 1st command did not have a 0 error code) 2nd command

`&&` : **AND operator**; will only execute the 2nd command if the 1st one runs w/out error codes (i.e. 1st command had a 0 error code)

```
false || echo "oops fail" # OUTPUT: oops fail
# bash ran 2nd command bec the 1st command has an error code of 1
true || echo "Will not be printed" # no output
# bash didn't run the 2nd command bec the 1st command has an
# error code of 0
```

```
true && echo "Things went well" # OUTPUT: Things went well
false && echo "This will not print"
```

`;` can concatenate commands in the same line with a semicolon `;`

```
false; echo "This always prints" # OUTPUT: This always prints
```

## 2.8 Command Substitution

- Command substitution is used to get the output of a command as a variable

`$(cmd)` : will execute `cmd`, get the output of the command (stored in a **variable**) and substitute it in place.

```
foo=$(pwd) # gets output of pwd and stores it in foo variable
echo $foo
```

```
echo "We are in $(pwd)" # OUTPUT: We are in /Users/admin/Documents
# Note: $(pwd) is expanded because we are using double quotes ""
```

## 2.9 Process Substitution

- Process substitution is useful when commands expect values to be passed by file instead of by STDIN

`<(cmd)` : will execute `cmd` and place the output in a **temporary file** and substitute the `<()` with that file's name

```
cat <(ls) <(ls ..) # OUTPUT: prints files in current dir and then
# files in parent dir
# ls-ing both current and parent directories and then storing
#output in temp file using process substitution <(cmd)
# cat then reads the output of the temp file
```

`/dev/null` : special UNIX null register used to discard data that we do not care about

`>` : redirects standard output `STDOUT`

`2>` : redirects standard error `STDERR`

```
#!/bin/bash

echo "Starting program at $(date)" # Date will be substituted

echo "Running program $0 with $# arguments with pid $$"

for file in $@; do
    grep foobar $file > /dev/null 2> /dev/null
    # When pattern is not found, grep has exit status 1
    # We redirect STDOUT and STDERR to a null register since we do
    # not care about them
    if [[ $? -ne 0 ]]; then
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
        # appends # foobar to end of file as a comment
    fi
done
```

- To see equality test flags, run `man test`
- When performing comparisons in bash try to use double brackets `[[ ]]` in favour of simple brackets `[ ]`. Chances of making mistakes are lower although it won't be portable to `sh`

## 2.10 Manipulating Files

`*` **globbing**; 0 or multiple character wildcard. When used with partial file name will expand to all files matching that pattern

`?` single character wildcard; only replaces 1 character (not 0 or more like with globbing)

`{}` used when you have a common substring that you want to expand automatically. Like for writing files with similar names but different extensions

```
ls *.sh # lists all files with .sh extension
```

```
# given files foo, foo1, foo2, foo10 and bar
rm foo? # deletes foo1 and foo2
rm foo* # deletes all except for bar
```

```
convert image.png image.jpg
convert image.{png,jpg} # equivalent to above line
# Remember: NO SPACES or else bash treats them as separate args
```

```
touch foo{,1,2,10}
touch foo foo1 foo2 foo10
```

```
# can also combine everything and at multiple levels
touch project{1,2}/src/test{1,2,3}.py

# globbing techniques can also be combined like this
mv *{.py,.sh} folder
# Will move all *.py and *.sh files
```

`..` expands into a range. `1..5` $\longrightarrow$ `1,2,3,4,5`

```
touch {foo,bar}/{a..j}
# expands into foo/a to foo/j and same with bar/a and bar/j
diff <(ls foo) <(ls bar) # compares output of 2 ls commands
```

## 2.11 Bash and Python Scripting

`#!` **shebang**; indicates that file is an executable and specifies which interpreter to use

- Can add shebang to python to make it executable from the shell

```
#!/usr/local/bin/python
# above line tells shell to use python as the interpreter
import sys
for arg in reversed(sys.argv[1:]):
  print(arg)
```

```
# can run above python file script.py as executable in shell
./script.py a b c # a,b,c are arguments passed to the script
```

```
# to avoid assuming where python is located, we can use the
# env command in python file

#!/usr/local/bin/env python
# give python as argument to env command
# output of env (location of python) becomes the interpreter
# specified by the shebang
import sys
for arg in reversed(sys.argv[1:]):
  print(arg)
```

`shellcheck` : useful CLI program to debug shell scripts; native shell doesn't give much useful error/debug statements

`tldr` : useful CLI program to get short documentation and examples for commands instead of using `man`

## 2.12 Shell Functions vs Scripts

1. Functions have to be in the same language as the shell, while scripts can be written in any language (ex. `python`)
- This is why including a shebang for scripts is important
2. Functions are loaded once when their definition is read. Scripts are loaded every time they are executed.
- This makes functions slightly faster to load but whenever you change them you will have to reload their definition
3. Functions are executed in the current shell environment whereas scripts execute in their own process
- Thus, functions can modify environment variables, e.g. change your current directory, whereas scripts can't.
4. Scripts will be passed by value environment variables that have been exported using `export`

## 2.13 Finding Files

`find` UNIX CLI tool that recursively searches thru all the files that match a certain pattern

`locate` uses a database updated using `cron` that is a a faster way of searching for files. To manually update database, run `updatedb` (Linux) or `sudo /usr/libexec/locate.updatedb` from root `/` for MacOS

- Tradeoff between `find` and `locate` is **speed vs freshness**

- Database may contain out of date info and needs to be updated

```
# Find all directories named src
find . -name src -type d
# Find all python files with a folder named test in their path
find . -path '**/test/**/*.py' -type f
# Find all files modified in the last day
find . -mtime -1
# Find all zip files with size in range 500k to 10M
find . -size +500k -size -10M -name '*.tar.gz'
```

```
# Delete all files with .tmp extension
find . -name '*.tmp' -exec rm {} \;
# Find all PNG files and convert them to JPG
find . -name '*.png' -exec convert {} {.}.jpg \;
```

## 2.14 Searching Within Files

`grep` UNIX CLI tool used for searching or matching patterns from input text

`rg` **ripgrep**; a CLI tool that improves `grep` by ignoring .git folders, using multi CPU support, etc.

Useful `grep` and `rg` flags

`-C n` gives n lines of **C**ontext around the matched string

`-v` in**v**erts the match, i.e. print all lines that do not match the pattern

`-R` **R**ecursively go into directories and look for text files for the matching string.

```
# Find all python files where I used the requests library
rg -t py 'import requests'
# Find all files (including hidden files) without a shebang line
rg -u --files-without-match "^#!"
# Find all matches of foo and print the following 5 lines
rg foo -A 5
# Print statistics of matches (# of matched lines and files )
rg --stats PATTERN
```

## 2.15 Searching Previous Shell History

`up arrow` : goes through previous commands line by line. Inefficient for very old commands

`history` : command that prints out most recent commands

`ctrl r` : backwards search fo previous command history and execute in place. Repetitive typing of `ctrl r` will give you next previous command

```
history 1 # prints all results since beginning of time
history 1 | grep convert
# search all history for commands using convert
```

### 2.16 Directory Structure

`tree` : pretty prints the directory structure

## 3 Vim Text Editor

### 3.1 Vim philosohpy

- Vim is a **modal** editor (multiple operating modes for inserting text vs manipulating text)
- Vim interface is like a programming language: keystrokes are commands and these commands can be composable
- Vim avoids use of mouse and arrow keys to speed up workflow; all vim functionality available from keyboard

### 3.2 Modal Editing

- Starts off in **normal mode**

`<ESC>` **Normal**; for moving around a file and making edits

`i` **Insert**; for inserting text

`R` **Replace**; for replacing text

`v, V, or <C-v>` **Visual (plain, line, or block)**; for selecting blocks of text

`:` **Command-line**; for running a command

- Note: `<C-v>` means Ctrl-v
- Note: keystrokes have different meanings in different modes
- Vim shows current mode in bottom left
- Usually use normal or insert mode

## 3.3 Vim buffers, tabs, and windows

- Vim maintains a set of open files called **buffers**
- A Vim session has a number of tabs, each with a number of windows (split panes)
- Each window shows only 1 buffer
- Note: a window is only a *view*
- A given buffer may be open in *multiple* windows (even in same tab)

## 3.4 Command-line

- Enter command mode by typing `:` in normal mode

`:q` quit (close window)
`:qa` close all windows and quit
`:w` save ("write")
`:wq` save and quit
`:e name of file` open file for editing
`:ls` show open buffers
`:help topic` open help
`:help :w` opens help for `:w` command
`:help w` opens help for the `w` movement

## 3.5 Movement Commands

- Spend most of the time in normal mode using movement commands (aka "nouns") to navigate the buffer
- Movements in Vim are also called "nouns", because they refer to chunks of text.

**Basic movement** `hjkl` (left, down, up, right)
**Words** `w` (next word) `b` (beginning of word), `e` (end of word)
**Lines** `0` (beginning of line), `^` (first non-blank character), `$` (end of line)
Screen `H` (top of screen), `M` (middle of screen), `L` (bottom of screen)
**Scroll** `Ctrl-u` (up), `Ctrl-d` (down)
**File** `gg` (beginning of file), `G` (end of file)
**Line numbers** `:{number}<CR>` or `{number}G` (line number)
**Editing parentheses and brackets** `\%` Jumps between matching brackets (),[]
**Find** `f{character}`, `t{character}`, `F{character}`, `T{character}` find/to forward/backward character on the current line `,` or `;` for navigating matches

**Search** : `/{regex}`, `n` or `N` for navigating matches

## 3.6 Text Selection

- Visual modes

1. Visual
2. Visual Line
3. Visual Block

- Can use movement keys in these modes to select text

## 3.7 Editing

- Vim's editing commands are also called "verbs" because verbs act on nouns

`i` enter insert mode
`o or O` insert line below/above
`d{motion}` delete motion
`dw` delete word
`d$` delete to end of line
`d0` delete to beginning of line
`c{motion}` change motion; like `d{motion}` followed by `i`
`cw` change word
`x` delete character (equal do
`dl` )
`s` substitute character (equal to
`xi` )
`u` undo
`<C-r>` redo
`y` to copy / "yank"
`p` paste
~ flips the case of a character

## 3.8 Repeated Actions with Counts

- Can combine nouns (movement command) and verbs (editing command) with a count
- Performs a given action a number of times

`3w` move 3 words forward

`5j` move 5 lines down

`7dw` delete 7 words

- Note: repeating a character twice applies that command to a whole line
- ex. `dd` deletes a whole line

### 3.9 Modifiers

- Can use modifiers to change meaning of a noun (movment command)
- ex. the `i` modifier means "inner" or "inside" and the `a` modifier means "around"

`ci(` change the contents inside the current pair of parentheses

`ci[` change the contents inside the current pair of square brackets

`da'` delete a single-quoted string, including the surrounding single quotes

### 3.10 Search and Replace

`:s` substitute

`%s/foo/bar/g` replace foo with bar globally in file

`%s/[.*]((.*))/1/g` replace named Markdown links with plain URLs

### 3.11 Multiple Windows

`:sp or :vsp` to split windows

`:tabnew` new tab

- Can have multiple views of the same buffer.

## 4 Data Wrangling

`journalctl` : view system logs

`ssh` : access computers remotely through commmand-line

`sed` : stream editor; make changes to stream. Usually use it to run replacement commands on input stream

`less` : pager to scroll through output and view data

- Can specify which commands to run on server when using `ssh` by using single quotes `''`

```
# Read server logs to see who is trying to log in
# This command uses pipes to stream a remote file through grep
# on local computer
ssh myserver journalctl | grep sshd

# This does filtering on the server and then displays data locally
# with the pager
ssh myserver 'journalctl | grep sshd | grep "Disconnected from"'
 | less
```

## 4.1 RegEx

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed 's/.*Disconnected from //'
# This uses the s substitution command for sed with regex (regular
# expressions)
```

`s/REGEX/Substitution/` **substitution** command in `sed`, where `REGEX` is the regular expression you want to search for and `SUBSTITUTION` is the text you want to substitute matching text for

- Regular expressions are usually surrounded by `/`
- Note: to use `sed` with modern regex (no escaping of characters with \), use `sed -E`

`.` means "any single character" except newline

`*` zero or more of the preceding match

`+` one or more of the preceding match

`?` zero or one of the preceding pattern; i.e. prevents regex from greedy matching as many occurrences as possible

`[]` one of many characters (specified inside square brackets `[]`)

`[abc]` selects any one character of a, b, and c

`[^abc]` selects any character that is **not** abc. The use of `^` in square brackets `[^]` means to exclude those characters in the match

`-` used to specify a range of characters

`[0-9]` selects any one number between 0 and 9

`(RX1|RX2)` either something that matches `RX1` or `RX2`

`^` matches the start of the line

`$` matches the end of the line

```
/.*Disconnected from /
# matches any text starting with any number of characters
# (.*) followed by the literal string "Disconnected from "
```

- Note: `*` and `+` are by default "greedy" (will match as many occurrences as possible)
- To avoid that, suffix `*` and `+` with `?` like `*?` or `+?` (not supported in `sed`)
- Recommended: use a regex debugger online to make sure the regex does what you want
- Recommended: use `^` and `$` to specify the beginning and end of the line to prevent users from doing weird stuff

```
| sed -E 's/.*Disconnected from (invalid |authenticating )?user
.* [^ ]+ port [0-9]+( \[preauth\])?$//'
# matches any text starting with any number of characters (.*)
# followed by the literal string "Disconnected from "
# then matches any of the user variants followed by matching any
# single word ([^ ]+), i.e. any non-empty sequence of
# nonspace characters, then the word "port" with some digits, then
# possibly the suffix [preauth], and finally the end of the line
# Note: square brackets [] are special characters in regex
# so we have to escape them
```

- Use **capture groups** in regex to store strings for use later

`()` any text matched by a regex surrounded by parentheses is stored in a numbered capture group. Available for substitution as `\1`, `\2`, `\3`, etc

```
| sed -E 's/.*Disconnected from (invalid |authenticating )?user
(.*) [^ ]+ port [0-9]+( \[preauth\])?$/\2/'
# does same matching as before but replaces each line with
# the 2nd capture group \2
# i.e. any text after user (.*), which is the username
```

## 4.2 Useful Data Wrangling Commands

`wc` wordcount program

`wc -l` gives number of lines

`sort` sorts lines of input (in ascending order by default)

`uniq` outputs the unique lines for a sorted list of lines

`uniq -c` outputs unique lines for a sorted list of lines with the number of occurrences

`sort -nk1,1` sorts numerically (`n`), for a white space separated column (`k`), starting and ending at the 1st column (`1,1`)

`awk` column based stream editor; operates on whitespaced separated columns

`paste` paste lines together with a delimiter

`bc` basic calculator; takes input from STDIN (use pipes)

`xargs` takes lines of input and turns them into arguments

`-` tells program to use STDIN or STDOUT instead of a given file; replaces a file argument (usually used with pipes)

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user
  (.*) [^ ]+ port [0-9]+( \[preauth\])?$/\2/'
 | sort | uniq -c

# takes usernames from before and sorts them (ascending
# alphabetically), but only keeps the unique ones and adds
# a count of occurrences
```

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user
  (.*) [^ ]+ port [0-9]+( \[preauth\])?$/\2/'
 | sort | uniq -c
 | sort -nk1,1 | tail -n10

# takes the alphabetically sorted list of unique usernames
# then sorts them again numerically based on the number of
# occurrences by using sort -nk1,1
# i.e. sort by only the first whitespace-separated column up to
# the 1st column
# tail then gives the last ones (the most common ones since sort
# is ascending)
```

```
rustup toolchain list | grep nightly | grep -vE "nightly-x86"
 | sed 's/-x86.*//' | xargs rustup toolchain uninstall
# uses xargs to pass certain versions as arguments to the rust
# uninstallation program
```

```
ffmpeg -loglevel panic -i /dev/video0 -frames 1 -f image2 -
 | convert - -colorspace gray -
 | gzip
 | ssh mymachine 'gzip -d | tee copy.jpg | env DISPLAY=:0 feh -'
# pipes useful to binary data
# here we used ffmpeg to capture webcam image, convert it
# to grayscale, compress it, send it to a remote machine over SSH,
# decompress it there, make a copy, and then display it locally
```

# 5 Command-line Environment

## 5.1 Job Control and Processes

- The shell uses a UNIX communication mechanism called a **signal** to communicate info to a 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 info that the signal delivered

`sleep` takes an integer argument specify the number of seconds that the process will "sleep"

`ctrl-C` ^C stops execution of a process by sending a SIGINT signal to tell the process to stop itself. The process is then ended.

`ctrl-Z` ^Z suspends the terminal by sending the process a SIGTSTP signal. The process is then stopped and put in the background, but its execution can be continued later

`ctrl-\` quits execution of a process by sending a SIGQUIT signal

`man signal` gives list of UNIX signals and their numbered identifiers

`kill -TERM <PID>` sends a SIGTERM signal to the process with process id `<PID>` to ask process to exit gracefully

## 5.2 Common Unix Signals

`SIGINT` : signal sent by terminal to *interrupt* execution of a process (i.e. *software interrupt*)

`SIGQUIT` : signal sent by terminal to *quit* execution of a program

`SIGHUP` : signal to indicate terminal *hangup*

`SIGSTOP` : pauses execution of a process to *stop*

`SIGTSTP` : sends a *terminal stop* (i.e. the terminal's version of `SIGSTOP`)

`SIGCONT` : *continues* execution of a stopped program at a later point in time

`SIGKILL` : causes a process to terminate immediately (i.e. *kill* the process). Unlike SIGINT, this signal cannot be caught or ignored because the receiving process cannot do any clean-up after receiving this signal

`SIGTERM` : a more generic signal to ask process to exit gracefully. Sent using `kill` command

- if there are still things running in your terminal when you close it, the program sends a SIGHUP to all processes to tel them to stop (i.e. had a hang-up in the command line communication)
- Can change the default behaviour of process upon receiving signals by using handlers in the program

**handler** captures signal and adds extra behaviour

**orphan process** when a process has other small children processes that it started, using SIGKILL to kill the 1st parent process will leave the child process still running (but without the parent). May lead to weird behaviour.

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

def handler(signum, time):
  print("\nI got a SIGINT, but I am not stopping")
  # handler captures SIGINT and ignores it
  # i.e no longer stops execution and continues running

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

# to actually stop this program, we need to use a SIGQUIT signal
# by typing ctrl-\
```

```
# if we run that program and send SIGINT twice, nothing happens
# it only stops when we give it SIGQUIT

$ 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
```

- `SIGINT` is like a "user-initiated happy termination" while `SIGQUIT` is like a "user-initiated unhappy termination" (both can be caught or ignored)
- `SIGTERM` terminates the process, gracefully or not, but allows it a chance to clean up (can be caught or ignored)
- `SIGKILL` kills the process immediately and is a last resort (process cannot catch signal or clean up)

## 5.3 Pausing and Background Processes

`fg` continues a paused job in the foreground

`bg` continues a paused job in the background

`jobs` lists unfinished jobs associated with current terminal session

`pgrep` finds process id (PID) of running jobs

`nohup` a wrapper for a command to ignore `SIGHUP`. Allows a process to continue running when shell closed (useful when working on remote machine in case you disconnect)

`disown` removes a process form the shell's job contorl and allows it to ignore `SIGHUP`

- Can refer to unfinished jobs using their pid or with a percent sign % and their job number
- Can refer to last backgrounded job with `$!` environment variable
- Adding an ampersand & suffix in a command will run the command in the background (will still use STDOUT but will give you the prompt back)
- To background an already running program, you can do `ctrl-Z` followed by `bg`
- Note: backgrounded processes are still children processes of the terminal and will die if you close the terminal (terminal sends a `SIGHUP` signal)

```
# example of jobs and foreground/background processes
$ 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 # run process 1 in the background
[1]  - 18653 continued  sleep 1000

$ jobs
```

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

$ kill -STOP %1 # stop process 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
```

## 5.4 Terminal Multiplexers - tmux

`tmux` terminal multiplexer that allows you to multiplex terminal windows using panes and tabs so that you can interact with multiple shell sessions

- `tmux` lets you manage shell sessions and is useful for remote machines since it eliminates the need to use `nohup`
- `tmux` uses keybindings of the form `<C-b> x` (`ctrl-b` release and another button `x`)
- Note: often remap `<C-b>` to `<C-a>` because it's faster and more ergonomic

### 5.4.1 Sessions

**session** an independent workspace with one or more windows

`tmux` starts a new session

`tmux new -s NAME` starts a session with that name

`tmux ls` lists the current sessions

`<C-b> d` detaches the current session

`tmux a` attaches the last session. You can use -t flag to specify which session to attach

### 5.4.2 Windows

**window** equivalent to tabs in editors or browsers

`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

`<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

### 5.4.3 Panes

**pane** like vim splits, pane lets 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

## 5.5 Aliases

**shell alias** a short form for another command that your shell will replace automatically for you

`alias alias_name="command_to_alias arg1 arg2"` command to create an alias. Note no spaces around equal sign `=` because `alias` only takes a single argument

```
# 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: aliases do not persist shell sessions by default
- Need to add an alias to shell startup files like `.bashrc` or `.zshrc` to have it persist

## 5.6 Dotfiles

**dotfile** plain-text file whose file name starts with a `.` (so that they are hidden in the directory listing `ls` by default). Used to configure many programs (ex. `~/.vimrc`)

**symlink** symbolic link a path to another path using `ln`. Kinda like a pointer where you can specify one path that links to the path where the file actually is

```
# create a symlink
ln -s path/to/file/you/want /symbolic/path/you/want
```

- Shells are configured with dotfiles (ex. `.bashrc`, `.bash_profile`, `.zshrc`) and reads these files to load its configuration on startup
- Can store environment variables in dotfiles
- Can add commands that you want to run on startup or modifications to your `PATH` environment variable (usually required by programs so that their binaries can be found)
- For better organization, it's recommended to organize dotfiles in their own folder (under version control) and symlinked into place using a script

- This is done for easy installation on new machines, portability, synchronization, and change tracking
- Note: dotfiles need to be in home directory `~/` (or use symlinks)

### 5.6.1 Portability

- Dotfile configurations may not work on all machines (ex. diff OS or shells)
- Can then make specific configurations using if-statements (if supported by config file)

```
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};
```

- If supported, also use includes for machine-specific settings (stored in another file)

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

- Can also share configurations across different programs
- ex. making both `bash` and `zsh` share the same set of aliases in `.aliases`

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

## 5.7 Remote Machines

`ssh` Secure Shell (SSH) used to interact with a remote server/computer

```
# ssh into a server by running either
ssh user@IP # ssh as user into server specified by this IP
ssh user@URL # ssh as user into server specified by  this URL

# Examples
ssh foo@bar.mit.edu # user is foo, server is the URL
ssh foobar@192.168.1.42 # user is foobar, server is the IP
```

### 5.7.1 Executing Commands

- Can run commands directly with `ssh`
- Also works with pipes to redirect input and output with local programs

```
# execute ls in the home folder of foobar
ssh foobar@server ls

# grep locally the remote output of ls
ssh foobar@server ls | grep PATTERN

# grep remotely the local output of ls
ls | ssh foobar@server grep PATTERN
```

### 5.7.2 SSH Keys

- Key-based authentication uses public-key cryptography to authenticate you to the server
- Allows you to avoid entering password every time
- Note: the secret private key (often `~/.ssh/id_rsa` and more recently `~/.ssh/id_ed25519`) is basically your password so treat it like so

### 5.7.3 Key generation

`ssh-keygen` Generates a public and private key pair

`ssh-agent` lets you skip typing your passphrase every time

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

- Recommended: use a passphrase to avoid someone who gets your private key to access authorized servers

```
# check if you have a passphrase and valid it
ssh-keygen -y -f /path/to/key
```

### 5.7.4 Key based authentication

- `ssh` will look into `.ssh/authorized_keys` (on the remote sever side) to determine which clients it should let in

```
# copy over your public key to .ssh/authorized_keys
# on the remote server
cat .ssh/id_ed25519.pub
  | ssh foobar@remote 'cat >> ~/.ssh/authorized_keys'

# can also use ssh-copy-id if available
ssh-copy-id -i .ssh/id_ed25519.pub foobar@remote
```

### 5.7.5 Copying files over SSH

`ssh+tee` use `ssh` command execution and STDIN input. `tree` then writes output from STDIN into a file

`scp` secure copy command useful for copying large amounts of files/directories (recurses over paths)

`rsync` improves upon `scp` by detecting identical files in local and remote to avoid duplicate copying. Provides more control over symlinks, permission, and extra features like `--partial` flag to resume a previously interrupted copy

```
# Copy a local file into a remote server file called serverfile
# using ssh+tee
cat localfile | ssh remote_server tee serverfile.

# Copy a local file into a remote server file
scp path/to/local_file remote_host:path/to/remote_file
```

### 5.7.6 Port Forwarding

- Often have software that listens to specific ports in a machine to function
- ex. `jupyter notebook`
- For local machines, you can just type the port `localhost:PORT` or `127.0.0.1:PORT`
- For remote servers, you need port forwarding (either Local Port Forwarding or Remote Port Forwarding)

**local port forwarding** : link a port in your local machine to the remote port for a service (forward local port)

**remote port forwarding** : link a remote port to the local port for a service (forward remote port)

- Usually use local port forwarding (ex. `jupyter notebook`)

```
# Execute jupyter notebook in remote server (listens to port 8888)
# Want to interact with jupyter notebook locally so forward
# the local port 9999 to the remote port 8888
ssh -L 9999:localhost:8888 foobar@remote_server
# Then navigate to localhost:9999 on local machine to use notebook
```

### 5.7.7 Local Port Forwarding

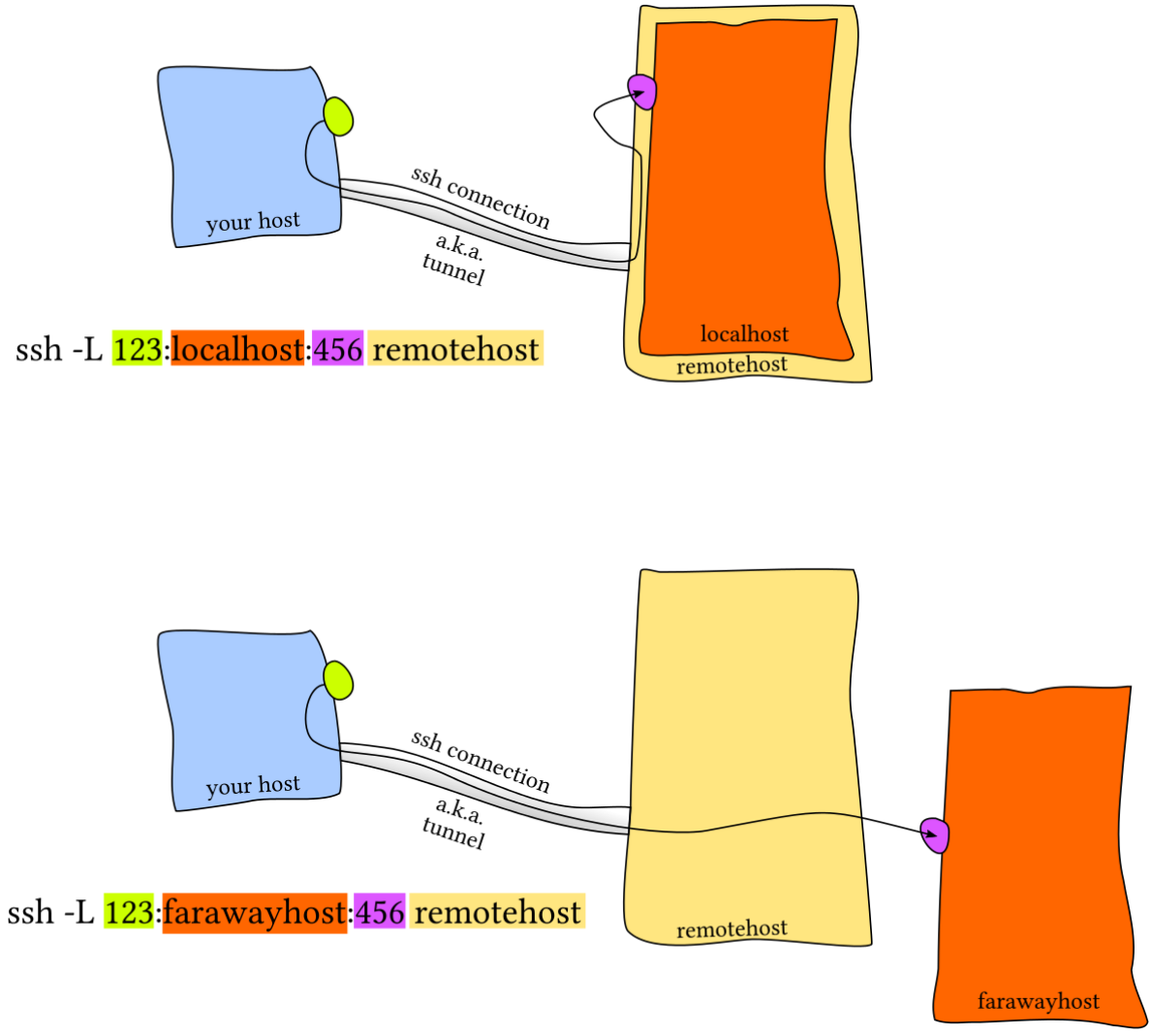


Figure 1: Local Port Forwarding

### 5.7.8 Remote Port Forwarding

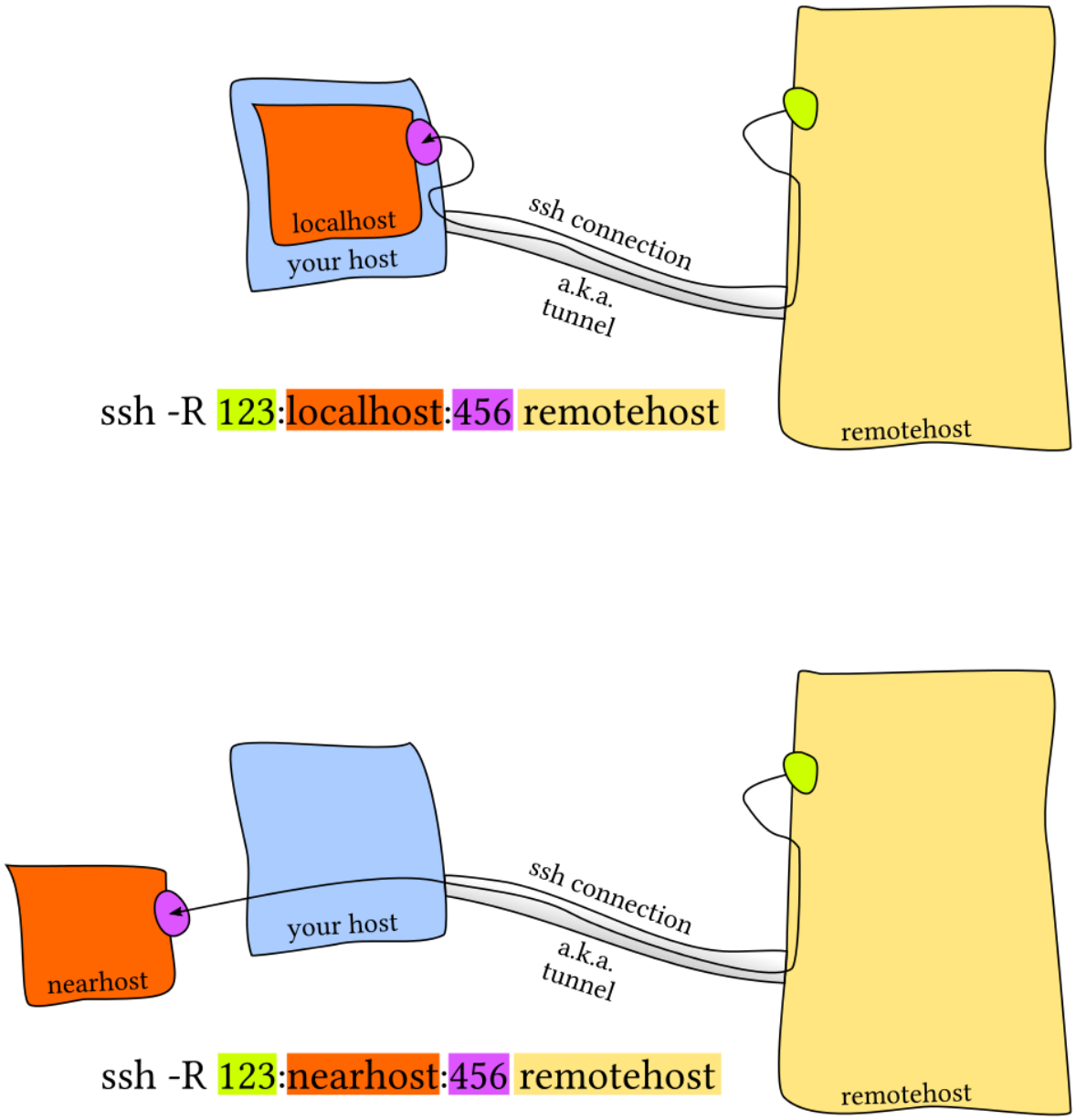


Figure 2: Remote Port Forwarding

### 5.7.9 SSH Configuration

~/.ssh/config Dotfile used to configure ssh. Also readable by other programs like scp, rsync, etc that can convert the settings into the corresponding flags

/etc/ssh/sshd_config Used for server side configuration. Can make changes like disabling

password authentication, changing ssh ports, enabling X11 forwarding, etc. Can also specify config settings on a per user basis

- Note: `~/.ssh/config` has some potentially private info that you might not want to share with other people

#### 5.7.10 Miscellaneous SSH Stuff

`mosh` mobile shell that improves upon `ssh` by allowing roaming connections, intermittent connectivity, and providing intelligent local echo

`sshfs` mounts a folder on a remote server locally, allowing you to use a local editor

### 5.8 Bash vs Zsh

- `bash` is the most common shell and is the default option on most shells

`zsh` a superset of `bash` that provides extra features like

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

## 6 Version Control (Git)

### 6.1 Version Control Systems

**Version Control Systems (VCS)** : tools to track changes to source code; maintain history of changes and improve collaboration

### 6.2 Git's Data Model

#### 6.2.1 Snapshots (commits)

- Git models history of a collection of files and folders inside a top-level directory as a series of snapshots (commits)

**blob** : a file in Git

**tree** : a directory; maps names to blobs or trees (directories can contain other directories)

**commit** : a snapshot of the top-level tree being tracked

```
<root> (tree) # top level directory
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

### 6.2.2 Git's model of history

- Git models history as a directed acyclic graph (DAG) of snapshots
- i.e. each snapshot in Git refers to a set of "parents" (older snapshots before it)
- Snapshots can have multiple parents (a set of parents) because a snapshot might descend from multiple parents
- ex. combining (merging) two parallel branches of development (2 parents)

```
# example of branching (newer commits on the right)
o <-- o <-- o <-- o
            ^
             \
              --- o <-- o
```

- The circle `o`'s refer to individual commits (snapshots of entire tree)
- Note: the arrows point to the parent of each commit (i.e. the previous commits) and the newer commits are on the right
- After third commit, the history branches into 2 separate branches (ex. 2 separate features independently developed in parallel)
- Can later merge parallel branches to create a new snapshot with both features

```
# merging parallel branches in Git
o <-- o <-- o <-- o <---- o
            ^             /
             \           v
              --- o <-- o
```

- Git commits are immutable
- i.e. "edits" to the commit history actually add new commits instead of changing old commits

- References are then updated to point to new ones

### 6.2.3 Data Model in pseudocode

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | file>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
  parent: array<commit>
  author: string
  message: string
  snapshot: tree
}
```

### 6.2.4 Objects and content-addressing

**object** a blob, tree, or commit

```
type object = blob | tree | commit
// these objects are all content-addressed and given a SHA-1 hash
```

- In Git's data store, all objects are content-addressable by their SHA-1 hash (160-bit string or 40 chars of hexadecimal)

```
objects = map<string, object>

def store(object):
  id = sha1(object)
  objects[id] = object

def load(id):
  return objects[id]
```

- Since blobs, trees, and commits are all objects they can reference other objects by their hash
- i.e. don't have to contain the on-disk representation of the referenced object

`git cat-file -p <SHA-1 hash>` visualizes the object pointed to by the hash

### 6.2.5 References

- All snapshots can be identified by their SHA-1 hash but for convenience, can make human-readable references

**references** : human-readable pointers to commits (mutable)

`master` : a reference that usually points to the latest commit in the main development branch. Created by default when you init a git repo

`HEAD` : a reference that points to "where we currently are" in history. Allows you compare current position with other snapshots in history

- Note: references are mutable (can point to other objects) but objects are immutable
- Can't change where hashes point to because the hash is determined from the object (which is immutable)

```
references = map<string, string>

def update_reference(name, id):
  references[name] = id

def read_reference(name):
  return references[name]

def load_reference(name_or_id):
  if name_or_id in references:
      return load(references[name_or_id])
  else:
      return load(name_or_id)
```

### 6.2.6 Repositories

**repository** data storing `objects` and `references`

- On disk, Git only stores objects and references
- All `git` commands manipulate the commit DAG by adding objects and adding/updating references

## 6.3 Staging Area

**staging area** allows you to specify which code changes should be included in the next commit

## 6.4 Git Command-Line Interface

### 6.4.1 Basics

`git help <command>` get help for a git command

`git init` creates a new git repo, with data stored in the .git directory

`git status` tells you what's going on

`git add <filename>` adds files to staging area for next commit

`git commit` creates a new commit Write good commit messages!

`git log` shows a flattened log of history

`git log --all --graph --decorate --oneline` visualizes history as a DAG

`git diff <filename>` show differences since the last commit

`git diff <revision> <filename>` shows differences in a file between snapshots

`git diff --cached` show what changes are staged for next commit

`git checkout <revision>` updates HEAD and current branch. Changes files in working directory to match the revision snapshot (where HEAD now points to). Note that it throws any current uncommited changes

### 6.4.2 Branching and Merging

`git branch` shows branches in repo

`git branch <name>` creates a branch. New branch points to the same current location in history (i.e. HEAD and new branch will resolve to same location)

`git checkout -b <name>` creates a branch and switches to it

- Equivalent to `git branch <name>; git checkout <name>`

`git merge <revision>` merges into current branch

`git mergetool` use a fancy tool to help resolve merge conflicts

`git merge --abort` aborts merge and puts you back in previous state before merge

`git merge --continue` finishes merge after merge conflict is resolved

`git rebase` rebase set of patches onto a new base

**fast forward** : if you merge a branch that has the the current commit (HEAD) as a predecessor, it will just move the references up to the merged branch (because no other changes required)

**Merge conflicts** : when you merge parallel branches, Git may get confused if you have contradictory changes

- For merge conflict, git will add conflict markers in the affected files to show incompatible code between the branches being merged

### 6.4.3 Remotes

- Remotes used to collaborate with other people
- `.git` folder contains entire repo history (objects, references, previous commits)
- Each person maintains their own copy of the git repo and they pass changes around with commits
- Remote repo (ex. GitHub) usually called `origin`

`git remote` list remotes

`git remote add <name> <url>` add a remote. Makes local repo aware of remote repo's

`git push <remote> <local branch>:<remote branch>` send objects to remote, and update remote reference

`git branch --set-upstream-to=<remote>/<remote branch>` set up correspondence between local and remote branch. Can then use shortened form `git push`

`git fetch` retrieve objects/references from a remote

`git pull` same as `git fetch; git merge`

`git clone` download repository from remote

### 6.4.4 Undo

`git commit --amend` edit a commit's contents/message

`git reset HEAD <file>` unstage a file

`git checkout -- <file>` discard changes

### 6.4.5 Advanced Git

`git config` customize Git; can also directly edit `~/.gitconfig`

`git clone --shallow` clone without entire version history (faster). Useful for big projects with many commits

`git add -p` interactive staging

`git rebase -i` interactive rebasing

`git blame` show who last edited which line

`git stash` temporarily remove modifications to working directory

`git stash pop` restore changes from `git stash`

`git bisect` binary search history (e.g. failed unit tests)

`.gitignore` file used to specify intentionally untracked files to ignore

# 7 Metaprogramming

## 7.1 Build Systems

**Build system** : automates building process to convert inputs (dependencies) to outputs (targets) using specified rules

`make` most common build system on UNIX. Good for simple to medium complexity

`cmake` another build system that is opinionated and optimized for specific tasks

`Makefile` files used to specify dependencies, targets, and rules for `make`

- Build system aims to do minimal work
- i.e. if a dependency has not changed, it will not rebuild the associated targets

```
# first directive
paper.pdf: paper.tex plot-data.png
  pdflatex paper.tex

# second directive
plot-%.png: %.dat plot.py
  ./plot.py -i $*.dat -o $@
```

**Directive** : rule for producing target (left-hand side of colon `:`) using the dependencies (right-hand side of colon `:`)

**Target** : desired output (ex. pdf, mp4)

**Dependency** : software that is required to built the target

**Rule** : specifies how to get target from dependencies (ex. run a `python` file or `pdflatex`)

- The indented block in each directive is a sequence of programs to produce the target from the dependencies
- Note: first directive in `make` defines teh default goal (what is built when you run `make` with no arguments)
- Can also build specific targets with arguments: `make plot-data.png`
- Note: `Makefile` requires tabs for the rules; spaces will not work

`%` wildcard that specifies "patterns" in a rule and matches the same string on the left and on the right

`$*` special variable that matches `%`

`$@` special variable for target

- ex. if the target `plot-foo.png` is requested, `make` will look for the dependencies `foo.dat` and `plot.py`

- Note: if your `Makefile` is super complicated, it probably means you should use `cmake` or something else
- Can use `make` at the top level and use it to call opinionated build systems like `cmake`

## 7.2 Versioning

- Versions are usually numerical and are used to ensure that software keeps working
- Helps users to determine what is compatible/incompatible when choosing which version of a dependency to install

### 7.2.1 Semantic Versioning

**Semantic versioning** : common version format of **major.minor.patch**

- Semantic versioning rules

1. if API is **unchanged**, increase the **patch** version
2. if API is changed (**backwards-compatible**), increase the **minor** version (reset patch version to 0)
3. if API is changed (**non-backwards-compatible**), increase **major** version (reset minor and patch version to 0)

- Specify dependencies with major and minor version numbers
- Can use any version of dependency with the same major and same or higher minor version
- Usually patch updates are for security fixes; the software will still run, but you might still want to update it
- The best type of dependency is when you depend on version X.0.0 because you can use any minor or patch version from major version X
- Python 2 and Python 3 are an example of a major version bump (incompatible code)

### 7.2.2 Lock files

**lock file** : file that lists the exact version you are currently depending on of each dependency. Dependencies are then only updated when you run the update program

**vendoring** : explicitly copy all the code of your dependencies in your own project. Gives you total control over any changes, but makes updating difficult

- Lock files are useful because they avoid unnecessary recompiles, having reproducible builds, and avoid automatically updating to potentially faulty versions

## 7.3 Continuous integration systems

**Continuous integration (CI)** : software (cloud build system) that runs whenver code changes

- Continuous integration can be general or specific to certain tasks (ex. run test suite after a code push)
- ex. dependabot is CI tool that scans a repository for newer versions of dependencies
- ex. GitHub pages is another CI action that runs the Jekyll static site generator on every push to `master` and loads the built site on a particular domain
- CI software usually give badges that you can add to README

## 7.4 Testing

**Test suite** : collective term for all the tests

**Unit test** : a "micro-test" that tests a specific feature in isolation

**Integration test** : a "macro-test" that tests if different features/components work together properly

**Regression test** : test that implements a specific pattern that previously caused a bug to ensure that the bug does not reappear

**Mocking** : replace a function, module, or type with a dummy implementation to avoid testing unrelated functionality