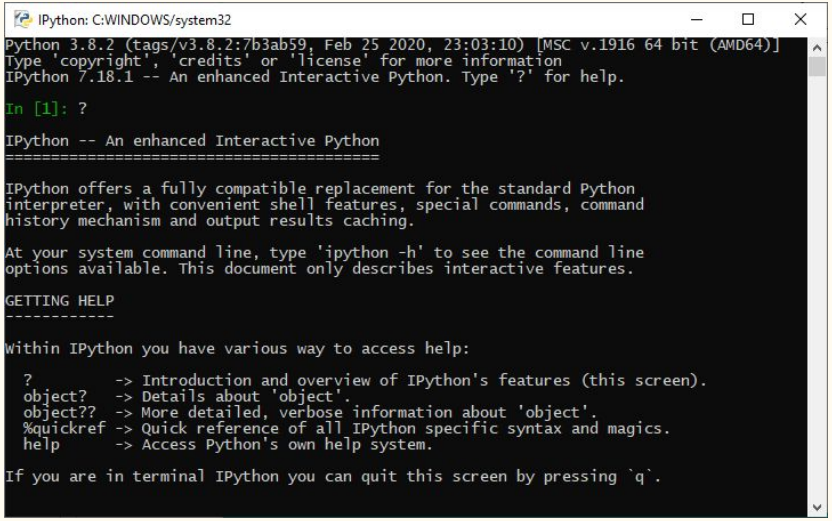
IPython and Jupyter

Janusz Jabłonowski, MIMUW

IPython

- An enhanced Interactive Python
- It started in 2001
- Invoking (from command line)

  `ipython`
- Getting help
  - -h in command line
    - description of command line parameters
  - ? for commands within IPython
  - history of commands
  - tab completion for names
  - syntax highlighting
- `quit` ends



IPython - syntax extensions

- IPython offers syntax extensions
  - they might be considered handy
  - but they are incompatible with Python!
  - hence I do **not** recommend using them
- Getting help
  - ? - general information
  - %quickref - commands quick reference
  - help()
- Auto-parentheses (sample extension)
  - %autocall 1
  - print 1,2 becomes print(1,2)

IPython

- Offers invoking shell commands

  !ls

- Lets define aliases

  %alias alias_name cmd

- Has a set of built in magic functions
- %magic - (long) description of magic functions and their usage

IPython - magic

- Convenient way to measure the execution time of command
- For simple commands

  `%timeit <command>`

```
In [69]: %timeit pass
8.61 ns ± 0.407 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)

In [70]: %timeit x = 100
15.3 ns ± 0.552 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)

In [71]: %timeit range(100)
169 ns ± 3.58 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [72]: %timeit range(10000)
212 ns ± 5.36 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [73]: %timeit list(range(100))
688 ns ± 20.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

IPython - magic

- For series of commands - cell magics

  ```
  %%timeit <command>
  <commands in multiple lines>
  ```

```
In [81]: %%timeit k=0
    ...: for i in range(100):
    ...:     k+=i
    ...:
4.72 µs ± 232 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

- The commands in the first line set the environment and are not being measured

Exercise

Measure on your machine and in your implementation average time of testing if a list is empty.

```
In [18]: timeit 1 if lst else 0
38 ns ± 2.49 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [19]: timeit 1 if lst else 0
40.7 ns ± 3.28 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [20]: timeit 1 if lst == [] else 0
55.2 ns ± 2.92 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [21]: timeit 1 if lst == [] else 0
55.7 ns ± 2.63 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [22]: timeit 1 if len(lst) else 0
82 ns ± 5.05 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [23]: timeit 1 if len(lst) else 0
80.9 ns ± 4.74 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Jupyter

- Jupyter project
  - started in 2014
  - by people from the IPython project
  - open source
  - language independent platform for interactive calculations
  - https://jupyter.org/
  - Name to honour Java, Python, R and Gallileos observations of Jupiter moons
- (Jupyter) notebook
  - interactive document with text, code, graphs, and results.
- Kernels
  - backend for notebooks
  - for various programming languages
  - implement Jupyter interactive computing protocol
  - there are many of them (several dozen)

More about Jupyter

- Support for data visualization

Installing

- conda

conda install -c conda-forge jupyterlab

- pip

pip install jupyterlab

- pipenv

pipenv install jupyterlab

pipenv shell

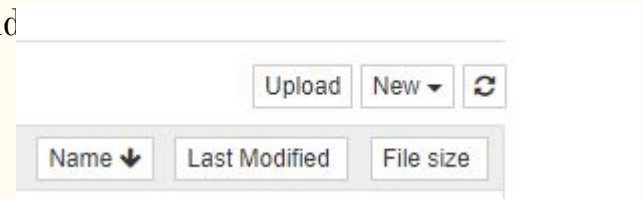
Jupyter vs Jupyterlab

JupyterLab

- The next generation of the Jupyter Notebook
- Uses the exact same file format as the classic Jupyter Notebook
- Is fully compatible with the existing Jupyter notebooks and kernels
- The classic Notebook and Jupyterlab can run side to side on the same computer
- Allows for notebook, console, terminal in consecutive tabs
- Returning to classic notebook: replace lab for tree in the page address

Jupyter

- Starting
  - `jupyter notebook`
  - Or
  - `jupyter lab`
  - on most systems starts Jupyter in a browser (https://localhost:8888)
  - it opens jupyter in current folder
  - note: while it is possible to change the folder it is not possible to change the drive :((
  - it is even possible to share notebooks over internet
- Creating the first notebook
  - click `New` in the right upper corner of the browser wind



Working with Jupyter

- Notebook structure
  - cells for code and results
  - cell contain (Python) code
  - `Shift/Enter` to execute the code in the current cell (or `Run` at the toolbar)
  - Jupyter calculates and displays the result
  - for multiple lines: all lines are executed, result of the last one is displayed
  - the value None is not displayed
  - functions and variables from other cells are visible
  - cells can be edited
  - cells may be run many times

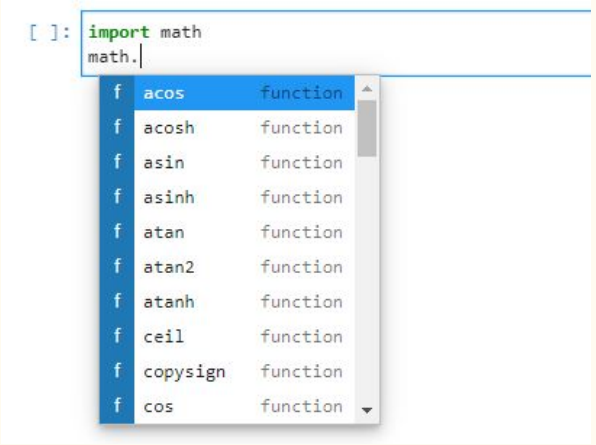
Saving and notebook file format

- Saving
  - File>Save and checkpoint (Ctrl-S)
    - it possible to revert to a checkpoint
  - File>Save as - saves with a given name
- Saved notebooks may be opened in other instances of Jupyter
- Checkpoints are stored in the `.ipynb_checkpoints` directory
- Format
  - extension .ipynb
  - text format (json) hence well suitable for git

Saving and notebook file format

- Other save formats
  - File>Download as
    - Python - program ready to run
    - Html - nice looking static page
    - Latex - human-unreadable but correct
    - Pdf - from .tex file, works fine on Linux, has / vs. \ problems on Windows, generates nice looking pdf files

Tab completion



- Is working!
- Works for built-in and user defined names
- Helps with keyword parameter names
- Works across cells
- Works also for paths (even in strings)
- Distinguishes lowers and upper case letters (in paths too)

Object introspection

- Question mark before/after a name
- Displays information about that variable/function
- See example on the next slide

Object introspection

```
[12]: lst = [1,2,3]
      lst?
```

```
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

```
[11]: print?
```

```
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

Object introspection

- For functions ? displays also their docstring
- Double question mark tries to display the source code (if available)

Object introspection

[23]:
```
def my_factorial(n=10):
    """Caluclates factorial of n, independently from the official factorial implementation"""
    res = 1
    for i in range(2,n+1):
        res *= i
    return res
my_factorial(5)
```

[23]:
```
120
```

[26]:
```
my_factorial?
```

```
Signature: my_factorial(n=10)
Docstring: Caluclates factorial of n, independently from the official factorial implementation
File:      m:\zajęcia\narzędzia\jupiter\<ipython-input-23-dc230989d16e>
Type:      function
```

[25]:
```
my_factorial??
```

```
Signature: my_factorial(n=10)
Source:
def my_factorial(n=10):
    """Caluclates factorial of n, independently from the official factorial implementation"""
    res = 1
    for i in range(2,n+1):
        res *= i
    return res
File:      m:\zajęcia\narzędzia\jupiter\<ipython-input-23-dc230989d16e>
Type:      function
```

Object introspection

- Also wildcard (*) can be used
- For example *.*os*? Gives a list of all available names from modules and having the sequence ‘os’ inside.

Object introspection

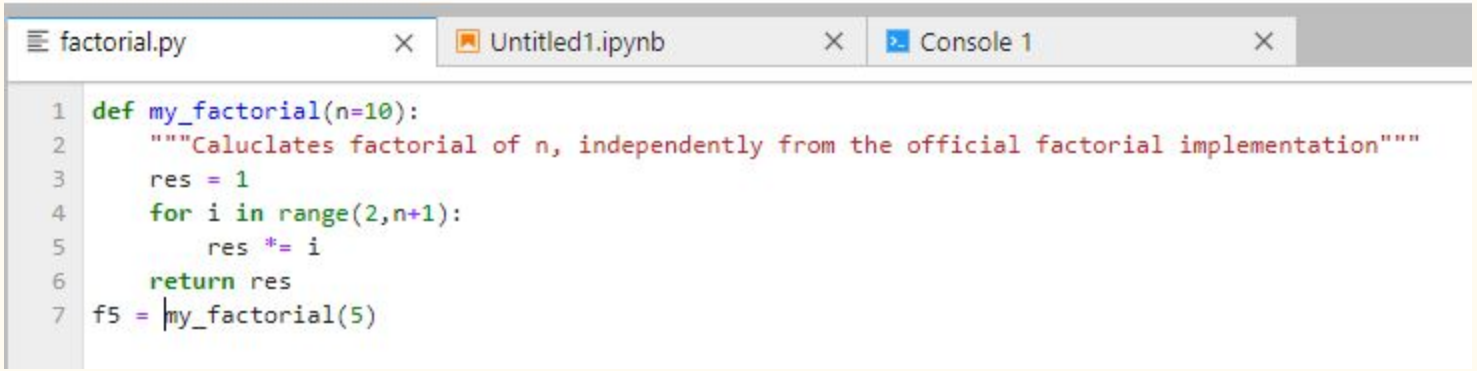
```
[39]: *.*os*?

False.__pos__
True.__pos__
__.__pos__
__IPYTHON__.__pos__
___.__pos__
__debug__.__pos__
bool.__pos__
complex.__pos__
display.__closure__
float.__pos__
int.__pos__
math.acos
math.acosh
math.cos
math.cosh
math.isclose
my_factorial.__closure__
```

Running scripts

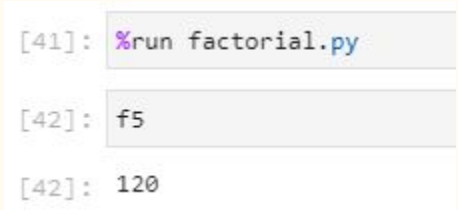
- The run command allows for calling scripts residing on disk

Running scripts

- Let's create a simple script (File>New>Text file)



- It can called and variable there defined used as follows



Running scripts

- Also command line parameters can be used

sum.py

```
1  import sys
2
3  def sum_arg():
4      res = 0
5      for elt in sys.argv[1:]:
6          res += int(elt)
7      return res
8
9  m = sum_arg()
10
```

- It can called and variable there defined used as follows

```
[48]: %run sum.py 12 4 8 123

[49]: m

[49]: 147
```

Running scripts

- Normally scripts are run without access to variables in calling notebook
- The -i option (`%run -i`) gives them access to notebook variables
- The `%load path` command allows for loading scripts
- The Ctrl/C key combination allows for breaking the execution of the invoked scripts (results in KeyboardInterrupt being raised)
- In very rare cases it may not work (Python code called from compiled modules), then the Python process killing may be used as the last resort

Plotting (graphs)

- Jupyter supports user interface integration with some important libraries
- First of all `matplotlib`!
- Before calling plotting from this library the magic function `%matplotlib` with parameter has to be called (otherwise results will not be visible)

Matplotlib

- Library for plotting graphs
- Over 70k lines of code
- Started in 2002 (to enable MATLAB-like plotting)
- Supports various backends
- Exports graphs in various formats (like pdf, jpg, svg, etc.)

Installing `matplotlib`

- Best install anaconda - it contains matplotlib
- To install separately issue the command

```
pip install matplotlib
```

- Usually imported as follows

```
import matplotlib.pyplot as plt
```

`pyplot`, `pylab`, `matplotlib` - how not to get confused (too often)

- **matplotlib** is the whole package
- **matplotlib.pyplot** is a module (in matplotlib)
- **matplotlib.pylab** is a module (in matplotlib)

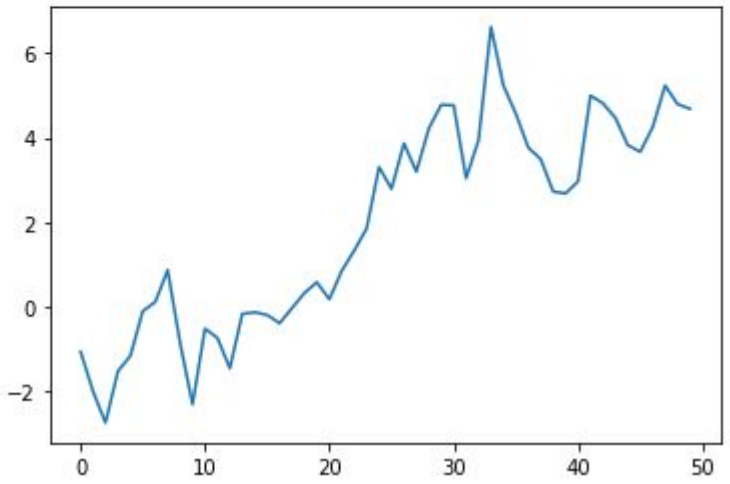
- pyplot - interface to the underlying plotting library in matplotlib
- pyplot interface is generally preferred for non-interactive plotting (scripts)
- pylab combines pyplot and numpy in a single namespace
- pylab interface is convenient for interactive calculations and plotting
- For ex-MATLAB users
- Not recommended nowadays
- The ipython -pylab option imports everything from pylab and makes plotting fully interactive

Simple example of plotting

```
%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

plt.plot(np.random.randn(50).cumsum())
```

Simple example of plotting

```
In [10]: %matplotlib inline
         import matplotlib.pyplot as plt
         import numpy as np

In [11]: plt.plot(np.random.randn(50).cumsum())

Out[11]: [<matplotlib.lines.Line2D at 0x157146afb50>]
```



Some examples

- The following examples assume that the following code has been executed

`%matplotlib inline` (or `%matplotlib notebook`)

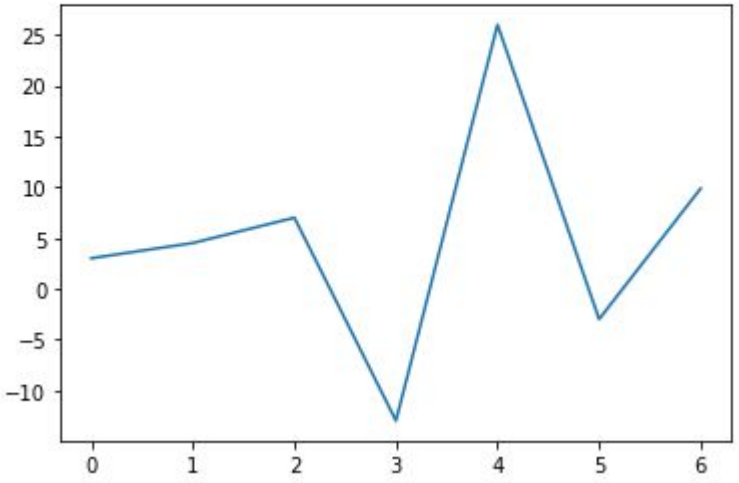
```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib
```

Some examples

- One-line plot

```
plt.plot([3, 4.5, 7, -13,
          26, -3, 9.87])
```

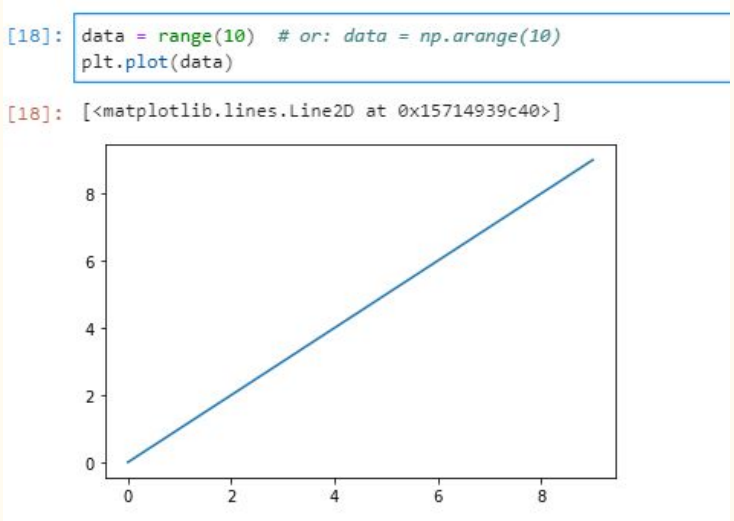
```
[21]: plt.plot([3, 4.5, 7, -13, 26, -3, 9.87] )

[21]: [<matplotlib.lines.Line2D at 0x157149f0b20>]
```



Some examples

- One of the simplest plots (just a line)

```
data = range(10) # or data = np.arange(10)
plt.plot(data)
```



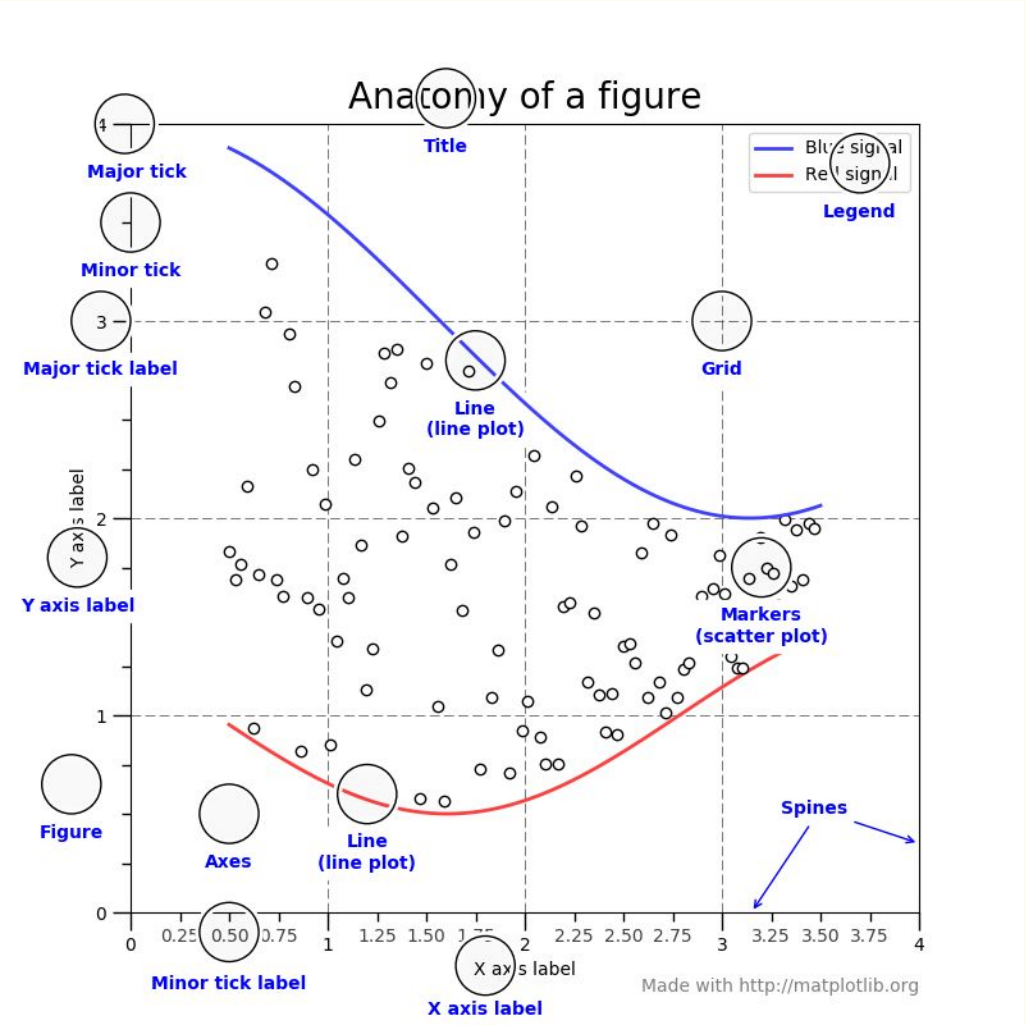
(Some) objects

- `matplotlib` is based on a hierarchy of objects
- But we do not cover OOP in this lecture
- Hence we'll try to avoid talking about objects, they are just *things* after all
- The Figure *thing* (aka *object*) represents the entire plot
- It usually contains several Axes *things*
- The name Axes is highly confusing, it does **not** mean plural of axis but something like one plot (which does have axes, sure, but also much more)
- In general a Figure *thing* contains nested structure of other things (like folders)
- Using this structure, dot notation and Jupyter hints it is possible to generate quite sophisticated plots

An anatomy class

Source:

https://matplotlib.org/examples/showcase/anatomy.html



Two notations

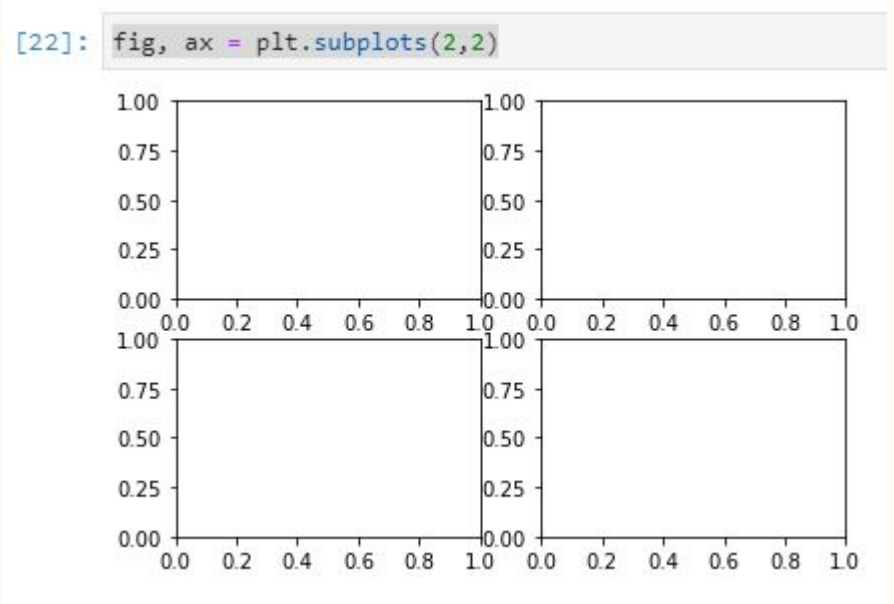
- `matplotlib` uses two notational styles
- From OOP (dot notation)
- Calling global functions operating on the global state
- The latter results in shorter code
- It assumes that there is the current figure thing, the current axes thing etc.

Let’s plot (graphs)

- We can easily create a plot consisting of many graphs

fig, ax = plt.subplots(2,2)

- Four charts were created
- fig is a figure, ax an numpy’s array of four axes

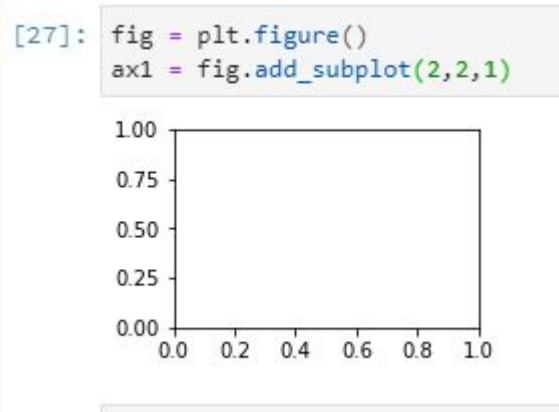


Let’s plot (charts)

- Plots can be also added one by one

```
ax1 = plt.subplots(2,2,1)
```

- There will be up to 4 charts, we address the first one (to add confusion they are numbered from 1)

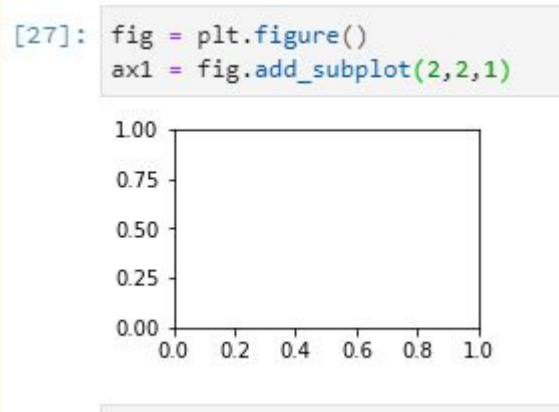


Let’s plot (graphs)

- Plots can be also added one by one

```
ax1 = plt.subplots(2,2,1)
```

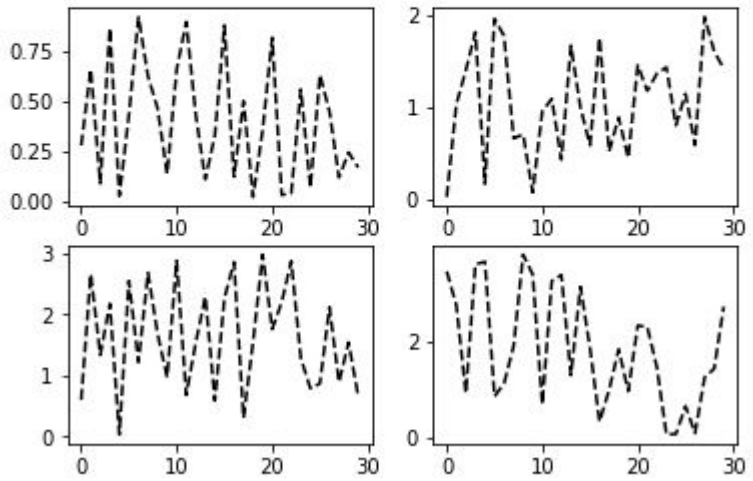
- There will be up to 4 charts, we address the first one (to add confusion they are numbered from 1)



Let's plot (in Jupyter)

- These enables for easy generating various graphs combined together (for example for comparison)
- k-- things are not what they seem to be

```
[44]: import random
      fig = plt.figure()
      for j in range(1,4+1):
          ax = fig.add_subplot(2,2,j)
          plt.plot([random.random()*j for i in range(30)], 'k--')
```



Let’s plot (for the future)

- This presentation just scratched the surface of the fascinating possibilities offered by matplotlib

Solving Jupyter annoyances [extension]

- It can be annoying - you can select files but cannot change the drive (!)
  - https://github.com/jupyter/notebook/issues/1334
  - `mklink /D "D_Drive" "D:\sources"`
- It can be annoying - poor clipboard integration
  - Copy using keyboard shorthands, not pop-up menu

A few remarks about debuggers and profilers

- History of the name (debugger)
- The dichotomy of debugging
  - Should we use debuggers?
  - We shouldn’t need to!
  - But we badly need ...
- Important notions
  - breakpoints
  - conditional breakpoints
- The need for profilers
  - not for entire code

Thank you for your attention!