UNIVERSITY OF TRENTO - Italy

Computational Microbial Genomics

March 1st, 2023

Francesco Asnicar - f.asnicar@unitn.it

COMPUTATIONAL METAGENOMICS SEGATA LAB

CIBIO

UNIVERSITY
OF TRENTO - Italy

Conda

CIBIO

Anaconda - conda - Bioconda

Anaconda (https://anaconda.com) is an environment manager and a Python/R data science distribution that comes with several (useful) packages already available

Conda (https://conda.io) is a package manager that allows you to build, install, update python packages and easily allows the creation and switch between environments on your local machine

Bioconda (https://bioconda.github.io) is a channel for the conda package manager specifically for bioinformatics software

Create the “cmg” conda environment

1. Download Anaconda (https://anaconda.com)

```
$ wget https://repo.anaconda.com/archive/Anaconda3-2022.10-Linux-x86_64.sh
```

2. Install Anaconda

```
$ sh Anaconda3-2022.10-Linux-x86_64.sh
```

3. Create a new environment named “cmg” with Python 3

```
$ conda create -n “cmg” python=3
```

4. Activate the environment

```
$ conda activate cmg  # run “conda deactivate” to deactivate the environment
```

5. Install Biopython

```
(cmg) $ conda install -c bioconda biopython
```

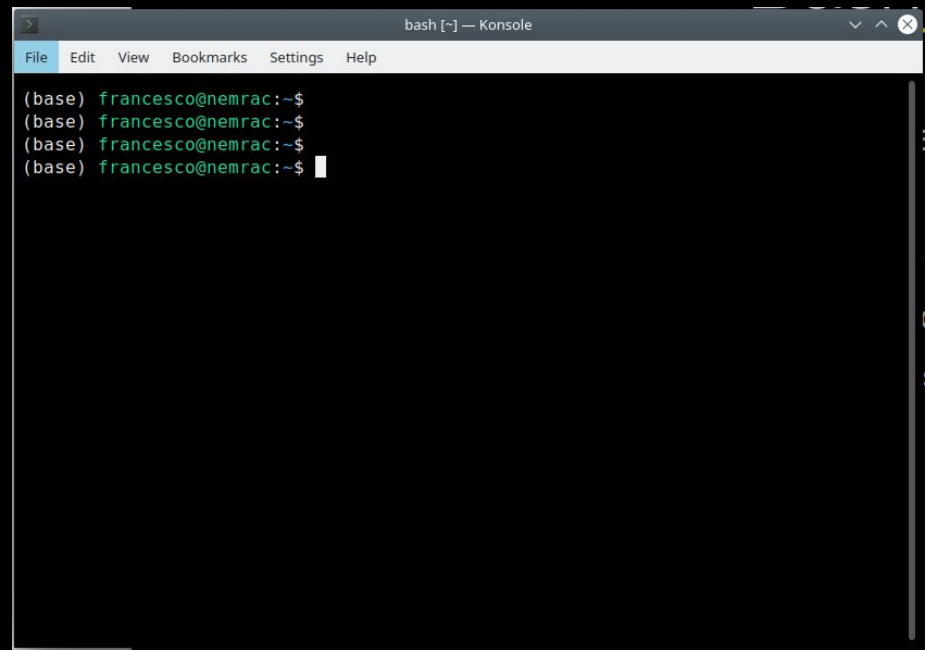
UNIVERSITY
OF TRENTO - Italy

Bash

CIBIO

Bash

- Command-line interpreter
- Allows to perform actions by inserting text commands
  - List directories and files
  - Move, rename, delete, edit and view files
  - Execute any type of software (even the ones with a graphical interface)
  - View information on the status of the system
  - Run a set of (my) commands
  - Programming



Print

- Print something to the shell

```
echo “Hello World”
```

- Print current time and date

```
echo $(date)
```

- Write some text to a file

```
echo “the text you want here” > output.txt
```

List

- List current directory

```
ls
```

- List only files in the current directory that ends with “`.txt`”

```
ls -lh *.txt
```

Copy

- Copying a file or a directory (`-r`)

```
cp <source> <destination>
```

- Copy a file to the parent directory

```
cp output.txt ../output.txt
```

Moving

- Moving a file

```
mv <source> <destination>
```

- Rename a file

```
mv output.txt output2.txt
```

- Move and rename a file to a different folder

```
mv output2.txt ../output3.txt
```

Change path

- Go to your home

  ```
  cd
  ```

- Go up (or back) to the parent folder

  ```
  cd ..
  ```

- Go to the “home” folder present in “/”

  ```
  cd /home/
  ```

Remove files

- Remove a file

```
rm <filename>
```

- Remove a directory

```
rm -r <directory>
```

Viewing (1/2)

- Visualize (without modify) the content of a file

```
less <filename>
```

- Print all the content of a file on the console

```
cat <filename>
```

Viewing (2/2)

- Print the first 10 (default) lines of a file on the console

```
head <filename>
```

- Print the last 10 (default) lines of a file on the console

```
tail <filename>
```

Concatenating commands

- Execute a series of commands for which the **output** of the previous command becomes the **input** of the following command

```
cmd1 | cmd2 | … | cmdN
```

- In bash, “ | ” is used to chain commands where the output of the left-side command becomes the input of the right-side command

- Examples

```
man less | head
cat output.txt | tail | head -n3
```

Output redirection

There are special characters

```
<
>
>>
```

that allow to redirect output to a file and/or command

- Examples

```
cat output.txt | grep “ASD” > to_a_file.txt
```

Text manipulation (1/2)

- Search for occurrences of `<what>` in `<where>`

```
grep <what> <where>
```

- Case-sensitive or not?

```
grep -i <what> <where>
```

- Invert the match (i.e., everything that does not match `<what>`)

```
grep -v <what> <where>
```

- Counts matched instances (`-c`)

```
grep -c <what> <where>
```

Text manipulation (2/2)

- Match whole word

```
grep -w <what> <where>
```

- Display the line number of the matches

```
grep -n <what> <where>
```

- Add context (lines) after (`-A`), before (`-B`), or both (`-C`)

```
grep -A3 <what> <where>
grep -B3 <what> <where>
grep -C3 <what> <where>
```

What do to if I need help?

- Use the man!

```
man <command>
```

- It provides you the manual for that command that explains what it does, what are the parameters, and other useful stuff

- Be aware that, in general, the man on a Linux system is simply displaying a text file using the `less` command. So, to exit just press `q`!

Let’s use the bash

1. Lets go to your home folder
2. Create a folder named “`tmp`”
3. Create a file that contains the manual of the `ls` command
4. Put this file into the “`tmp`” folder
5. Count how many occurrences of “`ls`” are present
6. Print the first 7 lines that matched
7. Save the output to a file named “`7.txt`” inside “`tmp`”
8. Print the first 3 lines of the last 10 lines that matched
9. Concatenate this output to the file “`7.txt`”
10. Now remove the “`tmp`” folder

UNIVERSITY
OF TRENTO - Italy

Python

CIBIO

Introduction

- Interpreted language
- Multiplatform, can run on Linux, Windows, and Mac
- Can be executed either in “Interactive mode” or in a “script mode”

```
09:22:49 | ~
$ python
Python 2.7.6 (default, Oct 26 2016, 20:30:19)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```python
#!/usr/bin/env python

from biom import load_table
from hclust2.hclust2 import DataMatrix
from argparse import ArgumentParser
from StringIO import StringIO
from re import compile

def parse_biom(filename) :
    biom_table = load_table(filename)
    strs = biom_table.delimited_self(header_value = 'TAXA', header_key = 'taxonomy')
    lst1 = [str(s) for s in strs.split('\n')]
    lst1 = lst1[1:] # skip the "# Constructed from biom file" entry
    biom_file = []

    pre_taxa = compile("._")

    for l in lst1 :
        lst = [str(s) for s in l.split('\t')[1:]] # skip the OTU ids

        # Clean an move taxa in first place
        taxa = '.'.join([s.strip().replace('[', '').replace('u\'', '').replace(']', '').replace(' ', '').replac
        taxa = pre_taxa.sub('', taxa)
        taxa = taxa.rstrip('.') # remove trailing dots

        biom_file.append('\t'.join([taxa] + lst[:-1]))

    with open('aaa.txt', 'w') as f :
```

Which version?

- Python2 vs. Python3
- There are some implementations differences and, in general, it is not always possible to execute a Python2 script with a Python3 interpreter, or vice versa

Variables

- *int*
  ```
  a = 3
  ```
- *float*
  ```
  b = 7.4
  ```
- *bool*
  ```
  c = True
  ```
- *complex*
  ```
  d = 2 + 3j
  ```

- No explicit definition of type
- Variable can be cast to be interpreted as a particular type
- Special value: `None`

Collections

Ordered

- *str*

```
e = “Hello world!”
```

- *list*

```
f = [1, 3.1, True, “ciao”]
```

- *tuple*

```
g = (“Hello”, “world!”)
```

Unordered

- *set*

```
f = set([1, 3.1, True,
         “ciao”])
```

- *dict*

```
f = {‘a’: 1, ‘b’: 3.1,
     ‘c’: True, ‘d’: “ciao”}
```

Operations

- Number

```
var1 + var2    var1 - var2    var1 * var2    var1 / var2
```

- String

```
str1 + str2          str1 + " " + str2    str1 - str2
str1 + repr(3.4)     str1 + str(3.77)     str1 + 4.99
```

- Print

```
print(str1)    print("hello" + "world")    print(3.5)
```

Indentation!!!

- To define block of codes Python uses indentation
- No parentheses are used to define block of codes (like happen in other languages)
- Be aware, do not mix tabs with spaces!

```
9  def parse_biom(filename) :
10     biom_table = load_table(filename)
11     strs = biom_table.delimited_self(header_value = '
12     lst1 = [str(s) for s in strs.split('\n')]
13     lst1 = lst1[1:] # skip the "# Constructed from bi
14     biom_file = []
15
16     pre_taxa = compile("._")
17
18     for l in lst1 :
19         lst = [str(s) for s in l.split('\t')[1:]] # s
20
21         # Clean an move taxa in first place
22         taxa = '.'.join([s.strip().replace('[', '').r
23         taxa = pre_taxa.sub('', taxa)
24         taxa = taxa.rstrip('.') # remove trailing dot
25
26         biom_file.append('\t'.join([taxa] + lst[:-1])
```

Block 1

Block 2

Control structures

If…else

```
if condition1:
    # block1
elif condition2:
    # block2
else:
    # block3
```

Loops

```
while condition:
    # block


for variable in iterable:
    # block
```

Conditions

Booleans

`x or y`

if **x** is false, then **y**, else **x**

`x and y`

if **x** is false, then **x**, else **y**

`not x`

if **x** is false, then **True**, else **False**

Comparisons

`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

List comprehension

- List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition

Join

- If I have a list of strings and I want to join it using a common separator, `join` is the function that I needed!

```
a = ['H', 'i', ' ', 't', 'h', 'e', 'r', 'e']
print(''.join(a))
```

```
print(',').join(a))
```

Range

- What if I want (to generate or iterate over) the first 10, 100, or 1000 numbers?

```
range(10)
```

```
range(10, 21)
```

Functions

- Group together lines of code that you need to execute often
- Assign a name that you can refer to
- You can specify many input arguments to pass to the function

```
def function_name(argument1, argument2, …):
    # body of the function
    return value # not always present!
```

Let’s start coding in Python

1. Open the Python console
2. Create a list of tuples as follows: $\{(a, a^2) \mid a >=1 \text{ and } a <=10\}$
3. Write the content of this list into a file where each line contains a tuple and the two values of the tuple are separated by a tab
4. Now, load the file just written into a dictionary, where the first number of each row is the key in the dictionary and the second number is the value.
5. Do the point above using a list comprehension
6. Loop on the dictionary and if the key is an odd number, print the value divided by 2

Script basic structure

```
1  #!/usr/bin/env python


2  import <module>
3  import <module> as <rename>
4  from <module> import <submodule>
5  from <module> import <submodule> as <rename>


6  # code and functions

7  if __name__ == '__main__':
8      # code
```

Shebang

```
1 | #!/usr/bin/env python
```

When executing the script, a special program recognizes the first line of your script and evaluate it

In the above example the command

```
/usr/bin/env python
```

Will be executed and its result will be the Python interpreter to use to execute the script

This first line is important to make sure which Python interpreter will be used to execute the code

Import external modules

```
2  import <module>
3  import <module> as <rename>
4  from <module> import <submodule>
5  from <module> import <submodule> as <rename>
```

(2) Import external functions implemented in a specific module

(3) To avoid clashes of names (or to shorten the module's name), one can define an alias to use in the code

(4) If you need only a specific function from a module, you can import only that function

(5) Also in this case, to avoid clashes of names you can rename the submodule

Script body

```
6 | # code and functions
```

Right after the imports and before the instruction in line number 7 there can be global code and the definition of functions

We'll see an example in a few slides!

Script body

```
7  if __name__ == '__main__':
8      # code
```

When you will execute your script, which part of the code will be executed first?

1. The first thing is the evaluation of the shebang

2. Then, the modules will be imported

3. All the code after the imports and before line 7

4. Finally, the line number 7 will be the next to be run. That will be the main entry point of your script

```python
#!/usr/bin/env python


import os
import sys
from time import time
from glob import iglob
from argparse import ArgumentParser


__date__ = '27 Mar 2015'
__email__ = 'f.asnicar@unitn.it'
__author__ = 'Francesco Asnicar'
__version__ = '0.01'


SUCCESS = 0
INVALID_ARGS = 1
FOLDER_NOT_FOUND = 2


def info(s, init_new_line=False):
    if s:
        nfo = '\n' if init_new_line else ''
        nfo += '[i] '
        sys.stdout.write(nfo + str(s) + '\n')
        sys.stdout.flush()


def error(s, init_new_line=False):
    if s:
        err = '\n' if init_new_line else ''
        err += '[e] '
```

```python
        for pcim in pcims:
            listt = []
            with open(folder+pcim, 'rU') as f:
                for r in f.readlines()[2:]:
                    listt.append(r.strip().split(',')[1])

            expansion_lists[pcim] = listt[:len(expansion_lists[aggregated])]

        output_file = folder+aggregated[:aggregated.rfind('.')]+"_similaritie

        if os.path.exists(output_file):
            info('file already exists, will be overwritten: "'+output_file+'"

        with open(output_file, 'w') as f:
            f.write(','.join(['aggregated'] + sorted(pcims))+'\n')
            similarities = []

            for pcim in sorted(pcims):
                similarities.append("{0:.2f}".format(jaccard_similarity(expan

            f.write(aggregated+','+','.join(similarities)+'\n')

        info('similarities file written: "'+output_file+'"')

    return SUCCESS


if __name__ == "__main__":
    t0 = time()
    args = read_params()
    status = main(args)
    info("total time: "+str(int(time()-t0))+"s")
    sys.exit(status)
```

How to run a script?

- Go to the folder where your script is located

```
cd path/to/the/folder/
```

- Check that you have the execution permissions

```
ls -l
```

- Run the script

```
./my_script.py
python my_script.py
```

What's the difference?

Let’s code!

Write a script that:

1. Takes as input a path to a folder (i.e., “`path/to/02_python_input`”)
2. Reads all the files contained in that folder and print on the console the current file being read
3. Store the content of each file into a dictionary (each filename can be considered a key)
4. And write the output file (if it doesn’t exists) “`results.tsv`” with the following structure for each input file read:

```
filename<TAB>number_of_lines<TAB>average_of_numbers
```

<u>Note</u>: each file has several rows (number not fixed) that only contains several integer numbers (number not fixed) TAB-separated