Computational Methods in Particle Physics

Part 1: Code and stuff

Drs Ben Hodkinson & Holly Pacey
Oxford HEP DPhil Programme
Hilary Term 2025

Course Overview

1. Concepts in Programming, Software design, Common tools and tips.
2. Data & Statistical analysis
3. Machine Learning

Each session: Lecture (ask questions!) + Practical exercises.

→ Aims: Introduce a broad range of concepts and skills that you are likely to use. Encourage good practice

Sorry we can't cover everything!
But we can help you find more resources for anything not covered.

Why is this important?

Software & Computing is a crucial part of performing research that is **reproducible** and **well-preserved**.

The better you can exploit S&C in your work and build your skills….
the higher your research **quality** and **quantity** will be, the more **robust** your results will be, and the more **employable** you will be after your PhD.

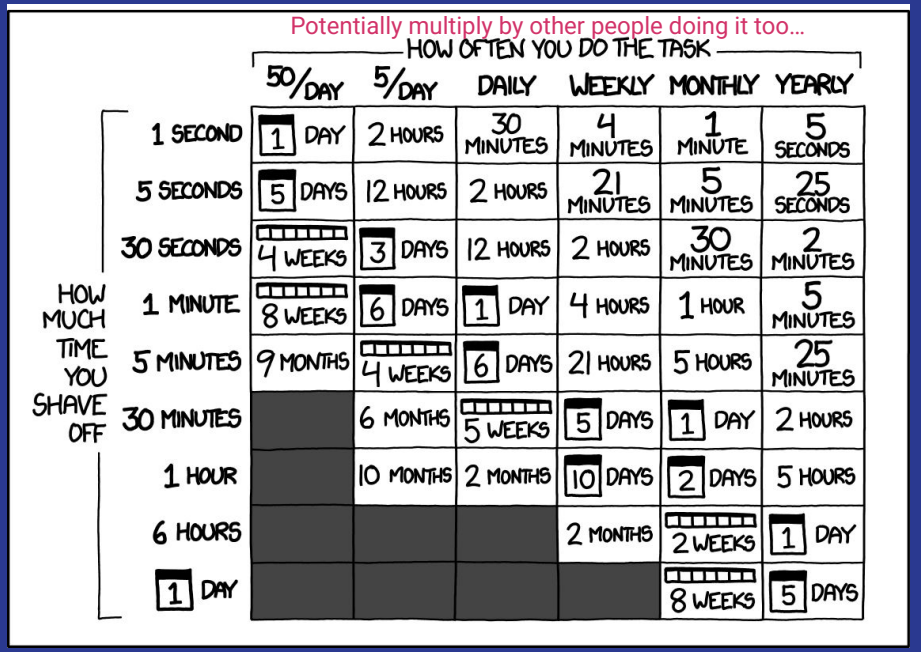
the more likely other people can build on your research and tools to increase their impact, learn faster themselves, and make HEP better for everyone.

Software can be **publishable in its own right** even beyond that used for a paper, funding bodies caring more about this idea than in the past.

&

Working more efficiently gives you more time to spend on other things

Potentially multiply by other people doing it too...



Admin

Any questions feel free to find us in DWB or email us:
holly.pacey@physics.ox.ac.uk, benjamin.hodkinson@physics.ox.ac.uk

All Tutorial materials here:....

Links to additional resources throughout.

Recommend Oxford DTC RSE Course, v. broad intro to many things. Can find all materials here: https://gutenberg.fly.dev/ Slides (and some is recycled here!)

This course uses python and C++, as the most common languages used in HEP.

Intro/Setup

Programming in HEP

- You are likely to be working collaboratively to process/analyse some kind of data in most areas of HEP.
- You will likely encounter:
  - Programming languages
    (mostly C++ and python, or Java/SQL in technical settings)
  - Shell environments + scripts
  - Virtual environments
  - Plotting and Statistical tools
  - Data management + Versioning tools
  - Python libraries for Machine Learning etc.
  - Build systems (cmake)
  - Batch/distributed computing systems (Condor, The Grid)
  - Different formats: markdown, yaml, json, …
  - Other people who could benefit from using your code (and visa versa)
  - Needing to rerun code you last used years ago (e.g. for your thesis)
  - Bugs, messy code, lack of documentation,
    seg faults, useless error messages.

Oxford PP Specifics

Connecting to PPLXINT
PPUNIX overall
PPUNIX background info
PPUNIX FAQ
PPUNIX Best Practice
Data management

- Structure:
  - data/ area to keep: data input/output; installations (miniconda, madgraph, …). Much more space but not backed up.
  - user/ area to keep: code, latex stuff, small things like plots. Limited space, but it is backed up.
- pplxint: our place to work, access e.g. ATLAS stuff, link to Condor batch system.
- You need to SSH into a ‘node’ to work on… several options to try.
  - ssh from a terminal: MAC terminal or use Git Bash For Windows
  - Basic terminal + open windows (slowly): Putty+xming
  - MobaXTerm: fancier terminal, has penguins, v. easy to transfer files between pplxint & laptop.
  - Full remote desktop (nice as doesn’t end if you lose wifi or close it, full desktop experience for opening other programmes), but needs good bandwidth: RDC for windows / for MAC
  - VSCode: ssh to e.g. a folder in pplxint, full IDE, not just a terminal.

Editors

Two main kinds to work in - suggest trying both, they have different benefits.

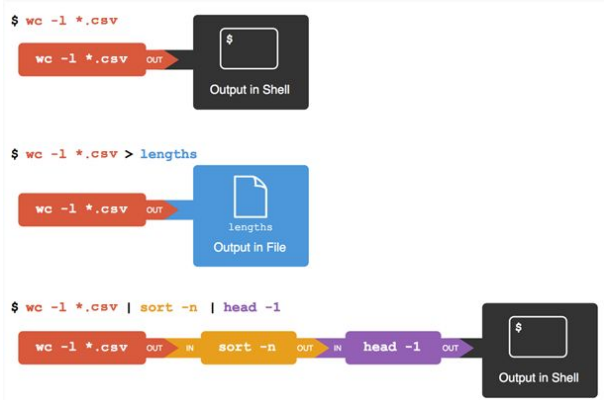
Terminal Based	Interactive Development Editor (IDE)
vim [cheatsheet, tutorial], nano, emacs[emacs can be GUI or terminal based]	VSCode
• More of a basic text editor appearance, specifically for writing and editing code. • Can define your own macros/shortcuts. • Use REGEX for complex find/replace/edit commands. • Keyboard-only • Probably better in bad-wifi situations, only need a light ssh terminal connection? • Good if you just want to quickly tweak something or write a bash script.	• More holistic workflow, with a particular dir/project/terminal/code open in one window. Use to write->debug->run code. • Get nice pointers on style/syntax, auto-indenting, spellcheckers, auto-fill, etc… • Can ssh directly. • Lots of extensions available (pdf view, tbrowser etc.). • Direct git interface • Debuggers, linters available.

Environments & management.

Bash Shell

The Bash shell is a command-line interface typically used in Unix-based operating systems such as Linux & macOS, but is also available for Windows. There are many useful short commands…

- create/move/copy/list/rename/… files
- Search for/in files using
- Find+replace/sort/retrieve-lines in files
- Open programmes and files
- setup/compile/run code
- Assess running processes, disk space, memory usage, …
- Print things into the terminal or into files



Which can be piped together to do more useful things!
`<command 1> | <command 2> | <command 3>`
will run these 3 commands sequentially, using the previous command's output as the input to the next command

Note you can tab complete file paths etc., Up and down arrows scroll through command history.

Useful Bash Commands:

- `ls <a>`: Lists the files and directories in dir <a> (-lrt will show details and order by time, -S orders by size, -a includes hidden files)
- `cd`: Changes the current directory to a specified directory. (.. = 1 dir up, . = current dir, ~ = home dir)
- `mkdir`: Creates a new directory.
- `rm`: Deletes a file or directory (-r recursive). BEWARE you cannot recover from this unless you are using a backed up disk
- `echo`: Prints a message to the terminal.
- `pwd`: prints path of current directory.
- `cp <a> <b>` : copies <a> to <b>, add -r if a is a folder
- `mv <a> <b>`: moves <a> to <b>, if <b> is a folder that exists <a> will be moved inside it.
- `ln -s <a> <b>`: creates a soft-link to pretend that <a> is in folder <b> / a file called <b>
- `du -sh <a>`: show how much disk space is taken up by file/folder <a>
- `cat <a>`: shows <a> file contents in terminal
- `* > moo.log`: write * command terminal output into moo.log (won't show it in the terminal). Can also use >> instead of >: If mo.log doesn't exist already, both > and >> will create one. If it does, then > will *overwrite* the contents of the file, whilst >> will *append* to the existing contents.
- `* |& tee moo.log`: write * command terminal output (out and error stream) into moo.log (will show it in the terminal)
- `sed "s/oink/moo/g" <a>:` print out <a> and replace all oinks with moos. (-i will instead do the replacement in <a>) This command uses regex so you can do complex things!
- `grep "moo" <a>:` prints out all occurrences of moo in file <a> (-i makes it case sensitive, -r for recursive search through folder <a>)
- `find [where to start searching from] [expression determines what to find] [-options] [what to find]` (e.g `find ./ -name moo.py` will look for files called moo.py from your current dir.)
- `wc <a>:` count words in <a> (-l count lines), can use wildcard in <a>.
- `sort <a>:` print <a> file contents with sorted lines (-n numerical sort, -u keep unique lines only)
- `head -<x> <a>:` print first <x> lines of <a>
- `chmod <a>:` sets read/write permissions for files cheatsheet

Bash Scripts

```
for filename in *.dat
do
    echo "$filename original-$filename"
    mv $filename original-$filename
done
```

```
#!/bin/bash

ID=0

# make your life easier by not typing out long strings
if [ $ID = 0 ]
then
  IDSTRING="AnalysisZero"
elif [ $ID = 1 ]
then
  IDSTRING="AnalysisOne"
fi

# do something with files in specific cases
if [ $ID == 0 ]
then
  rm ${IDSTRING}_*categoryXY*.txt
fi

# do something with files
for file in ${IDSTRING}*txt
do
  cat $file >> allData.txt
done
```

Bash Scripts are *.sh files that can be used to run a series of commands, they are great time savers, memory aids, and critical for batch working.

- Run with “source moo.sh”

When to use them:

- Submitting batch jobs
- Running a complex chain of commands
  - Automate a processing pipeline e.g. where the input depends on previous output.
  - To setup and compile a project.
- Simple commands that you’ll run often
  - Making use of regex pattern matching when e.g. moving/renaming/copying files
  - Run a command several times with variations: e.g. making use of conditionals/loops and regex to quickly vary commands.
  - To setup an environment.
  - A standard command you’d run repeatedly while testing code.
- As a part of documentation/collaboration

Bashrc & Aliases

These are good time-savers!

Aliases: shortcuts for bash commands that you use a lot

Don't make an alias that overrides an existing command!
`alias mv = 'mv -r'` BAD

Env. variables: a path label,
called via `${my_env_var}` or `$my_env_var`.

Define these in in ~/.bashrc and a new ~/.bash_alias file.

It is possible to also set up things you want to start automatically when you open a terminal, but be sure that these are things you will ALWAYS want, as they might interfere with your env.

In .bash_alias:

```
alias tb='tbrowser'
alias lt='ls -ltrh'
alias setupATLAS='source
${ATLAS_LOCAL_ROOT_BASE}
/user/atlasLocalSetup.sh
'
```

Then, in terminal I can run e.g.:

```
tb file.root
```

instead of

```
root -l file.root
New TBrowser()
```

In .bashrc:

```
# User specific aliases and functions
tbrowser () {
  # Check a file has been specified
  if (( $# == 0 )); then
    echo "No file(s) specified."
  else
    # For each file, check it exists
    for i; do
      if [ ! -f $i ]; then
        echo "Could not find file $i"
        return 1;
      fi
    done
    root -l $*
$HOME/.macros/newBrowser.C
  fi
}

export
ATLAS_LOCAL_ROOT_BASE=/cvmfs/atlas.cern.
ch/repo/ATLASLocalRootBase
source ~/.bash_alias
```

Regular Expressions

Regular expressions (regex or regexp) are extremely useful in extracting information from any text by searching for one or more matches of a specific search pattern (i.e. a specific sequence of ASCII or unicode characters).

Can define very complex and specific patterns for whatever you need!

Found in e.g.

- Find+replace functions in vim/VScode/…
- bash commands like sed
- Python with ‘re’ library

```
[m-o]+.o?
```
a letter between m and o
At least one of them
Any single character
o 0 or 1 times

regex cheatsheet, another regex cheatsheet,

Practice your understanding: https://regexcrossword.com/

Python Virtual Envs.

Often we want to make use of *external* libraries/packages that don't come with python as default. E.g. anytime you do you need matplotlib and numpy:

```
from matplotlib import pyplot as plt
import numpy as np
```

These are your *dependencies*.

Depending on your project, you will want a specific set of *dependencies*, possibly even specific versions of libraries too. Virtual Environments set up a particular python installation + the dependencies we want, to work on a given project.

Several tools exist, standard is 'venv', comes with python3.3+.

To install/update external packages, use python package manager 'pip'.
-> any package on here can be installed: https://pypi.org/
-> Guide
-> to better handle mixing different versions of things use e.g. pip-tools.

`python3 -m venv /path/to/new/virtual/environment`
Or just `python3 -m venv venv` is standard

Which creates a dir containing…

- `pyvenv.cfg` configuration file with a home key pointing to the Python installation from which the command was run,
- `bin` subdirectory (called `Scripts` on Windows) containing a symlink of the Python interpreter binary used to create the environment and the standard Python library,
- `lib/pythonX.Y/site-packages` subdirectory (called `Lib\site-packages` on Windows) to contain its own independent set of installed Python packages isolated from other projects,
- various other configuration and supporting files and subdirectories.

Activate our virtual environment via: `source venv/bin/activate` And exit via `deactivate`

Install the packages via
`pip install numpy`
`pip install matplotlib`
Or `pip install numpy matplotlib` Or `pip install -r requirements.txt`
Where requirements.txt has the list of packages you need (1/line)
Or to pick a particular version `pip install numpy==1.12.1 matplotlib>=1.20`

requirements.txt

```
pip==22.3.1
wheel==0.38.4
Pillow==9.5.0
setuptools==65.5.1
packaging==23.1
numpy==1.26.0
```

To check info about what version is installed: `Pip show numpy`
(Note if you later install something that requires a newer package version it will update it.)

Once you have fully set up what you need you can also generated the associated requirements.txt via `Pip freeze > requirements.txt`

You can put the requirements file on git, and write in the readme how to setup the venv with it :) Then others can recreate the same environment.

Conda

Anaconda is an open source Python distribution commonly used for scientific programming - it conveniently installs Python, package and environment management `conda`, and a number of commonly used scientific computing packages so you do not have to obtain them separately.

`conda` is an independent command line tool (available separately from the Anaconda distribution too) with dual functionality:

1. it is a package manager that helps you find Python packages from remote package repositories and install them on your system, and
2. it is also a virtual environment manager.

So, you can use `conda` for both tasks instead of using `venv` and `pip`.

Why use Conda instead of venv+pip?

- If you need packages other than python ones (e.g. ROOT)
- If you have a complex set up of specific versions needed

If you are using conda but need something only in pip, you can combine pip and conda but beware of version-compatibility mess, and we find it's better to do all the conda installation before all the pip installation.

cheatsheet

Co

Anaco
enviro

conda i

1.
2.

So, yo

Why u

-
-

If you
find it'

cheats

Install miniconda in your data/
(find here: https://docs.anaconda.com/miniconda/)

Recommend not to let anaconda go into your bashrc so it automatically turns on all the time (it will interfere with other software/envs).
Keep instead a separate bash script with :

```
eval "$(/data/atlas/users/you/miniconda/bin/conda shell.bash hook)"
```

To turn it on when you want. This also helps you to run conda in a condor job etc.

Create a virtual env:

```
conda create --name myenv
```

Activate it:

```
conda activate myenv
```

(use `conda deactivate` to switch off)

```
conda update -n base -c defaults conda
```

Example installations, from different sources:

```
conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch -c nvidia
conda install conda-forge::pytorch_geometric
conda install conda-forge::root
conda install -c conda-forge shap
conda install anaconda::pandas
```

You can also install things whilst defining the env:

```
conda create -c conda-forge --name my_env_name root pandas uproot python=3.8
```

What about sharing/preserving the set up? Instead of the requirements file you can just make a bash script with the instructions

```
conda create… conda activate… conda install bla…
```

Distributed Computing

Condor

Our PPLXINT system uses HTCondor as a batch system, allowing you to run multiple larger jobs on dedicated ‘node’ CPUs in a queue system. Generally, every system has a slightly different setup, or might use another system entirely but the principle is the same:

To run a condor ‘job' you need:

1. Your actual code.
2. A bash script that (1) goes to the right dir, (2) sets up your env., (3) runs your code, (4) deals with the output.
3. A list of settings for the condor batch system to organise and execute your code and output the log/err/job info.

Oxford instructions: batch farm, best practice (remember the chmod bit)
Lxplus HTCondor Docs: https://batchdocs.web.cern.ch/index.html

Always worth running a small test locally before submitting jobs!

Other systems

Won't go into here (though feel free to ask about later),
but there is a lot of documentation….

THE GRID

- Can use for large jobs, or e.g. if you need to access central ATLAS/LHCb data/MC samples you don't have stored locally.
- Requires a grid certificate to use - you should probably all get one via CERN/Oxford-UKRI.
- Then you join a VO (virtual Organisation) e.g. ATLAS-VO, to get authority to submit jobs and use particular resources.
- Ox intro, ox grid certificate/use intro, CERN grid certificate setup

Oxford ARC

- High Performance Cluster (HPC) of mainly GPUs.
- All Oxford people can get 10K hours for free (but in a big queue of jobs)
- If you need to do big ML stuff, is an option.
- getting-started, arc-user-guide, arc-software-guide.
- CERN is starting to get GPUs on lxplus-batch & the grid too.

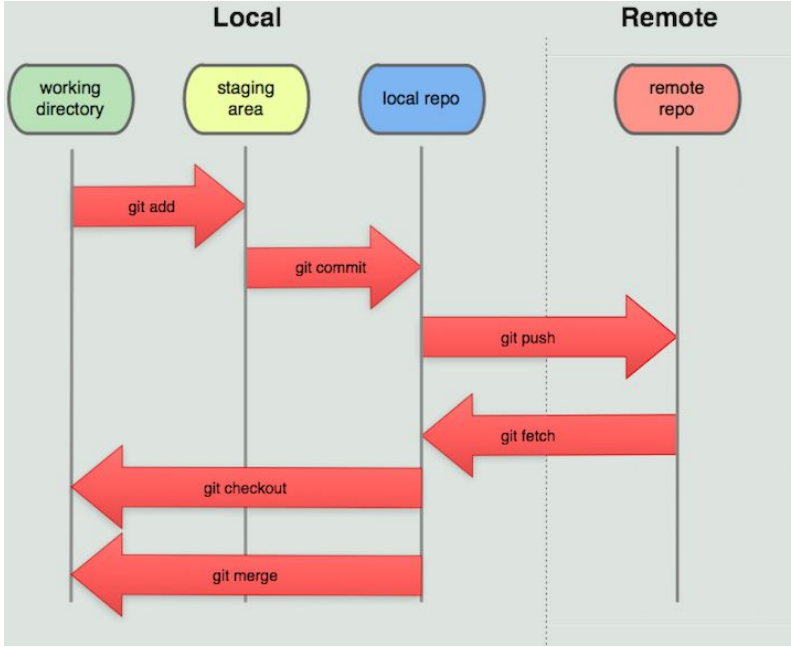
Collaboration & Version Control

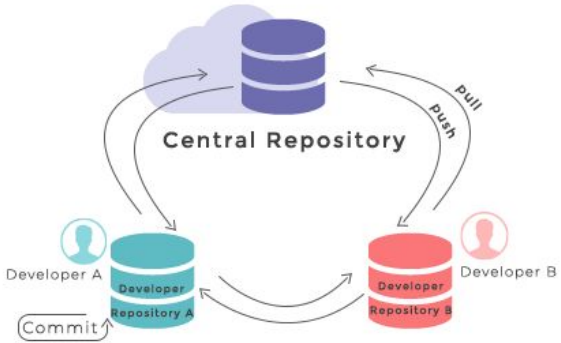
Git Intro

- Version control system to track/access/revert changes
- Records every change in the project history
  - You can always find the exact code used to make a particular version of your outputs.
- Coordinates work among multiple people
  - Supports non-linear + distributed development
- Still valuable for solo work
  - Won't accidentally delete/lose anything
  - Easy to show code to people helping you debug
  - Can move code easily between e.g. ppunix + lxplus
- gitlab for CERN people, github for anyone
- Tutorials: git-scm atlassian w3schools [ATLAS specific] HEP Software Foundation

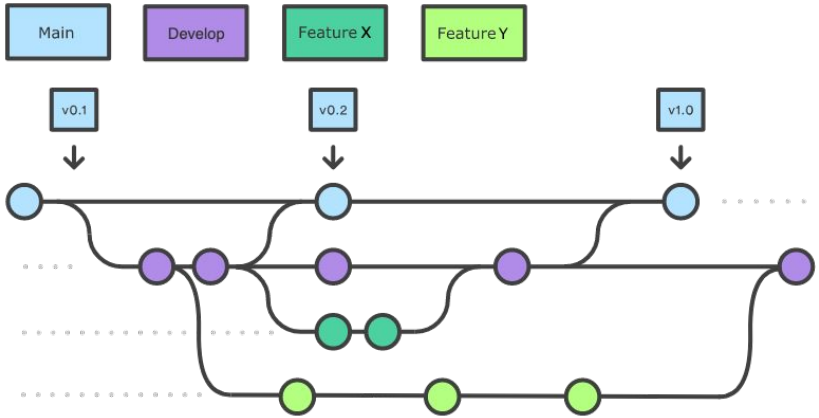
Basic Components

- Your Working directory (untracked)
- Staging area (index)
- Local Repository
- Remote Repository
    - Central Hub for collaboration and storage





Branches + Workflow



Main/Master branch

- Keep for stable, tested code that runs out the box.
- Update with new features when they are done and tested.
- Don't work on or push to directly
- For Users not Developers

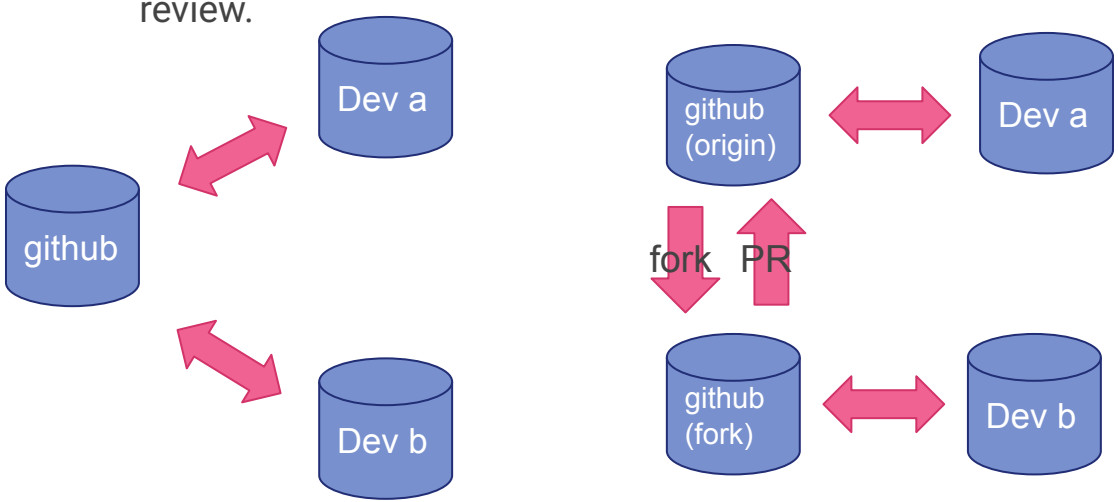
Feature branches

- Create new branch to work on new feature.
  - Each user can do this independently at the same time.
  - 'Push' your changes to your branch local->remote repo
- When ready, merge feature branch into master
  - Pull any updates to master first, resolve conflicts.
  - Merge/Pull 'request' allows everyone to review the changes, run any CI/Unit/Style tests.
- Feature branch is then removed.
- Create a new branch from master for next dev…

HEAD - reference to current commit in currently checkout out branch.

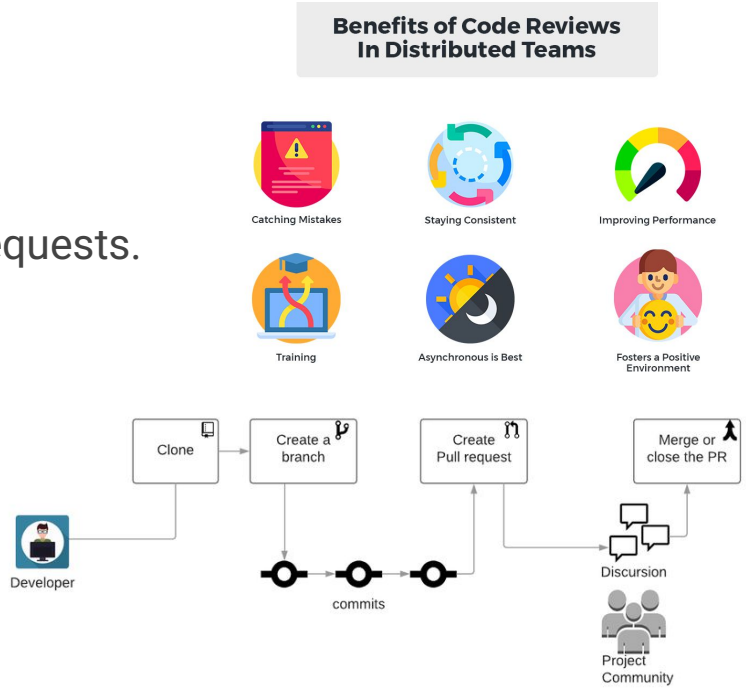
Collaboration Models

- As well as having 1 central repo and user branches…
- Can have ‘forks’ where users make their own cloned repo
  - More common in open source projects.
  - Provides more protection to central main repo/branch, harder to push straight to it without review.

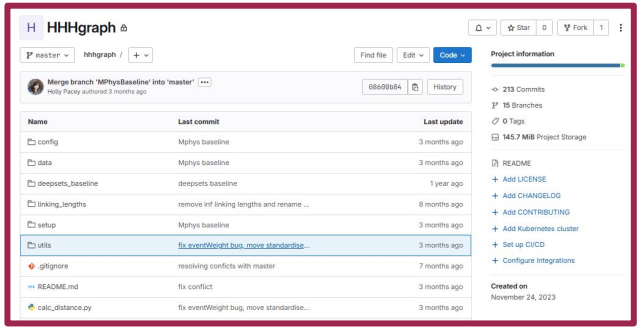


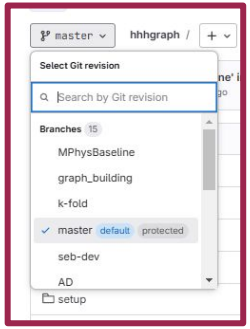
Project Management via Git

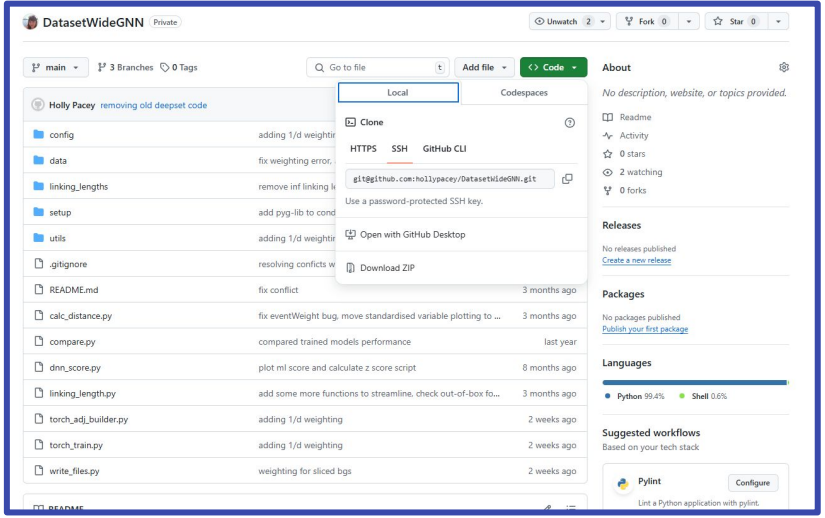
- Can use ‘Issues’ to track to-do list, bugs, feature requests.
  - Can assign to people
  - Can set ‘milestones’ to set a deadline goal
- Code Review
  - During Merge Request/Pull Request (MR/PR)
  - Can assign/tag people, can require approval.
  - Reviewers can comment on particular code lines
    - Spot bugs or suggest better approaches
    - Keep a consistent code style
  - You can still push new changes to the branch
    - they will automatically be included in the MR/PR
    - Reviewer can then ‘resolve’ their comments.
    - Directly editing the code in the browser is also possible!

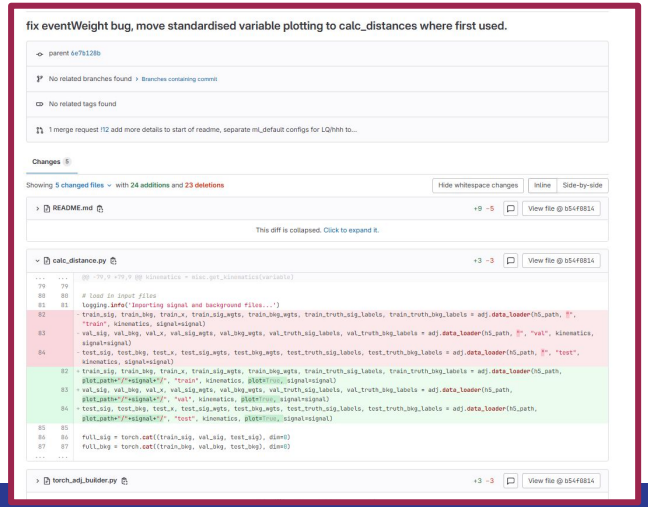


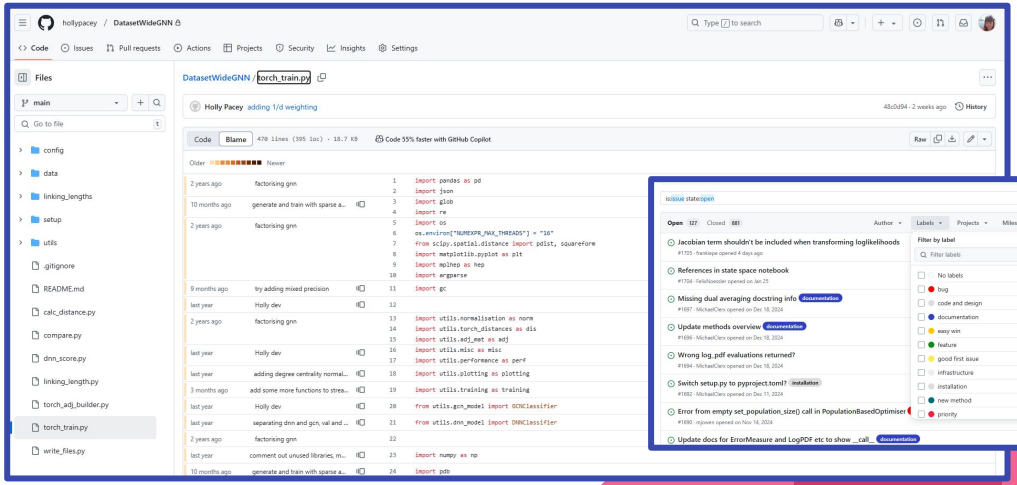
Git appearance











Basic Commands

Note: you must set your name and email using `git config` and `--global user.name`

command	details
`git init`	Sets up git in your folder, if you want to create a new git repo from your local code.
`git clone <code link ssh/https>`	Downloads the repo, on the master/main branch. To use ssh, requires setting up ssh keys on your machine: github keys gitlab keys
`git status`	View the status of your current branch compared to remote, points out all changes and whether they are committed or not.
`git diff`	See the content that has changed between your local/remote branches
`git branch -vv`	View all the branches in the local repo, the one you are on is highlighted. (-vv just gives more verbose info like latest commit message)
`git remote -vv`	View all the remote repos you are linked to (generally you would just have one, called 'origin', if you are using forks you might have two)
`git checkout -b mydev_120225`	Creates a new branch called mydev_120225 in your local repo from HEAD, and switches to it. If the branch already exists you don't need '-b'
`git add <filename>`	Will add your <filename> to the staging area. Can use wildcards, and even add entire folders. Avoid "git add *" you'll end up adding auxiliary things you don't want.
`git commit -m "my very helpful message"`	Collects all the changes in your staging area and applies them to your local repo.
`git push <remote> <branch>`	Pushes the changes from your local branch to the remote branch (will also create this branch in the remote repo if it is new)
`git tag -a my_tag_name -m "my tag description"`	Creates a 'tag' pointing to a specific state of the project history - use to mark code 'release' versions or e.g. versions of code used to make a particular set of analysis materials/results for bookkeeping.
`git stash`	Hide all your local unstaged changes in a 'stash'. Your changes can be later restored if you want.
`git reset`	Can undo changes to the staging/local area, several use options with flags.
`git fetch <remote> <branch>`	Pull changes from remote to local repo, can skip branch to fetch everything.
`git merge <branch>`	Merge branch <branch> local repo version into your working branch.
`git pull <remote> <branch>`	Pull changes from remote to local repo and then merge them into to your working branch. (pull = fetch + merge)

Git tips

```
<<<<<<< HEAD
this is what the existing
code looks like
=======
this is what your new code
looks like
>>>>>>> new_branch_to_merge
```

- What not to put on git? You can use `.gitignore` to manage this
  - Large files (e.g. big root files) gitlab repo has a 10Gb limit, for example.
  - Build/compiled files (*.o, *.pyc, *.out,), a build/run directory
  - Lots of hard-coded user specific paths etc. better to separate these into an untracked script or make user config scripts.
- Commits
  - These are how you add your local changes to the staging area.
  - Commit little and often
  - Use helpful commit messages
- Conflicts
  - Generally if you and others alter the same file, when you try to merge your changes on top of someone else's you can get conflicts you must resolve.
  - Your code will be annotated with "<<< HEAD…." and you need to choose whether to keep the stuff above or below the "=======", or write a 3rd option, and clean up the scripts. Then add/commit/push again.

Continuous Integration

Particularly with larger more collaborative projects we want to be sure that changes don't introduce unexpected problems. Continuous integration is about checking for these things automatically every time something changes.

- Does the change work on different OSes, software/library versions?
- Does it unexpectedly rely on user data on your local machine?
- Is it compatible with other user's changes?
- Is it backward compatible?
- Does the code compile and work on some standard test case?

Can setup on Git, such that a load of test scripts are run automatically on the new code version you are requesting to integrate with the main branch. Gitlab CI tutorial GitHub actions quickstart

Tests

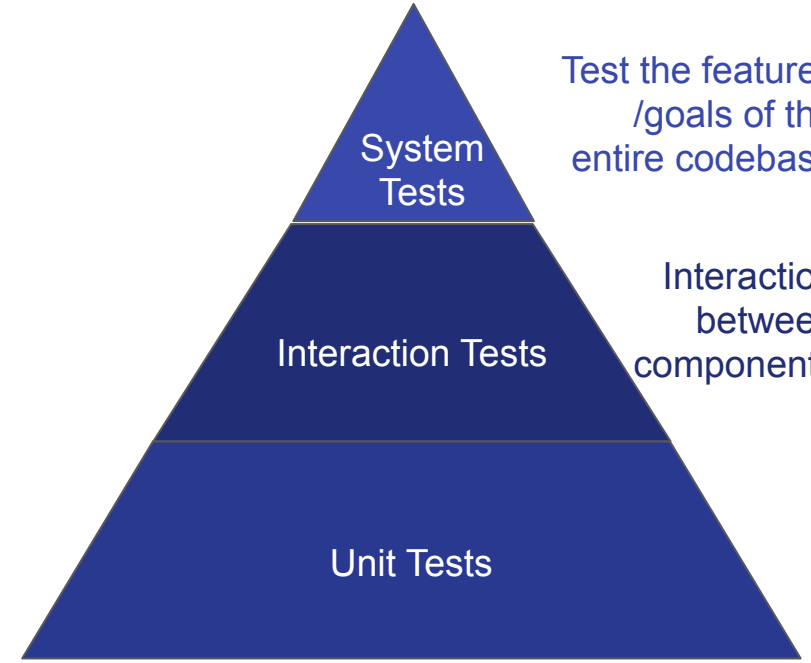
You should test your code manually as you develop it, but writing automated tests allow you to:

- Save time in the long run
- Remember to test for edge cases and all use cases.
- Sanity-check all the pieces of the code.

Types:

- **Unit tests**: Test specific units of functionality, ensuring expected outputs from given inputs.
- **Functional or Integration tests**: Test functional paths through the code, especially useful for exposing faults in inter-unit interactions.
- **Regression tests**: Ensure unchanged program output despite code modifications.

‘Code Coverage’ test can also be measured -> how much of your code is executed during automated tests. (test of tests :p)



Unit Tests

- Frameworks exist to help write/run tests: e.g. Pytest for python, Catch2 for C++
- Goal is to apply function to a known input, and compare the output to what we expect.
  - Test a range of inputs including edge cases e.g. 0s, -ve numbers, …
  - Intentionally test for problem cases - will the code break in the way you want?
    Division by 0, Nans, passing empty objects, …
  - If the test fails, try to be verbose with output so we can learn what failed and why.

```
import numpy as np
import numpy.testing as npt


def test_mean_zeros():
    """Test that mean function works for an array of zeros."""
    from mycode import my_mean_function
    test_input = np.array([[0, 0], [0, 0], [0, 0]])
    test_result = np.array([0, 0])
    # Need to use NumPy testing functions to compare arrays
    npt.assert_array_equal(my_mean_function(test_input), test_result)
```

Recording Environment Setup

For condor jobs, code sharing and saving time, make scripts to…

- Define Environment variables (can be user specific!)
- Define venv/conda environments (e.g. the venv requirements.txt)
- Set up grid envs like rucio
- clone/build/compile a c++ package for the first time.

ATLAS code package example:

first_time_setup.sh:

```
setupATLAS
mkdir -p build run
cd build
asetup
25.2.23,AnalysisBase
cmake ../source
make -j8
source */setup.sh
cd ../run
```

daily_setup.sh:

```
setupATLAS
cd build
asetup --restore
cmake ../source
make -j8
source */setup.sh
cd ../run
```

User specifics

1. Create user_Name.yaml/py files to store specific paths/setting
   In code, read in as a command line arg.
2. Create a setup_env.sh file to define these paths as environment variables, with user-based options defined via username
   In code, read in via OS.

User specif

In user_Holly.yaml:

```
GRIDNAME = "hpacey"
BASEDIR = “/data/atlas/users/pacey/emus/”
HISTDIR = BASEDIR + “/histograms/”
CODEDIR = “~/emus/”
```

In script:

```
import argparse
Import os

def GetParser():
    """Argument parser for reading Ntuples script."""
    parser = argparse.ArgumentParser(description="Reading command line
options.")

    parser.add_argument("--userconfig", "-u", type=str, required=True,
                        help="Specify the config for user’s paths”)
    args = parser.parse_args()
    return args

args = GetParser()

user_config_path = args.userconfig
user_config = misc.load_config(user_config_path)

hist_dir = user_config["HISTDIR"]
```

User specifics

1.

In setup_env.sh:

```
#!/usr/bin/env bash

# user-specific setup
if [[ $USER == "hodkinson" ]]; then
  export GRIDNAME="bhodkinson"
  echo "Welcome Ben!"
elif [[ $USER == "pacey" ]]; then
  export GRIDNAME="hpacey"
  echo "Welcome Holly!"
  export BASEDIR=/data/atlas/users/pacey/emus/
else
  export BASEDIR=/r10/atlas/emus/
fi

export HISTDIR=$BASEDIR/histograms/
export PYTHONPATH=$COMMONDIR:$PYTHONPATH

export CODEDIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null
&& pwd )"
export COMMONDIR=$CODEDIR/common/
```

In script:

```
import os
hist_dor = os.environ[“HISTDIR”]
```

Coding + Common Tools

Python vs C++

Generally, C++ for heavy lifting

- LHC data processing, reconstruction, Calibration, .. (looping over many events)
- Software-based Triggers
- Monte Carlo e.g. Pythia
- ROOT-based things e.g. Statistics tools.

Python becoming more common for…

- Plotting (ROOT -> MPL)
- Data Analysis via ML.
- Stats. E.g. PyHF, Cabinetry, …
- Documentation (ReadTheDocs, …)

Parameters	Python	C++
Code	Python has fewer lines of code	C++ tends to have long lines of code
indents+	White space matters, indentation must be consistent to define structures	; at end of each line, {} used for structures, white space not important.
files	Just the .py	Needs header.h files as well as code.cxx with all function declarations to tell compiler what to compile.
Compilation	Python is interpreted	C++ is precompiled
Speed	It is slower since it uses an interpreter and also determines the data type at run time	It is faster once compiled as compared to python
Efficiency	Specialized formatting not common in other languages, script-like language, OOP features, code reuse through libraries	C-like syntax, powerful OOP features and operator overloading, best compile-time optimizer
Nature	It is dynamically typed.	is statically typed
Functions	Python Functions do not have restrictions on the type of the argument and the type of its return value	In C++, the function can accept and return the type of value which is already defined
Scope of Variable	Variables are accessible even outside the loop	The scope of variables is limited within the loops
Variable Declaration	Variables are declared by simply writing their name with their datatype	While declaring a variable it is necessary to mention its datatype

More info…, and here

*Note C++ expertise becoming rarer and more prized in industry, good to gain some experience using it, regardless of future career plans!

Programming Paradigms

Program structure

Many ways to structure the same code… consider: Maintainability, Extensibility

No substitute for proper planning

- What types of things do we have to represent?
- What data do we need about them?
- What are the relationships between them?

Paradigms (Most modern languages support most paradigms):

- Declarative Family: Code describes **what** data processing should happen.

  E.g. Functional: functions are maths operations, code is data. Increasingly used for e.g. ML, parallel programming.

- Imperative Family; Code describes **how** data processing should happen.

  E.g. Procedural: code should be groups into procedures performing a single task.

  E.g. Object Oriented: data should be structured, code belongs with data..

Why? Using/knowing common patterns help you to read code & design it well.

Object Oriented Programming: Classes

Builds on procedural paradigm, focus on data.
Encapsulation: structure data + behaviour together into **Classes**

- Structuring data crucial for complex systems.
- Class blueprint ensures sensible physics-informed processing.
- **Attributes:** hold current state of class instance.
- **Methods:** define class behaviours.

Relationships?

- **Inheritance** - Identity relationship ('x is a y')
  - Class x gets all *public* Atts./Mets. from Class y + added X-specific ones.
  - Useful to control common/distinct features: e.g. electron is a particle, hadronic-jet is a particle, but electrons and jets have distinct properties (e.g. charge vs clustering radius)
- **Composition -** Ownership relationship ('x has a y')
  - Class X contains an object (or objects) of class Y.
  - Useful for complex/flexible structures. E.g. A collision has some electrons, jets, conditions,..

Object Oriented Programming: Classes

In Particle.h :

```
#include <TLorentzVector.h>

class Particle {
    public:

        //Particle() {};
        virtual ~Particle() {}

        TLorentzVector p4;
};
```

```
#include "xAODAnaHelpers/Particle.h"

class Electron : public Particle {
    public:

        // kinematics
        float  charge;

         // Reconstruction:
        std::map<std::string,
std::int> PID;

    public:
        int passPID(PIDwp wp) const;

};
```

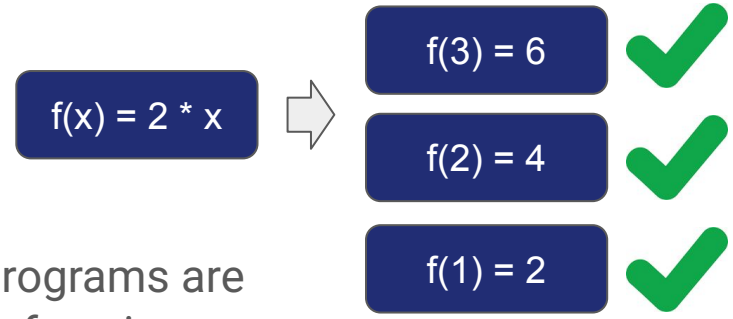
```
import TLorentzVector as TLV

class Particle:
    def __init__(self,
pfour):
        self.pfour = pfour
```

```
class Electron(Particle):
   // override init to add new attributes.
    def __init__ (self, pfour, charge, PID)
        Particle.__init__(self, pfour)
        // kinematics
       self.charge = charge

        // Reconstruction:
        self.PID =  PID

    def passPID(PIDwp  wp):
         ….
```

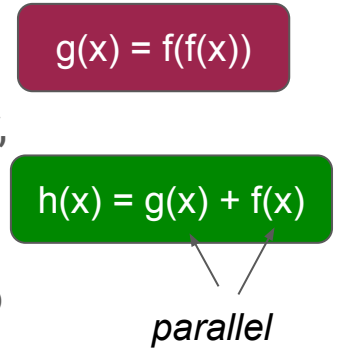
```
class Zboson {
    Public:
        Electron electron;
        Electron positron;
        double mass (p41, p42) const;
}
```

```
class Zboson:
    def __init__(self, pfour_e, pfour_p, charge_e, charge_p, PID_e, PID_p)
        self.electron = Electron(pfour_e, charge_e, PID_e)
        self.positron = Electron(pfour_p, charge_p, PID_p)
        self.mass = TLV.M(pfour_e + pfour_p)
```

Functional Programming



- **Definition:** functions are *first class objects* and programs are constructed by *applying and composing/chaining functions*.
- **Difference from Procedural Programming:** Focuses on what transformations are done to data, rather than how they're performed.
- **Data immutability:** Functional computations rely on input values only, do not modify anything outside the current functions. Otherwise we have **side effects**.
- **Pure functions**: always return the same output for the same input, no side effects.
  - Easily **testable**
  - Can be **compounded** into new pure functions.
  - Easy to **parallelise**



Types of Functions

- **First Class Functions** are functions that can be passed as arguments, returned from functions, or assigned to variables.
  - True in python, not quite in c++ although you can use function pointers similarly.

Python:

```python
# First Class Function
def greet():
    return "Hello"

hello = greet
print(hello())
# Output: Hello
```

C++:

```cpp
# First Class Function
std::string greet(){
    return "Hello";
}

std::string (*func)();
func  = greet;
std::cout<< func()<<std::endl
// Output: Hello
```

Types of Functions

- **Lambda Functions** are small, nameless functions defined in the normal flow of the program.

python:

```python
# Lambda Function
add = lambda x, y: x + y
print(add(2, 3))
# Output: 5
```

C++:

```cpp
// Lambda Function
int add = [] (int x, int y) {
   return x + y;
};

std::cout<<add(2,3)<<std::endl;
//Output: 5
```

Types of Functions

- **Higher Order Functions** are functions that take other functions as arguments.
  - Examples: Map is a built-in higher order function in Python that uses lazy (lambda) evaluation.
  - Decorators are higher order functions that take a function as an argument, modify it, and return it. (in c++ e.g. do via inherited classes)

C++ (rather stupid example but oh well):

```
#include <math.h>
float Square = [] (float x) { return pow(x,2)};
// Higher order function:
float Cube(float x, function<float(float)> square){
    return x*Square(x);
}
float result = Cube(3.,Square); // 9.
```

```
numbers = [1, 2, 3, 4, 5]

# Higher order e.g. map
squared = list(map(lambda x:
x**2, numbers))
print(squared)
# Output: [1, 4, 9, 16, 25]
```

```
# decorator
def my_decorator(func):
    def wrapper():
        print("Before
function call")
        func()
        print("After
function call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello")

say_hello()
```

Types of Functions

- **Comprehensions** are a more Pythonic way to structure map/filter operations over eg. lists and dictionaries.
- **Generators** are similar to list comprehensions but behave differently and aren't evaluated until iterated over.

```
# List Comprehension
squared = [x**2 for x in numbers]
print(squared)
# Output: [1, 4, 9, 16, 25]

# Dictionary Comprehension
squared_dict = {x: x**2 for x in numbers}
print(squared_dict)
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Generator
squared_gen = (x**2 for x in numbers)
for num in squared_gen:
    print(num)
# Output: 1 4 9 16 25
```

Some Coding Basics & Good Practice

C++/Python

Inbuilt Types

Just some common examples.

In python: Mutable (Immutable) objects can (can't) be altered after defined. Defined by their type.

In c++: All mutable by default. Use 'const' or 'constexpr' to make something immutable.

Types	C++	Python
Integer	int≥16bit (long≥32bit, short≥8bit)	int - no size limit
Floating point	float 32bit ~7 digits (double 64bit)	float 64bit (float32 & float64 in numpy)
bool	true/false. Also 0==false, !false=true. Ops: &&, ||, !	True/false. 0==False, !False=True. Ops: and, or, not
Ordered sequences	Vectors: can alter size after defining by adding to the END std::vector<int> x = {1, 2, 3}; can index: x[1]=2, x[-1]=3. Lists: can add to any position inside list no indices, can iterate. std::list<int> x = {1, 2, 3};	lists: mixable types, mutable after defining. x = ["a", 2]; x[1]=2 tuples: like lists but immutable: y = ("a", 2); y = tuple(x)
Unordered sequences	Set: no repeats, no indices. std::set<int> s{1, 2, 5}; values stored in asc. Order. Unordered set: same but random order. std::unordered_set<int> s{1, 2, 5};	Set: mutable, no repeats, no indices, iterable. s = set(x).
Key-value mapping	map: keys unique std::map<std::string,int> a {{"cow",1},{"pig",2}; a["cow"] -> "moo".	dictionary: keys unique and immutable. a = {"cow": "moo", "pig": "oink": "swan": 0} a["cow"] -> "moo".
Letter/word related	Builds on 'char' type. std::string s = "moo"; can iterate through them but not indexable like python.	String: works as a sequence, immutable ordered array of unicode characters, no char eqv. s = "moo" or s = 'moo'. s[0]=0.

Error Handling

```
try:
    age=15
    if (age >=18):
        print(“Access granted - you are old enough”)
    else:
        Raise exception(“Access denite - You must be
at least 18 years old. Age is {}”.format(age))
```

How to handle errors in a helpful way: Exceptions.
Inbuilt exceptions exist e.g. for div0, index errors, undefined objects,... but you can also define your own custom exceptions.
C++ Try & Catch

- The **try** statement: define a block of code to be tested for errors while it is being executed.
- The **throw** keyword: throws exception when problem is detected; lets us create a custom error.
- The **catch** statement: define a block of code to be executed, *if an error occurs in the try block*.

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```

Python Try & Except

- The **try** statement: define a block of code to be tested for errors while it is being executed.
- The **Except** keyword: throws an exception when a problem is detected, which lets us create a custom error.

Documentation

Documenting your software is crucial

1. Improves understanding: easier/faster for new users/devs (& your future self) to learn.
2. Easier to maintain and understand the design decisions, future proofs the code against loss of developers.
3. Aids collaborative working.
4. Easier to spot where you could reuse your code in other projects.
5. Better quality software design - helps you think about choices, commenting as you write helps break it down into considered steps.

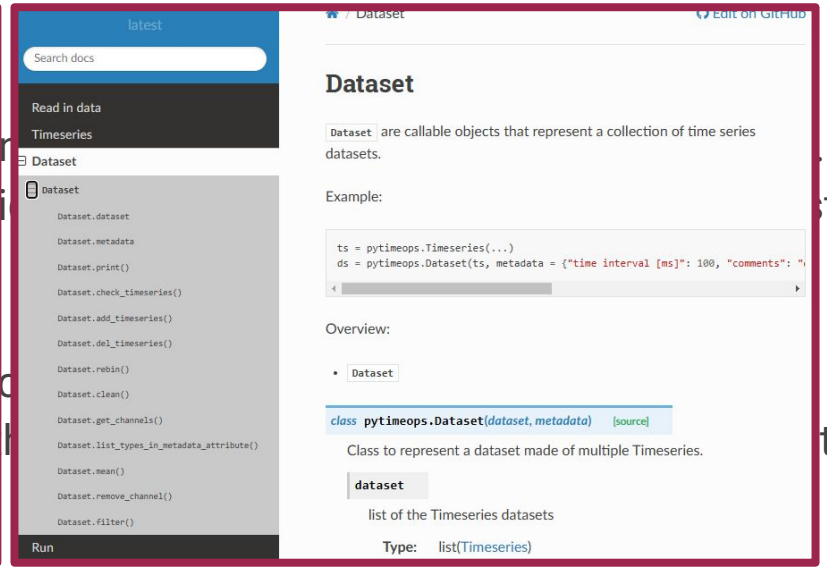
Minimum: Add code Comments + function docstrings; write proper README for setup/use
Use helpful names for functions/variables

Enhanced: Use tools Sphinx/ReadTheDocs/MkDocs/doxygen to make (more worthwhile for larger more collaborative projects)

Documentation

```
sphinx-build -b html docs/source/ docs/build/html
```

```python
class Dataset:
    """Class to represent a dataset made of multiple Timeseries.

    Attributes:
        dataset (list(Timeseries)): list of the Timeseries datasets
        metadata (dict): global metadata about the dataset including
            e.g. time_interval [ms], filename and comments.

    """
    def __init__(self, dataset, metadata, times):
        """
        Parameters
        ----------

        dataset (list(Timeseries)): list of the Timeseries datasets
```



Minimum: Add comments to functions etc., write proper README for setup/use

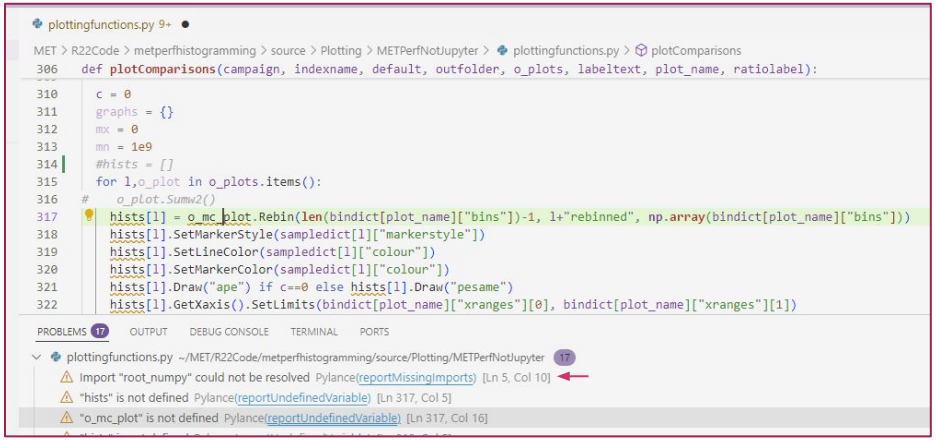
Enhanced: Use documentation tools Sphinx/ReadTheDocs/MkDocs
(more worthwhile for larger more collaborative projects)

Good Style Practices

- Always assume that someone else or your future self will want to look back and understand your code
  - Take the time to make your code efficient and good from the start to save yourself time overall, and minimise the chance of bugs
    - E.g. use config files & command line arguments instead of hard-coded settings/paths.
    - Use clear names and comment as you write to keep it readable.
    - Setup bash scripts to run through multiple steps and automate boring things.
- You have the same few lines of code repeated? Put it in a function.
- You frequently manipulate similar objects in of similar ways? Use classes
- Really long script doing many things?
  - split it up into modules with distinct purposes,
  - use utils scripts to separate standalone functions out from the main algorithms.
- Be consistent with tab/space indentations, layout of brackets, spacing, etc.
- Don't automatically overwrite outputs -> use descriptive file+folder names.

Linting

Linters are tools that analyse code for errors, inconsistencies and stylistic issues. They help readability, maintainability & adherence to best practices.



Many editors can check your code syntax and errors or provide linting:

- Emacs has a built-in linter
- VScode checks syntax, and you can use pylint extension (also can use pylint standalone).

Beware - depending on compiler environment and language version the errors/suggestions you get locally in your editor might not be the same as in your final environment. So this is not a substitute for actually testing your code!

How to improve

1. Practice.
2. Find existing code to use as a starting point and edit it, rather than starting from a blank page.
3. When you are given code to use try to actually understand it, ask questions about it.
4. Think about how to split the task up into small pieces, write comments/pseudo-code to map out the steps needed.
5. Do use AI Tools (chatGPT, git co-pilot, ...) to help generate code snippets etc., but..
    - Try to understand why/how this code works, so you actually improve your skills and understanding.
    - Use it more to save time on repetitive/lengthy things you know, or to get starting points to try something new, rather than making your code a black box.
    - Refine the output (names, comments etc.) to ensure clarity and style consistency in your use case.
    - Don't let it replace you practising coding. You will learn far more by coding yourself, from debugging your buggy attempts and working out why things *don't* work.

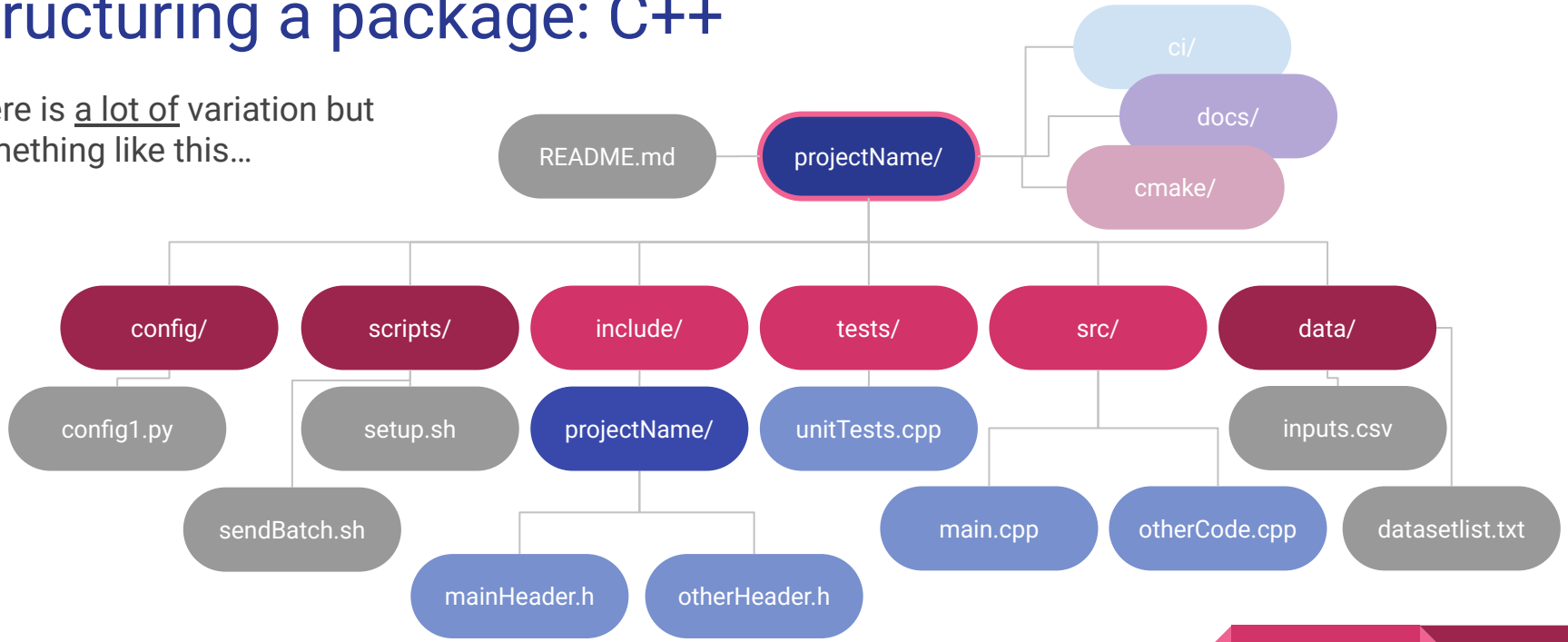
Do google it. -> stack overflow etc. can be helpful to debug.

Try to enjoy the process of creating something functional for its own sake.

Compiling & Packaging

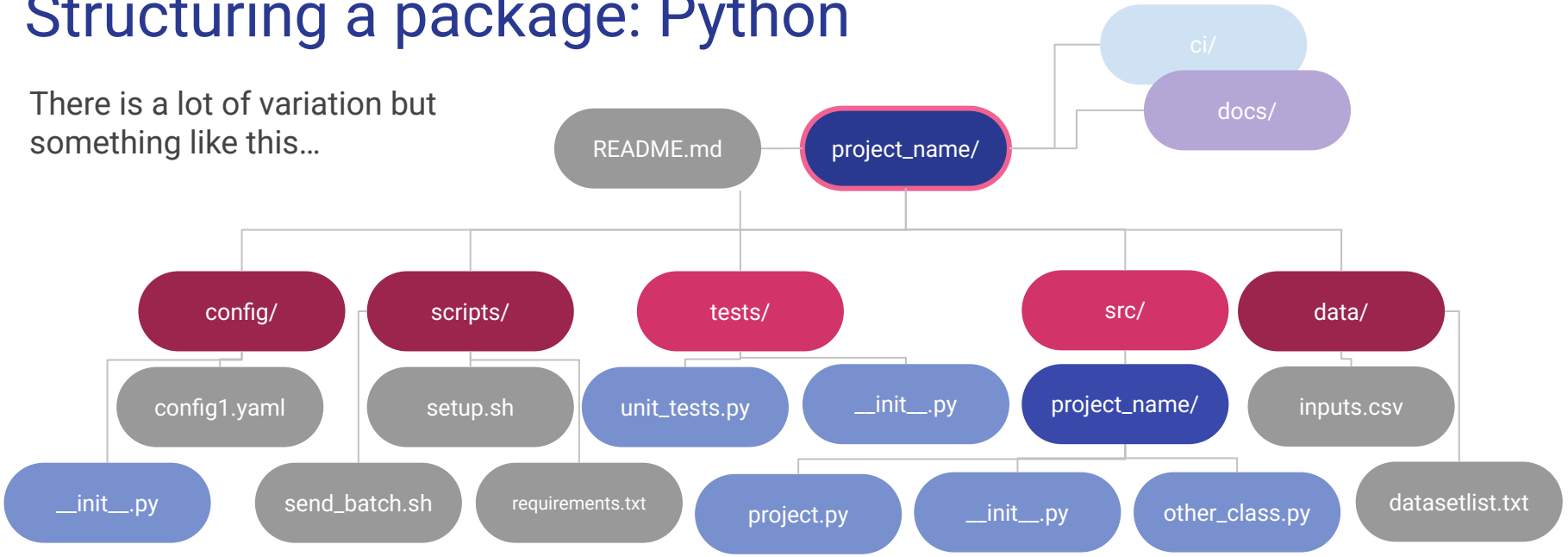
Structuring a package: C++

There is a lot of variation but something like this…



Structuring a package: Python

There is a lot of variation but something like this…



Compiling Standalone C++

Can use g++ for simply compiling 1 program Into an ‘executable’ which will run the main() Function.

Add libraries to add e.g. ROOT

```
// hello.cc
#include <iostream>
int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

```
g++ hello.cc -o hello

./hello
```

```
// trees.cc
#include <iostream>
#include "TH1.h"
int main()
{
  std::cout << "Hello World!" << std::endl;
  TH1D* hist = new TH1D("hist", "title", 100, 0, 100);
  hist->Fill(1,3);
  hist->Print();
  return 0;
}
```

```
g++ trees.cc -o trees # doesn’t work

root-config --cflags

g++ trees.cc -o trees `root-config --cflags` `root-config --libs`
```

Compiling Standalone C++

Can compile multiple files (still one main!).
Header files grabbed via #include. “Wall” turns on warnings

```
> g++ -c my_program.cpp
> g++ -c my_class.cpp
> g++ -o my_program my_program.o my_class.o
```

In one step:

```
> g++ my_program.cpp my_class.cpp -Wall -o my_program

> ./my_program
Doing something!
```

```
// my_class.cpp
#include "my_class.h" // header in local directory
#include <iostream> // header in standard library

using namespace N;
using namespace std;

void my_class::do_something()
{
    cout << "Doing something!" << endl;
}
```

```
//
my_program.cpp
#include
"my_class.h"

using namespace
N;

int main()
{
    my_class mc;

mc.do_something(
);
    return 0;
}
```

```
// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H

namespace N
{
    class my_class
    {
    public:
        void do_something();
    };
}

#endif /* MY_CLASS_H */
```

Compiling Standalone C++

OR use a Makefile, for more complexity.

- Encapsulates the settings to compile and what compiler.
- Define the target program to produce
- Instructions for how to compile and use the dependencies.
- Instructions to Clean if you want to undo compilation.

```
// Makefile
# Variables to control Makefile operation
CC = g++
CFLAGS = -Wall -g

# Targets needed to bring the executable up to date
TARGET = my_program

$(TARGET): my_program.o my_class.o
    $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).o
my_class.o

# compile the individual files
$(TARGET).o: $(TARGET).cpp my_class.cpp
    $(CC) $(CFLAGS) -c my_program.cpp
my_class.o: my_class.cpp
    $(CC) $(CFLAGS) -c my_class.cpp

clean:
    $(RM) $(TARGET) $(TARGET).o
    $(RM) my_class.o
```

```
[pacey@pplxint12 test]$ ls
Makefile  my_class.cpp  my_class.h  my_program.cpp
[pacey@pplxint12 test]$ make
g++ -Wall -g -c my_program.cpp
g++ -Wall -g -c my_class.cpp
g++ -Wall -g -o my_program my_program.o my_class.o
[pacey@pplxint12 test]$ ls
Makefile  my_class.cpp  my_class.h  my_class.o  my_program  my_program.cpp  my_program.o
[pacey@pplxint12 test]$ ./my_program
Doing something!
[pacey@pplxint12 test]$ make clean
rm -f my_program        my_program.o
rm -f my_class.o
[pacey@pplxint12 test]$ ls
Makefile  my_class.cpp  my_class.h  my_program.cpp
```

Build Systems

Complex (C++ based) analysis frameworks often require a build system that manages:

- Compilation,
- Dependencies and software version requirements,
- environment setups,
- Cross-platform issues, …

On a small scale often having a **Makefile** is sufficient. For more complex software build systems like **CMake, CMT, MRB**, etc. are available.

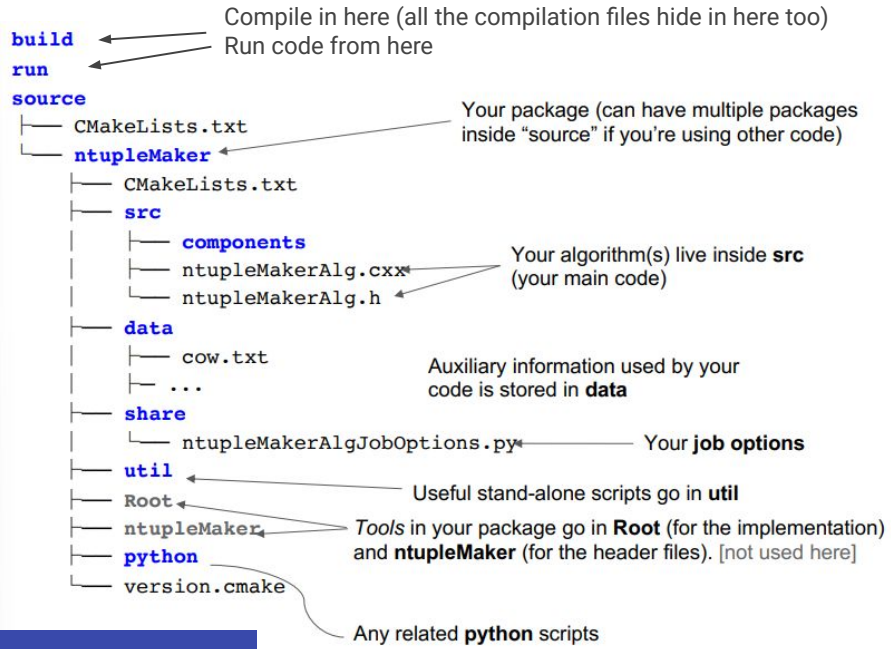
Usually your experiment's software tools come with their build system and you probably don't need to set it up yourself! Just know how to use it.

CMake example

Most large experiment data reconstruction/analysis software uses CMake (ATLAS Athena, LHCb Gaudi, DUNE duneana, …)

CMakeLists.txt & other cmake/ files instruct

- What is this project?
- What to compile - what executables and libraries?
- What are the dependencies
- What version of g++/gcc compiler is needed?
- …



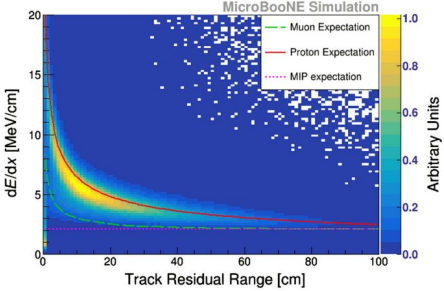

```
setupATLAS
mkdir -p build run
cd build
asetup 25.2.23,AnalysisBase
cmake ../source
make -j8
source */setup.sh
cd ../run
```

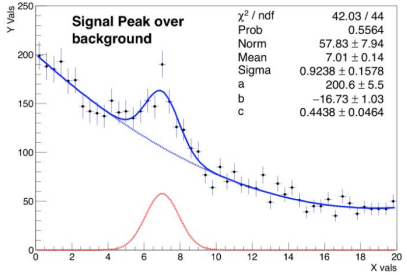
Common Tools

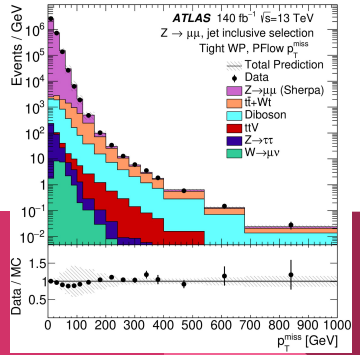
Root /1

ROOT primer , ROOT tutorials, another tutorial , new ROOT student course
An open-source data management system with a lot of functionality ideal for HEP experiments:

- Data management, representation, processing (central part of data reconstruction workflows in ~all HEP experiments, and related software e.g. Delphes/Geant4/...)
  - Basis is TTree: array of events and their properties (Tbranches) -> with nice flexibility that properties can be complex variable-length vectors etc. to represent eg. ‘muons’ in the event
  - Have standard int/float/… but also TLorentzVectors etc. for useful 4-mom variables, and can define custom C++ objects e.g. ‘Electrons’ with appropriate properties.
- Fast C++ interpreter. Can run Root in C++ via code or interactive GUI, or PyROOT in python.
- Visualisation
  - Plots! Big support for Histograms, fitting, and producing plots, can interface experiment style files for publication plots etc. Can save plot figures themselves and underlying ingredients to edit further.



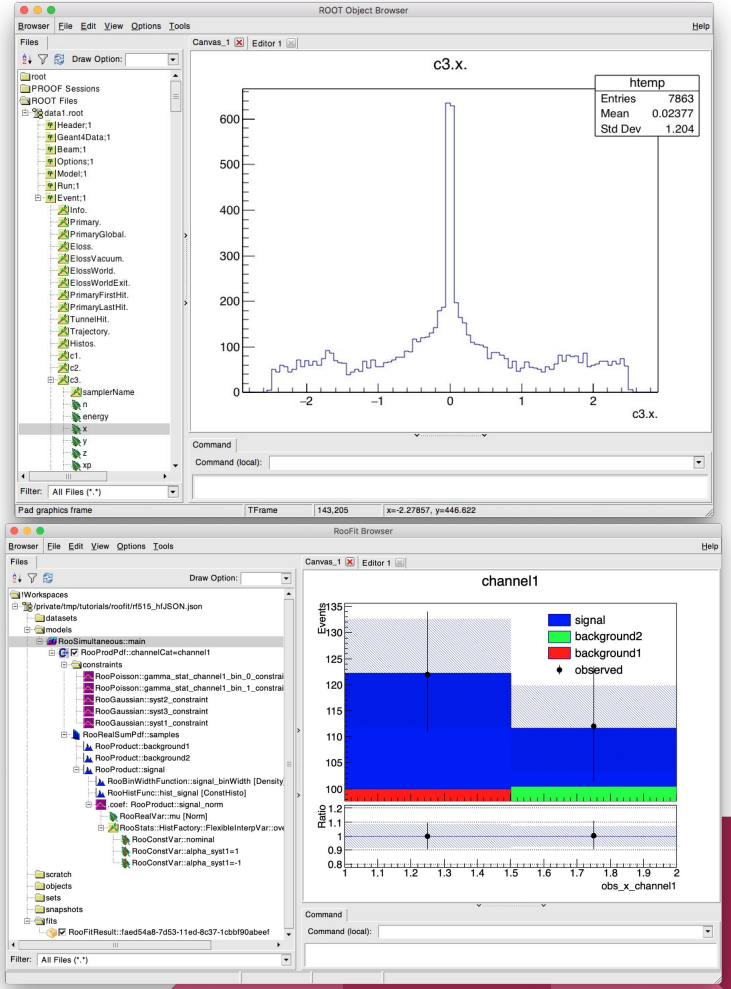




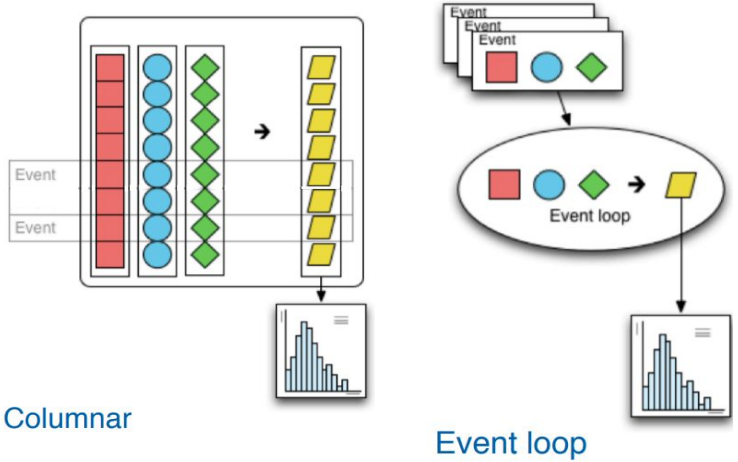
Root /2

ROOT primer , ROOT tutorials, another tutorial , new ROOT student course
An open-source data management system with a lot of functionality ideal for HEP experiments:

- Statistical analysis
  - E.g. RooStats, RooFit, and HistFactory are built on Root - creating likelihoods, PDFs, and performing fits.
- GUI
  - Can view TTrees, Histograms, workspaces etc. in an interactive TBrowser.
  - Can also run .C macros in an interactive root shell, for e.g. making plots / quick things, don't need compilation.
- Can move your data between ROOT formats and pythonic analysis/ML formats easily now:
  - Use Uproot to go between TTree and e.g. pandas dataframe / .h5 format.
  - Root also has 'dataframe' type structures to emulate pythonic-style formats that can benefit from vectorisation for speed.



Python Alternatives.



Libraries?

- Numpy/Pandas for big tabular arrays or ‘dataframes’
- Awkward arrays for non-tabular structures

Why use python instead?

- Easier to learn, do complex things with shorter code.
- More internet/AI-wide help available
- Integrate better with other modern tools like ML.
- ‘Default’ output of e.g. matplotlib, is nicer than root plotting.
- Vectorised ‘columnar’ calculations nice to read, can be fast for smaller datasets, easy when you have fixed-size tabular data format.

Why use root instead?

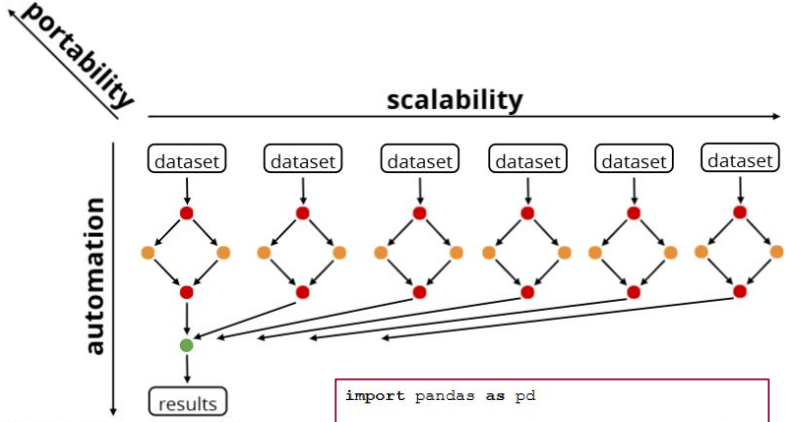
- Existing experiment code-bases more likely to be in ROOT; if you are using experiment specific data formats. Using HEP standard stats tools like RooStats/HistFactory/SPlot
- Still not THAT hard to learn, can have a lot of control over plots, can still use python if you want.
- Event Loop approach easier with complex variable-sized data and very complex selections to apply.
- Particularly if you use C++, fast on larger datasets.

Snakemake

LHCb analysis uses this a lot! Tutorial general slides

Snakemake is a workflow system for analysis.

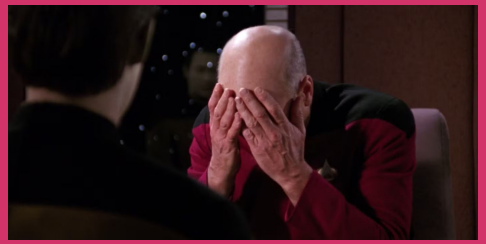
- Automate a pipeline!
- Human readable pythonic instructions to run a series of steps in terms of ‘rules’ for how to create output from input.
- Scalable to cluster/grid/parallelisation, can include env. setup instructions (e.g. via conda).
- Follows GNU Make (Makefile-based) paradigm.
- Jobs to run the pipeline get run based on given data are executed if and only if any of these conditions applies:
  - Output file is target and does not exist
  - output file needed by another executed job and does not exist
  - input file newer than output file
  - rule has been modified
  - input file will be updated by other job
  - execution is enforced




```
DATASETS = ["D1", "D2", "D3"]

rule all:
    input:
        expand("{dataset}.sorted.txt", dataset=DATASETS)
rule mytask:
    input:
        "path/to/{dataset}.txt"
    output:
        "result/{dataset}.txt"
    script:
        "scripts/myscript.R"


rule myfiltration:
    input:
        "result/{dataset}.txt"
    output:
        "result/{dataset}.filtered.txt"
    shell:
        "mycommand {input} > {output}"


rule aggregate:
    input:
        "results/dataset1.filtered.txt",
        "results/dataset2.filtered.txt"
    output:
        "plots/myplot.pdf"
    script:
```

```
import pandas as pd

data = pd.read_table(snakemake.input[0])
data = data.sort_values("id")
data.to_csv(snakemake.output[0], sep="\t")
```

Problems



What to do when it crashes

Don't panic. (1) find what line of code or eqv. failed. (2) find out why it failed.

First of all, help yourself.

- Scan your terminal output or log files from the top down - what is the first ERROR/FATAL/Exception you see?
- What is the first line of *your* code it points to?
- Are there any other WARNINGs?
- Has it failed to read/find something? Did you just not give it the right path to an input file?
- Try again from a fresh terminal, with a fresh setup and compilation (e.g. entirely delete the build/ dir). Often this does mysteriously fix everything!
- Add print statements to your code to help narrow in where it stopped.
  - Hide these under a `verbose = True; if (verbose): print("oink")` , if you might want to use these again for future debugging or generally give more info on the code's progress.
  - Or use an actual Debugger to step through the code (next slides).
- Google/AI the error message.
- Check the last thing you changed in the code (version control can help again here!); Test it with other settings and inputs to see the scope of the problem.
- Is it an environment/permissions problem (e.g. it worked locally but not on condor, it's not expecting our el9 OS, a package is missing, there's an I/O issue writing an output or storing it)?
- Is it a memory issue? (does it just say KILLED?) Can look at Profiling (next slides).
- **Do you have other ideas?**

Then ask for help… Give as much info as possible. Don't assume that sending someone an error message or seg fault will mean anything to them, even if it's from their code!

- What command did you run, what kind of input/config
- Send the complete set of errors/warning
  - Generate a log file: `Command |& tee moo.log`
- Send the code (ideally as a link to git)

Debuggers

Debuggers allow you to step through the code line-by-line, evaluate expressions at each point in the code, etc.

Some editors have built-in debuggers. Python has ‘pdb’, C++ has e.g, ‘gdb’

```
(base) eschopf@ppmaclap01 Documents % python -m pdb ~/Desktop/mytest.py
> /Users/eschopf/Desktop/mytest.py(4)<module>()
-> def main() :
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(13)<module>()
-> if __name__ == "__main__":
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(14)<module>()
-> main()
(Pdb) step
--Call--
> /Users/eschopf/Desktop/mytest.py(4)main()
-> def main() :
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(6)main()
-> counter = 0
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(7)main()
-> if counter > 0 :
(Pdb) p counter
0
(Pdb) step
> /Users/eschopf/Desktop/mytest.py(9)main()
-> print (myVariable)
```

Stepping through the code

Evaluate value of counter

mytest.py

```
1   #!/usr/bin/env python
2
3
4   def main() :
5
6       counter = 0
7       if counter > 0 :
8           print ("found event")
9       print (myVariable)
10
11      print ("done")
12
13  if __name__ == "__main__":
14      main()
15
```

Profiling

When you are performing millions of time/CPU/memory heavy operations, e.g. (processing millions of data events, calculated complex quantities or corrections, handling lots of copies of objects or very large objects)...

Can be helpful to understand which parts of the code consume the most resources, to…

- Better optimise things
- Spot and fix memory leaks (e.g. if you have a loop and each time you create something in memory but it doesn't get deleted so it just keeps growing.
  Hint: in c++ for every 'new' you have have a delete)
- Fix memory related crashes

Open-source tools are available for this e.g. valgrind.

The End!

Other resources

"The Missing Semester of Your CS Education"

HEP Software Foundation - C++ Course BASICS

HEP Software Foundation - C++ Course ADVANCED

CMake intro