Exercise 2 - Genome download & overview – how & what software?

**Day 1**

1. **Running commands:**

As per the lecture, linux commands take this form:

**COMMAND --FULLFLAG ARGUMENT1 -b ARGUMENT2 -c**

An example:

**RibDif --genus Phaeobacter -i .99 -f**

This command is for RibDif to run on Phaeobacter with an identity threshold of 99% and use fragmented genomes. Specifically, RibDif is the command and then a **full argument flag** is specified with 2 dashes (--genus) and its corresponding argument (Phaeobacter), followed by a **short argument flag** specified with 1 dash (-i) followed by its argument (.99). The last **short flag** (-f) has no argument.

Note that most flags have a long and short version (--genus or -g), and some flags require no arguments. The syntax is very much dependent on the author and must be individually investigated.

## Conda setup:

Make sure you have your conda environment set up. Without any exceptions, everything in these exercises are installed with the conda framework. If you start googling your way into ‘sudo apt-get’ commands, call for help. **I know you will be googling and ChatGPT-ing yourself into odd solutions, but ask us before you use ‘sudo’!**

Remember that we install things with this syntax (which is not a real command, just the syntax!):

**conda install PACKAGE -c REPOSITORY**

**PACKAGE** is here the package you want, and **REPOSITORY** is what database it is in. In our case, it is almost always ‘bioconda’ and sometimes also ‘conda-forge’. The homepage/github of the software will usually tell you.

**? Can you explain to you person beside you what conda is?**

## Package installation:

Let’s start by installing the program ‘seqtk’. We do that like this:

**conda install -c bioconda seqtk**

The program seqtk is now installed in your base environment. You can check if that is in fact the case by sampling running:

**seqtk**

Which should simply return a helpful set of commands on how seqtk can be run.

**? Without any googling, what does the program seqtk appear to be used for?**

## Environments

Now we create a specific environment for the next step. This keeps programs from interfering with each other, which is particularly important when it comes to versions of software which may not work with one another. Imagine package A (v1.2) requiring >1.5 of package B, but package C must have version 0.9 of package B. It simply won’t work. Instead, we make multiple different environments containing compatible packages for each purpose, and hence accept that we often have the same package installed multiple times.

This particular environment will be called ncbiDL, because we will **D**own**L**oad from **NCBI** using packages in this environment. You can call this environment whatever you want though – makes no difference whatsoever, but it certainly helps with a logical name if you wish to remember it tomorrow. Here, we use the command ‘create’ with conda and we then use the ‘-n’ flag (short for ‘--name’) to specify the name.

**conda create -n ncbiDL**

This is then activated like this:

**conda activate ncbiDL**

**? Do we need an environment for every single package we install?**

**? Considering that you should specify software versions in your paper, can you find an additional advantage of the environment concept?**

## Finding packages

Next, we need to find a good program to download genomes from NCBI. As we often do, we will google our way forward – try by googling ‘ncbi genome download’ and see if there is a good github repository that looks like it might fit (Hint: it should be the first hit on google by a nice user called Kai Blin, and the name of our environment is no coincidence). Follow the install instructions and remember we use conda for installation. Much of bioinformatics is looking through githubs and working through installation guides.

**? Are there other ways to install this package? Should we do this or do as Mikael tells us?**

**? How would you download these genomes without this package and how long would it take?**

## Project hygiene

As soon as you have the program installed, we will do some data hygiene. **First, make a folder (with ‘mkdir’) for this part of the summer school and enter it (with ‘cd’)**.

**mkdir MYFOLDER**

**cd MYFOLDER**

Call it whatever you want, but make sure it still makes sense for you next month when you have forgotten all you did in this course (hint: ‘MYFOLDER’ is a bad name. **How about day1?**). Within that folder (**did you remember to enter it?**), you should generally consider making a folder for the raw input – a good name for this folder could be ‘input’ or ‘genomes’ or similar. In this case, the next program will actually make the folder, so **not** necessary here.

Always consider the architecture of your folders when you run a project. It easily becomes very messy which is problematic because:   
1) You, yourself, will not be able to remember the details of what you did   
2) Your collaborators, possibly coders themselves, will struggle finding out what you did.   
3) Your supervisor, likely a very poor coder, will have absolutely no idea what you did.

Lets pretend you used what you learned here to do something genius and your supervisor is just itching to publish. Such a publication will probably be your front row ticket into that fancy PhD program you have been looking for, but you just got the paper back from review and the reviewers demand details on how you did that analysis months ago, and now you simply can’t remember – Why O Why didn’t you listen to Mikael and write it down ???

Commonly, my data-heavy projects look like this, which in this case is based on a Rstudio-project. This particular project is made for statistical analysis a large set of data that Polonca and our friend Anna Dragos have produced, which I need to keep it ready and reproducible for publication. The data given to me from Polonca and Anna had to undergo substantial bioinformatic analysis before we could do proper statistics, and the commands I used for that are given in the file “Workflow and commands.txt”. You won’t get to see that one just yet, because it is exactly what you need to do on this course.

Note how there are only very few loose files (for RStudio bookkeeping), and how most others are otherwise organized into main folders, such as raw data (Data), Outputs (Figures & Tables), analysis code (Scripts) and a README that reminds me what this project is about (README\_SlovPhages.txt). Since this is for a paper, there are archive folders to keep track of prior versions.

A screenshot of a computer

Description automatically generated

**? What did you call your project and why?**

**? What do you think should go in your README file?**

## Genome download

In your case, you should not have anything in your folders yet. So, now we download. The ncbi-genome-download package is a wonderful program that makes downloading genomes very easy. Getting genomes manually from NCBI is almost impossible in large scale without it.

First, check the possible arguments of the program like this:

**ncbi-genome-download -h**

Using either ‘-h’ or ‘--help’ is standard for getting programs to tell us how they want to be used.

The following command will download some genomes of P. piscinae. Take a close look at the arguments I have specified and make sure you know what they mean. Why do I specify the ‘-F’ argument like that? What do I mean with the ‘-l’ argument? What is the ‘-p’ argument and should you use the same value?

**ncbi-genome-download --genera "Phaeobacter piscinae" -F fasta --flat-output -o genomes -l complete -p 10 bacteria**

**? could you use the short version of the flag in some of the flags?**

**? why is there a ‘naked’ argument at the end without a corresponding flag?**

## Checking the files

Now we should have a couple of files in the ‘genomes’ folder. First, check out what you actually have now with

**ls -l**

Check out what is in your newly made ‘genome/’ folder with the command:

**ls -l genomes/**

These files are gzipped, evident by their file extensions (.gz). Gzipped files, analogous to zipped files on windows, are compressed and takes up less space, but are not readable. We will gunzip these files by use of the gunzip command, which is built-in in Ubuntu.

This will be our first use of the wildcard expansion of bash, e.g. the symbol ‘\*’, which stands for ‘everything’. Run this

**gunzip genomes/\***

which basically means “gunzip everything in this folder”. Check again if the files are now unzipped fasta files with the commands above.

**? although you can go in and out of folders to run commands where you actually are, what is the advantage of staying where you are and running commands from there?**

**? what is the meaning of the -l flag for ‘ls’? Try to combine the -l flag with -t, -S, -h, -r alone and in combination and see if you can guess what they do?**

## Installing non-standard software

Next we annotate. This used to be a little tricky, since the traditional program for this, PROKKA, is written in Perl, and Perl programs do not play nice with conda, which is mainly a python manager. Instead we will use BAKTA, the successor to PROKKA. In any case, the actual program finds all the coding regions of the genome, and then tries to identify what these genes are. It is fairly normal that most genes have an unknown function.

We will have to make a new environment and do some workarounds. The program is described here https://github.com/oschwengers/bakta

**Make a new environment and activate it**. **VERY IMPORTANT - do not just keep working in your previous environment!** A good name for the environment could be ‘bakta or ‘annotation’, your choice. As you will have noticed from reading the github, BAKTA requires download of a database. We will use the **‘light’** one here – it is not as thorough as the ‘full’ one, but it will be much faster and perfectly good for our use.

Install BAKTA using **conda** as described in the github in **a new environment (remember to activate it after you create it)**.

When you download the database, run something like this:

bakta\_db download --output bakta\_db --type light

Here, the database will be saved in the folder you are currently in, in a folder called bakta\_db.

Check that the installation worked by simply asking the program for help as described previously.

**? How did we ask a program for help? Remember the flags vi discussed!**

**! If bakta gives any other output than a list of commands, ask!**

## Annotating

Now we annotate all the genomes. We used to use PROKKA, but it is way too hard to install, and has now been replaced by a program called BAKTA.

This would be a good time to make an output folder to put all our annotations - ‘annotations’ sound like a reasonable naming choice. Bakta will make a new folder for each genome within this folder if it is specified like below. Make sure to check with the description of the arguments so you are sure what each argument means. Try for one of the genomes before trying the loops.

The basic command is (for a given genome file, which here is **symbolized** as GENOME.fna, and a given number of CPUs, which here is 8). ‘GENOME’ is not the correct name, nor is the GENOME.fna a correct file - you need to point to the correct genome as well as use a proper name for the ‘outdir’ and prefix. If the genome file is GCF\_0000001.fna, then a proper name could be GCF\_0000001 and so on. Remember that your genome files are in the ‘genome/’ folder. The **‘--db bakta\_db/db-light’** part is always needed to point to the database.

The basic command:

**bakta --db bakta\_db/db-light/ --threads 8 --output annotations/NAME --prefix NAME genomes/GENOME.fna**

An example command:  
**bakta --db bakta\_db/db-light/ --threads 8 --output annotations/GCF\_000826835 --prefix GCF\_000926835 genomes/GCF\_000826835.2\_ASM82683v2\_genomic.fna**

On my machine this takes around 2 minutes with 8 cores.

**? Can you successfully annotate a single genome?**

**? why do we use the –prefix flag?**

**? How many cores should you use?**

## Loops

Loops are great for doing the same thing many times. It might be avoidable here, but if you need to do basically the same operation a million times, then you need the loops. They are very fundamental in programming, and you will not get far without them.

The format of a bash loop is this, like you might know from other languages:

**for i in LIST; do**

**SOMETHING**

**done**

This can be run in one line, a so called one-liner (note the semicolons, ‘;’):

**for i in LIST; do SOMETHING; done**

Here, the LIST is any type of list that contains something to loop over – in this case, it will be the genomes. First, try this:

**for i in genomes/\*; do echo $i; done**

As we can see, this code will first make a list of everything in the genomes/ folder thanks to the ‘\*’ wildcard symbol, and store each element of this list sequentially in the $i variable at each round of the loop. In this simple loop, the current value of $i is merely printed in each iteration of the loop.

It is generally good practice to use the full path of whichever files you work on (e.g. ‘~/genomes/GENOME1.fna’ rather than just ‘GENOME1.fna’) to ensure that you refer to the correct files, but we need only the actual genome name for naming output folders. We will expand this with a little bit of extra code to fish out only the genome name, which is now captured in the variable $j by use of the ‘basename’ function which chops of the directory and, in this case, also the specified file extension ‘.fna’:

**for i in genomes/\*; do**

**echo $i;**

**j=$(basename $i ".fna");**

**echo $j;**

**done**

or as a one-liner:

**for i in genomes/\*; do echo $i; j=$(basename $i ".fna"); echo $j; done**

Your assignment is now to modify the last one-liner to run all genomes through BAKTA. In the above loop, you have the **full path to the genome given in $i**, you have the **NAME (hint: look at the ‘basic command’) of the genome given in $j**, and you know the full BAKTA command from earlier.

**This is pretty advanced stuff and way more than one would expect from people on their first day! Don’t be afraid to play around, talk to each other and ask the teachers!**

For the more advanced users with bigger machines, the job can be run with the parallel package like this:

**ls -d genomes/\* | parallel -j CPUS ' echo {} ; echo {/.}'**

where CPUS is the number threads you want and {} is analogous to $i and {/.} is analogous to $j.

**Done for the day!**

**Day 2:**

Today, the technical instructions will be much more sparse since you now know how to both install & run programs, as well as how to do loops.

## Phage prediction

1. First we will predict phages using the program ‘phispy’. Have a look online on how to install this program (github is a good bet). As usual, make a new environment for this.
   1. **But beware!** In this case the github suggest installation by “conda install -c bioconda phispy”, but this will install version 3.7 of phispy even though the newest version is 4.2! This is because the ‘bioconda’ repository does not contain the newest version of a program called ‘libgcc-ng’. Luckily for us, the ‘conda-forge’ repository does, so we will instead do **‘conda install -c bioconda -c conda-forge phispy’**
2. If you want a nice graphic interface, you can try uploading the genomes to PHASTER at <https://phaster.ca/> . It looks to me that phispy is the best algorithm, but PHASTER shows the results nicely although it is not useful for high throughput analysis.
3. As is good practice, make a new folder this analysis. Maybe something like “phages” could be good.
4. Ask phispy how it likes to be run, and also look closely at the github. Run the first genome and note that it would like to be fed a genbank-file (e.g. a file with a “.gbk” or **“.gbff”** ending), which is a very standardized annotation file. Perhaps the annotation we did yesterday gave us one…?
5. Phispy can do a bunch of different outputs. Apart from the coordinates of these phages, we are often particularly interested in the actual sequences of the predicted prophages, so see if you can get phispy to give you that in a .fasta file. Note that it uses a somewhat complex binary numbering system here.
6. Of course, the loop-logic we used yesterday can be used again when you are ready to run phispy on all your genomes. Same logic – loop over .gbff files and place the output in a folder somewhere it makes sense. The main challenge here, however, is the fact that the files are now in individual folders rather than nicely collected within one single folder. Luckily, this is such a common occurrence that Linux has a built-in function for it: the ‘find’ command, which finds files according to a pattern within folders (and much more). Surely the syntax for finding files with a given extension/format can be googled (such as .gbff-files). This can then be used in a loop like so and treated just like before:

**for i in $( FIND\_COMMAND ); do echo $i; j=$(basename $i .gbff); echo $j; done**

1. Make sure you understand, you will need it later. Start thinking of the biological inferences here before you get lost in all the data.

**? how many phages do you see in each genome?**

**? Are they in the same position on the genome?**

**? Is it the same phages in all the genomes?**

**? How would you compare them?**

**? Do you see the same phages as in PHASTER?**

## Prediction of Biosynthetic Gene Clusters

1. Now we run the antiSMASH algorithm on our genomes, which is the current gold standard for finding gene clusters that encode the production machinery to produce secondary metabolites, also called Biosynthetic Gene Clusters (BGCs).
2. Once again, installation might prove tricky, and the online guide may be outdated. Follow this installation code exactly (which might take a while):

**conda create -n antismash**

**conda activate antismash**

**conda install -c bioconda -c conda-forge antismash**

**download-antismash-databases**

1. Running antismash is however easy enough:

**antismash GBFF\_FILE**

1. Once again, ask antiSMASH how I would like to be run and then write a nice loop that can do it. Be aware that antiSMASH will use all your cores by default, so make sure to handle that appropriately. Also be aware that antiSMASH outputs a folder named by the input file unless you tell it otherwise.
2. AntiSMASH gives a bunch of outputs which are a little difficult to parse automatically. If you are comfortable with databases, the .json files contains the most information. For the rest of us, we can rely on the .html-files, which can be visualized in a standard browser. Navigate to the antiSMASH output-folder and use the following command to open the folder in windows and then simply inspect the html files in your browser of choice:

**explorer.exe .**

1. Here, we might be interested in the total number of BGCSs, which types they are and if we have any novel BGCs . Usually, the most interesting ones are NRPSs, PKSs and RiPPs, which are traditionally the ones we use as antibiotics and various other important functions.

**? What where the dominant classes of BGCs?**

**? Are there any BGCs common to all genomes?**

**? Can you check for BGCs on the phage regions by running antismash on the phages you found?**

## Whole genome phylogeny

1. Lastly, we will do the phylogeny, that is, estimate how closely related our genomes are. Here, the gold standard is ROARY, which will fish out all genes that are sufficiently common across our genomes to be compared. Then we align them and build the tree from that. In more detail, ROARY uses blast to compare every protein of every genome, and within a certain cutoff of similarity (95% by default), considers proteins to be core genes if they are present in most (99% by default) of genomes. The nucleotide sequences of these core proteins are then put back-to-back for each genome and this sequence is then aligned for all genomes. This is a fairly clever data-driven approach to compare genomes, since it keeps as much information as possible given the data. In contrast, older methods rely on single genes, such as the 16S rRNA gene or specific collections phylogenetically specific genes, e.g. MLST. **ROARY is, unfortunately, very slow and difficult to install.**
2. Instead, we will use **PANTA** which is fast, but requires some extra steps to install. Notably, it doesn’t have a full conda installation, so we need a slight workaround:
3. First, we ‘clone’ the PANTA repository with the ‘git’ program. This simply means that we download the current version of the source files, enter that folder and build the software from there.
   1. **git clone** [**https://github.com/amromics/panta.git**](https://github.com/amromics/panta.git)
   2. **cd panta**
4. Then we create the environment, and, importantly, specify the **version of python to use**
   1. **conda create -y -c conda-forge -c defaults --name panta python=3.10**
   2. **conda activate panta**
5. Next, we build all the dependencies for PANTA using a file in the folder
   1. **conda install -y -c conda-forge -c bioconda -c anaconda -c defaults --file requirements.txt**
6. Lastly, we do the final install with the program ‘pip’
   1. **pip install .**

**? Discuss with your classmates how this approached was different from a normal install?**

**? Why did we need to specify a python version? What version would be installed if we didn’t?**

**? What is in the file ‘requirements.txt’**

Now make sure you navigate back to the folder with your files.

Panta wants a dedicated folder of gff3-files as input. We will make the folder and copy our files to it:

1. first make a folder for these GFF3 files
   1. **mkdir gff**
2. Then find all files ending with ‘gff3’ and copy them to the gff-folder
   1. Here, we look in the ‘annotations’-folder for files having the ending ‘gff3’, which we then use the ‘exec’ flag to call the ‘cp’ (i.e. ‘copy’) function into our newly made ‘gff/’-folder.
   2. **find annotations/ -name "\*gff3" -exec cp {} gff/ \;**
3. Finally, we run panta
   1. Panta is a suite of programs, the most important one being ‘main’. In this case we run the ‘main’ routine of panta on a set of gff-files (flagged by -f), and put the analysis in the folder ‘panta’ (flagged by -o). We also tell the program to use ‘proteins’ for analysis.
   2. **panta main -o panta -g gff/\* -a protein**

**? What is a core gene in this case?**

**? would this work if we move up in phylogeny, such as families?**

**? inspect the ‘summary\_statistics.txt’ file – what does that tell you?**

**? What is in the ‘core\_gene\_alignment.aln’ file and why can we use it to build the tree?**

## Phylogenetic trees

The easiest way to visualize genome differences is with a phylogenetic tree. We will use FastTree, which is probably the only tree-builder that is fast enough for such a big alignment. Install fasttree with conda, find out how it works and run it on the aligned DNA from PANTA (you want the file ‘core\_gene\_alignment.aln’ and you should also specify the -gtr argument, which is the probably most robust ‘generalized time reversible’ algorithm). This will also take a while, so consider doing it at home if your machine is slow. **If you have 16Gb RAM and you close all other programs, you might be able to build the tree. If you have 32Gb or above, you will probably be fine. If you have less than 16Gb RAM and/use your machine at the same time, it will probably fail. In that case, you can download my premade one using this command:**

wget https://raw.githubusercontent.com/mikaells/SlovSchool24/main/Data/core\_gene\_alignment.tree

Now we have to look at the tree, which seems like a good time to exit the command line and enter the modern world – lets have a with a dedicated tree viewer. For an online solution, ITol is pretty good <https://itol.embl.de/> . If you are comfortable in R, there are multiple ways to make beautiful trees such as <https://guangchuangyu.github.io/software/ggtree/> .

CLC is also very good: <https://digitalinsights.qiagen.com/products-overview/discovery-insights-portfolio/analysis-and-visualization/qiagen-clc-genomics-workbench/>

## Free play

Next up is free play, see if you can annotate your tree with the information we got from the earlier analyses. **You are done when you have a beautiful tree**. You can now repeat the analysis with your favorite bacteria if you wrote all the code correctly! Feel free to find more interesting things to look for - regardless of what you find interesting, someone probably wrote some software for it. If not, maybe you are the one that should do it?

Some suggestions for further analysis:

1. ABRICATE for antibiotic resistance profiling?
2. DBCAN for profiling of carbohydrate active enzymes (CAZYmes)?
3. PyANI for average nucleotide identity comparisons between genomes?
4. Customized BLAST-searches for your favorite gene?

**Feel free to as the teacher for input – I’m very happy to help and I’m only here for one week!**