

Lecture 1

The build process, introduction to UNIX

Development Tools for Scientific Computing - SISSA, 2024-2025

Pasquale Claudio Africa, Dario Coscia

11 Feb 2025

Course overview

Practical info

- **Instructor:** Pasquale Claudio Africa pafrica@sissa.it
- **Assistant:** Dario Coscia dcoscia@sissa.it

Course material

- [GitHub](#): timetable, lecture notes, exercise sessions.
- Books (see course syllabus):
 - Parallel and High Performance Programming with Python*, Fabio Nelly.
 - Python Parallel Programming Cookbook*, Giancarlo Zaccone.
 - High Performance Python: Practical Performant Programming for Humans*, Micha Gorelick & Ian Ozvald.
- Internet (plenty of free or paid resources).

Practical info

Check out [GitHub](#) regularly for up-to-date timetable, rooms, lecture topics, and course material.

Course balance (approximate):

- ~24 hours.
- Frontal lectures: **40%**, practical sessions: **60%**.

For practical sessions please **bring your own laptop**.

Questions?

- Ask!
- Engage with each other!
- **Office hours**: send an email to book a session.

Syllabus (1/2)

Part I - Introduction

- Introduction to the UNIX shell.
- Version control (Git) and dependency management (conda, Docker).
- Hardware architectures and parallel computing paradigms.

Part II - Tools for scientific computing

- Python ecosystem for data science and scientific computing.
 - Data types for efficient computing.
 - NumPy, SciPy, scikit-learn, visualization tools.
 - Libraries for deep learning (PyTorch)
- Best practices for writing reliable code: error handling, unit testing, code profiling, optimization, and software deployment.

Syllabus (2/2)

Part III - Tools for High-Performance Computing (HPC)

- How to use HPC resources.
- Libraries for parallel computing (Numba, Lightning).

Required skills

- Former knowledge of programming fundamentals (syntax, data types, variables, control structures, functions).
- Prior experience with C, C++, Java, or Python, is **recommended**, but not mandatory.

Laptop configuration

Please **bring your own laptop** with a working UNIX/Linux environment, whether standalone, dual boot, or virtualized.

For beginners: <https://ubuntu.com/tutorials/install-ubuntu-desktop> .

You can write code using any **text editor** (such as Emacs, Vim, or Nano), or an Integrated Development Environment (**IDE**) (such as VSCode, Eclipse, or Spyder).

Requirements

- Python 3. The presence of Jupyter and conda is recommended.
- A C++ compiler installed with full support for C++17, such as GCC 10 or newer, or Clang 11 or newer. The presence of CMake is recommended.
- [Docker Desktop](#) . Please follow the instruction on the [official guide](#) and the [post-installation steps](#) thoroughly.

Windows users

- **Windows Subsystem for Linux (WSL2)** . Ubuntu version recommended, then follow Ubuntu-specific instructions.
- Virtual machine (such as **VirtualBox**).
- **(Expert users)** **Dual boot** .

macOS users

- **Xcode** : provides Clang.
- **Homebrew** : provides GCC, Clang, Python 3.

Linux users

- Install Python 3 and GCC using your package manager (such as apt, yum, pacman).

Why should I learn development tools in the AI-dominated era?

1. You can't really understand/modify/improve a text written in English, unless you are proficient in English!
2. Career opportunities:
 - Coding opens doors to a diverse array of high-demand careers in technology and data-driven sectors.
 - It cultivates critical thinking and problem-solving skills.
3. Artificial Intelligence (AI) and chatbots **lack** creativity!
 - They derive knowledge from historical data.
 - Innovation, idea generation, implementation of novel concepts through software and technology remain a **human** prerogative (at least for now 😊).
4. Understand how AI and chatbots work under the hood.

Popularity of programming languages

Source: <https://pypl.github.io/PYPL.html>

Curated lists of awesome C++ and Python frameworks, libraries, resources, and shiny things.

- [awesome-cpp](#)
- [awesome-python](#)
- [awesome-scientific-python](#)
- [awesome-scientific-computing](#)

Outline

1. The build process:
 - Compiled vs. interpreted languages.
 - Preprocessor, compiler, linker, loader.
2. Introduction to the UNIX shell:
 - What is a shell.
 - Variables.
 - Basic commands and scripting.
3. Shell scripting.

The build process: **Preprocessor, Compiler, Linker, Loader**

Compiled vs. interpreted languages

The build process

Preprocessor

- Handles directives and macros before compilation.
- Originated for code reusability and organization.

Preprocessor directives

- `#include` : Includes header files.
- `#define` : Defines macros for code replacement.
- `#ifdef` , `#ifndef` , `#else` , `#endif` : Conditional compilation.
- `#pragma` : Compiler-specific directives.

Macros

- Example: `#define SQUARE(x) ((x) * (x))`
- Usage: `int result = SQUARE(5); // Expands to: ((5) * (5))`

Compiler

- Translates source code into assembly/machine code.
- Evolved with programming languages and instructions.

Compilation process

1. Lexical analysis: Tokenization.
2. Syntax analysis (parsing): Syntax tree.
3. Semantic analysis: Checking.
4. Code generation: Assembly/machine code.
5. Optimization: Efficiency improvement.
6. Output: Object files.

Common compiler options

`-o` : Optimization levels; `-g` : Debugging info; `-std` : C++ standard.

Linker

- Combines object files into an executable.
- Supports modular code.

Linking process

1. Symbol resolution: Match symbols.
2. Relocation: Adjust addresses.
3. Output: Executable.
4. Linker errors/warnings.
5. Example: `g++ main.o helper.o -o my_program`

Static vs. dynamic linking

- Static: Larger binary, library inclusion.
- Dynamic: Smaller binary, runtime library reference.

Loader

- Loads executables for execution.
- Tied to memory management evolution.

Loading process

1. Memory allocation: Reserve memory.
2. Relocation: Adjust addresses.
3. Initialization: Set up environment.
4. Execution: Start execution.

Dynamic linking at runtime

- Inclusion of external libraries during execution.
- Enhances flexibility.

Introduction to the UNIX shell

What is a shell?

From <http://www.linfo.org/shell.html> :

A shell is a program that provides the traditional, text-only user interface for Linux and other UNIX-like operating systems. Its primary function is to read commands that are typed into a console [...] and then execute (i.e., run) them. The term shell derives its name from the fact that it is an outer layer of an operating system. A shell is an interface between the user and the internal parts of the OS (at the very core of which is the kernel).

What shells are available?

`Bash` stands for: `Bourne Again Shell`, a homage to its creator Stephen Bourne. It is the default shell for most UNIX systems and Linux distributions. It is both a command interpreter and a scripting language. The shell might be changed by simply typing its name and even the default shell might be changed for all sessions.

macOS has replaced it with `zsh`, which is mostly compatible with `Bash`, since v10.15 Catalina.

Other shells available: `tsh`, `ksh`, `cs`, `Dash`, `Fish`, `Windows PowerShell`, ...

Variables and environmental variables

As shell is a program, it has its variables. You can assign a value to a variable with the equal sign **(no spaces!)**, for instance type `A=1` . You can then retrieve its value using the dollar sign and curly braces, for instance to display it the user may type `echo ${A}` .

Some variables can affect the way running processes will behave on a computer, these are called **environmental variables**. For this reason, some variables are set by default, for instance to display the user home directory type `echo ${HOME}` .

To set an environmental variable just prepend `export` , for instance `export PATH="/usr/sbin:$PATH"` adds the folder `/usr/sbin` to the `PATH` environment variable. `PATH` specifies a set of directories where executable programs are located.

Types of shell (login vs. non-login)

- A **login** shell logs you into the system as a specific user (it requires username and password). When you hit `Ctrl+Alt+F1` to login into a virtual terminal you get after successful login: a login shell (that is interactive).
- A **non-login** shell is executed without logging in (it requires a current logged in user). When you open a graphic terminal it is a non-login (interactive) shell.

Types of shell (interactive vs. non-interactive)

- In an **interactive** shell (login or non-login) you can interactively type or interrupt commands. For example a graphic terminal (non-login) or a virtual terminal (login). In an interactive shell the prompt variable must be set (`$PS1`).
- A **non-interactive** shell is usually run from an automated process. Input and output are not exposed (unless explicitly handled by the calling process). This is normally a non-login shell, because the calling user has logged in already. A shell running a script is always a non-interactive shell (but the script can emulate an interactive shell by prompting the user to input values).

The shell as a command line interpreter

When launching a terminal a UNIX system first launches the shell interpreter specified in the `SHELL` **environment variable**. If `SHELL` is unset it uses the system default.

After having sourced the initialization files, the interpreter shows the **prompt** (defined by the environment variable `$PS1`).

Initialization files are hidden files stored in the user's home directory, executed as soon as an **interactive** shell is run.

Initialization files

Initialization files in a shell are scripts or configuration files that are executed or sourced when the shell starts. These files are used to set up the shell environment, customize its behavior, and define various settings that affect how the shell operates.

- **login:**

- `/etc/profile` , `/etc/profile.d/*` , `~/.profile` for Bourne-compatible shells
- `~/.bash_profile` (or `~/.bash_login`) for `Bash`
- `/etc/zprofile` , `~/.zprofile` for `zsh`
- `/etc/csh.login` , `~/.login` for `csh`

- **non-login:** `/etc/bash.bashrc` , `~/.bashrc` for `Bash`

Initialization files

- **interactive:**

- `/etc/profile` , `/etc/profile.d/*` and `~/.profile`
- `/etc/bash.bashrc` , `~/.bashrc` for `Bash`

- **non-interactive:**

- `/etc/bash.bashrc` for `Bash` (but most of the times the script begins with: `[-z "$PS1"] && return` , *i.e.* don't do anything if it's a non-interactive shell).
- depending on the shell, the file specified in `$ENV` (or `$BASH_ENV`) might be read.

Getting started


To get a little hang of the shell, let's try a few simple commands:

- `echo` : prints whatever you type at the shell prompt.
- `date` : displays the current time and date.
- `clear` : clean the terminal.

Basic shell commands (1/2)

- `pwd` stands for **Print working directory** and it points to the current working directory, that is, the directory that the shell is currently looking at. It's also the default place where the shell commands will look for data files.
- `ls` stands for **List** and it lists the contents of a directory. `ls` usually starts out looking at our home directory. This means if we print `ls` by itself, it will always print the contents of the current directory.
- `cd` stands for **Change directory** and changes the active directory to the path specified.

Basic shell commands (2/2)

- `cp` stands for **C**opy and it moves one or more files or directories from one place to another. We need to specify what we want to move, i.e., the source and where we want to move them, i.e., the destination.
- `mv` stands for **M**ove and it moves one or more files or directories from one place to another. We need to specify what we want to move, i.e., the source and where we want to move them, i.e., the destination.
- `touch` command is used to create new, empty files. It is also used to change the timestamps on existing files and directories.
- `mkdir` stands for **M**ake **d**irectory and is used to make a new directory or a folder.
- `rm` stands for **R**emove and it removes files or directories. By default, it does not remove directories, unless you provide the flag `rm -r` (`-r` means recursively).
 **Warning:** Files removed via `rm` are lost forever, please be careful!

Shell scripts

Commands can be written in a **script file**, i.e. a text file that can be executed.

Remember that the **first line of the script** (the so-called *shebang*) tells the shell which interpreter to use while executing the file. So, for example, if your script starts with `#!/bin/bash` it will be run by `Bash`, if it starts with `#!/usr/bin/env python` it will be run by `Python`.

To run your brand new script you may need to change the access permissions of the file. To make a file executable run

```
chmod +x script_file
```


Not all commands are equals

When executing a command, like `ls` a subprocess is created. A subprocess inherits all the environment variables from the parent process, executes the command and returns the control to the calling process.

A subprocess cannot change the state of the calling process.

The command `source script_file` executes the commands contained in `script_file` as if they were typed directly on the terminal. It is only used on scripts that have to change some environmental variables or define aliases or function. Typing `. script_file` does the same.

If the environment should not be altered, use `./script_file`, instead.

Built-in commands

Some commands, like `cd` are executed directly by the shell, without creating a subprocess.

Indeed it would be impossible to have `cd` as a regular command!

The reason is: a subprocess cannot change the state of the calling process, whereas `cd` needs to change the value of the environmental variable `PWD` (that contains the name of the current working directory).

Other commands

In general a **command** can refer to:

- A builtin command.
- An executable.
- A function.

The shell looks for executables with a given name within directories specified in the environment variable `PATH`, whereas aliases and functions are usually sourced by the `.bashrc` file (or equivalent).

- To check what `command_name` is: `type command_name` .
- To check its location: `which command_name` .

A warning about filenames

⚠ In order to live happily and without worries, **don't** use spaces nor accented characters in filenames!

Space characters in file names should be forbidden by law! The space is used as separation character, having it in a file name makes things a lot more complicated in any script (not just shell scripts).

Use underscores (snake case): `my_wonderful_file_name`, or uppercase characters (camel case): `myWonderfulFileName`, or hyphens: `my-wonderful-file-name`, or a mixture: `myWonderful_file-name`, instead.

But **not** `my wonderful file name`. It is not wonderful at all if it has to be parsed in a script.

More commands

- `cat` stands for **C**oncatenate and it reads a file and outputs its content. It can read any number of files, and hence the name concatenate.
- `wc` is short for **W**ord **C**ount. It reads a list of files and generates one or more of the following statistics: newline count, word count, and byte count.
- `grep` stands for **G**lobal **r**egular **e**xpression **p**rint. It searches for lines with a given string or looks for a pattern in a given input stream.
- `head` shows the first line(s) of a file.
- `tail` shows the last line(s) of a file.
- `file` reads the files specified and performs a series of tests in attempt to classify them by type.

Redirection, pipelines and filters

We can add operators between commands in order to chain them together.

- The pipe operator `|`, forwards the output of one command to another. E.g., `cat /etc/passwd | grep my_username` checks system information about "my_username".
- The redirect operator `>` sends the standard output of one command to a file. E.g., `ls > files-in-this-folder.txt` saves a file with the list of files.
- The append operator `>>` appends the output of one command to a file.
- The operator `&>` sends the standard output and the standard error to file.
- `&&` pipe is activated only if the return status of the first command is 0. It is used to chain commands together: e.g., `sudo apt update && sudo apt upgrade`
- `||` pipe is activated only if the return status of first command is different from 0.
- `;` is a way to execute to commands regardless of the output status.
- `$?` is a variable containing the output status of the last command.

Advanced commands

- `tr` stands for **translate**. It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters, and basic find and replace. For instance:
 - `echo "Welcome to Advanced Programming!" | tr [a-z] [A-Z]` converts all characters to upper case.
 - `echo -e "A;B;c\n1,2;1,4;1,8" | tr ", " "." | tr ";" ", "` replaces commas with dots and semi-colons with commas.
 - `echo "My ID is 73535" | tr -d [:digit:]` deletes all the digits from the string.

Advanced commands

- `sed` stands for **stream editor** and it can perform lots of functions on file like searching, find and replace, insertion or deletion. We give just an hint of its true power
 - `echo "UNIX is great OS. UNIX is open source." | sed "s/UNIX/Linux/"` replaces the first occurrence of "UNIX" with "Linux".
 - `echo "UNIX is great OS. UNIX is open source." | sed "s/UNIX/Linux/2"` replaces the second occurrence of "UNIX" with "Linux".
 - `echo "UNIX is great OS. UNIX is open source." | sed "s/UNIX/Linux/g"` replaces all occurrences of "UNIX" with "Linux".
 - `echo -e "ABC\nabc" | sed "/abc/d"` delete lines matching "abc".
 - `echo -e "1\n2\n3\n4\n5\n6\n7\n8" | sed "3,6d"` delete lines from 3 to 6.

Advanced commands

- `cut` is a command for cutting out the sections from each line of files and writing the result to standard output.
 - `cut -b 1-3,7- state.txt` cut bytes (`-b`) from 1 to 3 and from 7 to end of the line
 - `echo -e "A,B,C\n1.22,1.2,3\n5,6,7\n9.99999,0,0" | cut -d "," -f 1` get the first column of a CSV (`-d` specifies the column delimiter, `-f n` specifies to pick the n -th column from each line)
- `find` is used to find files in specified directories that meet certain conditions. For example:
`find . -type d -name "*lib*"` find all directories (not files) starting from the current one (`.`) whose name contain "lib".
- `locate` is less powerful than `find` but much faster since it relies on a database that is updated on a daily base or manually using the command `updatedb` . For example: `locate -i foo` finds all files or directories whose name contains `foo` ignoring case.

Quotes

Double quotes may be used to identify a string where the variables are interpreted. Single quotes identify a string where variables are not interpreted. Check the output of the following commands

```
a=yes  
echo "$a"  
echo '$a'
```

The output of a command can be converted into a string and assigned to a variable for later reuse:

```
list=`ls -l` # Or, equivalently:  
list=$(ls -l)
```

Processes

- Run a command in background: `./my_command &`
- `ctrl-Z` suspends the current subprocess.
- `jobs` lists all subprocesses running in the background in the terminal.
- `bg %n` reactivates the n -th subprocess and sends it to the background.
- `fg %n` brings the n -th subprocess back to the foreground.
- `Ctrl-C` terminates the subprocess in the foreground (when not trapped).
- `kill pid` sends termination signal to the subprocess with id `pid`. You can get a list of the most computationally expensive processes with `top` and a complete list with `ps aux` (usually `ps aux` is filtered through a pipe with `grep`)

All subprocesses in the background of the terminal are terminated when the terminal is closed (unless launched with `nohup`, but that is another story...)

How to get help

Most commands provide a `-h` or `--help` flag to print a short help information:

```
find -h
```

`man command` prints the documentation manual for command.

There is also an info facility that sometimes provides more information: `info command`.

Shell scripting

Functions

A **function** in a shell is a block of reusable code that you can define and call throughout your script. Functions are useful for organizing complex scripts and avoiding repetition. The general syntax for defining a function is:

```
function_name() {  
    # Commands to be executed.  
}
```

Example:

```
greet() {  
    echo "Hello, $1!"  
}
```

In this example, `greet` is a function that takes one argument and echoes a greeting message.

Input arguments in a script or in a function

- `$0` : The name of the script/function itself.
- `$1` , `$2` , `$3` , etc.: The first, second, third (and so on) argument passed to the script/function.
- `$#` : The number of arguments passed.
- `$@` : The list of all the arguments passed as a single string.
- `$*` : All the arguments as a single word (not often used).

Writing and running shell scripts

A shell script is simply a file containing a sequence of commands. It starts with a *shebang* (`#!`) that tells the system which interpreter to use.

Example:

```
#!/bin/bash
echo "Hello, World!"
```

Make the script executable and run it:

```
chmod +x my_script.sh
./my_script.sh
```


Variables and user input

Shell variables store data and can be set as follows:

```
name="Alice"  
echo "Hello, $name!"
```

Reading user input:

```
echo "Enter your name:"  
read user_name  
echo "Welcome, ${user_name}!"
```

Conditional statements: `if`

```
#!/bin/bash
echo "Enter a number:"
read num

if [ $num -gt 10 ]; then
    echo "Number is greater than 10."
else
    echo "Number is 10 or less."
fi
```

[Bash conditional expressions](#)

[POSIX shell cheat sheet](#)

Conditional statements: case

```
echo "Choose an option: start, stop, restart"
read action

case $action in
  start) echo "Starting service...";;
  stop) echo "Stopping service...";;
  restart) echo "Restarting service...";;
  *) echo "Invalid option";;
esac
```

Loops (for, while)

for loop:

```
for i in 1 2 3 4 5
do
    echo "Iteration $i"
done
```

while loop:

```
count=1
while [ $count -le 5 ]
do
    echo "Count: $count"
    ((count++))
done
```

Loops (until)

until loop (runs until condition becomes true):

```
num=0
until [ $num -eq 3 ]
do
    echo "Number: $num"
    ((num++))
done
```

Error handling and debugging

- `set -e` : Exit script on error.
- `set -x` : Enable debugging (prints each command before execution).
- `trap` : Catch errors and execute custom actions.

Example:

```
trap 'echo "An error occurred!"' ERR

function cleanup() { ... }
trap cleanup EXIT

set -e

mkdir my_directory
cd my_directory
rm nonexistent_file # This will trigger the trap.
```

Parsing command-line arguments

Using `$1`, `$2`, etc. to read input:

```
echo "First argument: $1"  
echo "Second argument: $2"
```

Scripts share the same syntax as functions for parsing arguments.

More advanced argument handling with `getopts`:

```
while getopts "u:p:" opt; do  
  case $opt in  
    u) username=$OPTARG ;;  
    p) password=$OPTARG ;;  
    *) echo "Invalid option" ;;  
  esac  
done  
  
echo "User: $username, Password: $password"
```

Functions in shell scripts

```
#!/bin/bash

my_function() {
    echo "Function name: $0"
    echo "First argument: $1"
    echo "Second argument: $2"
    echo "All arguments (\$@): @$@" # As separate strings.
    echo "All arguments (\$*): $*" # As a single string.
    echo "Number of arguments: $#"
```



```
}

my_function "Alice" "Bob" "Charlie"
```


➔ Introduction to `git`
