Pipeline development

This exercise assumes the following:
 * A native Linux environment / terminal
 * A functional micromamba installation
 * A few bacterial assemblies
 * All files are stored within the
`gebt` user files. Feel free to change locations according to own needs!

Authors

These exercises where authored and tested by Povilas Matusevicius and Kasper Thystrup Karstensen

Our very first pipeline

A brief disclaimer

Scripting and pipelines is a bit like cooking. You set one or more recipes (**scripts**), which the chef (**bash**) then follows. The ingredients (**files** and **file paths**) are usually listed at the top, and then the cooking utensils (**commands** and **software**-calls) are introduced later once you need these.
Now contrary to cooking,
scripts are interpreted literally down to the very last comma, so the chef will not use His/Her experience to guide the process. This means that any errors in the recipe will be followed and missing steps will be let out.
E.g. If the chef is supposed to put the turkey in the oven and crank the temperature up to a 200 Celsius for 20 minutes, this will neither ensure that the turkey will be taken out of the oven, nor that the oven will be turned off, unless it is stated in the recipe.

This can (and will) lead to a lot of frustration, as its hard to ensure that all details are correct.

This is quite alright. In fact, its impossible to learn coding without a lot of trials and errors!!

Pipelines starts with a script...

In order to make a pipeline we must provide at least one *script*, which can be followed, thus in this first exercise you will be guided through the process of generating a **bash** script.

The **bash** script is a simple text file which contains one or more lines of code, which will be executed in chronological order.

A very important aspect of making bash script is to ensure that the code is clean and concise for the future reader and developer. Which assuming long job contracts, very often could be yourself. One way of helping yourself making the code more reader friendly is to assign variables for containing **file paths** and **files**.

Setting up a characterization pipeline

Let’s make a script which utilizes some of the characterization tools which were introduced during previous days.

To demonstrate we will start out setting up ResFinder, and AMRFinderPlus

Prerequisites

For this exercise we will use following **files** and **file paths**.

- Sample: */home/gebt/BTG/Precomputed_Data/RawAssembliesSpades/careful_mode/Ec001.fasta*
- Path for output folder: */home/gebt/BTG/Day8_pipelines/output*
- Path to ResFinder database: */home/gebt/BTG/Day8_pipelines/dbs/resfinder_db*

Executing commands from different environments

A drawback of using conda/mamba environments, is that not all software can run in the same environments. When running bash scripts, the shell doesn't always know how to invoke the conda/mamba commands. With `mamba` though, there is a hack that can be utilized: Mamba allow for cherry picking commands from different environments with using `micromamba run`. This way environments are not required to be loaded. So in order to run e.g. AMRFinderPlus from within any (or no) environment, the following command can be used in the bash script.

```
micromamba run -n BTG_resistance amrfinder
```

Tasks: Variables

First, we will make variables which can make writing and interpreting the script easier.

1. Make a new folder in the `BTG` directory, call it `Day8_pipelines`. The full path to the folder should be `/home/gebt/BTG/Day8_pipelines`.
2. Navigate into the folder using `cd`.
3. Make a new file called `finders_pipeline.sh` within the folder, e.g. using `nano finders_pipeline.sh`
4. Copy and paste the code chunk below, and *then* fill out the missing information for the `rfdb_path`:

```
#!/bin/bash

# Define locations for input and output
sample="/home/gebt/BTG/Precomputed_Data/RawAssembliesSpades/careful_mode/Ec001.fasta"
rfdb_path=""
output_folder="/home/gebt/BTG/Day8_pipelines/output"
```

Task: Execution

The very first line of this script is called a **shebang**, and it's comparable to file extensions in Windows. In Windows. If you take a word document and rename its file extension form `.docx` to `.exe` , Windows will assume that it's a program that it will try to run once you double click it. However, it will fail, as it's in fact a document file and not a program.

In Linux, the shebang works by telling the **shell** which program is required to run the script, if no shebang line is added, you would have to tell the **shell** which program is used to execute the script, which could be helpful for non-programming users.

By using a shebang, scripts are easy to execute, you just have to point to the file with a preceding dot and forward slash (`./`)

1. Can you name the program which is used to execute this script?
2. Try to execute the script using `./finders_pipeline.sh`. Did any errors show up?

Right now the files permission is to read and write. However, as a safety mechanism in Linux files can't be executed unless you change their permissions to do so.

3. Allow execution of the file by running `chmod u+x finders_pipeline.sh`
    a. Explanation: ***u*** means for current user only, ***+*** means add permissions, ***x*** means execution permission.
4. Try to execute the script again. Did any errors show up this time?

Task: Telling the chef what to do...

Now where the ingredients list have been set up, lets start making the script usable.

Currently, we are pointing to a output **file path** which does not exist, like telling the chef to drop the dishes on an imaginary table, not very helpful!

A great start is then to ensure that the folder is created early on.

1. In the `finders_pipeline.sh` file, directly after the variables section, add the following lines:

```
# Create output folders
rf_out=$output_folder/rf
af_out=$output_folder/af
```

```
mkdir -p $rf_out
mkdir -p $af_out
```

♟️ A word about the quoted variables. Quotes on “$variables” are not required but recommended. Say that we have a `$output` variable which points to the file **.../SRR27240825.fa**. If you wrote `$output_results.txt` the script would look for a `$output_results` variable instead of `$output`. In order to prevent this behavior just add quotes: `"$output"_results.txt`

Now, its time to add some lines of code which executes some of the programs which we want to use for characterization, lets start with ResFinder.

2. Add the following lines to your script.

```
# Start characterization with resfinder
micromamba run -n BTG_resistance run_resfinder.py -ifa $sample -db_res $rfdb_path -o $rf_out -acq
```

3. Time to execute the script and see if things work. Save changes, and in another terminal run the script.

```
/home/gebt/BTG/Day8_pipelines/finders_pipeline.sh
```

💡 Did you make sure to fill out the database paths for ResFinder?

Running AMRFinderPlus

By now we are well on our way of setting up a small pipeline for characterizing isolates. We are not satisfied by only using ResFinder, so lets try to use AMRFinderPlus as well.

By calling the AMRFinder help page, we can inspect its usage. Here we have only included details for the two arguments we need:

❓ USAGE: amrfinder [--protein PROT_FASTA] [--nucleotide NUC_FASTA] [--gff GFF_FILE] [--database DATABASE_DIR] [--update] [--ident_min MIN_IDENT] [--coverage_min MIN_COV] [--organism ORGANISM] [--translation_table TRANSLATION_TABLE] [--plus] [--report_common] [--point_mut_all POINT_MUT_ALL_FILE] [--blast_bin BLAST_DIR] [--parm PARM] [--output OUTPUT_FILE] [--quiet] [--gpipe] [--threads THREADS] [--debug]

-n NUC_FASTA, --nucleotide NUC_FASTA | Nucleotide FASTA file to search

-o OUTPUT_FILE, --output OUTPUT_FILE | Write output to OUTPUT_FILE instead of STDOUT

1. Add the following AMRfinder call to the `finders_pipeline.sh` below

```
micromamba run -n BTG_resistance amrfinder -n $sample -o $af_out/amrfinder_results.txt
```

2. Ready to take the pipeline for a spin? Save and exit nano, then... Let’s GO

```
# Run this from terminal
/home/gebt/BTG/Day8_pipelines/finders_pipeline.sh
```

▼ Solution - Please minimize until you are done!

```
#!/bin/bash

# Define locations for input and output
sample="/home/gebt/BTG/Precomputed_Data/RawAssembliesSpades/careful_mode/Ec001.fasta"
```

```
rfdb_path="/home/gebt/BTG/Day8_pipelines/dbs/resfinder_db"
output_folder="/home/gebt/BTG/Day8_pipelines/output"

# Create output folders
rf_out=$output_folder/rf
af_out=$output_folder/af
mkdir -p $rf_out
mkdir -p $af_out

# Start characterization with finders
micromamba run -n BTG_resistance run_resfinder.py -ifa $sample -db_res $rfdb_path -o $rf_out -acq
micromamba run -n BTG_resistance amrfinder -n $sample -o $af_out/amrfinder_results.txt
```

Wrap up

Congratulations, you have made your very first pipeline. Provided you didn't introduce any errors, it should run the same way each time you execute the script. This is really useful for reproducibility and to semi-automate your own workflow.

Now the script is not very useful if you have samples other than *Ec001.fasta*, as you would have to change the `sample` variable in the script every time you wanted to run it on a different sample. Don't worry, there are very small changes required to achieve this, we will take a look at this next.

Making the pipeline run on other samples

Positional arguments

One way to make the pipeline easily usable one can replace the required input with **positional arguments**. **Positional arguments** is a way to make a script look at the extra arguments written by the user, when invoking the script.

Imagine we have a small executable **bash** script called `simple.sh`, it works like this:

```
#!/bin/bash

firstVar=$1
secondVar=$2

echo "The first variable is $firstVar. The second variable is $secondVar."
```

When you execute it:

```
./simple.sh Fish ImSecond
The first variable is Fish. The second variable is ImSecond.
```

1. Copy the `finders_pipeline.sh` script into a new file called `finders_positional_pipe.sh` using:

```
cp finders_pipeline.sh finders_positional_pipe.sh
```

2. Open the new file (`finders_positional_pipe.sh`) e.g. with nano.
2. Change the variable definitions so that `sample=` takes the first positional argument (`$1`) and the `output_folder=` takes the second positional argument (`$2`)
3. Save changes
4. In a new terminal execute the script providing the following **file** (a new file!) and **file path**, as first and second arguments respectively.
    a. Sample: */home/gebt/BTG/Precomputed_Data/RawAssembliesSpades/careful_mode/Ec001.fasta*
    b. Output_folder: */home/gebt/BTG/Day8_pipelines/output*

```
/home/gebt/BTG/Day8_pipelines/finders_positional_pipe.sh [sample] [output-folder]
```

Screening folder for samples

Another approach to enhance usability of your pipeline is to replace the input **sample file** with a **sample folder**, and then automatically screen this folder for relevant **sample files**.

Screening a folder for fasta files can be a bit out of the scope of this course, so we will provide the code necessary.

1. First copy the `finders_positional_pipe.sh` script into a new file called `finders_on_folder.sh` using `cp`
2. Open the new file (`finders_on_folder.sh`) e.g. with nano
3. Rename the `sample` variable to `sample_dir`. Remember to leave the remaining `$sample` variables in the remainder of the script.
4. Add the following lines to the script right after the variables and `mkdir` commands:

```
# Screen the sample_dir for fasta files
files=$(find "$sample_dir" -maxdepth 1 -type f -name "*.fasta" | sort)
```

- Explanation
  - -maxdepth 1 | parameter limits the search to exclude sub folders.
  - -type f | limits search to only files and not the folders
  - -name | defines name of the file or folder that has to be find
  - sort | A command which sorts all the output, in this instance from the `find` command

▼ Solution - Inspect after finishing step 4!

```
#!/bin/bash

# Assign values to the variables
sample_dir=$1
rfdb_path="/home/gebt/BTG/Day8_pipelines/dbs/resfinder_db"
output_folder=$2

# Create output folders
mkdir -p "$output_folder"/rf
mkdir -p "$output_folder"/af

# Screen the sample_dir for fasta files
files=$(find "$sample_dir" -maxdepth 1 -type f -name "*.fasta" | sort)

# Start characterization with finders
micromamba run -n BTG_resistance run_resfinder.py -ifa $sample --acquired -db_res $rfdb_path -o "$output
micromamba run -n BTG_resistance amrfinder -n $sample -o "$output_folder"/af/amrfinder_results.txt
```

5. The script does not yet work as the `$sample` variable is no longer defined, so lets comment out the lines which does not work. Add a `#` in front of all the finder commands so they will be ignored:

```
# Start characterization with finders
#micromamba run -n BTG_resistance run_resfinder.py -ifa $sample --acquired -db_res $rfdb_path -o "$output_f
#micromamba run -n BTG_resistance amrfinder -n $sample -o "$output_folder"/af/amrfinder_results.txt
```

6. Now its time to figure out whether the fasta file screener works or not. Add the following lines directly after the screening lines (`files=$(find ...`)

```
# Looping over each of the sample files individually
for sample in $files; do
  # Define the sample name from the file
  sample_name=$(basename "$sample" .fasta)

  # Print out helpfull message that this in fact works
  echo The sample $sample_name is located here: $sample
done
```

7. Save and exit nano, then execute the script by invoking:

```
/home/gebt/BTG/Day8_pipelines/finders_on_folder.sh /home/gebt/BTG/Precomputed_Data/RawAssembliesSpad
```

If things goes well, the script should print the file name for every single sample file located within the `sample_dir`. The second argument is provided to satisfy the `output_folder` variable which expects a second argument.

8. Reopen the script in, remove the `#` signs in front of the resfinder and amrfinder commands, and move the finder commands into the **for** loop. Remove the `echo $sample` line with the finder lines.

In its current state the script should automatically run the finders for each of the samples, however as the final output files have the same names, these will be overwritten. To prevent this, we must split these results into individual unique folders. Luckily, this was thought of in the for loop by defining the `sample_name` variable.

9. For each of the finder lines change the output arguments from the following:

   `-o "$output_folder"/Xf`
   to this:

   `-o “$output_folder”/Xf/"$sample_name"`
   Where
   `Xf` is denotes `rf` for ResFinder, and `af` for AMRFinder

10. There is one more issue, finders don’t create folder themselves, so it would give you an error. Lets tweak the output folder generation command chunk and then move them into the for loop. The commands should look like this
    - `mkdir -p $rf_out/$sample_name` for ResFinder
    - `mkdir -p $af_out/$sample_name` for AMRFinderPlus
11. Add a final victory message at the bottom of the file using `echo` e.g. `echo Jobs done!`

▼ Solution - Final script

```
#!/bin/bash

# Define locations for input and output
sample_dir=$1
rfdb_path="/home/gebt/BTG/Day8_pipelines/dbs/resfinder_db"
output_folder=$2

# Screen the sample_dir for fasta files
files=$(find "$sample_dir" -maxdepth 1 -type f -name "*.fasta" | sort)

# Looping over each of the sample files individually
for sample in $files; do
  # Define the sample name from the file
  sample_name=$(basename "$sample" .fasta)

  # Create output folders
  rf_out=$output_folder/rf/$sample_name
  af_out=$output_folder/af/$sample_name
  mkdir -p $rf_out
```

```
  mkdir -p $af_out

  # Start characterization with finders
  micromamba run -n BTG_resistance run_resfinder.py -ifa $sample -db_res $rfdb_path -o $rf_out
  micromamba run -n BTG_resistance amrfinder -n $sample --database $afdb_path -o $af_out/amrfinder_resu

done

echo Jobs done!
```

Congratulations on your first pipeline!!!