# lab notes

## Python scripting

Note that using Anaconda is not mandatory for the course, if you already have a python development environment set up that you are comfortable with, feel free to use that.

Anaconda has three basic features for interacting with python. All three have advantages and disadvantages, and are designed for different uses.

### Interactive interpreter prompt

ipython is an intractive python prompt, which is available on the command line, but spyder also has it built-in: start spyder, the python prompt in the lower-right corner uses ipython.

#### tab-completion

Start typing and press tab. For example, to get list of methods of some object:

```
In [2]: l = []
In [3]: l.
l.__add__ l.__eq__ l.__hash__ l.__lt__ l.__reversed__ l.__subclasshook__ l.remove
l.__class__ l.__format__ l.__iadd__ l.__mul__ l.__rmul__ l.append l.reverse
l.__contains__ l.__ge__ l.__imul__ l.__ne__ l.__setattr__ l.count l.sort
l.__delattr__ l.__getattribute__ l.__init__ l.__new__ l.__setitem__ l.extend
l.__delitem__ l.__getitem__ l.__iter__ l.__reduce__ l.__setslice__ l.index
l.__delslice__ l.__getslice__ l.__le__ l.__reduce_ex__ l.__sizeof__ l.insert
l.__doc__ l.__gt__ l.__len__ l.__repr__ l.__str__ l.pop
```

#### Use ? to explore an object:

```
In [5]: range
Out[5]:
In [6]: range?
Type: builtin_function_or_method
Base Class: <type 'builtin_function_or_method'>
String Form:
Namespace: Python builtin
Docstring:
range([start,] stop[, step]) -> list of integers
 Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3]. The end point is omitted!
These are exactly the valid indices for a list of 4 elements.
```

Note that if the documentation is long, you will not get the prompt back, you will have to press 'q' to close the documentation and go back to the prompt

## Spyder IDE

Edit script in the main area. When saving, use the file extension .py . To run a script, use the green triangle icon on the toolbar. To run only a few lines, select those and use the icon next to the green triangle. The script will be run in the interactive interpreter, so output will be printed there, and any variables or functions defined in the script will be accessible in the prompt afterwards.

## Jupyter notebook

To open a new notebook, run "jupyter notebook" from the command line, or use the appropriate icon. The notebook will open in a browser. Make sure to not close the command line window or exit the jupyter process as long as you use the notebook, since the notebook only provides the user interface, the code is actually run in a python process.

The notebook interface can be considered mix of the interactive interpreter and the script editor: input can be run in-line, but it is easier to go back and edit previous "cells" and handling multi-line cells is also easier.

Since cells can contain multiple lines, to close the current one and step to the next one (and, for cells containing python code, run the code), press Alt-enter.

Output of code in code cells is included after the cell, not only for things printed out, but also for plots (This is configurable in the settings; changing the backend in the preferences to an interactive one will use popup windows for the plots) .

All cells are editable (as opposed to the interactive interpreter, where you can't go back and edit a previous line), but make sure the re-run the given cell, or all cells below that (using Cell -> Run or "Run All Below" from the toolbar at the top of the notebook) so results will show the effects of the modification.

Note that when a cell in the middle of a notebook (i.e. there are cells before and after it) is edited and run, that won't update the output or state of other cells. Thus to get consistent results, you might need to run all cells from the beginning (and possibly also restart the kernel).

Notebooks can be saved as .ipynb format (which is the native, editable format), but can also be exported to .html files (which are not editable). To export the notebook to html, use File -> Download as -> HTML from the toolbar near the top of the notebook page.

# Introduction to networkx

see more detailed documentation at: https://networkx.github.io/documentation/stable/index.html, for example the tutorial at https://networkx.github.io/documentation/stable/tutorial.html

## Creating networks in networkx

creating an empty network (this will be a simple, undirected graph. See also: networkx.DiGraph, networkx.MultiDiGraph, etc.):

```
import networkx
g = networkx.Graph()
```

Adding nodes and edges:

```
g.add_node(1)
g.add_edge(1,2)
```

Note: when adding edges, missing nodes will be automatically added. Nodes can be anything that can be used as a key in a python dictionary, so for example numbers and strings.

Node and edge attributes can be added as well, simply by passing in additional attributes:

```
g.add_node(3, some_atttr=42)
g.add_edge(2,3, some_other_attr=137)
```

These attributes can be accessed via g.nodes and g.edges, as well as directly from the graph object:

```
g.nodes[3]
g.edges[2,3]
g[2][3]['some_other_attr']
```

Which also allows modifying these attribute values simply by assigning to them:

```
g[2][3]['some_new_attr'] = 'asdf'
```

### Iterating over nodes / edges:

```
graph.nodes() # an iterable of nodes
graph.edges() # returns iterable of edges
graph.edges(data=True) # returns iterable of edges, including the edge attributes

# to iterate over the edges, also accessing edge attributes:
```

```
for from_node, to_node, attributes in g.edges(data=True):
    edge_weight = attributes['weight']
    # do something with from_node, to_node and edge_weight
```

## Drawing networks in networkx

```
networkx.draw(g)
plt.show()
```

Note: make sure to import matplotlib first. "plt.show()" is often not needed in jupyter prompt, but is needed when running a python script.

networkx.draw() will automatically run a default layout to arrange the nodes. If you want to set parameters of this layout, or change which algorithm is used, you can call one of the functions in networkx.layouts, and pass the resulting dictionary (mapping nodes to coordinates) to the "pos" parameter of networkx.draw().

To export the image (i.e. instead of showing it, save it to a file), instead of calling plt.show(), use plt.savefig(). Pass in the filename as a string, the file format will be determined automatically based on the extension.

## Reading data into networkx

To read a list of edges into networkx, use:

```
graph = networkx.read_edgelist('word_association_graph_DSF.txt', create_using=networkx.DiGraph(), nodetype=str, data=[('weight', float),])
```

Where we are also specifying that we want a directed network (with the create_using parameter), that our node identifiers are strings (nodetype parameter), and that the third column of our file contains the edge weights as floats.

# Python / Jupyter features mentioned during the lab

## For loops

to iterate over a list or similar object, and do something for each element, use:

```
for element in elements:
 ... do something with element ....
```

If each element is actually a tuple or a list, the following is a nice pattern:

```
for a,b,c in elements:
 .. do something with a, b & c ....
```

### Switching between in-line and interactive plotting in a jupyter notebook

Running the magical command "%matplotlib" will switch to displaying plots in a new window, with some interactive features (like zooming). To switch back, run "%matplotlib inline"

### Hide the output of plt.hist() to avoid obscuring the plot and forcing long scrolling

The notebook interface will print the result (returned value) of function calls (if the last line in the given cell is a function call), which might be unwanted. To avoid having it printed above the in-line plot, simply assign the return value to some variable, i.e. "_ = plt.hist(...)" (see examples in the lab session jupyter notebook)

## Plotting the in-degree and out-degree distributions

For plots and code, see the jupyter notebook

### Selecting the number of bins

When plotting a binned histogram, one important task is to select the correct number of bins to use: too few, and we lose too much information because we are averaging too much: the resolution of the plot won't be very good, it will be too "blocky". Too many, and we will get either a noisy histogram (because we are not averaging enough), or even empty bins between non-empty one (where we had no datapoints falling into a given bin)

### Using logarithmic axis for plots

The in-degree distribution has values over a large range. Plotting it with linear axis makes all the datapoints get squeezed in the lower left corner of the plot. In such cases, allways use logarithmic axis to make the data more visible. For plt.hist(), use the "log=True" to set the vertical axis to logarithmic. To set the horizontal axis to logarithmic as well, use plt.xscale('log'). ( For plt.plot(), use plt.yscale('log') and plt.xscale('log'), respectively.)

### Describing the shapes of the distributions

For the out-degree distribution, we noted that the spike at the left hand side looks seperate from othe rest of the distribution: checking it in detail, we can see that this is due only to nodes with degree zero.

For the in-degree distribution, we didn't want to call the spike near zero to be a seperate part of the distribution: although it looks very large (especial with linearly scaled axes), it appears to be part of the smooth curve of the distribution.

## Python code written during the lab

See on the lab slides.