

Topic 2: Programming for biologists

Included in this topic

- Why command line
- Thinking ahead: Reproducibility
- Shell basics
- Common gotchas for newcomers
- Common languages

Non-command line options

- Galaxy (<https://www.galaxyproject.org/>)
- Open-source, run much of the same tools



No background in
bioinformatics required

- Commercial

Would you prefer to own both the experimental design and the data analysis? We're bringing sophisticated bioinformatics to the fingertips of the life scientists, making sure that critical information isn't lost.

[> Find out how](#)




- CLCbio Genomics Workbench

- Commercial, includes assembly










Galaxy

- Run pre-established tools, setup is form-based
- Jobs submitted to a cluster of servers (running the software stack)

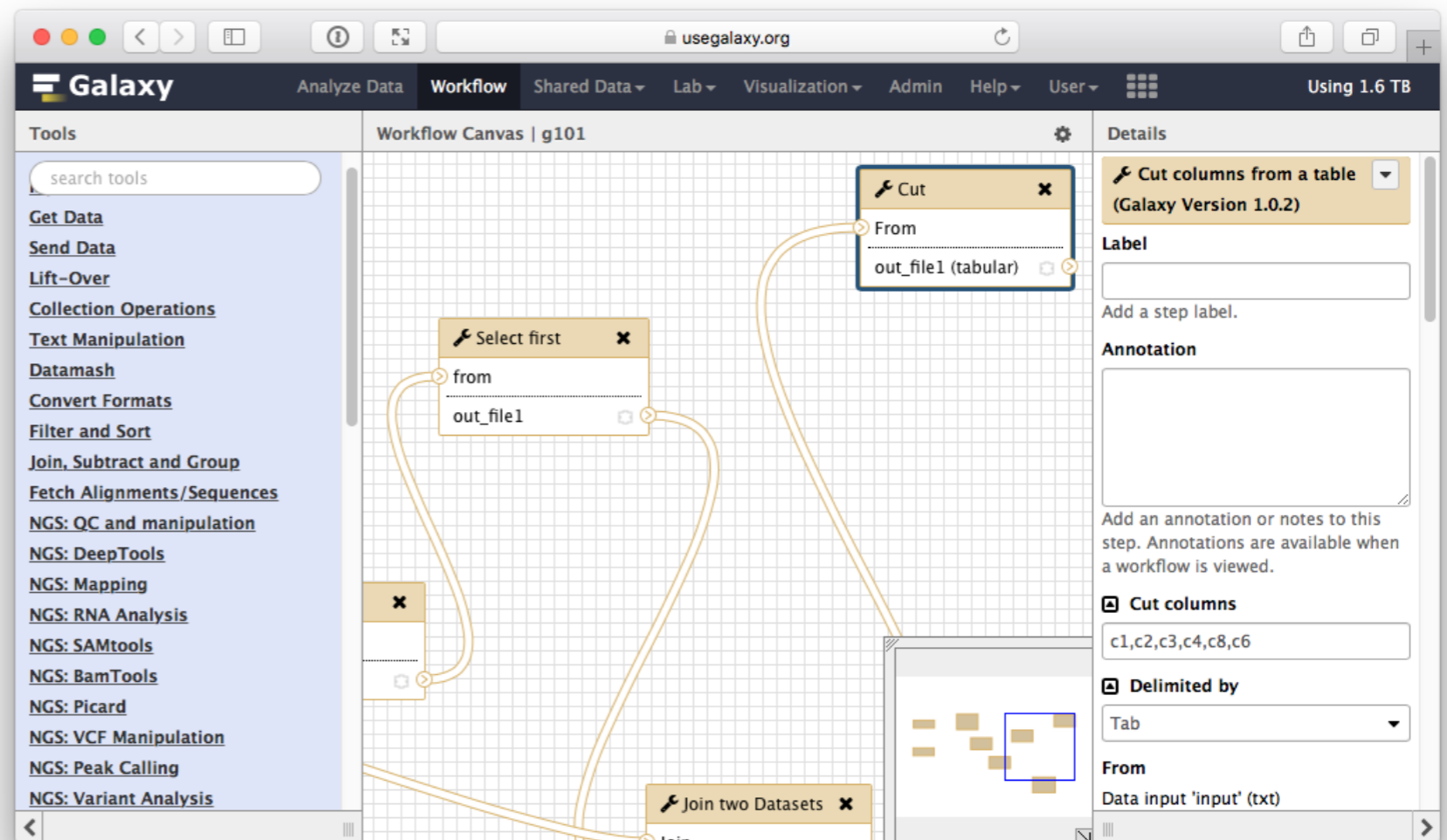
8: Cut on data 7   

5 regions
format: **bed**, database: **hg38**

display in IGB View
display with IGV [local](#)
display at UCSC [main](#)



1. Chrom	2. Start	3. End	4. Name
chr22	15528158	15529139	uc011agc
chr22	15690077	15690709	uc010gqp
chr22	15690245	15690709	uc062bek
chr22	22376182	22376505	uc062cbs
chr22	46256560	46263322	uc003bhf





Galaxy Analyze Data **Workflow** Shared Data Lab Visualization Admin Help User Using 1.6 TB

Tools search tools



Workflow Canvas | g101

Cut  

From
out_file1 (tabular)

Select first  

from
out_file1

Join two Datasets  

Details

Cut columns from a table (Galaxy Version 1.0.2)

Label

Add a step label.

Annotation

Add an annotation or notes to this step. Annotations are available when a workflow is viewed.

Cut columns

c1,c2,c3,c4,c8,c6

Delimited by

Tab

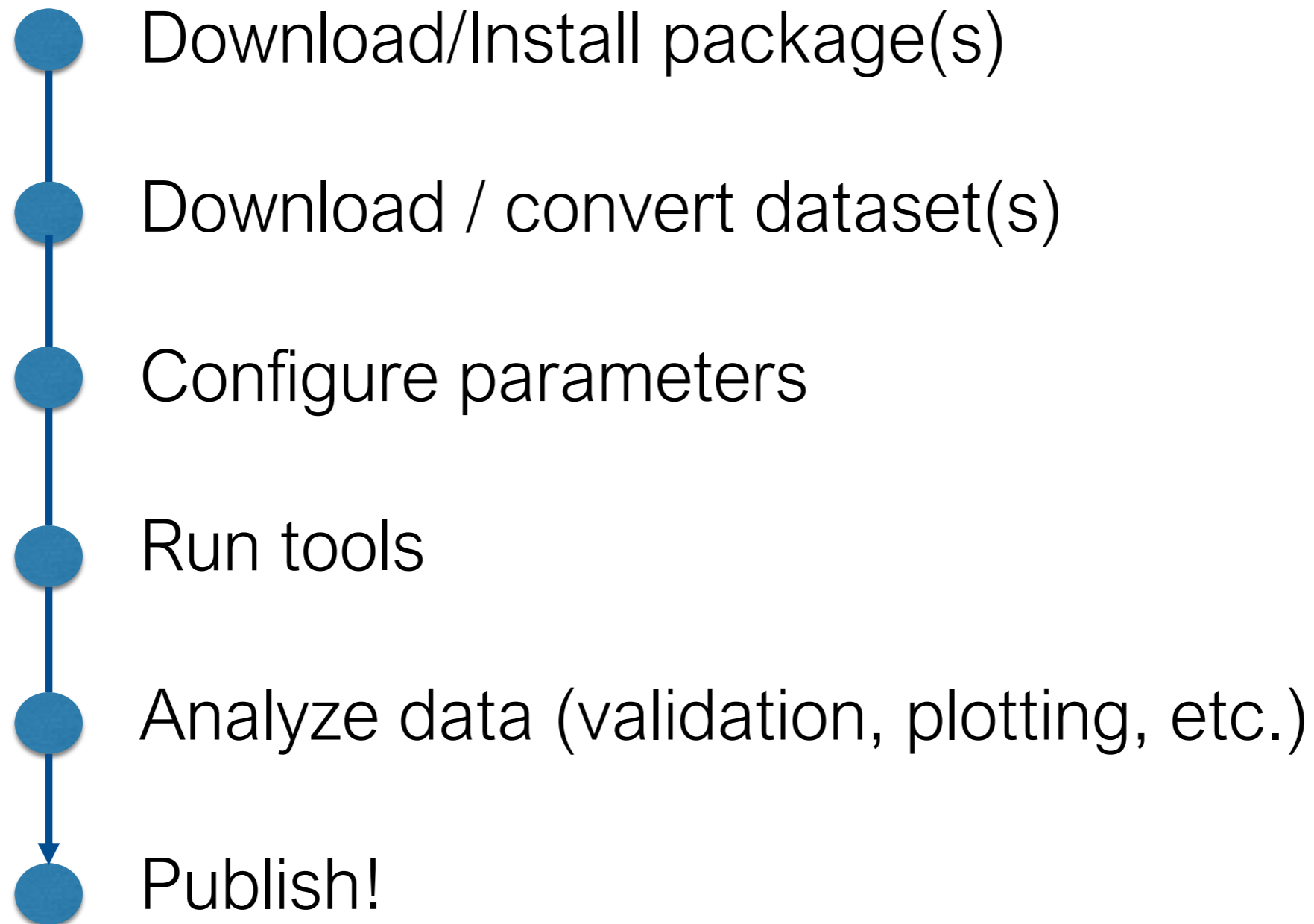
From

Data input 'input' (txt)

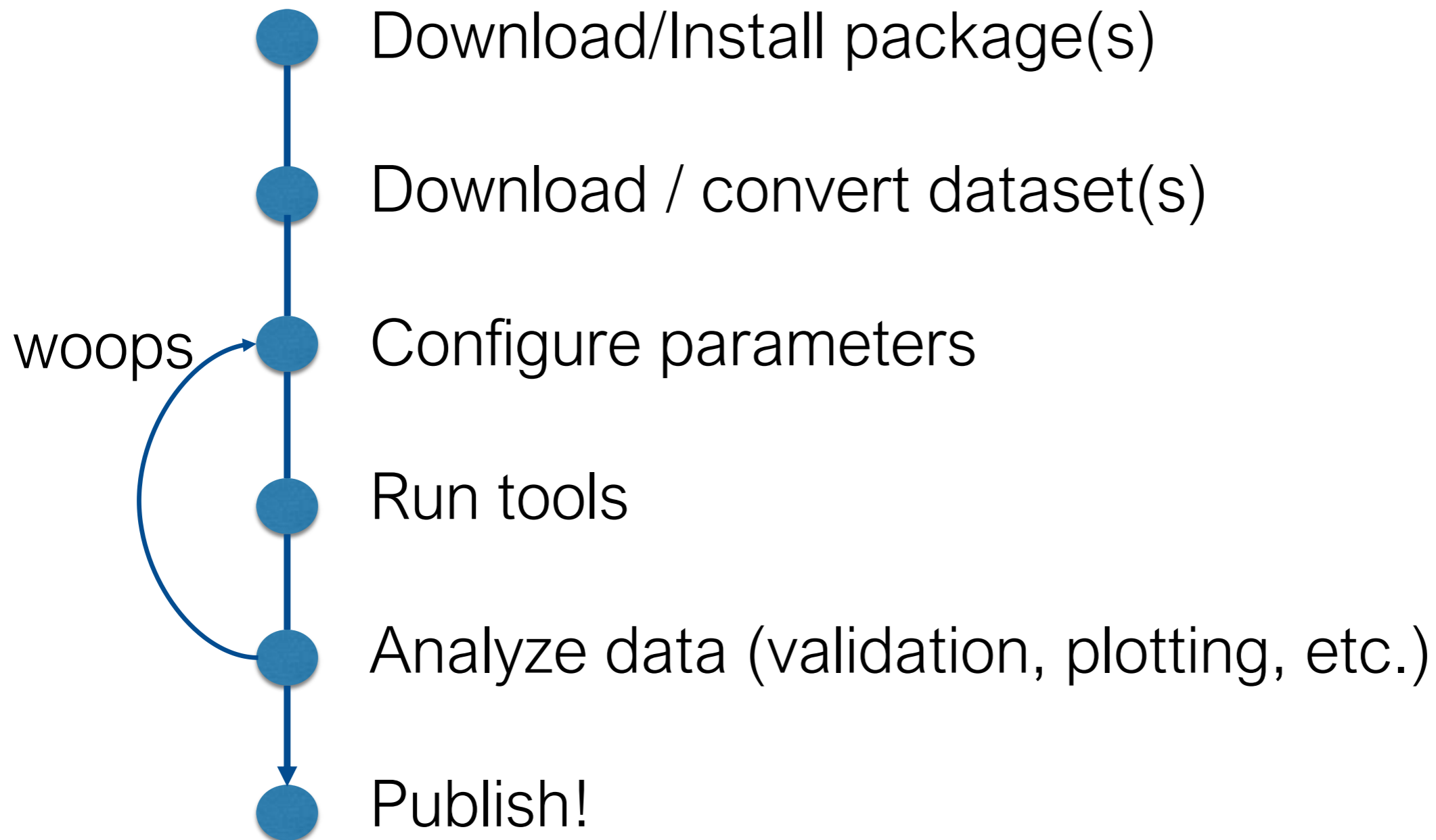
Why use command line?

- Extract subsets of a large file (line range, column)
- Steps recorded, repeatable, auditable
- Powerful text editing tools. Powerful code revision tools
- Generally faster than GUI based methods
- Most scientific programs do not have a GUI (necessity)
- Allows different programs to be combined arbitrarily
- Free

Command line actions



Command line actions



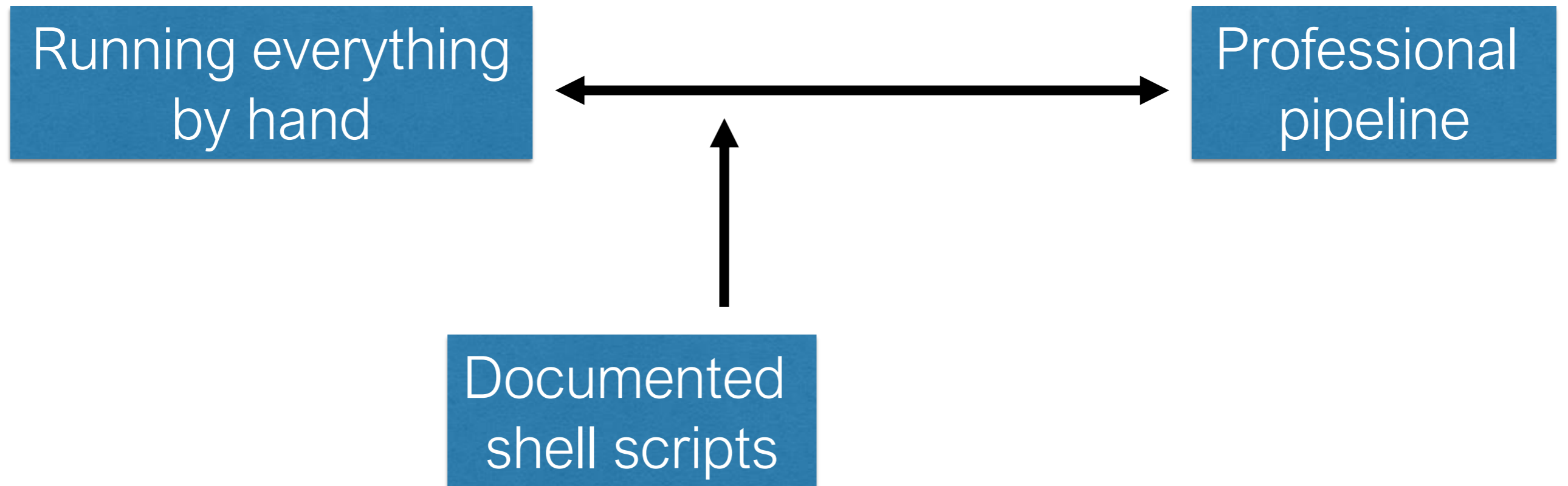
Axis of reproducibility

Running everything
by hand

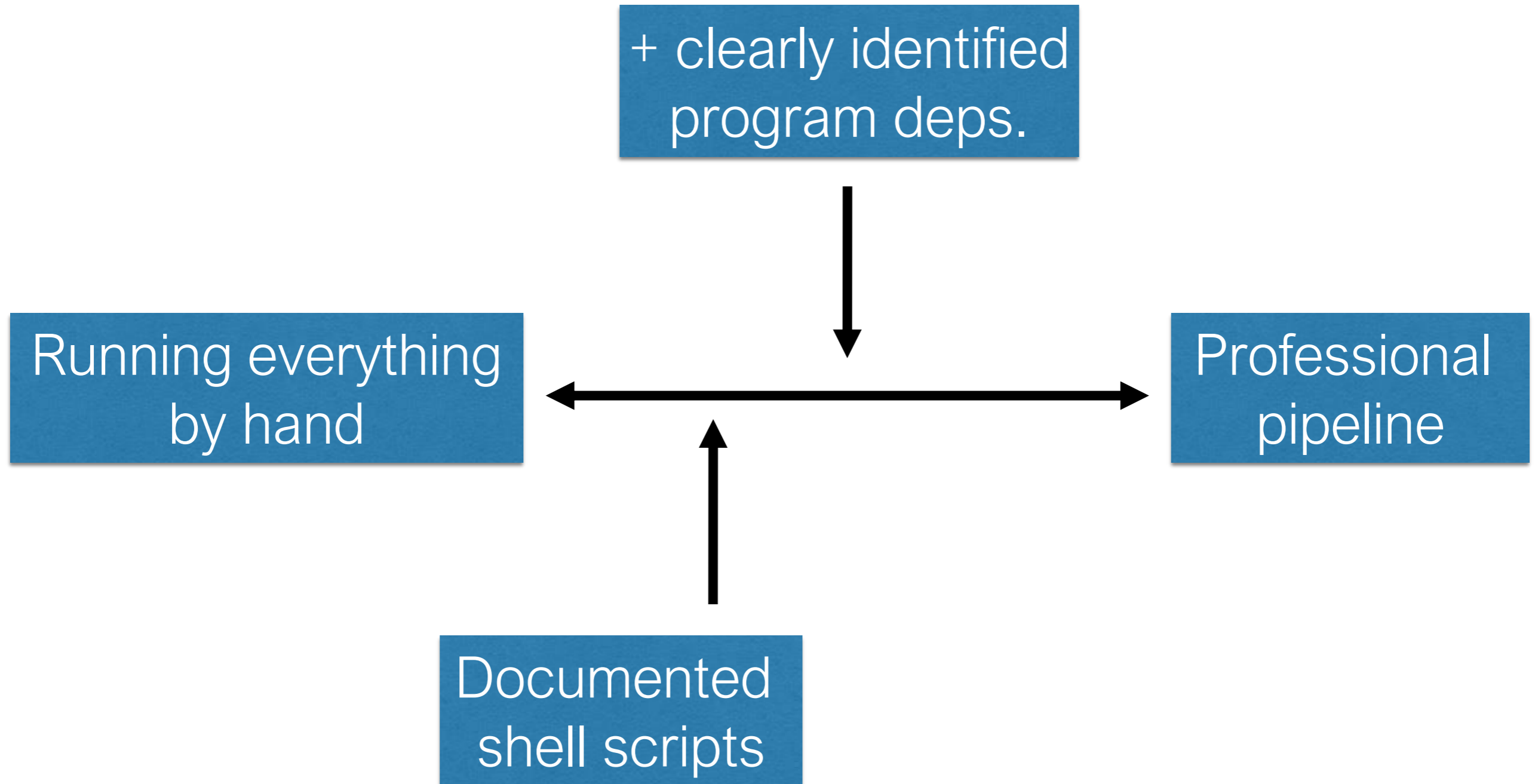


Professional
pipeline

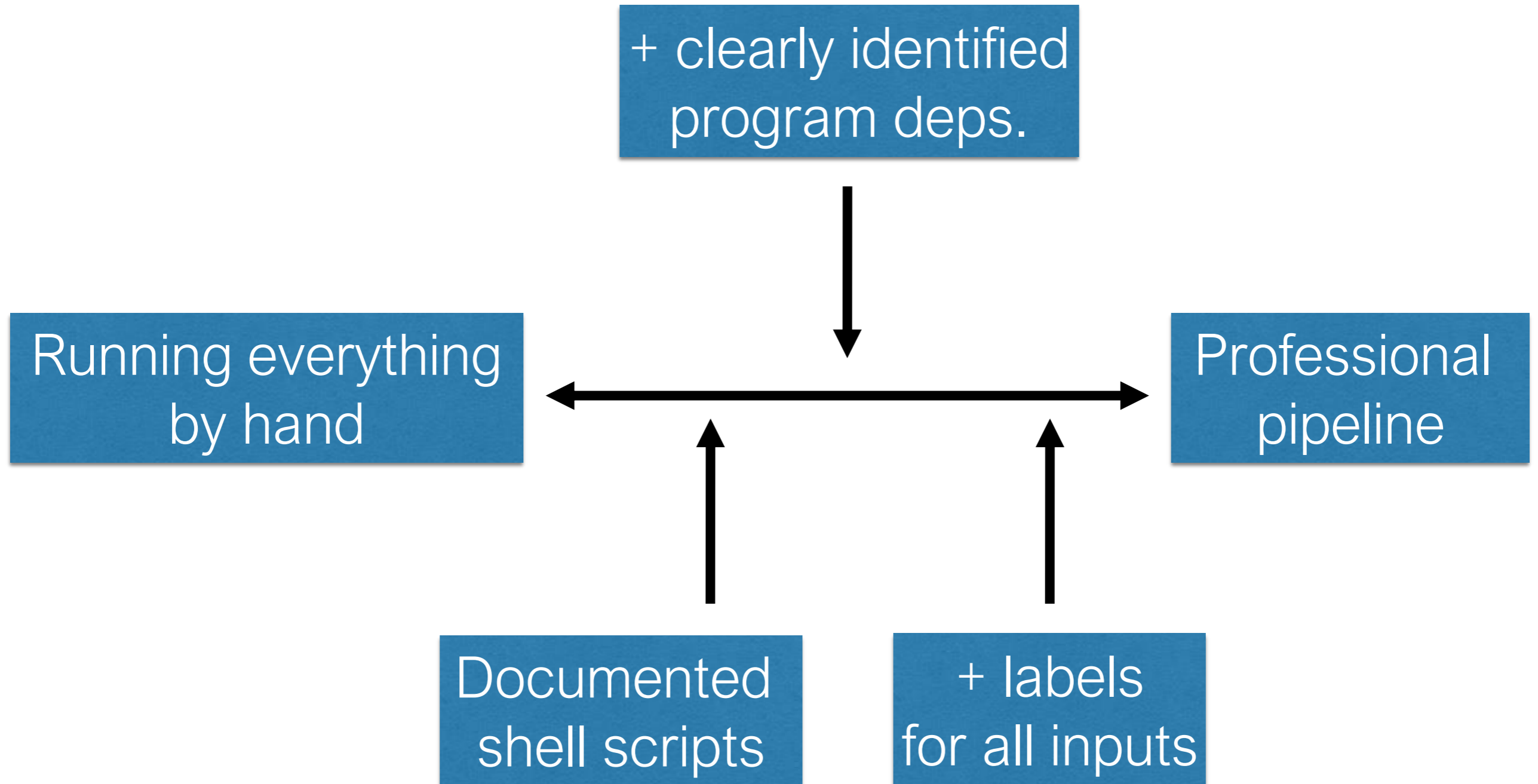
Axis of reproducibility



Axis of reproducibility

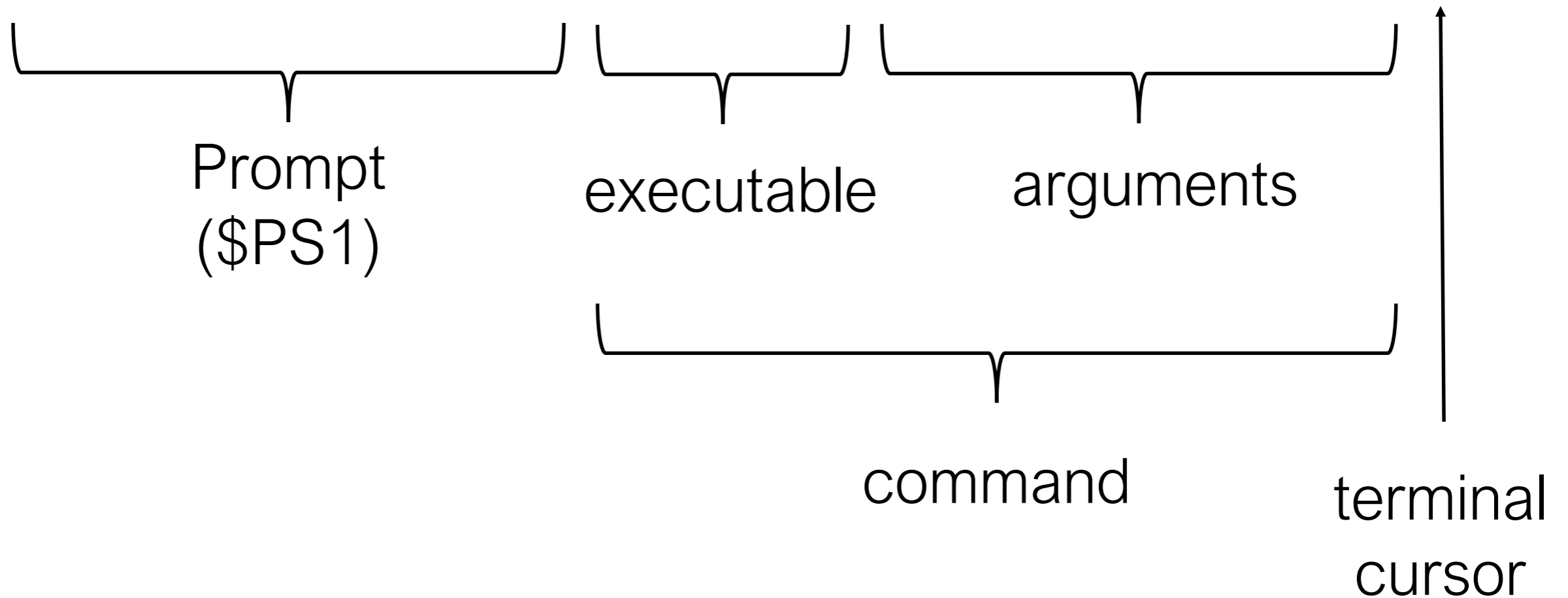


Axis of reproducibility

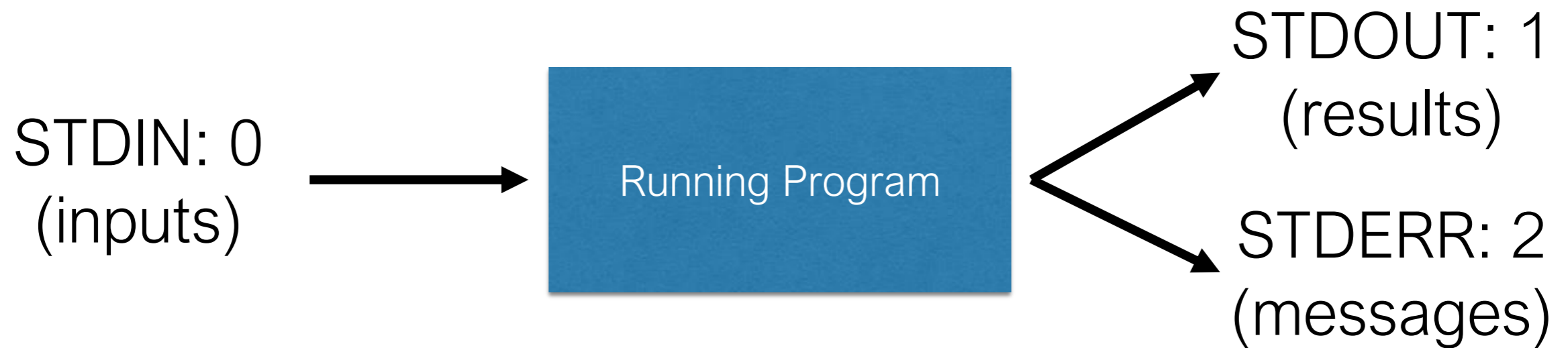


Interface: prompt

```
jslegare@meteoric:~/src$ my_program arg1 arg2 arg3 arg4 ...
```



Interface: stdio, exit codes



- On start, programs inherit 3 open files (They can open more)
- On exit, they provide an integer exit code for the parent
 - convention: 0 is success, non-0 is failure (i.e. 1 to 255)

Interface: signals

```
$ kill -KILL 100
```



- Signals are one-way messages sent to control running programs
 - SIGINT (interrupt) (CTRL-C on terminal)
 - SIGSTOP (pause) / SIGCONT (CTRL-Z on terminal / fg)
 - SIGKILL (kill the process immediately) (use `kill -9 PID`)

Interface: grammar (GNU syntax)

- Programs take options and parameters. Example:

```
rm -i -f --one-file-system --interactive=always A.txt B.gz
```

- `rm`: the executable/program. always first. the other arguments are specific to that program.
- `-i -f`: short options. single ``-``. relative order is generally unimportant. collapsible, e.g `-if`
- `--one-file-system`: long options. double ``-``. some short and long forms are equivalent.
- `=always`: some options take a param value. ``=`` introduces the value applied to the preceding long option. Short options can take a parameter, but without ``=``, e.g.: ``tar -cf F.gz mydir``
- `A.txt B.gz`: Positional parameters. Variable number of them can be provided. Order matters.

which mycmd

specify exactly which program to run by providing a file path
>\$ /path/to/mycmd -i

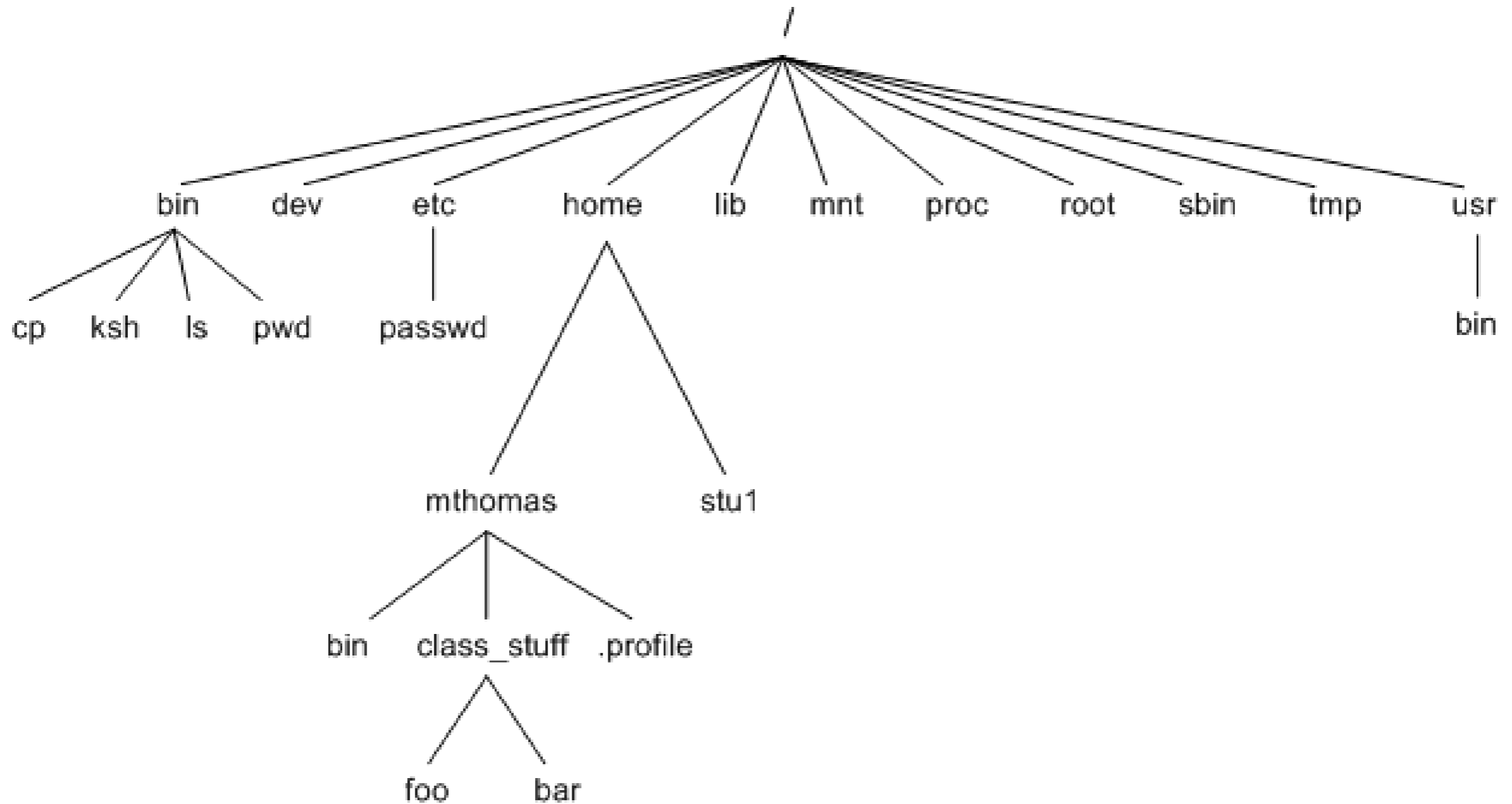
specify just the name - use lookup procedure to find it
>\$ mycmd -i

1. reserved names (builtins): cd, for, do, kill, help, etc.
2. aliases (user-defined shell macros)
3. shell functions
4. file "mycmd" under each dir found in variable \$PATH

If all else fails:
throws
"command not found"

/a/b/../c ./d/e/f

- Most files that you will analyze have a name (path) in a hierarchy of files (the filesystem). Root = “/”
- “Everything is a file”: Some files don’t have a name in the hierarchy. Network sockets, and pipes for IPC are examples of files. (more on this later)
- Paths starting with “/” are absolute. Else, relative.
- On unix filesystems, names can contain any character except ‘/’ and ‘\0’ (nil). *some names are reserved: “.”, “..”



[http://homepages.uc.edu/~thomam/Intro_Unix_Text/File_System.html]

Access control follows inheritance:

- Is /home/mthomas/bin (requires +x from / to leaf)
- Is /home/mthomas/bin (requires +r on the leaf)

Executable files

Not every file can be run as a standalone program. Files need to be marked as “executable”

```
$ ./mycmd
```

```
bash: ./mycmd: Permission denied
```

```
$ chmod +x mycmd # mark as executable
```

To open a directory, it must be marked executable.

Programming Languages

- C++ (.cpp)
- Shell scripting (.sh)
- Perl (.pl)
- Python (.py)
- R (.r)

Text Programs

myprog

```
#!/usr/bin/perl
$age = 25;           # An integer assignment
$name = "John Paul"; # A string
$salary = 1445.50;  # A floating point
print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";
```

- The `#!` shebang specifies the interpreter program
 - “./myprog” \approx “/usr/bin/perl ./myprog”

C++

- Compiles into very fast binaries.
- Few prepackaged tools
 - More control over use of computer resources.
 - Requires programmers to be explicit about everything. Longer development time.
- Harder to tweak and debug (due to the binary nature)
- Can be a good option for a low-level plugin/library

Shell scripting

- Basically, taking command line arguments and putting them in a reusable script
- Easy to write.
- Provides glue between programs written in different languages.
- Connects with external programs (sed, grep, awk) naturally
- Needs external programs to manipulate data.

Perl

- Excels at text processing.
- Some prepackaged functions.
- Lacks complicated data structures.
- Very concise (at times, at the expense of readability)
- Slower than C++, but still fast.

Python

- Slightly slower than Perl.
- Batteries included. Scientific methods and plotting.
- Popular.
- Allows rapid prototyping.
- Good for reformatting data. Good data structures.

R

- Many packages for specific scientific tasks and statistics.
- Great at plotting.
- Harder to use with big data (GB+ files), although work arounds exist.
- Slowest option.

Recommendations

- Genomic dabbler
 - Shell and R
- Genomic scientist
 - Shell, R and Python/Perl
- Bioinformatician
 - Shell, R, Python/Perl and C++

The Computer Language Benchmarks Game

“Which programming language is fastest?”

Should we care? How could we know?

“It's important to be realistic: most people don't care about program performance most of the time.”

“By instrumenting the ... runtime, we measure the JavaScript behavior of ... web applications... Our results show that real web applications behave very differently from the benchmarks...”

<https://benchmarksgame-team.pages.debian.net/benchmarksgame/>

- Numerous repetitive tasks demand speed
- Software development demands time

Recommendations

General order:

1. Make it work
2. Make it go fast
3. Make it pretty

Language choice considerations:

- Meld with existing codebase
- Your current and future colleagues (expertise)

Tools we will learn:

- Unix shell
- byobu
- grep and sed

Coding Example!

Compute Canada servers

- Uses a Slurm scheduling system
- Tasks must be submitted in specific bash scripts and will run when they get priority
- You must specify how much CPUs, RAM and time you need.
- You can use 'salloc' to get an interactive job, which is like working on your server.