CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 2

Thursday, September 9th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

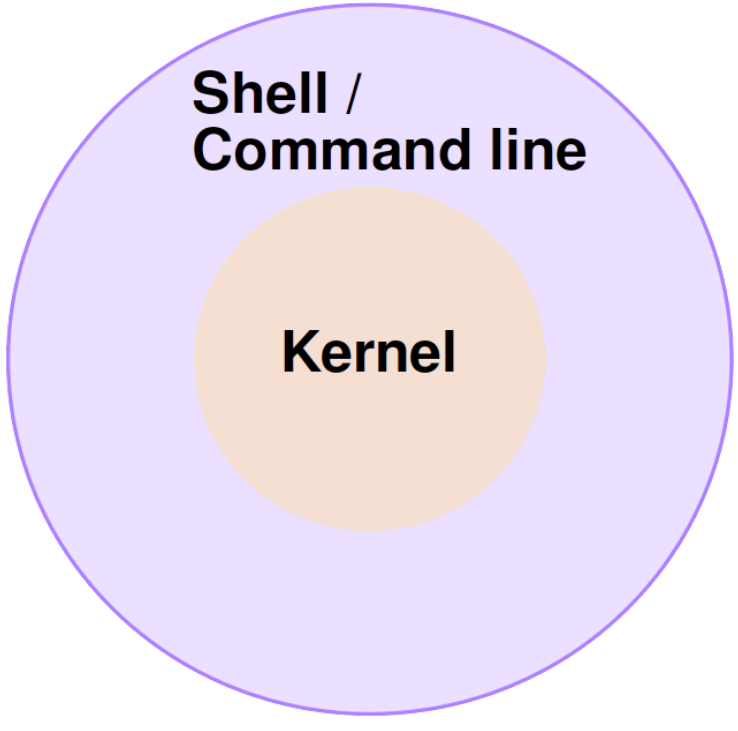
- More on Linux commands and the `man`-pages
- Working with the shell (Unix philosophy and pipes)
- Regular expressions and `grep`
- File attributes and permissions
- Short journey into text editors

OUTLINE

- Shell customization
- Input/Output (I/O) and redirection
- Job/process management
- Environment variables
- Shell scripting

SHELL CUSTOMIZATION

RECALL THE SHELL IS YOUR COMMAND INTERFACE



- The shell is your tool, just like an editor. *Make it your own.*
- There are different shell interpreters:
  - `sh`
  - `bash` (Mac: since OSX Jaguar)
  - `csh`
  - `ksh`
  - `zsh` (Mac: since OSX Catalina)
- Each of those shells executes a number of files at startup
- You can use them to run commands (`rc`) and configure your shell

SHELL CUSTOMIZATION

- *Examples for configuration:* user prompt, environment variables, auto-completion, command aliases, color theme and appearance, message of the day (motd), ...
- These customizations are implemented in startup files that are read by the `bash` shell when it starts:
  - Interactive login shell or with `--login` option:
    1. `/etc/profile`
    2. `~/.bash_profile` (if it exists, read and execute then stop)
    3. `~/.bash_login` (if it exists, read and execute then stop)
    4. `~/.profile` (if it exists, read and execute then stop)
  - Interactive non-login shell (e.g. a terminal emulator like `xterm`):
    1. ~/.bashrc

SHELL CUSTOMIZATION

- The files that are being read at shell startup depends on the shell you are using. This is a nice summary and overview for different types of shells.
- A **login** shell is one that you login to your system.
- If you use Ubuntu with a GUI front end like Gnome, you will not see this shell as the system boots directly into graphical mode.
- On a headless server you will be dropped into a login shell. This is called an *interactive login shell*
- The login shell allows you to create other shell instances, these are interactive **non-login** shells.

SHELL CUSTOMIZATION

Summary for `bash`:

- Files read for interactive login shell:
  1. `/etc/profile`
  2. One of (in that order): `~/.bash_profile`, `~/.bash_login`, `~/.profile`
- Files read for interactive non-login shell:
  1. `~/.bashrc`

Typically, `~/.bash_profile` contains this code:

```
1 [[ -f ~/.bashrc ]] && source ~/.bashrc  # if ~/.bashrc exists, source its contents
```

Conclusion: edit your `~/.bashrc` to customize your shell. `zsh` users edit `~/.zshrc` instead.

Bash reference: startup files

SHELL CUSTOMIZATION

A few comments about `bash` and `zsh`:

- Mac users will most likely be working with `zsh`, a newer shell with some additional features.
- The default shell on Linux is `bash`. You must install `zsh` from the package repo if you want to use it on Linux.
- While startup files may be different, most *scripts* should run with either shell.
- You will be confronted with `bash` on most remote machines and servers. Keep that in mind when you work with `zsh` and must be compatible with `bash`.

References worth checking:

- Moving to `zsh` - Scripting OSX
- What should/shouldn't go in `.zshenv`, `.zshrc`, `.zlogin`, `.zprofile`, `.zlogout`?
- About `.bash_profile` and `.bashrc` on MacOS

SHELL CUSTOMIZATION

Simple `.bashrc` example:

```
1  set -o vi # configure the shell prompt to behave like vi (normal and insert mode)
2            # default is set -o emacs
3
4  alias diff='diff --color=always' # alias for the diff command to enforce color
5
6  # These are some environment variables.  The export keyword is important
7  export EDITOR=vim
8  export GIT_EDITOR=vim
9  export VISUAL=vim
10 export BROWSER=qutebrowser
11 export DEFAULT_PDF=zathura
12 export PDFVIEWER=zathura
13
14 # update the PATH environment variable
15 export PATH=$HOME/bin:$HOME/.local/bin:$HOME/go/bin:$PATH
16
17 # set a custom primary prompt: promt is defined in the variable PS1, see `man bash`
18 export PS1='\e[36m\w\e[0m\$ '
```

- A useful adaptive prompt for `bash` and `zsh` : https://github.com/nojhan/liquidprompt
- Shell color themes based on 16 colors: https://github.com/chriskempson/base16-shell

SHELL CUSTOMIZATION

Work together with your neighbors and setup a shell startup file:

- You find yourself often typing `ls -l`. Create an alias named `ll` to minimize your future typing overhead.
- Figure out what the prompt from the previous slide is doing:

```
1 # set a custom primary prompt: promt is defined in the variable PS1, see `man bash`
2 export PS1='\e[36m\w\e[0m\$ '
```

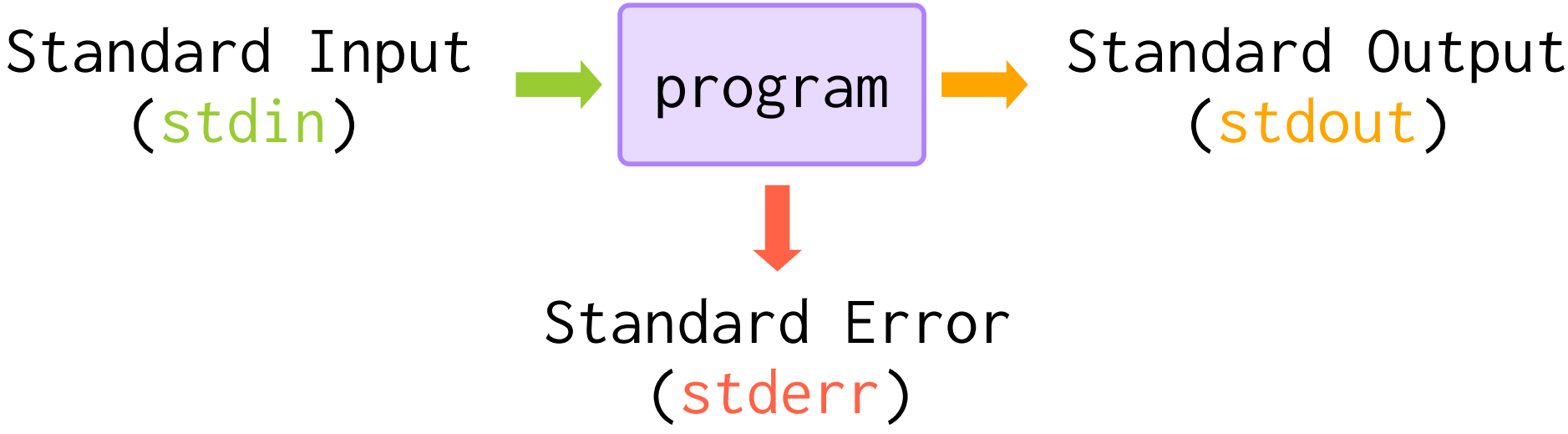
  Configure your own prompt. This page might be helpful.
- Try out the

```
$ set -o vi
$ set -o emacs
```

  settings. You can type them directly in the shell to test it and modify your startup file accordingly if you prefer either.

INPUT/OUTPUT (I/O) AND REDIRECTION

INPUT/OUTPUT (I/O) AND REDIRECTION



- Inside the shell, the `stdin` is from your keyboard
- A program generates two output streams:
  1. `stdout` : normal program output
  2. `stderr` : output associated to something gone wrong
- Recall the file descriptors: `stdin=0`, `stdout=1`, `stderr=2`
- To send and EOF (end-of-file) character, press `ctrl-d`

REDIRECTION OF I/O STREAMS

- You have learned about the pipe which streams the `stdout` of one program into the `stdin` into the next (see the " `SHELL GRAMMAR` " section in `man bash`)
- You can also *redirect* any stream ***to or from files***
- To redirect `stdout` to a file use the `>` operator:

```
$ ls -l > ls_long_output
```

- To redirect file contents to `stdin` use the `<` operator:

```
$ sort < some_data
```

- You can combine both in one go:

```
$ sort < some_data > some_sorted_data
```

REDIRECTION OF I/O STREAMS

- You can either ***create (or overwrite)*** files or ***append*** to existing files:
  - Use `>` to *create or overwrite* (it will delete previous contents)
  - Use `>>` to append to existing files (nice for logging)
- There are *two special data sinks* in Linux:
  1. `/dev/null` : data written to this device is *discarded*. Reading from this device always returns the end-of-file (EOF) character.
  2. `/dev/zero` : data written to this device is *discarded*. Reading from this device returns the `'\0'` (`NUL`) byte (e.g. see ASCII table).

Example: filter spam email and send it to `/dev/null`

```
$ script_to_filter_spam_email > /dev/null
```

REDIRECTION OF I/O STREAMS

SPECIAL CHARACTERS IN THE SHELL

- We have already seen the wildcard `*` that matches anything. E.g., list all python files

```
$ ls
exercise_1.py  exercise_2.py  README.md
$ ls *.py
exercise_1.py  exercise_2.py
```

- ***White space in filenames:*** although common on Windows, it is *bad practice* to create filenames with spaces. In the shell, white space separates arguments to commands:

```
1 $ touch exercise 1.py; ls
2 1.py  exercise  README.md  # touch creates 2 files: 1.py and exercise
3 $ touch exercise\ 1.py; ls # you must escape white space to get a single file
4 1.py   exercise  'exercise 1.py'   README.md
```

JOB/PROCESS MANAGEMENT

JOB/PROCESS MANAGEMENT

- Any process or job is assigned a unique `PID`:

```
$ sleep 100
```

```
$ ps aux | grep sleep # this is run in another bash instance, you see why in a second
fabs      145691  0.0  0.0   5364   688 pts/4    S+   12:19   0:00 sleep 100
```

  The `PID` for this `sleep` process is `145691`
- The shell gives you some tools to manage such processes:
  - Run them in the background
  - Move a job to the foreground
  - Suspend a job
  - Terminate a job
- Running a process will *block* your prompt by default. For example: the command `sleep 100` you will give you back control only after 100 seconds have passed.

JOB/PROCESS MANAGEMENT

- Running a process will *block* your prompt by default. For example: the command `sleep 100` you will give you back control only after 100 seconds have passed.
- You can get back control immediately by appending a `&`

```
$ sleep 100 &
[1] 153514  # the shell notifies us that PID 153514 is running in background
$ jobs      # we check that the job is running indeed
[1]+  Running                 sleep 100 &
...         # 100 seconds later the shell will tell you that the process has concluded
[1]+  Done                    sleep 100
```

  This way you can continue with work in the current shell session.
- Appending a `&` will put the job in the *background*, you could exit the shell and the job would continue to run. (Only if the shell you are quitting is a *non-login shell*!)

JOB/PROCESS MANAGEMENT

- You can *suspend* a job to get back control of the shell by pressing `Ctrl-z`.

```
$ sleep 100
^Z  # here I pressed Ctrl-z
[1]+  Stopped                 sleep 100
$ jobs
[1]+  Stopped                 sleep 100
```

  A stopped (or suspended) job *does not make progress*! If you want to quit the current shell (even if inside an interactive non-login shell) it will warn you the first time you try:

```
$ exit
exit
There are stopped jobs.
$ exit  # the second time you call exit (or Ctrl-d) the shell will quit without warning
```

- You can bring a stopped job back to foreground by using the `fg` command.

JOB/PROCESS MANAGEMENT

Stopping a job with `Ctrl-z` and resuming with `fg` is very useful.

Example: When you launch `vim` to edit files and you quickly need to go back to the shell where you came from, but you do not want to quit `vim`, press `Ctrl-z` while inside `vim`. When you want to resume `vim`, simply type `fg` on the shell prompt.

JOB/PROCESS MANAGEMENT

What to do when you want to keep a job running but you need to exit an *interactive login-shell*?

Where does this scenario happen in the first place?

- You are in the middle of work on a remote machine but you must go get groceries right now. When you are home you want to continue work. Assume that on your way you will lose your internet connection.
- You want to run a simulation for, say, 24 hours on a local workstation without job scheduling. During that time you want to be able to logout and login whenever you want to check on your results.

JOB/PROCESS MANAGEMENT

What to do when you want to keep a job running but you need to exit an *interactive login-shell*?

Two solutions:

1. Use the `nohup` (no hangup, see `man nohup`)

```
$ nohup my_prog &> my_output &
```

   Make sure to redirect the output, both `stdout` and `stderr`, to a file for later reference.
2. Use a *terminal multiplexer* on the target machine.
   - `screen`: one of the first, it has seen years.
   - `tmux`: a more recent multiplexer with more features than `screen`.
   - `tmate`: a fork of `tmux` designed to collaborate with other mates.

JOB/PROCESS MANAGEMENT

You can list and display running jobs and processes in several ways:

- `jobs` (see `man jobs`): displays the status of jobs in the current session (running, stopped, terminated)

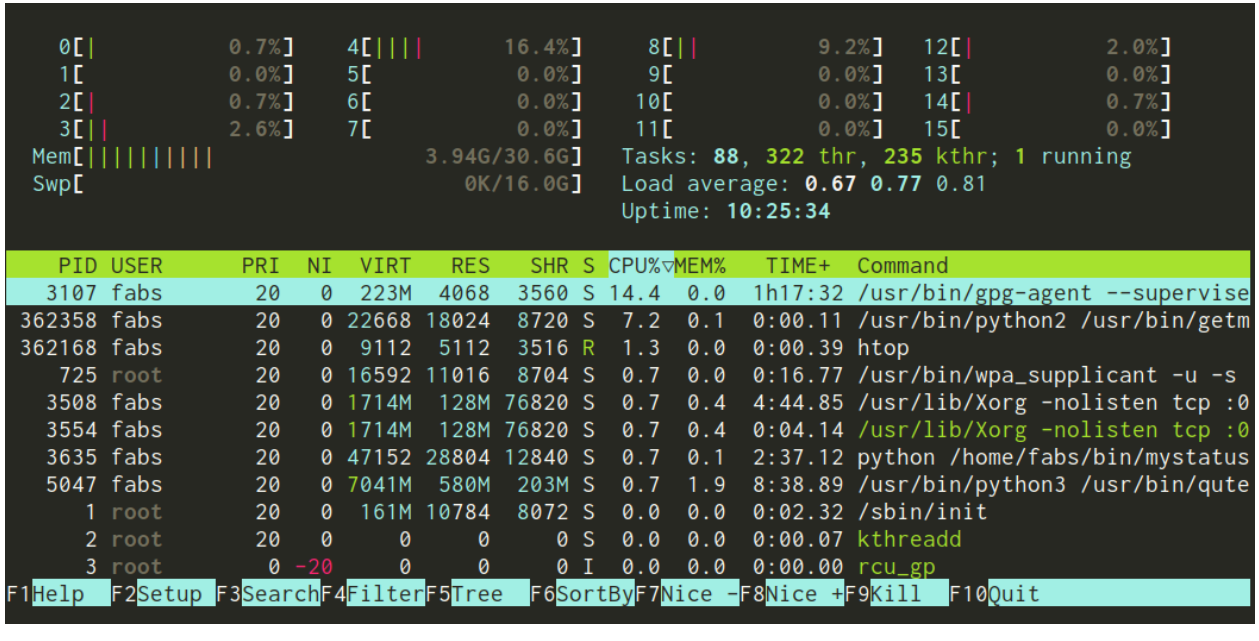
```
$ jobs
[1]-  Running                 /bin/zathura leiserson2020a.pdf &
[2]+  Stopped                 vim newton_iterations.cpp
```

- `ps aux` (see `man ps`): list all running processes on `stdout`. You may need to filter through `grep` to find what you are looking for.
- `top` (always available on Linux, see `man top`) or better UI use `htop` (you will need to install it, see `man htop`): list running processes with the ones consuming most resources at the top

JOB/PROCESS MANAGEMENT

You can list and display running jobs and processes in several ways:

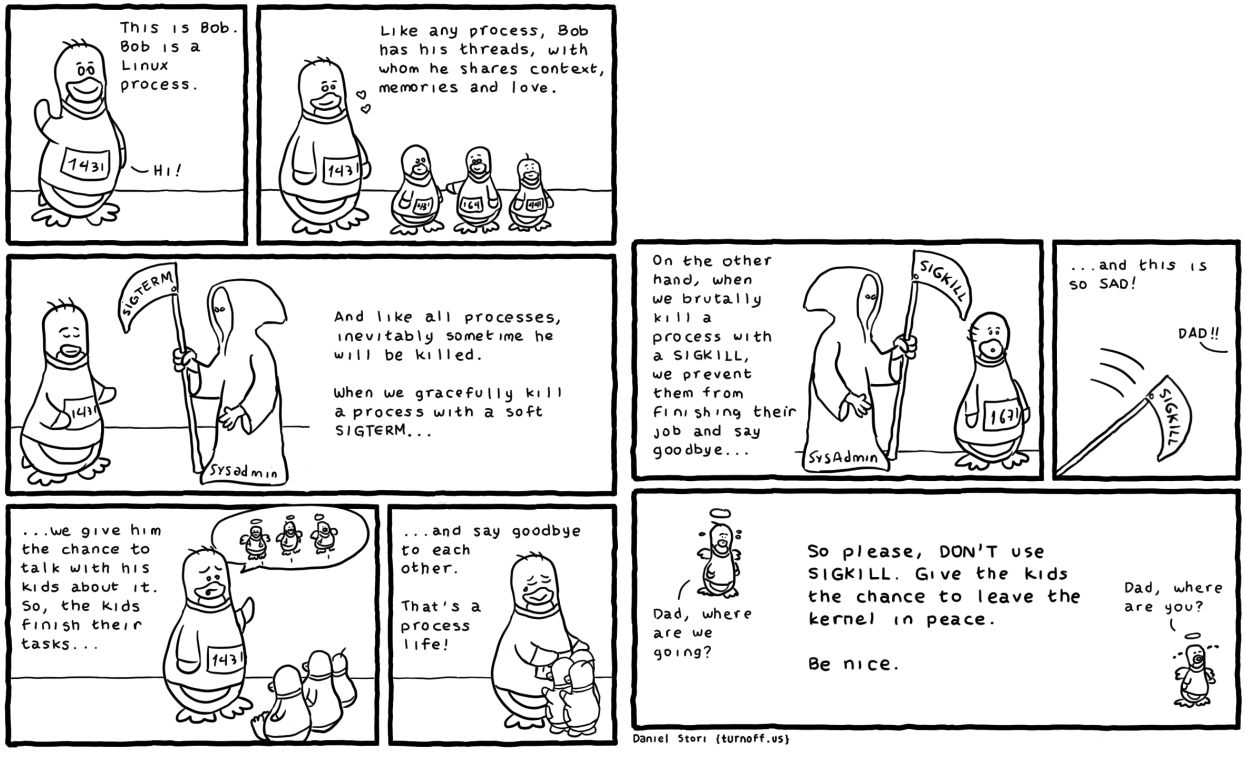
- `top` (always available on Linux, see `man top`) or better UI use `htop` (you will need to install it, see `man htop`): list running processes with the ones consuming most resources at the top



htop

JOB/PROCESS MANAGEMENT

You can terminate any job you have appropriate permissions. You can graciously terminate a job or forcefully kill it. You should only use the latter when there is no hope (e.g. system starts to become unresponsive due to memory leak). The former will make sure that claimed resources are freed correctly and child processes are shutdown first.



JOB/PROCESS MANAGEMENT

- You terminate jobs by sending a *signal* to them through the `kill` command. See:

```
$ whatis signal
signal (7)           - overview of signals
signal (2)           - ANSI C signal handling
signal (3p)          - signal management
$ man 7 signal
$ man kill
```

- By default `kill` will send a `SIGTERM` signal which is what you want (the mean one is called `SIGKILL`)
- You can specify the signal with the `-s` switch (be sure to get the `PID` right! Use `ps` or `top` to get it):

```
$ kill -s SIGKILL <PID>  # only do this when nothing else works anymore
```

- If you are sure that, for example, `python` is causing you trouble, you can send a `SIGTERM` by name which is easier and more verbose

```
$ killall python
```

JOB/PROCESS MANAGEMENT

- You terminate jobs by sending a *signal* to them through the `kill` command. See

```
$ whatis signal
signal (7)           - overview of signals
signal (2)           - ANSI C signal handling
signal (3p)          - signal management
$ man 7 signal
$ man kill
```

- You can send an *interrupt* signal (`SIGINT`) by pressing `Ctrl-c`
- A `SIGINT` can be *catched* and processed differently by interactive software. E.g., a hanging python script will not always terminate with `Ctrl-c` because the interpreter will catch the signal and decide what to do with it. Use `killall python` instead.
- In most cases a `SIGINT` translates to `SIGTERM`

JOB/PROCESS MANAGEMENT

Work together with your neighbors and practice job management:

1. Use the `sleep` command to sleep for `1000` seconds
2. Suspend the job (stop it from running)
3. Open `vim` and suspend it too
4. List your jobs with `jobs`
5. Continue running the first job (`sleep`) by sending it to the background with the `bg %n` command. `n` is the job ID listed by `jobs`
6. Bring `vim` back to the foreground using `fg %n` (what happens if you omit `%n` ?)
7. Exit `vim` by pressing `:q!` followed by `ENTER`
8. Forcefully kill the running `sleep` command (you need to find its `PID` and then use the `kill` command)

ENVIRONMENT VARIABLES

ENVIRONMENT VARIABLES

- You can customize your environment by setting the values of certain *environment variables*
- You have already seen them when customizing your prompt by setting the value of `PS1` accordingly
- You can get a list of all environment variables and their corresponding value with the `env` command
- Any variables in a shell script (not only environment variables) can be *dereferenced* by prepending a `$` character:

```
$ my_var='Hello CS107!'
$ echo $my_var
Hello CS107!
$ echo my_var
my_var
```

- *Environment variables are usually set in* ALL CAPS

ENVIRONMENT VARIABLES

The role of the `PATH` is to specify the search path(s) used by the shell to find executable programs.

- For every command you enter, the shell checks if this command is a built-in command (see the "`SHELL BUILTIN COMMANDS`" in `man bash`)
- If not found, it will check the path(s) defined in `PATH` to see whether it can find the executable
- Finally, the shell will give up:

```
$ this_command_is_hypothetical
bash: this_command_is_hypothetical: command not found
```

- Each path specified in `PATH` must be *delimited* by a colon "`:`". This is true for any environment variable that can hold a list of paths, e.g. `MANPATH`, `INFOPATH`, `PYTHONPATH` and others

ENVIRONMENT VARIABLES

- By default, `PATH` holds at least the relevant paths for your system commands. It is a good idea to extend it in your `.bashrc` as follows:

```
PATH=$HOME/bin:$HOME/.local/bin:$PATH
export PATH
```

- The `export` ensures that your customized `PATH` is available in other shell instances as well
- **Can you guess what `HOME` is?**
- `$HOME/bin`: a standard path in your home directory for executable scripts or programs
- `$HOME/.local/bin`: default path used by `python` to install packages in a user directory (some of those packages come with executables and you want to access them). E.g., this command installs the `python` package "`package_name`" below your `$HOME/.local` by default:

```
$ python -m pip install --user <package_name>
```

ENVIRONMENT VARIABLES

- By default, `PATH` holds at least the relevant paths for your system commands. It is a good idea to extend it in your `.bashrc` as follows:

```
1 PATH=$HOME/bin:$HOME/.local/bin:$PATH
2 export PATH
```

Order is important: You must dereference `PATH` in order to keep what was previously defined in it. Append it *at the end* to ensure that your custom executables (with possibly the same names as already existing ones) will be picked up by the shell first! *Once the shell has found a match in the search path, it will not look any further*

SETTING VARIABLES

- You can omit the `export` keyword. In that case the variable will *only* be available in the current shell instance:

```
$ my_var='Hello CS107!'  # set a variable in the current shell (no spaces between '=')
$ bash                   # create another shell instance
$ echo $my_var           # my_var is empty
```

- With `export`:

```
$ export my_var='Hello CS107!'  # set a variable in the current shell
$ bash                          # create another shell instance
$ echo $my_var                  # value of my_var is propagated
Hello CS107!
```

  You must use `export` in your `.bashrc` or `.zshrc` files to ensure the settings propagate correctly.
- You can delete any variable using the `unset` command:

```
$ my_var='Hello CS107!'  # set a variable in the current shell
$ unset my_var           # unset it again in the current shell
$ echo $my_var           # my_var is empty
```

SHELL SCRIPTING

SHELL SCRIPTING

- Typing out a series of commands that do complex tasks is not convenient
- Shell scripting (and also `python` scripts) is a powerful tool to perform all kinds of automation tasks, often repetitive in time
- By setting variables from the previous slides, you have already seen an important part of shell scripting

A shell script is an *executable* file that contains commands together with pipes and file redirection to perform (more complex) tasks in the command line.

SHELL SCRIPTING INTERPRETER

You should be specific about which shell (interpreter) you want to target in your scripts. This ensures *portability* of your scripts.

- You specify the interpreter with a *shebang*. The general form is:

```
1 #!interpreter [optional arguments]
```

  which you must write *at the very beginning* of your script.
- Here are a few examples:

`bash`	`#!/usr/bin/env bash`
`zsh`	`#!/usr/bin/env zsh`
`python`	`#!/usr/bin/env python3`

- **Note:** Use the `/usr/bin/env` tool to resolve the actual path of the interpreter you target. Some users might have custom installations for these interpreters in their `PATH`. Hard-coding a path like `/bin/bash`, for example, would ignore that and possibly annoy users of your script.

SHELL SCRIPTING INTERPRETER

Your script must be executable, like any other program. By now you know how to do that. Here is an example script called `cs107.sh`:

```
1 #!/usr/bin/env bash
2 echo "I am script $0, running inside $PWD."
3 echo "The following arguments were given:"
4 for arg in "$@"; do
5     echo $arg
6 done
```

- I save the script inside `$HOME/bin` because this path is in my `PATH` lookup. The verbose `.sh` is optional, you can choose any name you want. It is just another file.
- Make it executable and run it:

```
$ chmod 755 ~/bin/cs107.sh
$ pwd
/home/fabs
$ cs107.sh C S 1 0 7
```

- **What output do you expect?**

SHELL SCRIPTING INTERPRETER

- Make it executable and run it:

```
$ chmod 755 ~/bin/cs107.sh
$ pwd
/home/fabs
$ cs107.sh C S 1 0 7
```

- **What output do you expect?**

```
I am script /home/fabs/bin/cs107.sh, running inside /home/fabs.
The following arguments were given:
C
S
1
0
7
```

SHELL SCRIPTING SPECIAL VARIABLES

There are some special variables that you can make use of in your scripts and functions:

`$@`	Expands to quoted arguments. For previous example: `"C" "S" "1" "0" "7"`
`$0`	The full path of the script. Always use `$0` for your help messages in case you rename your script later.
`$1`, ..., `$9`	The first nine script arguments. For previous example: `$1=C`, `$5=7`
`$#`	The number of arguments given to the script

SHELL SCRIPTING FOR-LOOPS

Often you want to loop over a list of items:

```
#!/usr/bin/env bash
dir=$1  # what does this line do?
for f in $(find $dir -maxdepth 1 -type f -name "*.py"); do
    # f:  iteration variable
    # in: expects a list of items (for iteration)
    echo $f  # of course you do something more meaningful
done
```

The `$(...)` executes the statement inside the parenthesis in a sub-shell and returns the `stdout`. You can use pipes inside the parenthesis as well. Running such sub-commands is very useful in scripting.

SHELL SCRIPTING IF-CONDITIONALS

The general form for an `if` -conditional looks like this:

```
if [ condition_A ]; then
    # execute this block when condition_A is true
elif [ condition_B ]; then
    # execute this block when condition_B is true
else
    # execute this block otherwise
fi # except for loops, the end-delimiter of constructs is the construct name in reverse
```

Main reference for `if` -conditionals:
https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

SHELL SCRIPTING IF-CONDITIONALS

String comparisons:

`[ STRING1 == STRING2 ]`	Test for *equality*. For strict POSIX compliance you may use " = " instead of " == "
`[ STRING1 != STRING2 ]`	Test for *not* equal
`[ -z STRING ]`	True if the *length* of the string is zero
`[ -n STRING ]` or `[ STRING ]`	True if the length of the string is *non-zero*

SHELL SCRIPTING IF-CONDITIONALS

String comparisons: Example

```
1 #!/usr/bin/env bash
2 if [ "$1" == 'Hello CS107!' ]; then
3     echo 'Success!'
4 else
5     echo 'Got an unexpected argument'
6 fi
```

- **What output do you expect from the following invocation:**

```
$ example.sh Hello CS107!
```

SHELL SCRIPTING IF-CONDITIONALS

String comparisons: Example

```
1 #!/usr/bin/env bash
2 if [ "$1" == 'Hello CS107!' ]; then
3     echo 'Success!'
4 else
5     echo 'Got an unexpected argument'
6 fi
```

- **What output do you expect from the following invocation:**

```
$ example.sh Hello CS107!
Got an unexpected argument
```

- **How can we fix it?**

```
$ example.sh 'Hello CS107!'
Success!
```

SHELL SCRIPTING IF-CONDITIONALS

Integer comparisons: the general form is

```
[ INT1 OP INT2 ]
```

where `OP` is one of the following:

`-eq`	`INT1` is *equal* to `INT2`
`-ne`	`INT1` is *not equal* to `INT2`
`-lt`	`INT1` is *less than* `INT2`
`-le`	`INT1` is *less than or equal* to `INT2`
`-gt`	`INT1` is *greater than* `INT2`
`-ge`	`INT1` is *greater than or equal* to `INT2`

SHELL SCRIPTING IF-CONDITIONALS

Integer comparisons: Example

```
1 #!/usr/bin/env bash
2 if [ $# -gt 2 ]; then
3     echo "Number of arguments $# is larger than two"
4 else
5     echo "Number of arguments $# is less than or equal to two"
6 fi
```

Testing with different number of arguments:

```
$ example.sh a b
Number of arguments 2 is less than or equal to two
$ example.sh a b c
Number of arguments 3 is greater than two
```

SHELL SCRIPTING IF-CONDITIONALS

Often you need to test if files exist:

`[ -d FILE ]`	True if `FILE` exists and is a *directory*
`[ -f FILE ]`	True if `FILE` exists and is a *regular file*
`[ -e FILE ]`	True if `FILE` exists
`[ -r FILE ]`	True if `FILE` exists and is *readable*
`[ -w FILE ]`	True if `FILE` exists and is *writable*
`[ -x FILE ]`	True if `FILE` exists and is *executable*

Note that instead of `FILE` (which is some path to a file) you can also specify a *file descriptor* using `/dev/fd/n` with `n` the file descriptor ID. (`stdin=0`, `stdout=1`, `stderr=2`, ...)

SHELL SCRIPTING IF-CONDITIONALS

Testing for files: Example

```
#!/usr/bin/env bash
if [ $# -ne 1 ]; then
    cat <<EOF
    USAGE: $0 <path/to/file>

    More documentation here.  The form used here is called a here-document.
    They are very useful to write longer strings and expanding variables like
    \$0 above.  See https://tldp.org/LDP/abs/html/here-docs.html
EOF
    exit 1 # exit with failure code
fi

if [ -f $1 ]; then
    echo "File $1 exists and is a regular file"
elif [ -d $1 ]; then
    echo "File $1 exists and is a directory"
elif [ -e $1 ]; then
    echo "File $1 exists and is an unknown file"
fi
```

RECAP

- Take advantage of the shell customization capabilities
- I/O redirection is a powerful tool that you must master when you spend the majority of time in the shell
- Process management and suspension. Be considerate when terminating your processes.
- Environment variables and essentials of `bash` shell scripting