Best Practices in Scientific Computing

Sam Gilmour

January 27, 2022

A Nightmare

- John has worked tirelessly on his project and is ready to produce final results …
- … but a few days before the deadline, his computer dies 💔.
- MIT provides a loaner, so he installs the latest Julia, clones his GitHub repo into `~/mit-johndoe/project` and runs his main script.

A NIGHTMARE

- John has worked tirelessly on his project and is ready to produce final results …
- … but a few days before the deadline, his computer dies 💔.
- MIT provides a loaner, so he installs the latest Julia, clones his GitHub repo into `~/mit-johndoe/project` and runs his main script.
- Make sure you're sitting down…
- …and then open `bad/script.jl`
- **What's wrong with it?** So many things.

Topics and Objectives for Today

- The class is called “Best Practices in Scientific Computing”.
- We will focus on three ideas (each of them, of course, linked):

1. Maintainability (of your project)
2. Reproducibility (of your results)
3. Efficiency (of your computation)

TOPICS AND OBJECTIVES FOR TODAY

- The learning objectives are:

1. Maintainability (of your project)
    - Understand the importance of writing clearly documented and modular code.
    - Understand the reasons for using a sensible directory structure to organize a project.

Topics and Objectives for Today

- The learning objectives are:

2. Reproducibility (of your results)

    - Understand the features and benefits of package managers and virtual environments.
    - Practice creating virtual environments and using them to set up a project in multiple locations.
    - Understand the purpose and benefits of using environment variables.

Topics and Objectives for Today

- The learning objectives are:

3. Efficiency (of your computation)

    - Formulate the flow of information in a simple project as a directed, acyclic graph (DAG).
    - Practice using a pipeline management tool, Snakemake, to execute a project.

Some Comments Before we Get Started

- Windows users: consider Windows Subsystem for Linux (WSL)!
  - Setup instructions here.
  - Allows you to run a Linux system on your PC.
  - WSL 2 introduced support for GUI apps (I’m not sure how stable it is).
- There are no ‘best practices’. You’ll need to figure out what the correct tools for your project are.
- But you should always be mindful of the *maintainability*, *reproducibility* and *efficiency* of your project.

WHAT IS MAINTAINABILITY?

> *"The ability to easily modify and extend code without it breaking."*

- Why should you care about maintainability?

1. Saves time for you and others (in the long run).
2. Looks great on your GitHub for industry job applications.
3. Helps you to not hate writing code.

What is Maintainability?

> *"The ability to easily modify and extend code without it breaking."*

- Why should you care about maintainability?

1. Saves time for you and others (in the long run).
2. Looks great on your GitHub for industry job applications.
3. Helps you to not hate writing code.

- Maintainability follows from several practices:

1. Writing modular code.
2. Writing unit tests (in our line of work, where possible).
3. Documentation!
4. **Using a sensible project structure.**

PROJECT STRUCTURE

- Here is the structure of an example project:

```
8_best_practices
├── README.md
├── data
│   ├── models
│   ├── prepared
│   ├── processed
│   ├── raw
│   └── results
├── notebooks
├── plots
├── references
├── scripts
└── src
```

TASK

Look at the `README.md` files in each directory and think of your own examples in which each directory could be useful.

Storing Data

Question

In a collaborative project, should your data be stored on Git?

Storing Data

Question

In a collaborative project, should your data be stored on Git?

- When you have small data files, it’s OK to store them on Git.
- But data can be MASSIVE.
- Consider keeping your data in the cloud (e.g. with the Dropbox plan provided by MIT).
- Use environment variables (which we will cover) to point to your data.

Aside: Data Version Control

- There are many(!) tools out there for version controlling data.
- Think: “Git for data.”
- Can be a great idea to use one when working with data sources that constantly update or change.
- But, for us, this is not usually the case…
- Some examples:
  - DVC
  - Git Large File Storage (LFS)

MAINTAINABILITY: KEY POINTS

1. Document everything!
   - You might think something is obvious, but it almost never is.
2. Do your best to write modular code.
3. Use a standard directory structure for your projects (but modify it where needed).
   - Remember that raw data is immutable.
4. Store large data outside the Git repository.

Reproducibility

> *"The ability to obtain consistent results no matter the machine a project runs on."*

- Why should you care about reproducibility?

1. Makes collaboration possible.
2. Fundamental to being a good scientist!

REPRODUCIBILITY

> *"The ability to obtain consistent results no matter the machine a project runs on."*

- Why should you care about reproducibility?

  1. Makes collaboration possible.
  2. Fundamental to being a good scientist!

- Let's assume you run the same code (Git!) on the same raw data (cloud storage!).

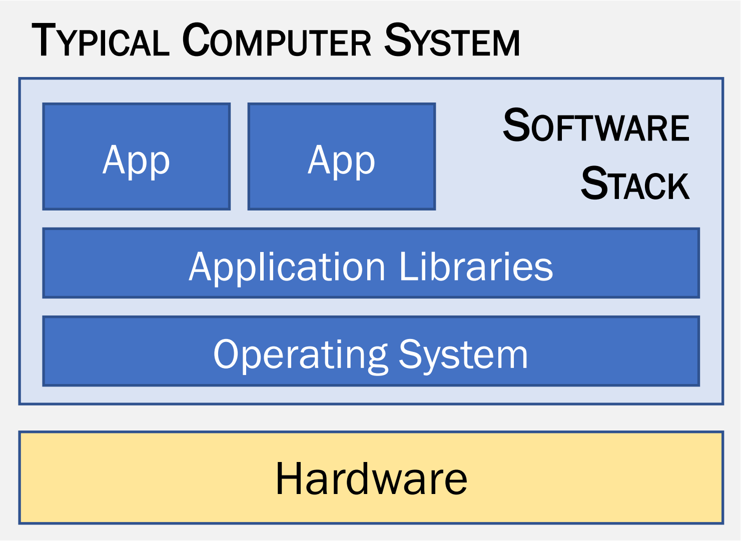
- Reproducibility boils down to you maintaining a consistent **computing environment.** In general, this can be very difficult.

What is a Computing Environment?

- There are two components in a computing environment:

  1. Hardware

  2. Software (operating system, application libraries)

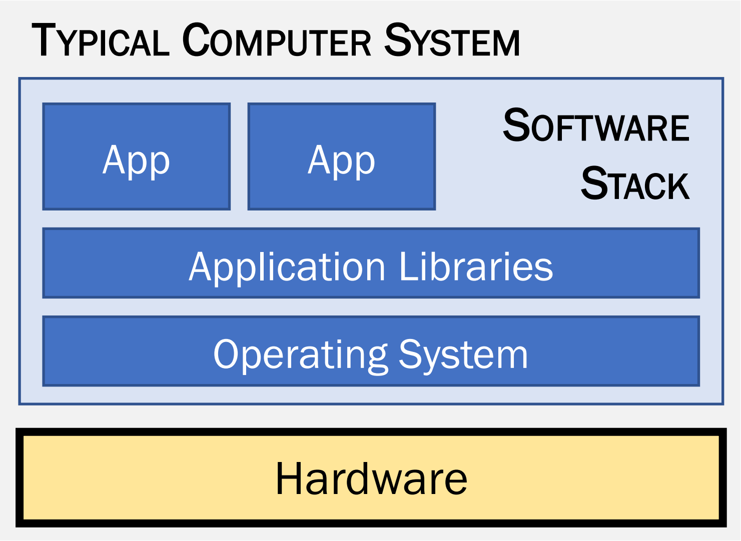


Note

Blocks above *request resources* from blocks below.

Computing Environment

- There are two components in a computing environment:
  1. Hardware
  2. Software (operating system, application libraries)

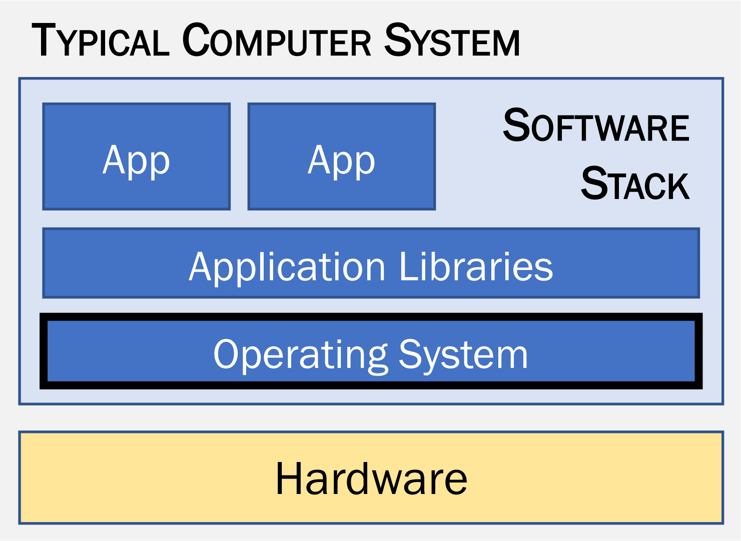


Hardware

Physical components of the machine: processor, RAM etc.

Computing Environment

- There are two components in a computing environment:
  1. Hardware
  2. Software (operating system, application libraries)



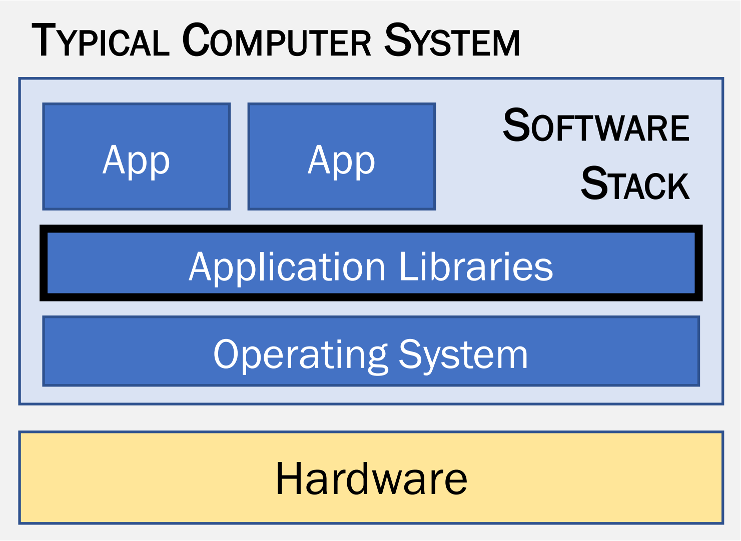
Operating System

Windows, Linux, macOS etc.

Computing Environment

- There are two components in a computing environment:

1. Hardware
2. Software (operating system, application libraries)

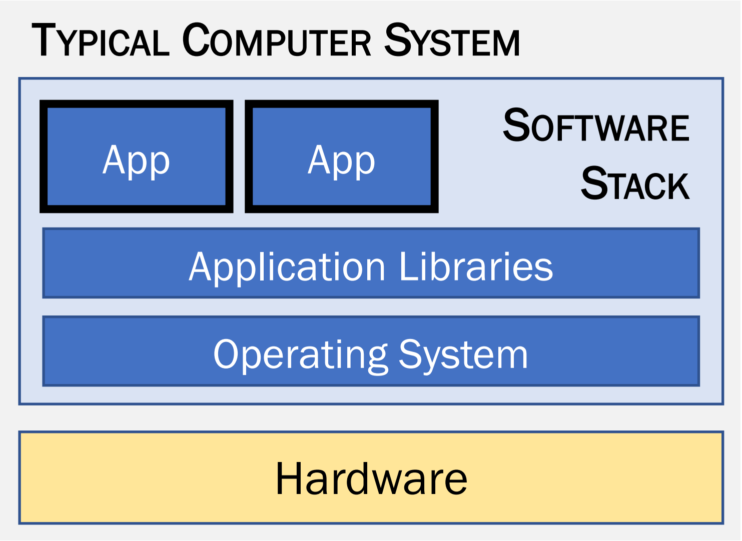


Application Libraries

For example: the Julia executable and packages being used to run a script.

Computing Environment

- There are two components in a computing environment:
  1. Hardware
  2. Software (operating system, application libraries)

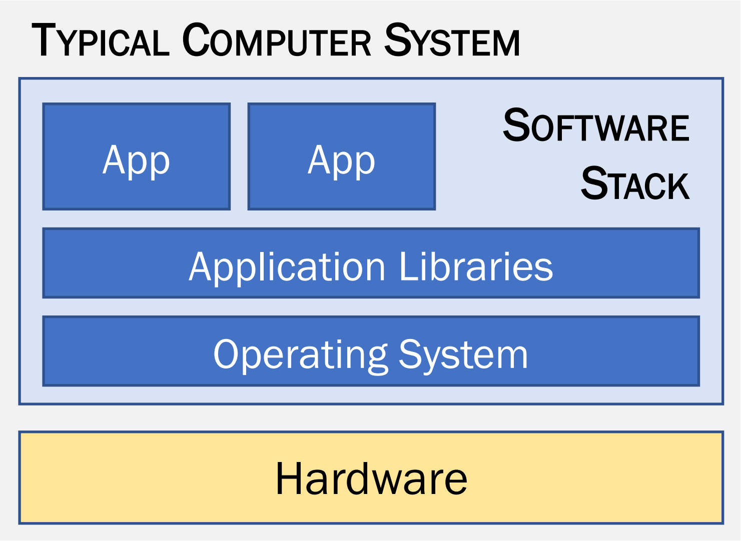


Apps

Collection of scripts and data that produce your results.

Computing Environment

- There are two components in a computing environment:
  1. Hardware
  2. Software (operating system, application libraries)

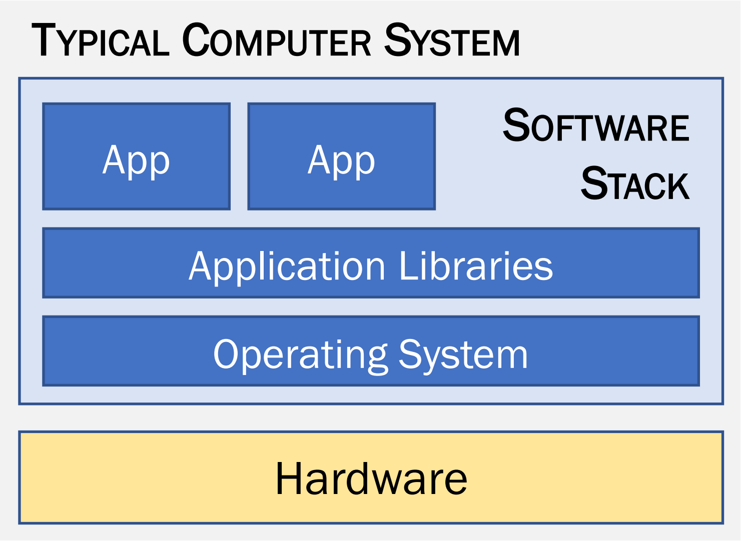


Question

Which of these components can we control?

Virtual Environments

- A virtual environment is an isolated set of application software living on a machine.



Typical Computer System

App

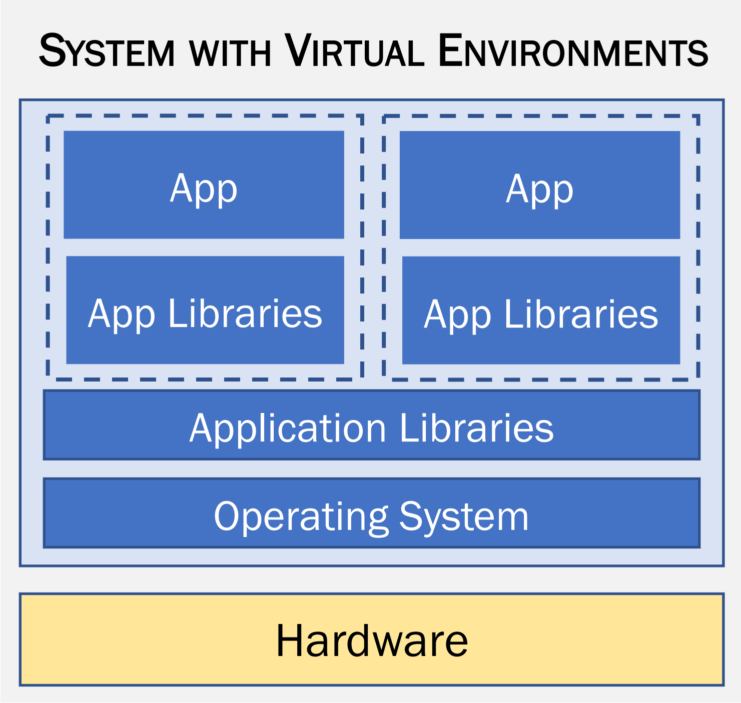
App

Software Stack

Application Libraries

Operating System

Hardware



System with Virtual Environments

App

App Libraries

App

App Libraries

Application Libraries

Operating System

Hardware

Virtual Environments

Question

Some of you probably have multiple versions of Python/Julia installed on your machine. What do you do when you want to run code in a particular version?

VIRTUAL ENVIRONMENTS

QUESTION

Some of you probably have multiple versions of Python/Julia installed on your machine. What do you do when you want to run code in a particular version?

- Why are virtual environments useful?
  - They provide a **clean** and **uncomplicated** way to install multiple versions of software on the same system.
  - Sure, you could install 10 python versions and run commands like `python3.6 script_1.py` and `python3.8.1 script_2.py`…
  - But it's not ideal!

PACKAGE MANAGERS

- Package managers provide three major benefits:
  1. Download, install and update the application libraries you tell it to.
  2. Record a snapshot of the precise software versions on your system.
  3. Quickly instantiate packages on a new system from a snapshot.
- You're probably already familiar with many package managers.
  - **Python:** pip
  - **Linux:** APT, Pacman, …
  - **macOS:** Homebrew
  - **Julia:** Pkg
  - **R:** (no name)
  - **iOS:** App Store

Package Managers

- Package managers provide three major benefits:
  1. Download, install and update the application libraries you tell it to.
  2. Record a snapshot of the precise software versions on your system.
  3. Quickly instantiate packages on a new system from a snapshot.
- You’re probably already familiar with many package managers.
  - **Python:** pip
  - **Linux:** APT, Pacman, …
  - **macOS:** Homebrew
  - **Julia:** Pkg
  - **R:** (no name)
  - **iOS:** App Store
- When using a package manager inside a virtual environment, we can record a snapshot of the virtual environment itself.

WHAT IS CONDA?

- Conda is a command line tool for managing virtual environments and packages.
- Anaconda is a collection of 160+ Python packages for data science.
- When you install Anaconda, you install Conda + these packages.
- Conda allows you to:
  - Have a single tool for package management and virtual environments.
  - Easily install software on systems where you don’t have root access.
  - Install software, not just packages for a particular language.

When to use Conda?

- Conda may not be the best tool for your particular project.
  - It *excels* at managing Python versions and packages.
  - Pretty good at managing R versions and packages.
  - Usable for Julia versions (provided you are on Linux/macOS) but not packages.
- Nonetheless, learning about Conda provides exposure to concepts that transfer over to many other tools.

Conda Channels

- One last thing before we get into some commands…
- A Conda **channel** is a repository of software (like a TV channel).
  - Gurobi is the channel maintained by Gurobi for their software.
  - conda-forge is a community-led collection of packages (where Julia can be found).
- When checking to see if software is provided by Conda, search through all channels on the Anaconda website.

Creating an Environment and Installing Packages

Terminal

```
$ conda create -n iap_env
$ conda activate iap_env
(iap_env) $ conda config --env --add channels bioconda
(iap_env) $ conda config --env --add channels conda-forge
(iap_env) $ conda info
(iap_env) $ conda install python=3.8
(iap_env) $ python --version
(iap_env) $ which python
(iap_env) $ julia --version
(iap_env) $ which julia
(iap_env) $ conda deactivate
$ python --version
$ which python
```

ENVIRONMENT VARIABLES

- You're probably familiar with the age-old StackOverflow exchange:

Q: *"Why doesn't my software ___ work?"*

A: *"Have you tried adding it to your* `PATH`*?"*

Environment Variables

- You’re probably familiar with the age-old StackOverflow exchange:

Q: *“Why doesn’t my software ___ work?”*

A: *“Have you tried adding it to your* `PATH`*?”*

- `PATH` is the environment variable your shell uses to search for executables.
- It likely includes some directories that are unique to your system: for example, subdirectories of your home directory.
- Almost all software uses environment variables to tailor itself to your system.

CONDA ENVIRONMENTS ARE…

- …made up of two simple components:
  1. A directory of application libraries
  2. Scripts to set up environment variables that point to this directory (when you run `$ conda activate iap_env`)

TERMINAL

```
$ which python
$ echo $PATH
$ conda activate iap_env
(iap_env) $ which python
(iap_env) $ echo $PATH
```

Aside: Using Environment Variables in Code

- As we have discussed: software uses environment variables to tailor itself to your system.
- We can do the same!

Aside: Using Environment Variables in Code

- As we have discussed: software uses environment variables to tailor itself to your system.
- We can do the same!
- It is good practice to create environment variables for such tasks as **setting paths to data directories.** Then, to access in code:
  - **Python:** `import os; os.environ[“MY_ENV_VAR”]`
  - **Julia:** `ENV[“MY_ENV_VAR”]`
  - **R:** `Sys.getenv(“MY_ENV_VAR”)`
- It’s also possible to specify environment variables that are created upon activating a Conda environment.

Recording and Instantiating Environments

Terminal

```
(iap_env) $ conda env export > environment.yml
(iap_env) $ cat environment.yml
(iap_env) $ conda env export --from-history
(iap_env) $ conda deactivate
$ conda env create -n new_env --file environment.yml
$ conda env list
$ conda activate new_env
(new_env) $ which python
(new_env) $ conda deactivate
```

Recording and Instantiating Environments

Terminal

```
(iap_env) $ conda env export > environment.yml
(iap_env) $ cat environment.yml
(iap_env) $ conda env export --from-history
(iap_env) $ conda deactivate
$ conda env create -n new_env --file environment.yml
$ conda env list
$ conda activate new_env
(new_env) $ which python
(new_env) $ conda deactivate
```

Question

Which different use cases do the first and third commands serve?

Other Conda Commands

Terminal

```
$ conda remove -n new_env --all
$ conda env list
$ conda activate iap_env
(iap_env) $ conda list
(iap_env) $ conda remove python --dry-run
```

ENVIRONMENTS AND PACKAGES IN JULIA

- When Conda doesn’t provide a package or software you need for your project, you’ll have to mix tools.
  - For example: it doesn’t provide a Windows build of Julia.
- Fortunately, Julia has its own tool: `Pkg`.

TERMINAL

```
$ julia
julia> ]
pkg> activate .
pkg> add DataFrames
```

TASK

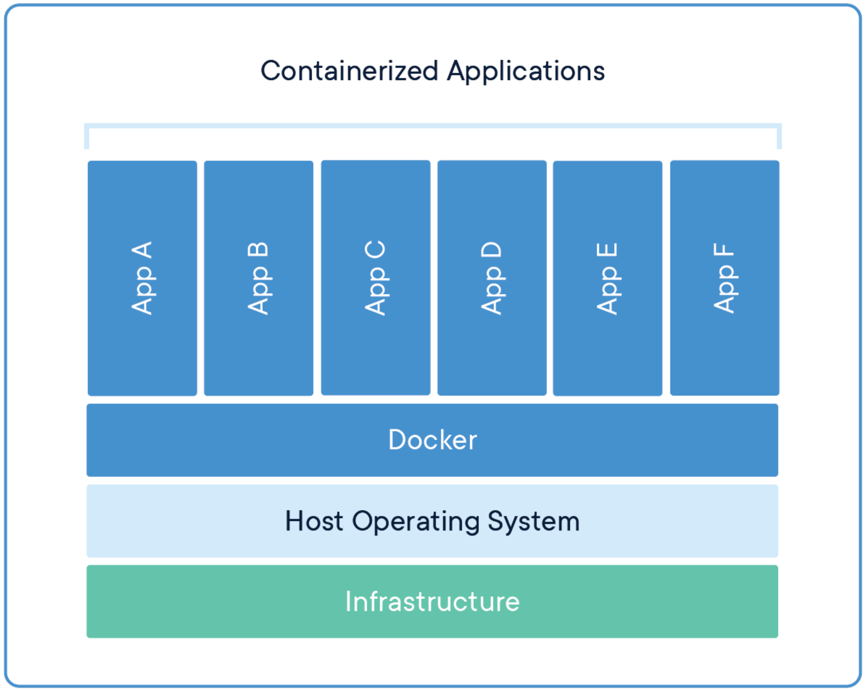
Inspect the `Project.toml` and `Manifest.toml` files that were created by these commands.

Dependency and Lock Files

- `Project.toml` keeps track of exactly what you specified.
  - This is for **flexibility**, called a ‘project’ file.
- `Manifest.toml` records exact versions that are installed.
  - This is for **precision**, called a ‘lock’ file.
- Most modern package managers use this two-file method.
- When you know the *exact* machine you’ll be instantiating an environment on, you have the luxury of using the lock file.
- Otherwise, use the dependency file: it is more forgiving.
  - More in the Pkg documentation.

ASIDE: DOCKER

- You’ve probably heard of Docker: the “industry standard for containerization”.
- It’s a virtual environment on steroids, which also separates your application from the operating system and hardware.



REPRODUCIBILITY: KEY POINTS

1. ALWAYS try to set up a virtual environment for each project.
   - This may require you to use and mix several tools.
2. **Document** your installation procedure.

Exercise: Installation

Task

Make sure your `iap_env` produces a similar export to mine:

```
channels:
  - conda-forge
  - bioconda
dependencies:
  - python=3.8
  - r=4.0
  - snakemake
  - r-purrr
  - r-dplyr
  - r-readr
  - r-ggplot2
  - r-forcats
```

EFFICIENCY

> *“Obtaining results quickly and without unnecessary computation.”*

- A useful mental model for scientific computation is a **directed acyclic graph** (DAG) (referred to as a **pipeline** or **workflow).**
- Nodes are *jobs* with inputs + outputs that generate results (and usually record these results).

EFFICIENCY

"Obtaining results quickly and without unnecessary computation."

- A useful mental model for scientific computation is a **directed acyclic graph** (DAG) (referred to as a **pipeline** or **workflow).**
- Nodes are *jobs* with inputs + outputs that generate results (and usually record these results).
- Two fundamental ideas to keep in mind:
  - **Splitting** the steps in a workflow into multiple jobs allows computation to be re-started from 'checkpoints'.
  - Placing jobs in **parallel** allows results that don't depend on each other to be computed concurrently.

Exercise: Design a DAG

Task

Here’s a small project, given the 3 ‘datasets’ in `data/raw`:

1. Process each input file to replace all instances of the string “###” with a decimal point and save the new datasets.
2. For each dataset, indexed by `i`, add `i` to each number and square the result. Find the sum of the resulting numbers and save it.
3. Create a column chart that visualizes the three results.

Design a DAG for this project.

Pipeline Tools

- These are tools that ‘simplify’ running the steps of your analysis.
- There are MANY (and most require the DAG to be specified).

Luigi

Spotify

Apache Airflow

Airbnb

METAFLOW

Netflix

Pipeline Tools

- These are tools that ‘simplify’ running the steps of your analysis.
- There are MANY (and most require the DAG to be specified).

luigi

Spotify

Apache Airflow

Airbnb

METAFLOW

Netflix

- What can a pipeline tool help with?
  - Re-running only steps that failed.
  - Easy parallelization on your own machine!
  - Forcing you to think about your DAG.

SNAKEMAKE

- Snakemake is a pipeline tool (that is by no means the best).
- Developed (and used heavily) by the bioinformatics community.
- Why are we learning to use Snakemake, in particular?
  - Easy to install (hopefully).
  - Simple, built-in visualization of DAGs.
  - Python scripting in the pipeline file.
  - Easy-to-learn syntax whose core ideas translate to other tools.
  - Integration with clusters running Slurm.

THE SNAKEFILE

- The Snakefile defines your pipeline.
  - Worth remembering: a `Snakefile` is essentially a Python script.

TERMINAL

```
(iap_env) $ touch Snakefile
```

RULES

- A Snakefile contains **rules**, which are *nearly* equivalent to jobs.
- Each rule applies a **process** to turn a set of **inputs** into **outputs.**
- We will first look at the **shell** block as an example of a process.
- Here's an example of the syntax:

```
rule name_of_rule:
    input:
        “in_A.txt”,
        “in_B.txt”
    output:
        “out.txt”
    shell:
        “$(cat {input}) > {output}”
```

Exercise: Writing a Rule

Task

Write a rule in your `Snakefile` called `process_first` that completes Step 1 of the example project on the first dataset (`data/raw/data_dump_1.txt`).

(You might need to Google for some Bash shell commands.)

Executing a Pipeline

- In order to execute, Snakemake needs a **target.**
- A target is what we hope to produce as the final result of computation, and it can be a **file** or a **rule.**
- In a Snakefile with a complicated DAG and many rules, specifying a target tells Snakemake to only execute up to a certain node.

Terminal

```
(iap_env) $ snakemake --np data/processed/1.txt
(iap_env) $ snakemake --np process_first
(iap_env) $ snakemake process_first --cores 1
```

WILDCARDS

- Snakemake provides some nice syntax for pattern matching, which is useful for specifying parallel jobs.
- Matched patterns are called **wildcards.**
- Here’s an example of the syntax:

```
rule name_of_rule:
    input:
        “in_{id}.txt”
    output:
        “out_{id}.txt”
    shell:
        “$(cat {input}) > {output}”
```

Exercise: Using Wildcards

Task

Rename your rule from the previous exercise to `process_all` and modify it to complete Step 1 of the example project on all 3 datasets.

Exercise: Using Wildcards

Terminal

```
(iap_env) $ snakemake process_all --cores 1
```

Question

Why doesn’t this command run properly?

EXPANSION VS. WILDCARDS

- Snakemake needs to have enough information **in the `Snakefile`** to construct the full DAG.
- It is useful to think of the process as ‘starting at the end and propagating backwards’.
- The `expand` function can be useful. Here’s some syntax:

WITHOUT EXPAND

```
rule name_of_rule:
    input:
        “in_A.txt”,
        “in_B.txt”
    output:
        “out.txt”
    shell:
        “$(cat {input}) > {output}”
```

WITH EXPAND

```
rule name_of_rule:
    input:
        expand(“in_{let}.txt”, let=[”A”, “B”])
    output:
        “out.txt”
    shell:
        “$(cat {input}) > {output}”
```

Exercise: Expansion

Task

Write a new rule called `all` that allows you to complete Step 1 of the example project on all 3 datasets.

VISUALIZING THE DAG

- It can be very useful in large experiments to visualize the DAG.
  - For debugging your Snakefile
  - To see which jobs already have results
- Snakemake makes this easy:

TERMINAL

```
(iap_env) $ snakemake --dag all | dot -Tsvg > plots/first_dag.svg
(iap_env) $ snakemake all --cores 1
(iap_env) $ snakemake --dag all | dot -Tsvg > plots/second_dag.svg
```

Using Scripts

- So far, we have used the shell block to turn inputs into outputs.
- You can run your scripts from the shell block, passing command line arguments for paths to input and output files.
- Snakemake also provides a convenience block called **script**.
- Strings for the input, output and additional parameters are supplied to the script itself in the form of a snakemake object.

```
rule name_of_rule:
    input:
        “in_{id}.txt”
    output:
        “out_{id}.txt”
    params:
        id=“{id}”
    script:
        “my_script.py”
```

Referencing in a Script

- In Python:

```
snakemake.input   # List of input files
snakemake.output  # List of output files
snakemake.params  # Dict of parameters
```

- In Julia:

```
snakemake.input   # List of input files
snakemake.output  # List of output files
snakemake.params  # Dict of parameters
```

- In R:

```
snakemake@input   # List of input files
snakemake@output  # List of output files
snakemake@params  # List of parameters
```

Exercise: Scripts

Task

Write a Python script called `sum_data.py` and a corresponding rule called `compute` to complete Step 2 of the example project on all datasets.

Then, write an R script called `plot.R` and a rule called plot to complete Step 3. Execute the pipeline.

Exercise: The Power of Snakemake!

Task

Duplicate `data/raw/data_dump_1.txt` as `/data/raw/data_dump_4.txt`.

Experiment with the Snakemake tools: plot your DAG to see what must be updated, try the `--summary` flag to see what it outputs.

Finally, execute the pipeline and note what happens.

Exercise: The Power of Snakemake!

Task

Duplicate `data/raw/data_dump_1.txt` as `/data/raw/data_dump_4.txt`.

Experiment with the Snakemake tools: plot your DAG to see what must be updated, try the `--summary` flag to see what it outputs.

Finally, execute the pipeline and report what happens.

Question

Why do we observe this behavior, and how can we ‘fix’ it?

ASIDE: EXECUTING SNAKEMAKE ON THE CLUSTER

- Snakemake functions by wrapping the shell and script blocks in your rules with some additional code before submitting.
- To be used on a Slurm cluster, all it has to do is wrap with the familiar srun command.
- Here is a tutorial.

EFFICIENCY: KEY POINTS

1. Think carefully about your DAG – it may make your life much easier!
   - This means both **checkpoints** and **parallelism.**
2. Pipeline tools can greatly assist with managing your DAG.
   - But there is no need to use them for simple projects.

FINAL THOUGHTS

1. ALWAYS prioritize maintainability, reproducibility and efficiency.
   - If you can’t prioritize due to a deadline, make sure to review when you have the chance (before your project gets out of hand…).
2. Stay curious – there are a ton of tools out there that aim to make your life easier. Experiment with them!
3. Pick the best tools for your project, and don’t be afraid to tailor everything we have covered today.

Thanks for Listening!