Additional resources for learning Python

I think the w3school is a good resource to learn python in details: https://www.w3schools.com/python/

SPOJ provides many problems examples - https://www.spoj.com/problems/classical/

Practical info about questions in Versioning with Git class

1. about the core.autocrlf thing in the configuration part, there is a detailed blog on this topic: https://www.aleksandrhovhannisyan.com/blog/crlf-vs-lf-normalizing-line-endings-in-git/

2. about the ssh-keygen, the wiki page https://en.wikipedia.org/wiki/Ssh-keygen and the ssh academy https://www.ssh.com/academy/ssh/keygen provide more details. They are more detailed but contain too much information.
To summarize, ed25519 is the algorithm (key type) specified for the generation. With the command we used
today:
```
ssh-keygen -t ed25519 -C "zdshao.teach@gmail.com" -f ~/.ssh/id_ed25519 -N ""
```
-t indicates the key type;
-C as the comment, and it does not have to be the email, while it is in practice; -N indicates the Passphrase, where "" (empty string) indicates no Passphrase; -f for the saving path;

POST WORKSHOP SURVEY:

https://forms.gle/aHdxA188BNGX4wjBA

Interaction during the workshop:
- zoom interactins: yes and no buttons, raise hands, etc.
- chat for questions

If you have problem with something, please indicate the helpers and "join" the breakout rooms with your operating system (windows, mac, or linux)

schedule: https://4turesearchdata-carpentries.github.io/2022-05-16-tudelft-online/

Day 4 - Git

Instructor: Zongru (Doris) Shao
swc lesson page: <https://swcarpentry.github.io/git-novice/>
lesson code repo: <https://github.com/zdshaoteach/2022-05-16-tudelft-online>
lesson contents during the class:: <https://github.com/zdshaoteach/2022-05-16-tudelft-online/blob/main/git-day-4.ipynb>
lesson slides full version: <https://zdshaoteach.github.io//2022-05-16-tudelft-online/>

Lesson 1: Intro to Git/GitHub

Today, we will need our GitHub username and the email address that was used to sign up to GitHub.

Repo is short for "repository"

Git is best at handling anything that is stored in plain text format, e.g. code and latex files.

Lesson 2: Setting Up Git

We will use the shell again today (GitBash on Windows, Terminal on MacOs and Linux).
Check that git is installed by running:

- `git --version`

Configuring git for the whole machine:

- `git config --global user.name "your-github-account-name"`
- `git config --global user.email "your.email@address.com"`

(account name and email address both in quotes!)

Checking the config:

- `git config --global --list`

Line endings differ between Windows and macOS/Linux, so different configurations have to be made:

- Windows:

        - git config --global core.autocrlf true
    - macOS/Linux
        - git config --global core.autocrlf input

Setting the default branch:

- git config --global init.defaultbranch "main"

Creating a new repository. Step 1: creating a new directory in your workshop folder:

- mkdir planets

Move into it:

- cd planets

Initialise the repo:

- git init

- Output:
    - Initialized empty Git repository in <path>/planets/.git

Check setup:

- ls -a

If you just use "ls" you will not see the .git file.

- Desired output:
    - . .. .git
    - (or ./ ../ .git/)

If you want to check, wether git is correctly set up (or later to check what is going on) you can use the command

- git status
- -> On branch main
- no commits yet

Now, we want to create a small testfile that we can later add to git

- echo "this is the first edit" > mars.txt

Then we add this file to git using

- git add mars.txt

You might get a warning about Line Endings depending on your configuration and operating system. This can be ignored.

If you now check

- git status

- -> new file: mars.txt

To save the file to git you need to make a **commit**:

- git commit -m 'first edit'

The -m flag is to add a message ("first edit") to your commit.
Hint: Help your future self and use meaningful commit messages - not"editing", "hopefully fixed the bug" but more "fix xyz for bug abc".

If you now check the status again

- git status
- -> nothing to commit, working tree clean

You want to see what has been going on in the past on your git instead of the current status, you can use

- git log
- -> will give you an overview over the newest commits

HEAD is git is the term for what/the version you are currently looking at.

We now add another line to our mars.txt

- echo "this is the second edit" >> mars.txt

and then check the status

- git status
- -> modified: mars.txt

We commit our second edit:

- git commit -m "second edit"
- -> 1 file changed, 1 insertion
- git log
- -> you now can see both commits

How to see what has changed in a file?

- git diff --staged
- -> you should get no output right now
- git diff HEAD mars.txt
- -> you should get no output right now as well
- git diff HEAD~1 mars.txt
- -> you can see what happend from the current **HEAD** to one version prior i.e. which lines have been editedt
- -> HEAD~1 refers to one commit before the **HEAD**, HEAD~2 would refer to two versions before

Once you add something to git, it is called being **staged**.

To make a multi line commit message, you can do this by not closing the quotation marks and hitting enter:

- git commit -m "this is the first line of the commit message
- > this is the second"

You can also add multiple files right after eachother or in one commit

- git add mars.txt
- git add venus.txt
- or
- git add mars.txt venus.txt

If you want to go back to a previous version of a file you can use:

- git checkout [commit number] mars.txt

You can access the commit number with

- git log

and then copy the commit number using right-click +copy

When coding, there are some file you likely do not want to push to git, but **ignore** them. This could be e.g. the .pyc files when coding in python or .class files when coding in Java.

If you add such an undesired file, e.g. by

- touch a.dat

and then check git status you will see that are untracked files.

To ignore file, you need to tell git which file to ignore.

- echo ".dat" > .gitignore

This creates a .gitignore file that will ignore all files file the .dat extension.

Now you have to add the git ignore file to git:

- git add .gitignore

and commit it:

- git commit -m "adding gitignore"
- git status

-> now any changes to *.dat files will be ignored by git.
Of course you can add other file types, specific file names etc. to your gitignore which you want to hide.

We now go to github.com again and create new empty repository. Please do not check any readme files, gitignore etc. For simplicity reasons we will name it "practice".

To work online with git, we need to have a SSH Key set up. If you already have a SSH key connected to your github account, you can skip these steps.

To generate a new SSH key, we type

- ssh-keygen -t ed25519 -C "your@email.com" -f ~/.ssh/id_ed25519 -N ""

Hint: You NEED to use the same e-mail you configured for git!

You can check if the SSH key was generated with

- ls ~/.ssh/
- -> id_ed25519 id_ed25519.pub

A key always consists of a **public (id_ed25519.pub)** and a **private (id_ed25519)** key

The **private** key is always meant to keep **private**. The **public** key we will later add to git, it will be used for authentication.

We check the key out with

- cat ~/.ssh/id_ed25519.pub
- and use right click + copy to copy the answer to the clipboard

Then you can add it on github (Click Git Profile in the upper right corner > Settings > SSH and GPG keys) to the **github settings** with "**new SSH key**" . You need to enter your password again to add the key.
You can choose your own title like "My work laptop" to indicate which computer this key authenticates.

Now we check if it worked on our computer:

- ssh -T git@github.com
- yes
- -> You are successfully authenticated

If it askes for a passphrase, try enter a empty space.

If you now go to github and your empty repository and copy the command to push an existing repository:

- git remote add origin https://github.com/YourUsername/practice.git

You can check with

- git config --list

This is only to check that the remote.origin.url is well set.

And then copy the two follwing commands from the github instructions:

- git branch -M main
- git push -u origin main
- -> Then your repository is all setup

Now our remote repository is setup correctly locally - congratulations!

To actually push something later to our remote repository, we need to create a new file/edit our old file:

- echo "this is yet another edit" >> mars.txt
- git add mars.txt
- git commit -m "another commit message for our latest commit"

Commits are so far only local savings of our work. To actually send it to the remote server we need to do a so called **PUSH**.

- git push

If you now refresh your github webpage, you can see how your files appeared on the remote page.

On the web page, when you click on "history" you can explore what you did (your changes) locally in a nice UI.

How to add a colaborator?

Click on profile -> Settings -> Collaborators
Click "add people"
Search for the respective username
Select the person and then click "Add .... to this repository"

Part of Partner A to make a change:

- echo "this is the third edit" > mars.txt
- git add mars.txt
- git commit -m 'third commit'
- git push

-> If you refresh the remote github website, you can seethe latest change

Partner B now needs to clone via SSH A repository.
Click on Clone > Change to SSH and copy the link.

back in your command line:
change to a new folder

- cd ..

Clone your Partners Repository

- git clone [add the url]

-> the repository is getting cloned and you can check with

- ls
- -> you should now see the "Practice" repo from your partner

Now A people can clone their Partners Repository the same way

A now can cd into As repository again and do a small change, add, commit and push it, just like we did before.

If you refresh your repository on the website now, you can see the newest commit.

For B the situation is now that A had made a change, but this is not visible on your local version.

To update your local version, you need to do

- git pull

in the local folder of Partner As repository

-> Now you also have the newest version of partners A repository

Now A and B both make changes to As repository.

A can push already, but B has to wait a little bit.

When B now pushes, you will get an error message! This is what we want :)
This a conflict, since both A and B tried to push to the same repo and the same file. To resolve the conflict you (B!) do

- git pull

Now we have to resolve the conflict on the mars.txt

- cat mars.txt
- -> Remove <<<<<<<<<<<<<<<<<<<<<<<HEAD
- and >>>>>>>>>>>>>>>>>>>>>>>>> 746scyd56as2d
- 6c5w

e.g. using

- echo "This is the new edit
- this is the second edit" > mars.txt

So now that we resolved the conflict manually, we need also to send it to git again.

- git add mars.txt
- git commit -m "resolved conflict"
- git push

Now the other partner can pull again and see how the conflict was resolved.

FEEDBACK FORM:
https://forms.gle/aHdxA188BNGX4wjBA

Day 3 - Python 2

Instructor: Zongru (Doris) Shao
lesson code slides: https://github.com/zdshaoteach/2022-05-16-tudelft-online/blob/main/python-key-links.md
https://zdshaoteach.github.io//2022-05-16-tudelft-online/
lesson code repo: <https://github.com/zdshaoteach/2022-05-16-tudelft-online>
Lesson page: https://swcarpentry.github.io/python-novice-inflammation/
Data: https://swcarpentry.github.io/python-novice-inflammation/data/python-novice-inflammation-data.zip
notes during the workshop has been uploaded to <https://github.com/zdshaoteach/2022-05-16-tudelft-online/blob/main/python-day-3.ipynb>

Reminder: Starting Jupyter Lab:

Starting JupyterLab -> UTwente JupyterLab (vpn outside of UT)
or

macOS and Linux:

1. Open a terminal
2. run jupyter lab

or

Windows:

1. open the Anaconda Prompt
(Start Menu -> Anaconda -> Anaconda Prompt)
2. run jupyter lab

Revisting the JupyterLab Interface

- blue plus in the top left: open a new launcher, e.g. to open a new notebook
- in the top right of your notebook, you should see the kernel called "Python 3"
- a green circle in front of the file name in the file overview indicates that the kernel is running
    - if you move the file while the kernel is still runing, thing might not work properly
    - try restarting the kernel via the menu at the top: Kernel > Restart Kernel ...

Recap Python Day 1

Things we saw yesterday:

- variable data types
    - numerical: int float
    - list (iteration)
    - string
    - numpy loadtxt
    - range
    - slices
- control over code:
    - loops
- file system
    - glob
- other functionality
    - plot

Lesson 7 - Making Choices

terminology for conditions:

- condition: if a condition is fulfilled, I want to do one thing, otherwise another thing
- code branch: the code that is executed depending on whether a condition is fulfilled or not

general syntax:

- if condition:
    - ... # this is a branch
- elif condition:
    - ...
- elif condition:
    - ...
- else:
    - ...

boolean values: either True or False, no other values

trying out booleans in conditionals:

- if True:
    - print(1)
- elif True:
    - print(2)
- elif True:
    - print(3)
- else:
    - print(4)

The code above prints 1. In order to print the second, the first condition has to not be fulfilled, e.g. by changing it to "False".

- num = ?
- if num > 100:
    - print('greater than 100')
- elif num > 50:
    - print('50 < num <= 100')
- elif num > 10:
    - print('10 < num <= 50')
- else:
    - print('num <= 10')

Numbers that we can use to print the different branches:

- first branch ('greater than 100'): 105
- last branch ('num <= 10'): -7

Combining several conditions:

- condition1 **and** condition2 **and** condition3
    - **every condition** has to be true, in order for the whole to be true
- condition1 **or** condition2 **or** contition3
    - **at least one** condition has to be true, in order for the whole to be true

Using **and** to check whether a value is within a certain range:

- num = ?
- if num > -1 and num < 1:
    - print('-1 < num < 1')
- elif -3 < num and num < 3:
    - print('num in (-3,3) but not in (-1,1)')
- else:
    - print('num is not in (-3, 3)')

to print the first statement:

- num = 0

to print the second statement:

- num = -1

to print the last statement:

- num = 7

Math notation detour: [lower, upper] includes lower and upper in range, (lower, upper) does not include lower and upper in range

Adding a cell below the current one:

- select the cell before and click on the "plus" at the top, next to the saving icon

Checking what functionality numpy offers to us:

- import numpy
- print(dir(numpy))

This will give us a list of everything that numpy provides. We can e.g. look for "load" in there, to find our "loadtxt" from yesterday.

If we know the name of the function, we can use the help function:

- help(numpy.loadtxt)

Let's load the data from yesterday:

- import numpy
- data = numpy.loadtxt('data/inflammation-01.csv', delimiter=',')
- max_cols = numpy.max(data, axis=0)
- if max_cols[0] == 0 and max_cols[20] == 20:
    - print('suspicious')
- elif numpy.sum(numpy.min(data, axis=0)) == 0:
    - print('minimum adds up to 0')
- else:
    - print('data looks good')

numpy.sum() calculates the sum of the input data

Difference between = and ==:

- = is an assignment, it assigns the value on the right, to the variable on the left, e.g.
    - text = "Hello"
- == is a boolean operator, checking if the left is equal to the right
- other boolean operators:
    - > (greater than)
    - < (less than)
    - >= (greater or equal)
    - <= (less or equal)
- the output of boolean operators is always True or False

Suspicious looking files:

- inflammation-01.csv
- inflammation-02.csv

Lesson 8 - Creating Functions

Functions allow us to package pieces of code, so that we can repeat them. They can have any number of return values. If a function does not have a return value, Python will always fill in the return value "None".

General syntax:

- def function_no_return(parameter1, parameter2, ...):
    - ... # do something

Let's write a function that converts Fahrenheit to Celsius:

- def fahr_to_celsius(temp):
    - return (temp-32) (5 / 9)

When we run a cell with a function definition, there is no output. We have only given a definition to Python but we haven't executed the function itself yet. In order to **use** the function, we have to call it as follows:

- temp_fahr = 32
- fahr_to_celsius(temp_fahr)

The input that we give to the function will be filled into the "temp" variable before the calculation is made. The output of the above is 0.0.

Exploring return types:

- def test_none_return(temp):
    - temp_converted = (temp-32)(5/9)
- test_none_return(32)

The above does not give any output, because we have not specified a return value, using the **return** keyword.

Comparing the function for which we have specified a return value and the one without a return value:

- temp_cel = fahr_to_celsius(32)
- temp_none = test_none_return(32)
- print(temp_cel)
- print(temp_none)

Output:

- 0.0
- None

Variable Working Scope:
-> which part of my program knows about a certain variable and where is it unknown?

- def test_scope(temp):
    - temp_1 = 4
    - print('temp', temp, 'temp_1', temp_1, 'temp_2', temp_2)
- test_scope(32)

Output:

- name 'temp_2' is not defined

Updated version:

- def test_scope(temp):
    - temp_1 = 4
    - print('temp', temp, 'temp_1', temp_1, 'temp_2', temp_2)
- temp_2 = 5
- test_scope(32)

Output:

- temp 32 temp_1 4 temp_2 5
- --> i.e. temp_2 is now available

What happens if we print temp_1 after running the function?

- temp_2 = 5
- test_scope(32)
- print(temp_1)

Ouput:

- name 'temp_1' is not defined
- --> temp_1 is only visible within the function, we can't access the value after the function is over

Adding temp_2 to the function:

- def test_scope(temp):
    - temp_1 = 4
    - temp_2 = 45
    - print('temp', temp, 'temp_1', temp_1, 'temp_2', temp_2)
- temp_2 = 5
- test_scope(32)
- print('outside: temp_2:', temp_2)

Output:

- temp 32 temp_1 4 temp_2 45
- outside: temp_2: 5
- --> the value of temp_2 differs, depending on whether we are inside or outside a function

If we want to overwrite a global variabel from within our function:

- def test_scope(temp):
    - global temp_2
    - temp_1 = 4
    - temp_2 = 45
    - print('temp', temp, 'temp_1', temp_1, 'temp_2', temp_2)

Output:

- temp 32 temp_1 4 temp_2 45
- outside: temp_2: 45

Using the global keyword, any changes that we make inside the function to temp_2 will also affect the value of temp_2 outside of the function
Everything defined outside of a function is residing in the global scope. In order to modify these variables, we do not have to use the global statement, e.g.:

- def test_scope(temp):
    - global temp_2
    - temp_1 = 4
    - temp_2 = 45
    - print('temp', temp, 'temp_1', temp_1, 'temp_2', temp_2)
- temp_2 = 5
- temp_2 = fahr_to_celsius(temp_2)
- print(temp_2)
- test_scope(32)
- print('outside: temp_2', temp_2)

Output:

- -15.0
- temp 32 temp_1 4 temp_2 45
- outside: temp_2 45

Break until 11:00
Exercises in breakout rooms

Lesson 9 - Errors and Exceptions

Original function (with errors):

```
• def order_ice_cream(customer, my_choices)
•     ice_creams = [
•         'chocolate',
•         'vanilla'
•         'strawberry',
•   ]
•
•     print('the full menu is:')
•     print('-----------------')
•         for index, item in enumerate(ice_creams)
•     print(index, item
•     print('---end of menu---')
•
•                 print('my choices are', my_choices)
•         price_to_pay = unit_price len(my_choices)
•     order = [ice_creams.index(i) for i in my_choice]
•     # write order to file
•     with open('orders/orders.txt', 'r') as f:
•         f.write("{}'s orders: ice cream items {}".format(
•             customer, order))
•
•     return order, price_to_pay
```

Identified errors:

- before execution, compiling stage:
    - we need a colon (:) after the function definition and after the for loop definition
    - inconsistent use of tabs and spaces in indentation (for loop definition)
    - indentation needed in for loop body
    - missing closing bracket in for loop body
- occurs during execution:
    - name not defined (unit_price)
    - misspelling (my_choice)
    - missing comma in "ice_creams" ("'vanilla' is not in list")
    - directory "orders" does not exist (--> create "orders" in working directory)
    - file "orders.txt" does not exist
    - file is not writable, read ("r") and write("w") are confused in the line opening the file

Fixed function:

```
• def order_ice_cream(customer, my_choices):
•     ice_creams = [
'chocolate',
'vanilla',
'strawberry',
'coffee'
]
print('the full menu is:')
print('-----------------')
for index, item in enumerate(ice_creams):
print(index, item)
print('---end of menu---')
print('my choices are', my_choices)
unit_price = 1
price_to_pay = unit_price len(my_choices)
order = [ice_creams.index(i) for i in my_choices]
# write order to file
with open('orders/orders.txt', 'w') as f:
f.write("{}'s orders: ice cream items {}".format(
customer, order))
return order, price_to_pay
```

Example execution of the funtion:

```
order_ice_cream('Doris', ['chocolate', 'vanilla'])
```

Another type of error:

```
def square(x):
    return x + x


square(2)
square(3)
```

Output:

```
4
6
```

--> the code runs just fine but it does not deliver the correct output (instead of addition it should be multiplication)

Lesson 10: Defensive Programming

(continuing with the previous "ice cream" example)

we can use the assert statement to check certain assertions. If the assertion is not met, the program will stop running and we will get an error message, e.g.:

```
assert choice in ice_creams, 'choice is not in the menu'
```

If something is not given in ice_creams, we will get the output: "AssertionError: choice is not in the menu"

```
def order_ice_cream(customer, my_choices):
    ice_creams = [
        'chocolate',
        'vanilla',
        'strawberry',
        'coffee'
    ]
    for choice in my_choices:
        assert choice in ice_creams, 'choice is not in the menu'
    print('the full menu is:')
    print('-----------------')
    for index, item in enumerate(ice_creams):
        print(index, item)
    print('---end of menu---')
    print('my choices are', my_choices)
    unit_price = 1
    price_to_pay = unit_price len(my_choices)
    order = [ice_creams.index(i) for i in my_choices]
    # write order to file
    with open('orders/orders.txt', 'w') as f:
        f.write("{}'s orders: ice cream items {}".format(
            customer, order))
    return order, price_to_pay
```

Example execution that will trigger the assertion error:

```
order_ice_cream('Doris', ['chocolate', 'vanilla', 'cherry'])
```

Question:
Why does the following not work?

```
help(assert)
```

assert is a Python keyword; help() does not work on keywords
You can find documentation for the assert statement (and many other things) here:
https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

Testing our square function:

```
def square(x):
    return x + x
```

```
assert square(0) == 0, 'square(0) should be 0'
```

- assert square(2) == 4, 'square(2) should be 4'
- assert square(3) == 9, 'square(9) should be 9'

Recommendation: do not use the same variable name inside and outside a function

Lesson 11 - Debugging

Options for debugging:

- printing intermediate values
- using the logging system to track values
- writing test cases to check which parts of your code work as expected and which don't
- using a debugger

Code for debugging demo:
print('start')
num = 2
if True:

- print(1)

elif True:

- print('a')

else:

- print(2)

Starting the debugger in JupyterLab:

- top right: click the little "bug" icon next to the Python kernel info (the bug should turn orange)
- the view of your cell will change, showing line numbers
- you can now click on the left of a line number and put what is called a breakpoint (a red dot will appear next to the line)
- when we run a cell with a breakpoint, the debugger will stop on that line and we can inspect the values of the different variables
- using the debugging controls we can step through our code line by line, function by function, etc.

Question: How can we delete/clear existing variables?

- Using the delete statement as follows:
    - del existing_variable_name
- Demo:
    - num = 5
    - print(num)
    - del num

    - print(num) # will raise: NameError: name 'num' is not defined

Lesson 12 - Command-Line Programs

Open a terminal by opening a new Launcher via the blue plus in the top left

How to use Python in the command line:

- python filename.py hello

-> python is the keyword to call a python script
-> filename.py is the name of the python script
-> hello is a paramter that we are passing to our python script

Making a Pyhon module:

1. Create a new folder, called python12
2. in the folder, create a new file, called __init__.py (note the **double underscore**!)
3. in the folder, create a new file, called modulev1.py

tree view of the folder and contents:

```
python12
├── __init__.py
└── modulev1.py
```

double-clicking on the file "modulev1.py" will open a text editor in JupyterLab. File content:

- def print_something(param):
    - print(param)
- print_something('hello')

run the file from the terminal with the following command:

- python path_to_your_file
- e.g.:
- python Desktop/swc-workshop/python12/modulev1.py

Output:

- hello

We have now created a module, similar to e.g. numpy. This can be imported in our **notebook** via:

- import python12.modulev1
- python12.modulev1.print_something('Doris')

When we import the module, it will print "hello", because we call our function in the script.

Day 2 - Python 1

Instructor: Raphaela
Lesson page: https://swcarpentry.github.io/python-novice-inflammation/
Data: https://swcarpentry.github.io/python-novice-inflammation/data/python-novice-inflammation-data.zip

Lesson 0: Starting up & General Jupyter Information

To start your Jupyter lab:

Starting JupyterLab -> UTwente JupyterLab (vpn outside of UT)
or

macOS and Linux:

1. Open a terminal
2. run jupyter lab
or

Windows:

1. open the Anaconda Prompt
(Start Menu -> Anaconda -> Anaconda Prompt)
2. run jupyter lab

to download the data file on a server, you could use:

- wget https://swcarpentry.github.io/python-novice-inflammation/data/python-novice-inflammation-data.zip
- unzip python-novice-inflammation-data.zip -d ./

Let's start:

First notebook - Start Python 3 notebook
The data format for a notebook is **.ipynb**.

In Jupyter lab you can access the typicall "actions" (save, load, settings) through the top left. We will use a few of these today, but most of them are not that frequently needed.

You can re-name your notebook by right-clicking on the tab name and then select rename.

Hint: Always give your notebooks good descriptives names so you can find them again later :) For today "Python_day_1" is probably a good name, but for the future: Experiment_xy_version_1 or similiar might be more useful.

Further, under the tab name of your notebook you have usual copy, paste, cut etc. buttons.

A "**cell**" in this context is automatically created when you create a new notebook - you can see the brackets on the left and then type in the main window.
You can create more cells by clicking on the plus.
Press **ESC** to exit the cell.

In Jupyter lab, you have (for today) two important options on **cell** settings:
Code and **Markdown**. You can change from markdown to code and vice versa either using the dropdown, or, when exited the cell press **M** to change to markdown and **Y** to go back to code.

Markdown is basically writing text with a few styling options, like **bold**, *italic* etc.
Unlike in e.g. MS Word you have to format in markdown using symbols (similiar to Latex):

- # Headline Nr. 1
- ## Headline Nr. 2
- ### Headline Nr. 3
- italic* _italic_
- *bold** __bold__
- `code`

You can find a more extensive cheat sheet here: https://www.markdownguide.org/cheat-sheet/

In this context, we use markdown to comment our code, add story telling to our code etc.

Lesson 1: Let's start coding!

If you open a new **code cell** and add

- 3+4

and then hit Shift+ENTER it will display

- 7

in the next line.

Hint: The numbers left to the cells are NOT the cell numbers, but the order of execution.

Variable in python are a possibility to store values.
If you e.g. enter

- age = 30

and hit enter, there will be no output directly but the notebook has now saved a variable called 'age' with the value 30.

Hint: Variables in python may contain letters, digits and underscore, may NOT start with a digit and are case sensitive! So AGE is not the same variable as age!

If you want to ask for the value of a variable you can print it using

- print(name_of_variable)

or if you want to print more than one

- print(variable1, variable2, variable3)

If you want to create decimal numbers, you use a point

- height = 1.6

and for strings you use " " or ' '

- name = "Raphaela"

Hint: To run a cell and create a new one, press SHIFT + ENTER

With these variables you can print more complicated constructs, like:

- print("Hi, my name is", name, "I'm", height, "m tall")

You can simply add more information with a "," - text always needs to be enclosed in " abc ".

Hint: Make sure to have no typos in your variable names!

You can always change the value of a variable by assigning a new one:

- age = 30
- print(age)
- -> 30
- age=31
- print(age)
- -> 31
- age = age + 2
- print(age)
- -> 33

Python supports the normal calculation operations + - * / ^

If you want to now the type of a variable you can ask this by

- type(variable)

Which type is what?

- **String**: normal text like "Raphaela"
- **Integer** (often called **int**): whole numbers without decimals, like 42
- **float**: Numbers which may contain decimals, like 42.42 but also 42.0

Lesson 2: First Data Analysis/Analysing Patient Data

Many people have already implemented lots of stuff in python - so you do not have to do it yourself again. Therefore these functionalities that have already been programmed are collected in so-called **libaries**.
You can import **libaries** to your notebook to use the functionality in your code.

Hint: In general, if something is available as a libary and it seems not so "shady" - use it and do not implement it yourself, the wheel has already been invented here :)

To import something in python type

- `import name_of_libary`

A really useful libary for number processing is numpy.
We install ist using

- `import numpy`

If it is not installed on your machine, you can try to install it using

- `!pip install numpy`

Hint: Programming people are always lazy people and really like abbreviations and shortcuts. "Numpy" is commonly referred to as "np".

You can set the "nickname" for a libary with

- `import numpy as np`

To access now functionality from numpy you use

- `np.name_of_functionality`

To load our sample data use

- `np.loadtxt(fname="data/inflammation-01.csv", delimiter=",")`

where "fname" is the name and path to the file you want to load the data from and delimiter is the separation sign used in that particulat file.
If the autocomplete with TAB key does not work, this page may help to resolve it:
https://stackoverflow.com/questions/33665039/tab-completion-does-not-work-in-jupyter-notebook-but-fine-in-ipython-terminal

If you press **ENTER**, an array of the loaded data will be displayed.

To load the data to variable (in this case "data"), type

- data = np.loadtxt(fname="data/inflammation-01.csv", delimiter=",")
- print(data)
- -> array of data is shown
- print(type(data))
- -> class numpy.ndarray - this is the internal numpy name for number arrays

To know the dimensions of your number array, you can use the function "shape"

- print(data.shape)
- -> (60,40)

If you want to access a specific value from the data array, use [x_value, y_value]

- print(data[x_value, y_value])

Hint: Python (and most other programming languages) starts counting by 0, so the upper left corner is 0,0.

To access the data for one patient:

- patient_0 = data[0,:]
- print(patient)

-> You get all the data from row zero, so the data for patient 0 for all 40 days.

If you just want to access the first 10 days instead of all 40

- patient_0_10_days = data[0,0:10]
- print(patient_0_10_days)

-> You get only the first 10 entries for patient 0.

The format for accessing data like this is

- data[row, column_start:number_of_steps]

This also works for strings:

- element = "Oxygen"
- print("first three letters", element[0:3])

-> first three letters Oxy

- print(element[1:-1])

-> xyge (From the second element (we start with 0), to the last)

If you want to know the **mean** of some date, you can use the function "**mean**" with

- print(np.mean(data))
- -> mean of all values
- print(np.mean(patient_0))
- -> Mean value of patient 0
- print(np.mean(patient_0_10_days)
- -> Mean value of patient zero for 0 days

This works the same way with **max** (maximum), **min** (minimum) , **std** (standard deviation) instead of **mean**. E.g.

- print(np.max(patient_0))
- -> Max value of patient 0

If you want to work on one axis of the data instead of all of it, you can give additional arguments to your **function cal**l (function call = fancy name for using a function).

- print(np.mean(data, axis=0))
- -> Mean values along the axis 0 -> Average per day
- -> axis = 1, would give you the mean per patient

Comments in Python can be added using #

- # this is a comment and will not be executed

Hint: Use comments on a regular basis to describe what your code is doing or what you think it does - your future self will thank you!

Quick Summary:

- #general numpy methods
- np.mean(data)
- np.max(data)
- np.min(data)
- #numpy methods in a certain direction
- np.mean(data, axis=0)
- # slicing numpy data
- data[0,:] # all days for the first patient
- data[0,0] # first patient, first day
- data[0,0:10] # first patient, first ten days
- # some info about numpy arrays
- data.shape # number of rows, number of columns

If you want to get more information and documentation about numpy, you can go to https://numpy.org/

Lesson 3: Visualizing data

Another useful libary to plot data is **mathplotlib** (again - do not reinvent the wheel but use existing libaries when possible!)

Import it with

- import matplotlib.pyplot as plt

For our first image plot, we use the following function:

- plt.imshow(data)

Where y axis shows the patients, x axis shows the days.

Caution: This works fine in Jupyterlab - for "real" python scripts an additional line is needed:

- plt.imshow(data)
- plt.show()

If you do not like the color-map that is set for default, you can exchange it against other color maps :)

- average_inflammation = np.mean(data, axis=0)
- print(average_inflammation)
- plt.plot(average_inflammation)
- -> will print out the array first and then a 2D graph for this

You can also directly combine it in one line:

- plt.plot(np.min(data, axis=0))

Always make sure to close all the brakets :)

Now, we want to combine all the three plots into one figure:

- fig = plt.figure(figsize=(10.0,3.0))

If you want to get more information, you can put the cursor to "figure" and the click **Shift** + **Tab**.

Creating three plots in one figure:

- # adding subplots to our figure
- subPlot1 = fig.add_subplot(1,3,1)
- subPlot2 = fig.add_subplot(1,3,2)
- subPlot3 = fig.add_subplot(1,3,3)
- # plotting the first plot
- subPlot1.set_ylabel("average")
- subPlot1.plot(np.mean(data, axis=0))
- # plotting the second plot
- subPlot2.set_ylabel("maximum")
- subPlot2.plot(np.max(data, axis=0))

- #plotting the third plot
- subPlot3.set_ylabel("minimum")
- subPlot3.plot(np.min(data, axis=0))

- # just for beautiful layout purposes :)
- fig.tight_layout()

To make the y-axis all the same, we can edit the code a bit:

- # adding subplots to our figure
- subPlot1 = fig.add_subplot(1,3,1)
- subPlot2 = fig.add_subplot(1,3,2)
- subPlot3 = fig.add_subplot(1,3,3)
- # plotting the first plot
- subPlot1.set_ylabel("average")
- subPlot1.plot(np.mean(data, axis=0))
- **subPlot1.set_ylim(0,22)**
- # plotting the second plot
- subPlot2.set_ylabel("maximum")
- subPlot2.plot(np.max(data, axis=0))
- **subPlot2.set_ylim(0,22)**

- #plotting the third plot
- subPlot3.set_ylabel("minimum")
- subPlot3.plot(np.min(data, axis=0))
- **subPlot3.set_ylim(0,22)**

- # just for beautiful layout purposes :)
- fig.tight_layout()

You can find more information for matplob, visit: https://matplotlib.org/stable/gallery/index.html

Lesson 4: Lists

To create a list in python, you can use [ ]

- odds=[1,3,5,7]
- print("odds are", odds)
- -> odds are [1, 3, 5, 7]

- print("first element", odds[0])
- print("last element", odds[-1])
- -> first element: 1
- -> last element: 7
- names = ["Darwin", "Lovelace", "Curie"]
- print("names originally are:", names)
- -> names originally are ['Darwin', 'Lovelace', 'Curie']

- names[0] = "Turning"
- print("names now are:", names)
- -> names now are ['Turing', 'Lovelace', 'Curie']

BUT you cannot access characters in strings with [-1], this only works for lists.

You can also nest lists into eachother:

- movingBoxes = [["pots", "pants", "cutlery"], ["TV", "books", plants", "sofa"]]

Hint: The lists do not need to contain the same amount of elements!

To now access an element you use:

- print(movingBoxes[1][0])
- -> 'TV'

Different here from the numpy array, we pick the lists after each other and not like [1,0] for coordinates.

Python does not care about the type of values that insert into a list, you can easily add them:

- sample_ages = [10, 12.5, "prefer not to tell"]

To add a new value to the list, you can use the **append** function:

- sample_ages.append(90)

To remove an item, you can use the **pop** function

- removed_element= sample_ages.pop(0)
- print(sample_ages)
- -> [12.5, "prefer not to tell", 90]
- print(removed_element)
- -> 10

You can reverse a list using the **reverse** function:

- sample_ages.reverse()
- print("after reversing", sample_ages)
- -> [90, 'prefer not to tell, 12.5]

You can also **combine** lists

- new_ages = [1,2,3]
- combined_samples = sample_ages + new_ages
- -> [90, 'prefer not to tell, 12.5, 1, 2, 3]
- combined_samples.extend([4,5,6])
- -> [90, 'prefer not to tell, 12.5, 1, 2, 3,4,5,6]

Caution! If you "multiply" a list with x, it will get appended x times.

- counts = [2,4,6,8,10]
- repeats = counts 2
- print(repeats)
- -> [2,4,6,8,10,2,4,6,8,10]

Terminology: **Overloading** - an operation that does different things for different types of variables like * normally does multiplication, but does append lists in this context.

Slicing for strings:

- bionomial_name= "Drosphilia melanogaster"
- group = binomial_name[0:10]
- print("group", group)
- species = binomial_name[11:23]
- print("species", species)
- -> group Drsophila
- -> species elanogaster

This works the same way for lists:

- counts = [2,4,6,8]
- print("first two numbers", counts[0:2])
- -> first two numbers [2,4]

You can also define the step size in which you want to do the slicing, with giving a third parameter like this **[from:to:step_size]**

- primes = [2,3,5,7,11,13,17]
- subset = primes**[0:7:2]**
- print(subset)
- -> [2,5,11,17]

If you do not want to specify a beginning or an end, but just the step size - you can just skip those:

- subset = primes**[::2]**

A useful function to find out the **length** of a string/construct is **len**:

- len("species")
- -> 7

Hint: Think of strings as lists of characters - that is why you access them so similiar.

Lesson 5: Loops

Now we will be iterating/looping over elements:

- odds = [1,3,5,7,9,11]
- **for** num **in** odds**:**
    - print(num)

->

- 1
- 3
- 5

- 7
- 9
- 11

Hint: In Python, if you enter the "body" of a loop you need to indent one level. If you forget this, python will complain.
Other programming languages might e.g. use curly brackets.

If you want to count how many times a loop is run, you can do this:

- counter = 0
- word = "oxygen"
- for char in word:
  - counter = counter + 1
  - print(char)
- print(counter)
- # notice that we are now outside of the loop again, because the indentation is gone!

```
->
o
x
y
g
e
n
6
```

How to sum all numers in a list:

- numbers = [1,2,3]
- summed = 0
- for num in numbers:
  - summed = summed + num # body of the loop with indentation
- print(summed) # outside of the loop
- -> 6

Scope of the loop variable:

- name = "Lovelace"
- print("before loop", name)
- for name in ["Turning", "Curie", "Darwin"]:
  - print("in loop", name)
- print("after loop", name)

Result:

- before loop Lovelace
- in loop Turing
- in loop Curie
- in loop Darwin

- after loop Darwin

-> At the end of the loop, the variable will contain the last variable from the looping, and not go back to Lovelace!

Hint: If you want to see all the variables you have defined in your notebook you can use %who **and** %whos. **You can find further information here: https://www.geeksforgeeks.org/viewing-all-defined-variables-in-python/**

To create sequences of numbers without typing them out one by one, you can use the function **range**():

- # range(n) gives n numbers from 0
- list(range(3))
- -> [0,1,2]

- # range(start, stop) gives numvers from start until stop-1
- list(range(2,5))
- -> [2,3,4]

The lower end is inclusive, the upper end is exclusive.

You can also define the step size:

- #range(start, stop, step) like range(start, stop) but with stepsize step
- list(range(2,10,2))
- -> [2,4,6,8]

How to print the numbers 1,2,3 using range()?

- for number in range(1,4):
    - print(number)

or

- for i in range(3):
    - print(i+1)

Lesson 6: Analysing data from multiple files

Again, we will be importing a new libary called **glob** to specify patterns to collect files (instead of telling the actual data file names)

- import glob
- print(glob.glob("data/inflammation.csv"))
- -> This gives us a list of all available files in data/ that start with inflammation and end with csv, but unsorted
- filenames = sorted((glob.glob("data/inflammation.csv")))
- print(filenames)
- -> gives us the filenames in a sorted way
- # load three files and print their names and data
- smallFilenames = filenames[0:3]

- for filename in smallFilenames:
    - print(filename)
    - data = np.loadtxt(fname=filename, delimiter=",")
    - print(data)
- # create a plot for min, max, mean in one figure for each of the three loaded files
- smallFilenames = filenames[0:3]
- for filename in smallFilenames:
    - print(filename)
    - # load the data from one file per iteration
    - data = np.loadtxt(fname=filename, delimiter=",")
    - # create a figure
    - fig = plt.figure(figsize=(10,3))
    - # adding three subplots to ist
    - plot1 = fig.add_subplot(1,3,1)
    - plot2= fig.add_subplot(1,3,2)
    - plot3=fig.add_subplot(1,3,3)
    - # plotting for the average
    - plot1.set_ylabel("average")
    - plot1.plot(np.mean(data, axis=0))
    - # plotting for the maximum
    - plot2.set_ylabel("maximum")
    - plot2.plot(np.max(data, axis=0))
    - #plotting for the minimum
    - plot3.set_ylabel("minimum")
    - plot3.plot(np.min(data, axis=0))
    - # just for layout purposes :)
    - fig.tight_layout()

To save your notebook after a long day of programming you can hit CTRL + S.

FEEDBACK FORM: https://forms.gle/wPjtXJdse5ARscPY9

Day 1 - Unix shell/ Shell novice

- Instructor: Raphaela
- lesson page : The Unix Shell https://swcarpentry.github.io/shell-novice/
- setup : please follow setup on the page https://swcarpentry.github.io/shell-novice/setup.html, and you may download a windows installer/portaable version on page https://github.com/git-for-windows/git/releases/tag/v2.36.1.windows.1

The idea of this workshop is to "code along". Raphaela will type things on her shell and you can try it out directly yourself as well. If you have questions - do not hesitate to ask, that is what the helper are here for :)

Data link: https://swcarpentry.github.io/shell-novice/data/shell-lesson-data.zip
We will work with the sample data today to demonstrate the advantages of shell usage.

Normally, you interact with your computer using the **Graphical User Interface (GUI)**. This is really convenient for "normal" task, but less convenient for repetitive task. The **Unix shell**, a **command-line interface**, is a great alternative for these tasks. Additionally, the shell is often a great way to interact with remote machines/servers/supercomputers.

A popular 'version' of the Unix shell is **Bash**.

How to access the shell for today's lesson:
Windows: use git for bash
Linux: normal new terminal
macOs: type terminal in spotlight, if on Catalina (10.15) or newer -> type bash and hit enter as a first step

Lesson 1: Navigating files and directories

Your terminal always starts with a naming in the top left. Normally it is
`username@computername:~$`

The $ sign here is the so-called **command prompt**.

The first command to learn is

- `ls`

which is sort for *listing* and will list all the files and directories in directory you are currently in. This is the same thing as if you would have opened the folder in the GUI you are currently in.

In case you ever mistype a command (due to a typo), this is not a problem at all, the command line will tell you "command not found" and then you can retype it.

To see "where" in your computer you are currently in your terminal, you can type

- `pwd`

which stands for *print working directory*.

The highest directory in your system, indicated by / is called the "**root**" directory.

So e.g. /Users/nelle/ is the folder nelle, which is in a folder Users, which is located in the root directory. Using the GUI being in Users you would click double first of Users and then on nelle to access the data.

How do I know if something is a folder or a file?
Type

- `ls -F`

then every folder is indicated by a / behind it. So *"Downloads/*" is your downloads folder, but *"Downloads"* is a file.

Adding something with a "-" to a command, like -F in the latest example is called a **flag**. This is an additional option.

With the flag *--help* you can get additional information on how to use every command in the shell. In most cases this also shows you all the available flags.

e.g.

- `ls --help`

In case you make a typo while writing a flag, the command line will inform you that this is not an available option.

In addition to --help you can also type

- `man ls`

or

- `man pwd`

which will give you the manual to your command (ls or pwd in this case). Of course you can also just google the manuals to any commands. :)

To take a sneak peak into another directory without navigating there, directly you can type:

- `ls Desktop`

This will give you all the files and directories on your desktop.
If you are lazy and do not want to type out e.g. "*Desktop*" you can just type *ls Desk* and then hit "tab". The terminal will fill out the nearest possible completion, so you can easily avoid typos. If there are multiple options, you can go through all of them by hitting tab multiple times.

To change the directory you type

- `cd new-place`

e.g.

- `cd Desktop`

to go there. cd stands for "**change directories**".
After changing directory you can use *pwd* (print working directory) to see where you are or *ls* (listing) to see the directories and files available.

If you get the "no such file or directory" error, check if you made a typo in the file name or if your data is in the directoy that it you think it is (you can check this with "*ls*").

To clean your teminal you can use the command

- clear

then it scrolls up.

If you want to go one folder up again ("go back"), type

- cd ..

so from

Users/Desktop you can type *cd* .. and then will be in *Users/* again.

If you type

- ls -F -a

it will give you the same results as ls -F , plus ./ and ../,
with ./ indicating the current directory and ../ inidcating one directory up.

Just

- cd

without any further arguments/names you get back to your root directory. So if you ever get lost, you can use just *cd* a trick to get back to a "safe point".

Another option for ls is -r which displays the output in reverse order, e.g.

- ls -r

Prints all the files and directories in the current directory, but in reverse order.

To sum up, with

- $ ls -F /Desktop
- $ is the **prompt**
- ls is the **command**
- -F is the **option** or **flag**
- /Desktop is the **argument**

Depending on the command you can have multiple options/flags or multiple arguments.

Lesson 2: Working with Files and Directories

To create a new directory from the shell, you can use the command

- mkdir name

to create a directory called *name*.
If you then run *ls* again, you can see your new, empty directory.

A useful flag for *mkdir* is *-p* (**parent**), which will automatically create all directories missing on the way as well. So

- mkdir -p ../project/data

will create two folders: project/ in your current folder and data/ within the project folder.

Hint: Do try not to name your folders with spaces, as it can confuse the shell. Good alternatives a - or _

Hint: Do not start your folder names with a - , as this normally indicates the shell that an option is coming instead of a name.

To create a text file, instead of a directory, you can use the command

- nano draft.txt

To create a text file called *draft.txt* and directly open the command line text editor.
At the bottom you will see some available options, like **CTRL+O** to save your file. Nano will then ask you where to save it.
To leave nano press **CTRL+X** to get out of nano again.

The read what is your freshly saved text file, run

- cat draft.txt

This will print out the content of the file to your command line.

To create an empty file, without openening it with nano, use

- touch name.txt

To move a file from one place to another you can use the **move** command *mv*.

- mv filename destination

E.g.

- mv draft2.txt ..

moves the draft2.txt file one level up.

The *mv* command can also be used to re-name files ("*moving them to a new name*") with

- mv old_name new_name

e.g.

- mv draft2.txt draft3.txt

renames draft2.txt to draft3.txt

To copy files you can use the **copy** command *cp*.

- cp filename destination

e.g.

- cp thesis/paper.txt thesisdraft.txt

will copy the *paper.txt* from the *thesis/* folder in the current directory and will name it *thesisdraft.txt*.

Hint: If you use a name for your copy that is already existing in your target destination, you will replace the existing file!

To remove files, you can use the command rm (this does not work for directories with content!).

- remove name.txt

Will remove name.txt from your directory.

Hint: There is no waste bin - once you delete a file, it is gone!

To remove everthing in a directory and its complete content, use

- remove -r

with -r stands for recursive.

A handy thing while operating with files is * - called wild card. * stands for one or more characters. If you want exactly 1 character, you can use ? instead.
So e.g.

- ls p.txt

or

- ls p?.txt

lists all the .txt files in the directory that start with p or all the txt files which start with p, then one character and than .txt - so it would list p1.txt but not ppp.txt.

Lesson 3: Pipes and Filters

The get the word count from your file, you can use

- wc cubane.pdn

which gives you lines, words, characters of that file (in this case cubane.pdn)

This can also be done for multiple files at once with e.g.

- wc .pdb

With the flag -l you can only let yourself output the number of lines.

- wc -l .pdb > filename.txt
- e.g., wc -l *.pdb > lengths.txt means write the output of wc -l *.pdb to the file lengths.txt — it is a one-line command

prints all the line numbers of *.pdb to a txt file named filename.

The output of a file can be **sorted** with

- sort numbers.txt

The normal sort is a string sort, so 10, 2, 11 sorts to 10, 11, 2.

To sort this numerical you can use the -n option

- sort -n numbers.txt

To write the sorted content to a file name, you can us

- sort -n length.txt > sorted_length.txt

which sorts the content of length numerically and save it to sorted_length.txt

To take a sneak peak into a document you can use the command

- head -n 1 sorted_length.txt

To get the first line of sorted_length.txt

- head -n 2 sorted_length.txt

to get the first two lines of sorted_length.txt.

Without any option

- head sorted_length.txt

just gives you a general sneak peak into the document.

The oppositive command to **head** is **tail**, which gives you the last lines.

To print something to the command line you can use

- echo

like

- echo Hello World

print Hello World to the shell.

With

- echo test > testfile.txt

you can write *test* to a file *testfile.txt*.

- echo another_test >> testfile.txt

appends the line *another_test* to *testfile.txt*.

To append commands to eachother you can use the **pipe** |

- sort -n length.txt | head -n 1

first sorts the content of the file and then directly gives you the first entry.

Lesson 4: Loops

To apply the same command or commands to multiple files, **loops** can be used to automate this.

As an example you can type

- head -n 5 file1.txt file2.txt file3.txt

will give you the first 5 lines from each of these files.

The general structure of a loop is:

- for filename in file1.txt file2.txt file3.txt

where,
for is the keyword for a loop
filename is your "nickname" (parameter) for a file within the iteration
in is another keyword followed by all the files you want to loop over

After hitting enter for your loop, your terminal will change to > and then you can add

- > do
- > head -n 5 $filename | tail -n 1
- > done

Of course, you can exchange the third line for anything you can imagine :) Just start with **do** and end with

done.

Another example :

- for datafile in NENEA.txt NENE*A.txt

to loop over all the NENE-something files that end with B or A.

- > do
- > echo $datafile
- > done

Will give you all of the file names that end with A or B.

Within your loop, you can always use $datafile as a piece of text, e.g. if you write echo abc-$datafile it will put abc- in front of every filename.

Hint: to check, if your potential command will be correct, you can check with

- > echo "cat $datafile >> all.pdb"

which will then give you e.g.

- cat NENE01729A.txt >> all.pdb

printed to the screen instead of running the command directly.

Lesson 5: Shell Scripts

The idea of a scripts is to package a list of commands into a small file, to not have to remeber all the commands and execute them manually but just run the whole script once. Scripts are also handy to send them to colleagues.

The create a script, type

- nano middle.sh

which creates a new file and opens this in the nano editor.

Then you can add your desired commands to the *middle.sh*, like

- head -n 15 octane.pdb | tail -n 5

Of course you can add any other commands here, depending on what you want to do.
Do not forget to save in nano with **CTRL+O** and close with **CTRL+X.**

You can than execute the script with

- bash middle.sh

To make scripts a bit more flexible, we will introduce **inputs** and change the script to e.g.

- head -n "$2" "$1" | tail -n "$3"

where $1 is the first input, $2 is the second input and $3 is the third input.

Hint: The "" around the inputs will ensure that you could use spaces in the input.
More information on when to use " " (double quotes**) and when to use ' ' (**single quote**) in bash, can be found here: https://stackoverflow.com/questions/6697753/difference-between-single-and-double-quotes-in-bash**

In this example, if you execute

- `bash middle.sh pentane.pdb 14 2`

would actually execute

- `head -n 14 pentane.pdb | tail -n 2`

Hint: If you want to add comments to your scripts (which is highly appreciated by your future self!), you can add them with # , so

- # this is a comment
- # this is another one

will be ignored while executing the script.

In the context of shell scripts, "$@" means all the command line arguments given when executing the script. Just like $1 takes the first argument $@ takes all arguments.

Nested loops in scipts and calling scripts from scripts:

To create a new script:

- `nano do-stats.sh`

- `for datafile in "$@"`
- `do`
    - `echo $datafile`
    - `bash goostats.sh $datafile stats-$datafile`
- `done`

Hint: ***bash goostats.sh $datafile stats-$datafile*** **means that** ***goostats.sh $datafile stats-$datafile*** **is that executed in the bash context.**

and then execute it with

- `bash do_stats.sh NENEA.txt NENE*B.txt`

This will then keep you sort of updated on the process, by printing out the name of the file it is currently working on.
Further it will have created a stats-NENE*A.txt and stats-NENE*B.txt for every file it has processed with the goostats.sh.

Lesson 6: Searching

To search for a word, you can use the command **grep**:

- `grep word-you-are-looking-for where-you-are-looking`

e.g.

- `grep not haiku.txt`

will search all the lines where "not" is present in haiku.txt. It will also put out places, where "not" is within a word.

To search for complete words, use the **-w** flag

- `grep -w "is not" haiku.txt`

will search for all the places where "is not" is used.

FEEDBACK FORM:

https://docs.google.com/forms/d/e/1FAIpQLSdB5kDPg4APRz1oU7s484vfL9VGZsl5hhmXGuTPGPAwd48DBQ/viewform

---