Prerequisites

Installing conda

```
$ mkdir ap_test
$ cd ap_test

Linux:
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ chmod u+x Miniconda3-latest-Linux-x86_64.sh
$ ./Miniconda3-latest-Linux-x86_64.sh

Mac:
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh -O ~/miniconda.sh
$ bash ~/miniconda.sh -b -p $HOME/miniconda
```

~~Read through license terms~~ press q and type “yes”.

Select default install location.

Type “yes” when prompted to initialize conda in .bashrc

Creating our environment

```
$ git clone ssh://git@stash.desy.de:7999/~murphyco/ap_startertalk.git
$ conda env create
```

See you in 10!

Analysis preservation (AP)

Colm Murphy (IPMU)

Belle II StarterKit

What is analysis preservation?

Over the course of your analysis you will produce **thousands of lines of code**, and tens or hundreds of scripts.

What is analysis preservation?

Over the course of your analysis you will produce **thousands of lines of code**, and tens or hundreds of scripts.

You may also produce large quantities of data and/or MC, with very **specific requirements** and **conditions**.

What is analysis preservation?

Over the course of your analysis you will produce **thousands of lines of code**, and tens or hundreds of scripts.

You may also produce large quantities of data and/or MC, with very **specific requirements** and **conditions**.

How can we ensure analyses are **reproducible**? (the scientific method!)

*__Preserve__ your **code** and **data** so your analysis can be re-run in future*

Why should we make analyses reproducible?

For you!

Why should we make analyses reproducible?

For you!

When it comes to review, you may have to make a change to an early stage of the analysis. **If you have a reproducible analysis, propagating this change will be quick and easy**.

Why should we make analyses reproducible?

For you!

When it comes to review, you may have to make a change to an early stage of the analysis. **If you have a reproducible analysis, propagating this change will be quick and easy**.

For others!

Why should we make analyses reproducible?

For you!

When it comes to review, you may have to make a change to an early stage of the analysis. **If you have a reproducible analysis, propagating this change will be quick and easy**.

For others!

Have you had to inherit complicated code from a previous student? Without AP it can be difficult or impossible to understand how an analysis works. **Following AP helps us collaborate**.

How do we make our analyses reproducible?

It is much easier to make a new analysis reproducible!

1. Organize our code in a **git repo**, and make it available to others

How do we make our analyses reproducible?

It is much easier to make a new analysis reproducible!

1. Organize our code in a **git repo**, and make it available to others
2. Make detailed notes on "**data provision**", i.e. exactly what commands you use to get your data/MC, and what computing environment you used.

How do we make our analyses reproducible?

It is much easier to make a new analysis reproducible!

1. Organize our code in a **git repo**, and make it available to others
2. Make detailed notes on "**data provision**", i.e. exactly what commands you use to get your data/MC, and what computing environment you used.
3. Use a **virtual environment** to describe exactly which software you are using (e.g. virtualenv, pip, conda)

How do we make our analyses reproducible?

It is much easier to make a new analysis reproducible!

1. Organize our code in a **git repo**, and make it available to others
2. Make detailed notes on “**data provision**”, i.e. exactly what commands you use to get your data/MC, and what computing environment you used.
3. Use a **virtual environment** to describe exactly which software you are using (e.g. virtualenv, pip, conda)
4. Use a **workflow manager** to automate your analysis procedure. (e.g. Snakemake, Luigi)

How do we make our analyses reproducible?

It is much easier to make a new analysis reproducible!

1. Organize our code in a **git repo**, and make it available to others
2. Make detailed notes on “**data provision**”, i.e. exactly what commands you use to get your data/MC, and what computing environment you used.
3. Use a **virtual environment** to describe exactly which software you are using (e.g. virtualenv, pip, conda)
4. Use a **workflow manager** to automate your analysis procedure. (e.g. Snakemake, Luigi)

1 and 2 are already strongly encouraged. 3 and 4 may be soon, so get ahead!

1. git

Belle II defines some standards for creating an analysis repository. You are encouraged to follow this procedure from the beginning.

This presentation will assume a basic familiarity with the principles behind git, and remote hosted repositories. (All git commands needed will be supplied)

Exercise: look up usage of the command `b2analysis-create`

2. Data provision

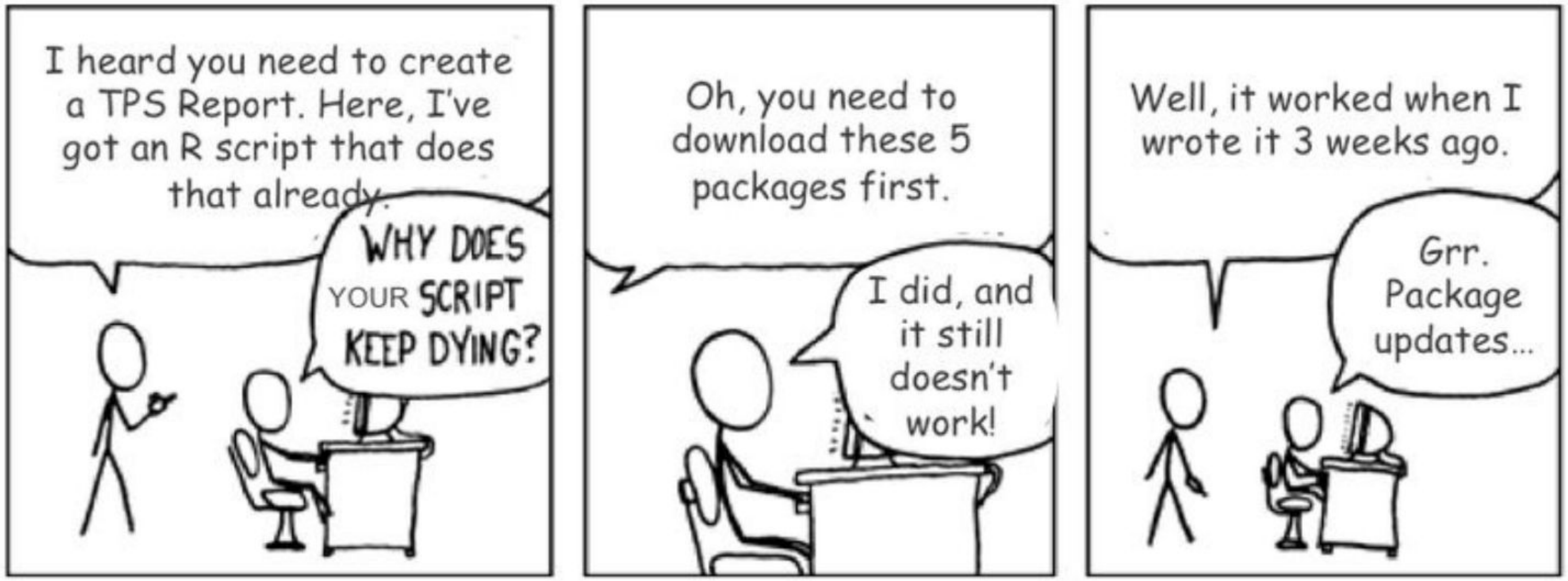
This is harder to formalize, and will differ slightly depending on each case.

Essentially, the analyst must make it possible for a future analyst to download/produce the same data/MC and they did, with the same conditions.

This must be coordinated with the experiment at large, who e.g. maintain the conditions database.

3. Environment manager (**Conda** example)

Why use an environment manager?


I heard you need to create a TPS Report. Here, I've got an R script that does that already.
WHY DOES YOUR SCRIPT KEEP DYING?
Oh, you need to download these 5 packages first.
I did, and it still doesn't work!
Well, it worked when I wrote it 3 weeks ago.
Grr. Package updates...

3. Environment manager (**Conda** example)

What is an environment?

A list of software, packages, tools etc and their version numbers.

3. Environment manager (**Conda** example)

What is an environment?

A list of software, packages, tools etc and their version numbers.

What is an environment manager?

A tool which helps you find and install packages, and switch between isolated environments. This helps stop ‘dependency hell’ and incompatibilities.

3. Environment manager (**Conda** example)

What is an environment?

A list of software, packages, tools etc and their version numbers.

What is an environment manager?

A tool which helps you find and install packages, and switch between isolated environments. This helps stop ‘dependency hell’ and incompatibilities.

Environment managers help us do AP because they let other scientists replicate our exact run-time environments for analysis programmes.

Open a terminal!

The next slides will be more interesting

Installation:
Detailed instructions at: https://conda.io/projects/conda/en/latest/user-guide/install/index.html

Let's test out an installation of Miniconda (smaller footprint) on KEKCC

```
$ mkdir ap_test
$ cd ap_test
```

Installation:

Detailed instructions at: https://conda.io/projects/conda/en/latest/user-guide/install/index.html

Let’s test out an installation of Miniconda (smaller footprint) on KEKCC

```
$ mkdir ap_test
$ cd ap_test
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
```

Installation:
Detailed instructions at: https://conda.io/projects/conda/en/latest/user-guide/install/index.html

Let’s test out an installation of Miniconda (smaller footprint) on KEKCC

```
$ mkdir ap_test
$ cd ap_test
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ chmod u+x Miniconda3-latest-Linux-x86_64.sh
```

Installation:
Detailed instructions at: https://conda.io/projects/conda/en/latest/user-guide/install/index.html

Let's test out an installation of Miniconda (smaller footprint) on KEKCC

```
$ mkdir ap_test
$ cd ap_test
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ chmod u+x Miniconda3-latest-Linux-x86_64.sh
$ ./Miniconda3-latest-Linux-x86_64.sh
```

Installation:
Detailed instructions at: https://conda.io/projects/conda/en/latest/user-guide/install/index.html

Let's test out an installation of Miniconda (smaller footprint) on KEKCC

```
$ mkdir ap_test
$ cd ap_test
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ chmod u+x Miniconda3-latest-Linux-x86_64.sh
$ ./Miniconda3-latest-Linux-x86_64.sh
```

~~Read through license terms~~ press q and type "yes".
Select default install location.
Type "yes" when prompted to initialize conda in .bashrc

We now have a working, minimal, installation of Anaconda.
We can now create separate environment for different analyses, without fear of breaking anything.

Let’s see what conda can do by making a simple test environment. We’ll use this environment for the rest of the talk.

Let’s see what conda can do by making a simple test environment. We’ll use this environment for the rest of the talk.

Let’s see how to write an “**`environment.yml`**” file.

Let’s see what conda can do by making a simple test environment. We’ll use this environment for the rest of the talk.

Let’s see how to write an “**`environment.yml`**” file.

Conda expects to find a file with this name, and can **deploy the environment for you automatically.**

Let's see what conda can do by making a simple test environment. We'll use this environment for the rest of the talk.

Let's see how to write an "**`environment.yml`**" file.

Conda expects to find a file with this name, and can **deploy the environment for you automatically.**

These are **human readable specifications** of the software/python packages you want installed.

Let’s see what conda can do by making a simple test environment. We’ll use this environment for the rest of the talk.

Let’s see how to write an “**`environment.yml`**” file.

Conda expects to find a file with this name, and can **deploy the environment for you automatically.**

These are **human readable specifications** of the software/python packages you want installed.

They’re **easily shareable with collaborators** so perfect for AP!

```
$ nano environment.yml
$ nano environment.yml
```

It uses the YAML markup language to specify the following:

1. A name for the environment, which helps us see clearly which environment we're currently working in from the terminal

   ```
   $ name: myb2env
   ```

```
$ nano environment.yml
```

It uses the YAML markup language to specify the following:

1. A name for the environment, which helps us see clearly which environment we're currently working in from the terminal

   ```
   $ name: myb2env
   ```

2. A list of "channels", which are semi-official repositories of software which conda will search to find our packages. In HEP lots of packages are in a non-default channel called "conda-forge".

   ```
   $ channels:
   $   - defaults
   $   - conda-forge
   $   - bioconda
   ```

Needed to install workflow manager "snakemake", see later.

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

The syntax for version specification is the following:

```
    -package_name = major.minor.patch           Use this exact version.
    -package_name = major.minor                 Use latest patch of this version
    -package_name = major                       Use latest minor version.
    -package_name >= major.minor.patch  etc     Use most recent compatible
                                                version later or same as this.
    -package_name                               Use latest compatible version
```

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

The syntax for version specification is the following:

```
    -package_name = major.minor.patch          Use this exact version.
    -package_name = major.minor                Use latest patch of this version
    -package_name = major                      Use latest minor version.
    -package_name >= major.minor.patch  etc    Use most recent compatible
                                               version later or same as this.
    -package_name                              Use latest compatible version
```

Conda will try and “resolve” and environment which satisfies all your requirements. Note that if you are too strict with exact version numbers, it may be impossible to create the environment. For this reason you should minimize non-essential version specifications.

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

The syntax for version specification is the following:

`-package_name = major.minor.patch`	`Use this exact version.`
`-package_name = major.minor`	`Use latest patch of this version`
`-package_name = major`	`Use latest minor version.`
`-package_name >= major.minor.patch  etc`	`Use most recent compatible version later or same as this.`
`-package_name`	`Use latest compatible version`

Conda will try and “resolve” and environment which satisfies all your requirements. Note that if you are too strict with exact version numbers, it may be impossible to create the environment. For this reason you should minimize non-essential version specifications.

When you finish your analysis, you can “print” the exact version of the environment file to ensure 100% reproducibility.

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

For this example we will use python3. Let’s specify a modern (but not too modern) version, 3.6.
Let’s install the infinitely useful package “root_pandas”, which also installs a copy of ROOT into your environment.

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

For this example we will use python3. Let’s specify a modern (but not too modern) version, 3.6.
Let’s install the infinitely useful package “root_pandas”, which also installs a copy of ROOT into your environment.

Note that these packages will share some dependencies, for example, numpy.

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

For this example we will use python3. Let's specify a modern (but not too modern) version, 3.6.
Let's install the infinitely useful package "root_pandas", which also installs a copy of ROOT into your environment.

Note that these packages will share some dependencies, for example, numpy.

The beauty of conda is that we only need to give the minimum level of detail needed, and everything else is calculated for us. In this example, numpy and matplotlib will be downloaded for us.

3. A list of dependencies. This is where we list the software, python version, and packages we want for our analyses.

For this example we will use python3. Let's specify a modern (but not too modern) version, 3.6.
Let's install the infinitely useful package "root_pandas", which also installs a copy of ROOT into your environment.

Note that these packages will share some dependencies, for example, numpy.

The beauty of conda is that we only need to give the minimum level of detail needed, and everything else is calculated for us. In this example, numpy and matplotlib will be downloaded for us.

```
$ dependencies:
$     - python=3.6
$     - ipython
$     - root_pandas
$     - pandas
$     - seaborn
```

The final product! Note, we can add packages to the file as we need to, then recreate the environment.

```
-- environment.yml ---
name: myb2env

channels:
      - defaults
      - conda-forge
      - bioconda

dependencies:
      - python=3.6
      - root_pandas
      - snakemake
```

The moment of truth: let’s use our environment.yml file to create our environment:

```
$ conda env create
```

The moment of truth: let's use our environment.yml file to create our environment:

```
$ conda env create
```

Now let's wait. If it takes a computer this long to solve the dependencies, think how impossible it is without a tool…

The moment of truth: let’s use our environment.yml file to create our environment:

```
$ conda env create
```

Now let’s wait. If it takes a computer this long to solve the dependencies, think how impossible it is without a tool…

Done?

Let’s fire up the environment:

```
$ conda activate myb2env
```

The moment of truth: let's use our environment.yml file to create our environment:

```
$ conda env create
```

Now let's wait. If it takes a computer this long to solve the dependencies, think how impossible it is without a tool…

Done?

Let's fire up the environment:

$ conda activate myb2env

If we need to switch to another/our base environment: **$ conda deactivate**

Exercise: look up the conda command to list all available environments

The important bit! Making the environment *completely* reproducible!

The important bit! Making the environment ***completely*** ***reproducible!***

We need to create an environment.yml file to send to our scientist friends so they can reproduce our environment. The most-recent solved environment may change in the future. We want to “freeze” the exact environment we have now:

The important bit! Making the environment *completely* reproducible!

We need to create an environment.yml file to send to our scientist friends so they can reproduce our environment. The most-recent solved environment may change in the future. We want to “freeze” the exact environment we have now:

```
$ conda env export > my_environment.yml
```

The important bit! Making the environment *completely* reproducible!

We need to create an environment.yml file to send to our scientist friends so they can reproduce our environment. The most-recent solved environment may change in the future. We want to “freeze” the exact environment we have now:

```
$ conda env export > my_environment.yml
```

I’ve given it a new name so it doesn’t overwrite our environment.yml

This file can be shared and used to create a complete replica of your working environment

Exercise: look at the output of the previous command - how is it different from our previous dependencies.yml file?

Now we have a reproducible environment!

Next step: workflow manager. Let's use **snakemake**

3. Workflow manager (**Snakemake** example)

What is a workflow manager? A software tool which allows the analyst to describe a series of steps, input files, output files, and configurations, which constitute their analysis.

3. Workflow manager (**Snakemake** example)

What is a workflow manager? A software tool which allows the analyst to describe a series of steps, input files, output files, and configurations, which constitute their analysis.

Why is it useful? It serves as excellent documentation. It allows you to automatically re-run your analysis ~~if~~ when you need to.

3. Workflow manager (**Snakemake** example)

What is a workflow manager? A software tool which allows the analyst to describe a series of steps, input files, output files, and configurations, which constitute their analysis.

Why is it useful? It serves as excellent documentation. It allows you to automatically re-run your analysis ~~if~~ when you need to.

Using a workflow manager (snakemake in my case) has improved my quality of life. **The tools exist to automate our workflows - let's use them!**

Snakemake is a simple tool which allows you to build a workflow using **rules**

Snakemake is a simple tool which allows you to build a workflow using **rules**

Each rule has:

1. **`input`**/s (optional)
2. **`output`**/s
3. A **`shell`** command which takes the input and produces the output. We will use exclusively python scripts with an `` `if __name__ == "__main__":` `` clause to turn them into command line tools. But they don't need to be python! E.g. "hadd {output} {input}".

Example rule

```
rule my_rule_name:
     input: "my_input.txt"
     output: "output_file.root"
     shell: "python my_script.py --input {input} --save_to {output}
```

Snakemake is a simple tool which allows you to build a workflow using **rules**

Each rule has:

1. **`input`**/s (optional)
2. **`output`**/s
3. A **`shell`** command which takes the input and produces the output. We will use exclusively python scripts with an `` `if __name__ == "__main__":` `` clause to turn them into command line tools. But they don't need to be python! E.g. "hadd {output} {input}".

Example rule

```
rule my_rule_name:
     input: "my_input.txt"
     output: "output_file.root"
     shell: "python my_script.py --input {input} --save_to {output}
```

Rules must be written in a text file called "**`Snakefile`**"

Snakemake example

Snakemake is a simple tool which allows you to build a workflow using **rules**

Each rule has:

1. **`input`**/s (optional)
2. **`output`**/s
3. A **`shell`** command which takes the input and produces the output. We will use exclusively python scripts with an `` `if __name__ == "__main__":` `` clause to turn them into command line tools. But they don't need to be python! E.g. "hadd {output} {input}".

Example rule

```
rule my_rule_name:
     input: "my_input.txt"
     output: "output_file.root"
     shell: "python my_script.py --input {input} --save_to {output}
```

Rules must be written in a text file called "**`Snakefile`**"

Snakemake allows one to **automate one's analysis**, and easily extend, re-run, and modify steps with minimal overhead.

Snakemake is a simple tool which allows you to build a workflow using **rules**

Each rule has:

1. **`input`**/s (optional)
2. **`output`**/s
3. A **`shell`** command which takes the input and produces the output. We will use exclusively python scripts with an `` `if __name__ == "__main__":` `` clause to turn them into command line tools. But they don't need to be python! E.g. "hadd {output} {input}".

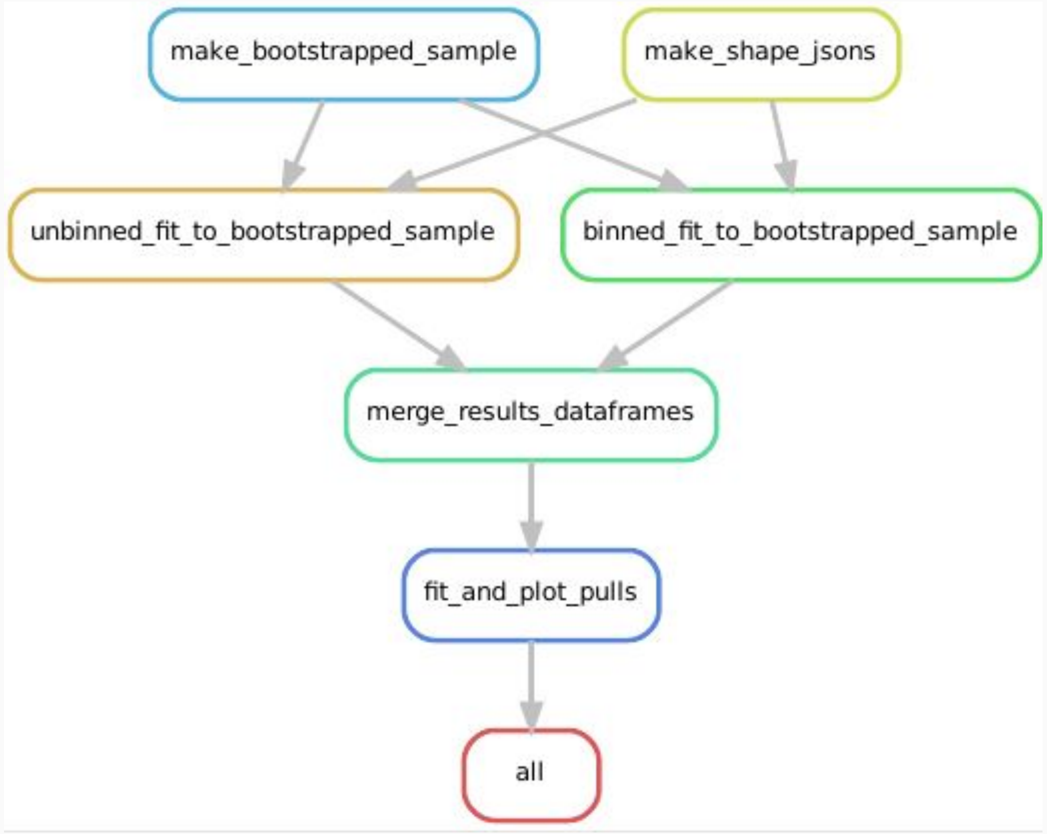
Example rule

```
rule my_rule_name:
     input: "my_input.txt"
     output: "output_file.root"
     shell: "python my_script.py --input {input} --save_to {output}
```

Rules must be written in a text file called "**`Snakefile`**"

Snakemake allows one to **automate one's analysis**, and easily extend, re-run, and modify steps with minimal overhead.

Snakemake isn't dependent on programming language at all. All you need is a shell command which produces the correct output. Snakemake will monitor your filesystem and wait for files to appear before sending off the next step. We will use python for these examples.



Example from real-life.

Snakemake can build a visualisation of the steps in your analysis.

This helps other people (and you!) understand your workflow.

This example creates toy datasets, and template PDFs as the first stage (no input required).

It then performs fits to each toy dataset and then calculates and fits pull distributions.

Before we jump into making **`rule`**s for our Snakemake workflow, we need some **scripts**, and some **data**!

Before we jump into making **`rule`**s for our Snakemake workflow, we need some **scripts**, and some **data**!

~~Un~~fortunately we don’t have enough time to write some dummy analysis code together, so please download my pre-made git repo:

```
$ git clone ssh://git@stash.desy.de:7999/~murphyco/ap_startertalk.git
```

Before we jump into making **`rule`**s for our Snakemake workflow, we need some **scripts**, and some **data**!

~~Un~~fortunately we don't have enough time to write some dummy analysis code together, so please download my pre-made git repo:

```
$ git clone ssh://git@stash.desy.de:7999/~murphyco/ap_startertalk.git
```

We should see four files (and a README)

1. generate_mc.py
2. fit_and_plot_mc.py
3. Snakefile
4. environment.yml

Before we jump into making **`rule`**s for our Snakemake workflow, we need some **scripts**, and some **data**!

~~Un~~fortunately we don't have enough time to write some dummy analysis code together, so please download my pre-made git repo:

```
$ git clone ssh://git@stash.desy.de:7999/~murphyco/ap_startertalk.git
```

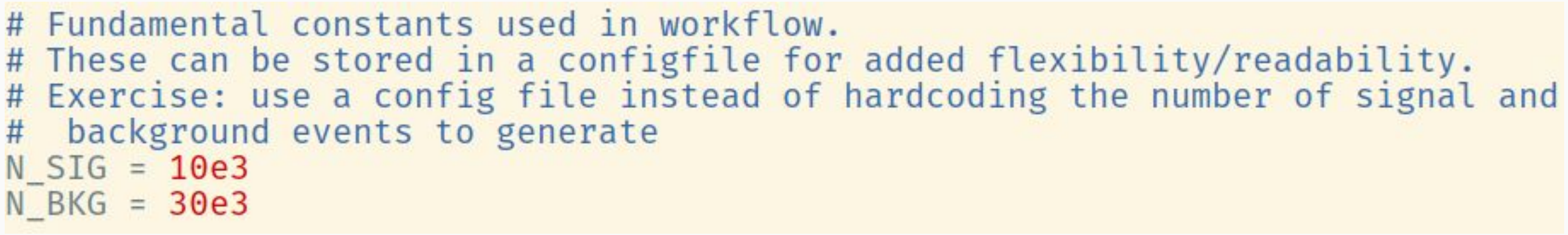
We should see four files (and a README)
1. generate_mc.py
2. fit_and_plot_mc.py
3. Snakefile
4. environment.yml

1 and 2 are simple scripts which generate some toy MC from a RooFit PDF and fit back to the distribution. We will use this trivial workflow as a way of understanding Snakemake.

Please open up "Snakefile" in your editor of choice.

Snakemake is built upon python, so normal python3 code can be used to extend it.

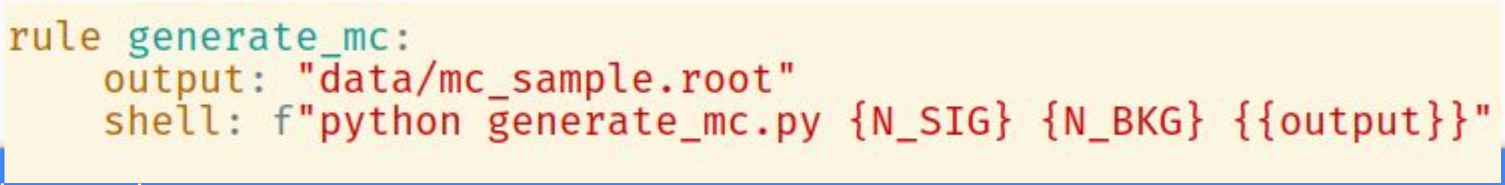
At the top of the file we define some constants needed as input to the generate_mc.py script



```
# Fundamental constants used in workflow.
# These can be stored in a configfile for added flexibility/readability.
# Exercise: use a config file instead of hardcoding the number of signal and
#  background events to generate
N_SIG = 10e3
N_BKG = 30e3
```

Exercise: look up “Configuration” on the Snakemake documentation. How could we achieve the same thing with a config file?

Fundamentally, Snakemake works by matching wildcards in **`input`** fields to those in **`output`** fields.

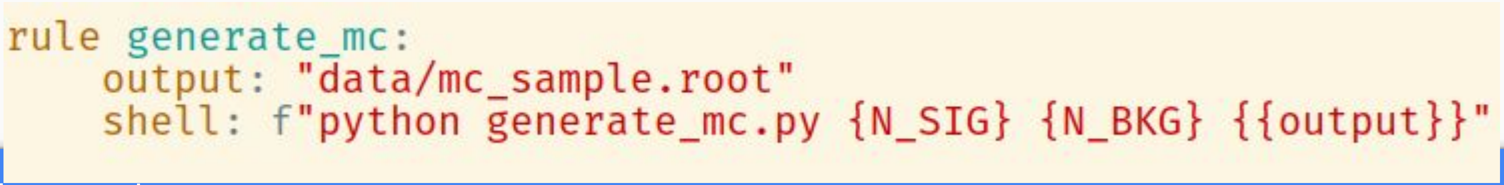


```
rule generate_mc:
    output: "data/mc_sample.root"
    shell: f"python generate_mc.py {N_SIG} {N_BKG} {{output}}"
```

Snakemake example

Fundamentally, Snakemake works by matching wildcards in **`input`** fields to those in **`output`** fields.

Wildcards are denoted with surrounding curly braces, **`{my_wildcard}`**

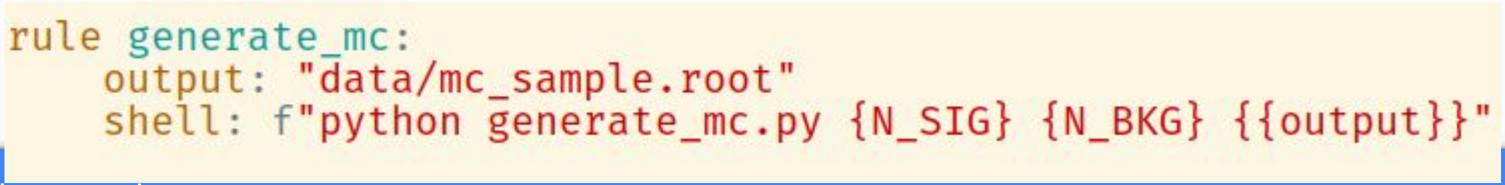


```
rule generate_mc:
    output: "data/mc_sample.root"
    shell: f"python generate_mc.py {N_SIG} {N_BKG} {{output}}"
```

Fundamentally, Snakemake works by matching wildcards in **`input`** fields to those in **`output`** fields.

Wildcards are denoted with surrounding curly braces, **`{my_wildcard}`**

Let’s define a simple **`rule`**, based on the script “generate_mc.py”.



```
rule generate_mc:
    output: "data/mc_sample.root"
    shell: f"python generate_mc.py {N_SIG} {N_BKG} {{output}}"
```

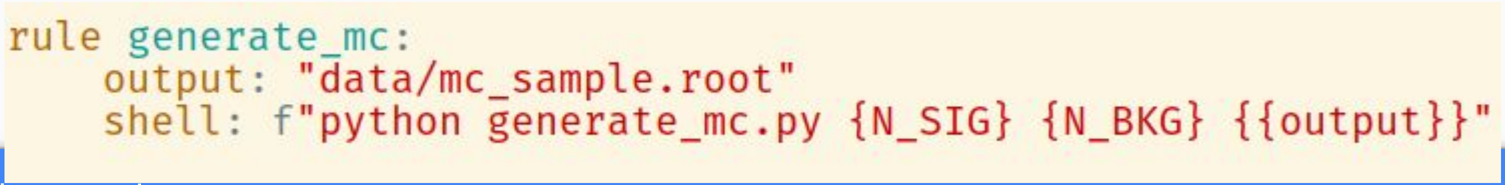
Fundamentally, Snakemake works by matching wildcards in **`input`** fields to those in **`output`** fields.

Wildcards are denoted with surrounding curly braces, **`{my_wildcard}`**

Let's define a simple **`rule`**, based on the script "generate_mc.py".

If you look in the script you will see it expects three command line arguments:

1. Signal yield
2. Background yield
3. Output directory (to save RooDataSet)



```
rule generate_mc:
    output: "data/mc_sample.root"
    shell: f"python generate_mc.py {N_SIG} {N_BKG} {{output}}"
```

Snakemake example

Fundamentally, Snakemake works by matching wildcards in **`input`** fields to those in **`output`** fields.

Wildcards are denoted with surrounding curly braces, **`{my_wildcard}`**

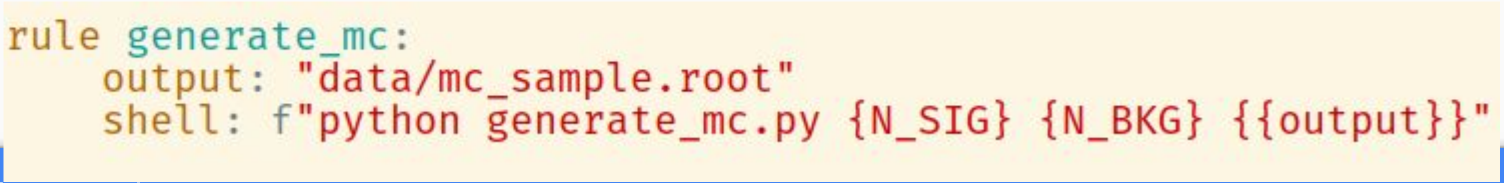
Let's define a simple **`rule`**, based on the script "generate_mc.py".

If you look in the script you will see it expects three command line arguments:

1. Signal yield
2. Background yield
3. Output directory (to save RooDataSet)

We don't need any input files. We just run the script and generate output.

So, when we define our rule, we do not need an **`input`** field



```
rule generate_mc:
    output: "data/mc_sample.root"
    shell: f"python generate_mc.py {N_SIG} {N_BKG} {{output}}"
```

Snakemake example

Fundamentally, Snakemake works by matching wildcards in **`input`** fields to those in **`output`** fields.

Wildcards are denoted with surrounding curly braces, **`{my_wildcard}`**

Let's define a simple **`rule`**, based on the script "generate_mc.py".

If you look in the script you will see it expects three command line arguments:

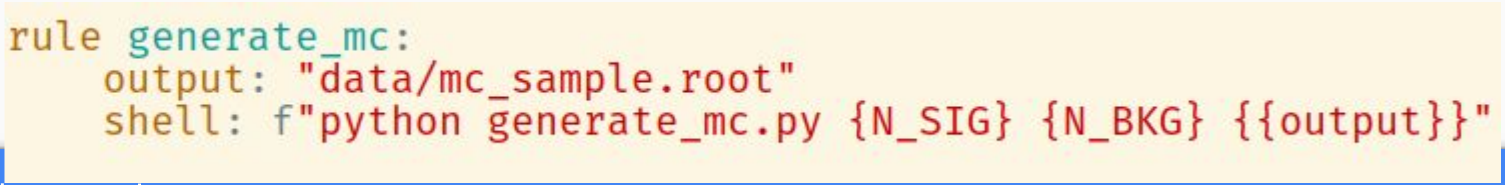
1. Signal yield
2. Background yield
3. Output directory (to save RooDataSet)

We don't need any input files. We just run the script and generate output.

So, when we define our rule, we do not need an **`input`** field

Here we use an f-string to format in the numbers for N_SIG and N_BKG, we don't expect these to change.

This is just a slightly more flexible way of writing:
`shell: "python generate_mc.py 10000 30000 {output}`

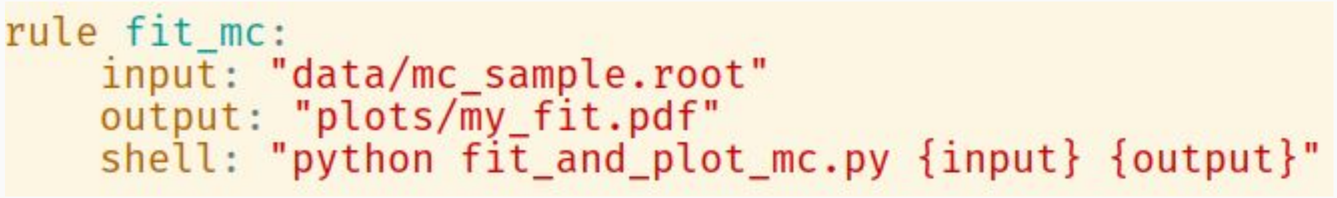


```
rule generate_mc:
    output: "data/mc_sample.root"
    shell: f"python generate_mc.py {N_SIG} {N_BKG} {{output}}"
```

Snakemake example

Let’s add a second rule, for the next distinct stage in our workflow: fitting the MC and making a plot.

As always, let’s first think conceptually about the input and output.



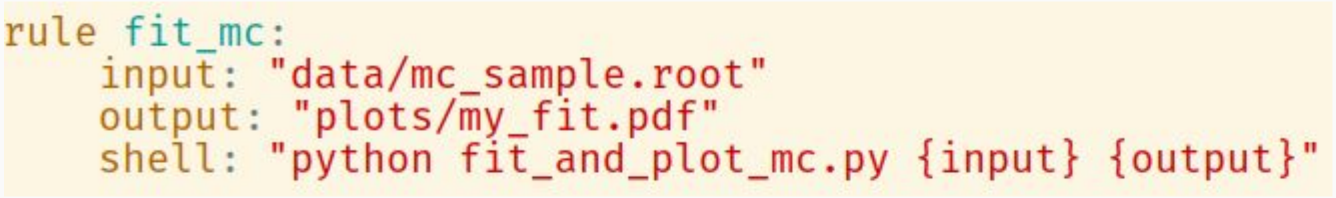
```
rule fit_mc:
    input: "data/mc_sample.root"
    output: "plots/my_fit.pdf"
    shell: "python fit_and_plot_mc.py {input} {output}"
```

Let’s add a second rule, for the next distinct stage in our workflow: fitting the MC and making a plot.

As always, let’s first think conceptually about the input and output.

The script fit_and_plot_mc.py expects two arguments.
1. Path to RooDataSet .root file
2. Path to save plot

Clearly we can see that the first argument corresponds to an “**`input`**”, and the second argument corresponds to “**`output`**”.



```
rule fit_mc:
    input: "data/mc_sample.root"
    output: "plots/my_fit.pdf"
    shell: "python fit_and_plot_mc.py {input} {output}"
```

Let’s add a second rule, for the next distinct stage in our workflow: fitting the MC and making a plot.

As always, let’s first think conceptually about the input and output.

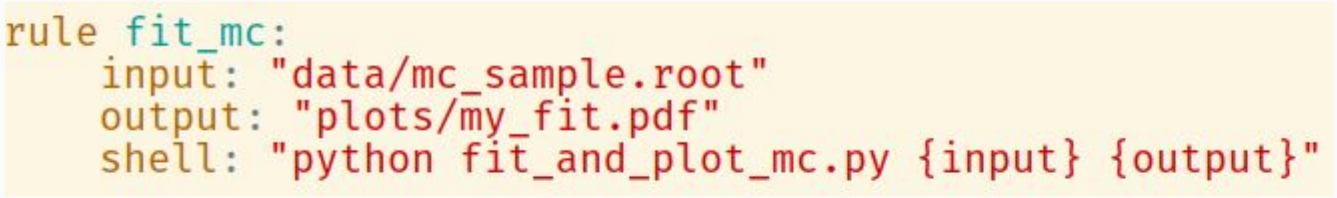
The script fit_and_plot_mc.py expects two arguments.

1. Path to RooDataSet .root file
2. Path to save plot

Clearly we can see that the first argument corresponds to an “**`input`**”, and the second argument corresponds to “**`output`**”.

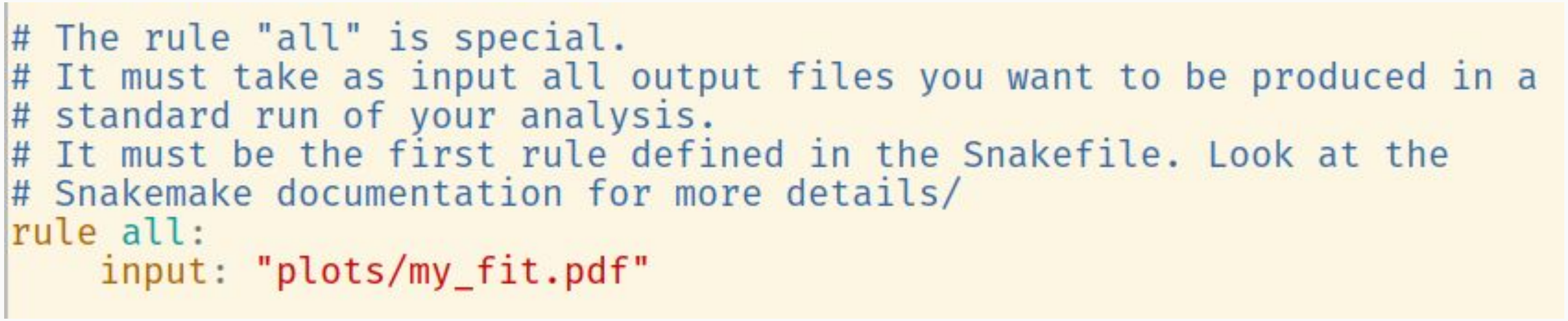
In this simple example, we can see that the “input” of this rule, corresponds to the “output” of the last rule. This is our workflow in action.

As before, the **`shell`** field is simply invoking our python script with the input and output command line arguments.



```
rule fit_mc:
    input: "data/mc_sample.root"
    output: "plots/my_fit.pdf"
    shell: "python fit_and_plot_mc.py {input} {output}"
```

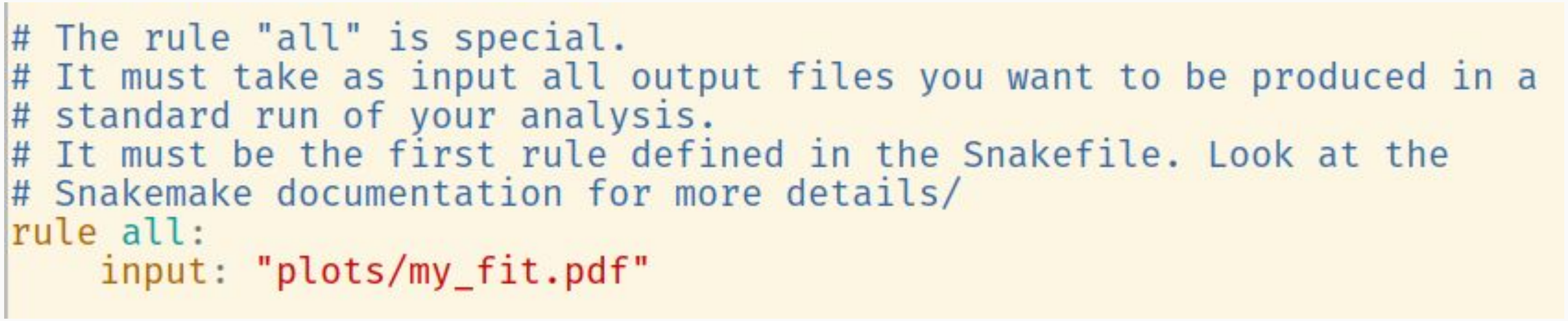
The final stage is to force snakemake to run the whole procedure by creating a rule whose **`input`** is all of our desired output.



```
# The rule "all" is special.
# It must take as input all output files you want to be produced in a
# standard run of your analysis.
# It must be the first rule defined in the Snakefile. Look at the
# Snakemake documentation for more details/
rule all:
    input: "plots/my_fit.pdf"
```

The final stage is to force snakemake to run the whole procedure by creating a rule whose **`input`** is all of our desired output.

This “completes the circuit” of our workflow.



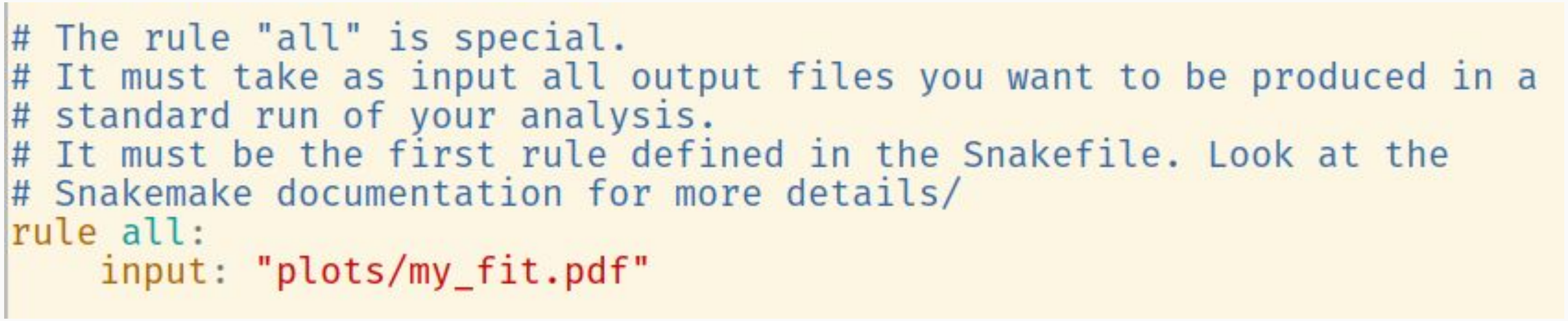
```
# The rule "all" is special.
# It must take as input all output files you want to be produced in a
# standard run of your analysis.
# It must be the first rule defined in the Snakefile. Look at the
# Snakemake documentation for more details/
rule all:
    input: "plots/my_fit.pdf"
```

The final stage is to force snakemake to run the whole procedure by creating a rule whose **`input`** is all of our desired output.

This “completes the circuit” of our workflow.

This rule is given the special name “**`all`**” and must be placed at the top of our Snakefile.

In our case, we want the plot of our fit.



```
# The rule "all" is special.
# It must take as input all output files you want to be produced in a
# standard run of your analysis.
# It must be the first rule defined in the Snakefile. Look at the
# Snakemake documentation for more details/
rule all:
    input: "plots/my_fit.pdf"
```

Try it out!

```
$ snakemake
```

This barely scratches the surface of what snakemake can do.

For even more complicated workflows, the tool “Luigi” can be used.

Note
It is possible to use KEKCC batch computing with snakemake.

Because snakemake creates a graph of processes, it knows what can be run in parallel automatically.

```
$ snakemake --cluster bsub -j 999
```

Max number of consecutive jobs

Now we see the true power of combining git, environment managers, and a workflow manager.

In only three commands, one can reproduce another user’s analysis.

```
$ git clone ssh://git@stash.desy.de:7999/~murphyco/ap_startertalk.git
$ conda env create
$ snakemake
```

Let’s write reproducible analyses!

Thank you for listening!

Bonus snakemake exercises

1. Look up the “`--dag`” and “`--rulegraph`” options in the snakemake documentation. Visualise the rulegraph for the test workflow in these slides

2. How can wildcards be used in the “`shell`” field of a rule? (wildcards object)

3. What does the “expand” keyword do?

4. Let’s say you have a preprocessing step which is intermediary, but produces large files that you want to delete afterwards. How do you mark those files as temporary in snakemake? (“temp” keyword)

5. Look up how conda environments can be integrated into specific rules