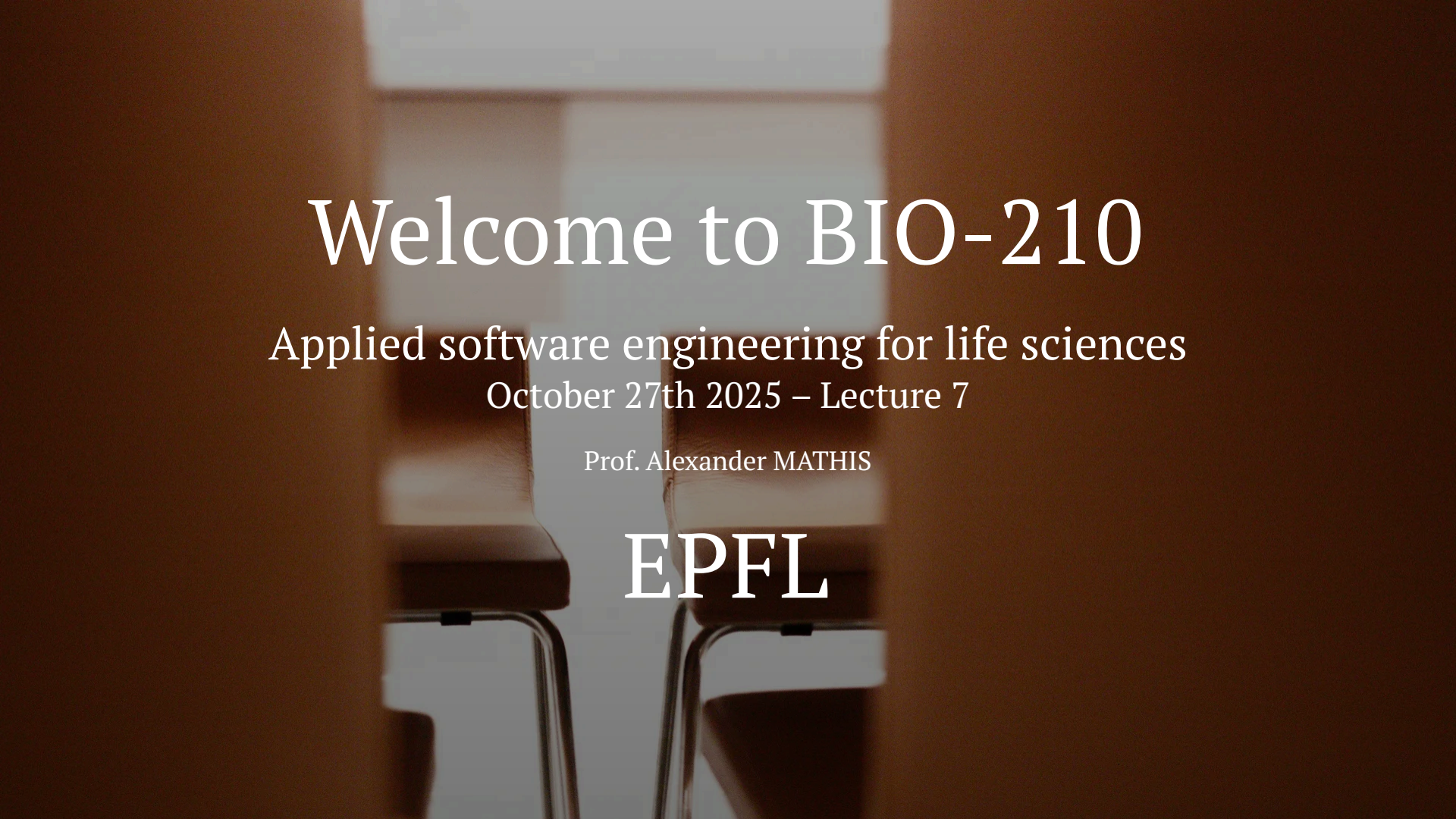


Welcome to BIO-210

Applied software engineering for life sciences

October 27th 2025 – Lecture 7

Prof. Alexander MATHIS

EPFL

Announcements I

- Today the second quiz starts at 3pm --> Moodle; you have time until Friday (at midnight) to fill it out!

Announcements II

- v2 of your project was due at 10am today (not graded, but reviewed), check out release guide.
- Final room assignment - Please note that we altered the room assignment for the exercises slightly. Check here. NOTE: you need your EPFL login to see it!
- try to do the majority in the exercise session (we share the problem set in advance, at least by Friday, so you can prepare better)
- what if you get stuck? -> discuss with your teammates, ask on ED, … and always *release* your best version on Monday at 10am
- make sure you get feedback about your latest version on Monday from the SA/TAs! You can release a bugfix/patch/update, e.g. `v2.1`, see details on releases
- We will provide code review for your v2. Incorporate the feedback as soon as you see it. How? Let's discuss this.

What is a code review?

A **code review** is when someone examines your code to provide feedback on:

- Code quality and readability
- Potential bugs or issues
- Design decisions and architecture
- Adherence to project standards

Why are code reviews useful?

- Catch bugs before they reach production
- Share knowledge across the team
- Maintain code quality and consistency
- Learn from each other’s approaches

How to address a code review?

When you receive feedback:

1. **Read carefully** - understand the reviewer's concerns
2. **Ask questions** - if feedback is unclear, request clarification
3. **Make changes** - implement suggested improvements
4. **Respond** - explain your changes or reasoning for not making changes
5. **Be professional** - feedback is about the code, not you!

Good response example:

```
Good catch! I've updated the function to handle edge cases.
See commit abc123.
```

Remember: Code reviews make everyone better programmers!

What is a GitHub Issue?

An Issue is:

- A way to track bugs, features, or tasks
- A discussion thread for your project
- Documentation of what needs to be done

Examples: Numpy Issues, DCL2action

When to create an issue?

- You find a bug 🐛
- You have a feature idea 💡
- You want to discuss an improvement 🚀
- You need to track a task ✓

GitHub Issue Example: Bug Report (made up)

Title: `Data loading fails for CSV files with special characters`

Description:

```
## Bug Description
The `load_data()` function crashes when CSV files contain
special characters (é, ñ, ü) in column headers.

## Steps to Reproduce
1. Create CSV with column name "Température"
2. Run `data = load_data('test.csv')`
3. Program crashes with UnicodeDecodeError

## Expected Behavior
Should load CSV files with any UTF-8 characters

## Actual Behavior
`UnicodeDecodeError: 'ascii' codec can't decode byte...`

## Environment
- Python 3.11
- pandas 2.0.0
- macOS 14.0

## Proposed Solution
Add `encoding='utf-8'` parameter to `pd.read_csv()`
```

Real, recent example: Likelihoods in RTMpose models in DeepLabCut.

GitHub Issue Example: Feature Request (made up)

Title: `Add plotting function for gene expression data`

```
## Feature Request
Add a function to visualize gene expression levels across samples

## Use Case
Users need to quickly visualize their results without writing custom plotting code each time.

## Proposed API
plot_expression(data,genes=['BRCA1', 'TP53'],samples=['tissue1', 'tissue2'],output='expression_plot.png')

## Acceptance Criteria
- [ ] Function creates bar plot or heatmap
- [ ] Supports multiple genes and samples
- [ ] Saves to file or displays in notebook

## Additional Context

Similar to what seaborn.heatmap() does but specialized for our data format.
```

GitHub Issue Example: Task/TODO (made up)

Title: `Update documentation for v2 release`

Description:

```
## Task
Update all documentation files for the v2.0 release

## Checklist
- [ ] Update README.md with new features
- [ ] Add docstrings to new functions
- [ ] Update installation guide
- [ ] Add examples for new API

## Deadline
Before October 29, 2025 (release date)

## Assigned to
@AlexEMG (that's me on GitHub)

## Related
- Relates to #156 (v2 feature list)
- Blocks #160 (v2 release)
```

Best Practices for Issues

Good issue titles: ✅ `Data loading fails for files >1GB` ✅ `Add support for Python 3.13` ✅ `Improve performance of distance calculation`

Bad issue titles: ❌ `Bug` ❌ `Help!!!` ❌ `It doesn't work`

Tips:

- Be specific and descriptive
- Include relevant code/error messages
- Add labels (bug, feature, documentation)
- Reference related issues with #number
- Use task lists for complex issues
- Keep discussion focused and respectful

Pull Requests (PRs) and Code Reviews

A Pull Request is:

- A request to merge your code into the main branch
- The primary mechanism for code review on GitHub
- A place for discussion about proposed changes

Basic PR workflow (also see previous exercises):

1. Create a branch for your feature
2. Make commits to your branch
3. Open a Pull Request on GitHub
4. Team members review and comment
5. Address feedback and push updates
6. (Ideally) PR gets approved and merged. Otherwise close it.

Linking PRs and Issues

Why link them?

- Track which code addresses which problems
- Provide context for reviewers
- Automatically close issues when PRs merge

How to link?

In commit messages:

```
git commit -m "Add data validation function

Fixes #42 by adding input validation"
```

You can also put such links in the PR description. GitHub will automatically link and close issues when the PR merges (e.g. see this issue). You can also cross-link by putting hyperlinks: example. Both of these mechanisms support working in teams.

Useful conventions for developing a project

New feature development:

- for new features make branches. Give the branch a good name, e.g. *your_name/novel_featurename*
- once you're ready you make a pull request and assign your collaborators for review (e.g., see this example PR)
- here is an example for the demo project

✅ Good names:

- `alex/add-plotting`, `feature/gene-expression-viz`, `bugfix/csv-encoding`, `docs/update-readme`

❌ Bad names:

- `test`, `new-branch`, `fix`,...

Why? Clear names help teammates (and your future self) understand what you're working on!

Remember, for our automatic grading (which will start soon!)

- make sure the application programming interface (API) follows the specification in the problem sets
- if you have additional parameters, make sure they have defaults

🧩🤔: Calculating statistics for arrays?

You have a dataset representing the expression levels of 5 genes across 4 tissue samples. Each row corresponds to a gene, and each column corresponds to a tissue sample.

```
import numpy as np
expression_levels = np.array([
    [5.1, 2.3, 3.4, 6.5],  # Gene 1
    [1.5, 3.5, 2.4, 4.6],  # Gene 2
    [3.2, 5.1, 1.6, 3.8],  # Gene 3
    [4.1, 3.2, 4.5, 2.2],  # Gene 4
    [2.8, 1.5, 3.1, 5.0],  # Gene 5
])
```

Write a program to calculate the standard deviation of the expression levels of each gene.

```
In [1]: np.std(expression_levels,axis=1)
Out[1]: array([1.60370664, 1.16404467, 1.25772612, 0.88600226, 1.2509996 ])

In [2]: np.sqrt(np.var(expression_levels,axis=1)) #if you don't know the std command
Out[2]: array([1.60370664, 1.16404467, 1.25772612, 0.88600226, 1.2509996 ])
```

Comments in Python code

We already learned that "#" allows to put comments in code.

```
# Hello world <--- this will not be interpreted!
# "#" also allows multi-line comments

# we have also seen inline comments...
a=3       # you can also make inline comments!
b=4       # assigning b to 4.

''' single quotes
You can also make long comments ... everything is "ignored" by python!
a=f2d123ee1505
b=123
'''
c=a+b

""" quotation marks
Alternative,
multiline comment
"""
```

Some guidelines (not rules)

From pep 8 = Style Guide for Python Code

- Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers, aka keywords, module names, etc.).
- Ensure that your comments are clear and easily understandable to other speakers of the language you are writing in.
- You can look for typos by using the pip package codespell.
- **Comments that contradict the code** are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

P.S.: PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. The PEP should provide a concise technical specification of the feature and a rationale for the feature – from PEP 1.

To comment or not to comment?

```
# Bad: obvious comment
x = x + 1  # Increment x

# Good: self-documenting code
sample_count = sample_count + 1

# Good: explain WHY, not WHAT
# Use log scale because expression values span several orders of magnitude
expression_log = np.log10(expression_data)
```

🧩🤔: How do you create a np.array ...

... starting at 12, ending at 176 containing every third number?

```
In [1]: import numpy as np
In [2]: np.arange(12,177,3)
Out[2]:
array([ 12,  15,  18,  21,  24,  27,  30,  33,  36,  39,  42,  45,  48,
        51,  54,  57,  60,  63,  66,  69,  72,  75,  78,  81,  84,  87,
        90,  93,  96,  99, 102, 105, 108, 111, 114, 117, 120, 123, 126,
       129, 132, 135, 138, 141, 144, 147, 150, 153, 156, 159, 162, 165,
       168, 171, 174])
```

But what if you forgot how to use `np.arange`?

```
help(np.arange)
np.arange? # in IPython/Jupyter
```

The displayed help is actually the `docstring`

```
In [3]: help(np.arange)

Help on built-in function arange in module numpy:

arange(...)
    arange([start,] stop[, step,], dtype=None, *, like=None)

    Return evenly spaced values within a given interval.

    Values are generated within the half-open interval ``[start, stop)``
    (in other words, the interval including `start` but excluding `stop`).
    For integer arguments the function is equivalent to the Python built-in
    `range` function, but returns an ndarray rather than a list.

    When using a non-integer step, such as 0.1, the results will often not
    be consistent.  It is better to use `numpy.linspace` for these cases.

    Parameters
    ----------
    start : integer or real, optional
        Start of interval.  The interval includes this value.  The default
        start value is 0.
    stop : integer or real
        End of interval.  The interval does not include this value, except
```

Essential documentation: Docstrings

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

- Docstrings provide help for your code (so you (and others) can re-use it in the future!)

```
In [4]: def myfun(x):
   ...:     ''' identity function '''        # Docstrings are defined like comments!
   ...:     return x
   ...:
   ...: print(myfun.__doc__)       # Docstrings are assinged to the attribute `__doc__`
   ...:
 identity function

 In [5]: help(myfun)               # They become accessible via help!
 Help on function myfun in module __main__:

 myfun(x)
     identity function
```

Python’s recommendations for docstrings

- Write docstrings for all public modules, functions, classes, and methods.
- Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.
- PEP 257 immortalizes Python’s docstring conventions
- For mathematical functions (like in our projects) the detailed numpy style guide is excellent to follow

What should be contained in a docstring?

- A **Short summary** (for basic, simple functions)

```
1   def add(a, b):
2       """The sum of two numbers.
3
4       """
```

- **Extended summary** contains among others:
  - a simple description (clarify functionality, not implementation details those belong to Notes)
  - parameters
  - returns
  - examples
  - notes
  - references

Details are available in the Numpy doc style guide

Example (*shortened* from np.arange)

```
In [13]: np.arange?
Return evenly spaced values within a given interval.
!!OMMITTED for space reasons!!
Parameters
----------
start : integer or real, optional
    Start of interval.  The interval includes this value.  The default
    start value is 0.
stop : integer or real
!!OMMITTED for space reasons!!

Returns
-------
arange : ndarray
    Array of evenly spaced values.
!!OMMITTED for space reasons!!

Examples
--------
>>> np.arange(3)
array([0, 1, 2])
Type:      builtin_function_or_method
```

Documentation

- Note that the (numpy) docstrings are also (html-rendered) on the web, e.g., for np.arange
- this is all automatically generated with Sphinx, see https://github.com/numpy/doc

Docstrings in action

- Today you will work on docstrings for your functions!
- Compare to the demo-project

🧩🤔: How do you define a function that can offset the output by a specific parameter with default 2?

```
In [1]: def f(x,offset = 2):
   ...:     return x+offset
   ...:

# Testing our function:
In [2]: f(0)
Out[2]: 2

In [3]: f(3)
Out[3]: 5

In [4]: f(2,5)
Out[4]: 7
```

🧩🤔: How do you write docstrings for this function?

```
def offsetter(x,offset = 2):        # use a good name!
    """ Function that offsets input by default value (offset)
    Parameters
    ----------
    x      : array or float
    offset : float, optional
             default 2
    Returns
    -------
    numpy array or float
         x + offset
    Examples
    -------
    >>> offsetter(2)
    4
    >>> offsetter(2,3)
    5
    """
    return x+offset
```

Questions?

Visualization

- is crucial in science and beyond ("a picture is worth 1000 words...")
- python has strong support for plotting with maptlotlib (our focus), seaborn (neat interface on top), Majavi (esp. 3D visualization), Plotly (esp. web), Bokeh (esp. web), Pandas, gnuplot, ...

P.S. A formula is worth a thousand pictures... (by Edsger W. Dijkstra)

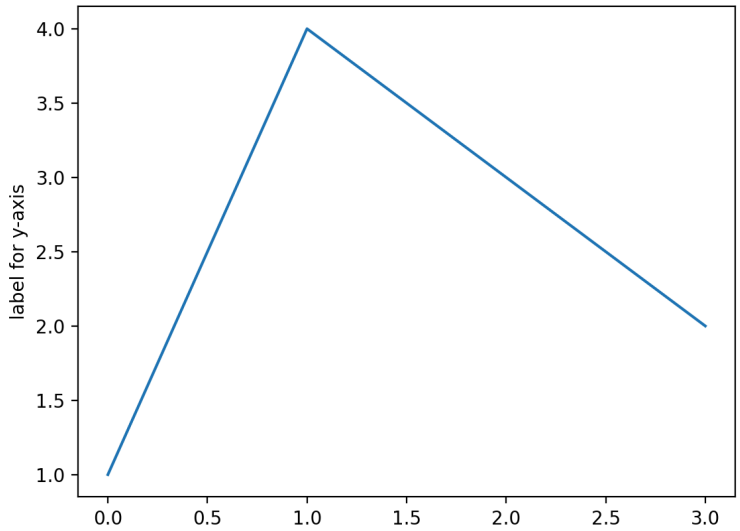
P.P.S. Just like Numpy, maptlotlib is a library you need to install --> `pip install matplotlib`

Pyplot: simple plotting in matplotlib

```
1   import matplotlib.pyplot as plt     # Importing matplotlib.pyplot
2   # Note: we use all functions from this library with plt.XYZ
3   plt.plot([1, 4, 3, 2])              # Plotting x vs. y data
4   plt.ylabel('label for y-axis')      # Making a label for y
5   plt.show()                          # Display all open figures
```

Pyplot: simple plotting in matplotlib

```
1   import matplotlib.pyplot as plt     # Importing matplotlib.pyplot
2   # Note: we use all functions from this library with plt.XYZ
3   plt.plot([1, 4, 3, 2])              # Plotting x vs. y data
4   plt.ylabel('label for y-axis')      # Making a label for y
5   plt.show()                          # Display all open figures
```

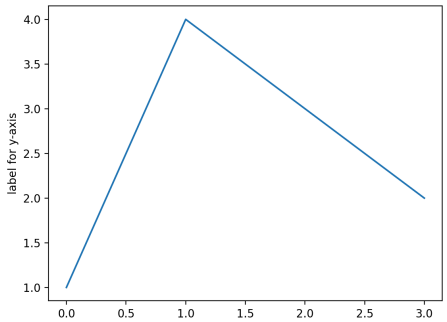


🧩🤔: Why does the plot look like this?

```
1   import matplotlib.pyplot as plt     # Importing matplotlib.pyplot
2   # Note: we use all functions from this library with plt.XYZ
3   plt.plot([1, 4, 3, 2])              # Plotting x vs. y data
4   plt.ylabel('label for y-axis')      # Making a label for y
5   plt.show()                          # Display all open figures
```

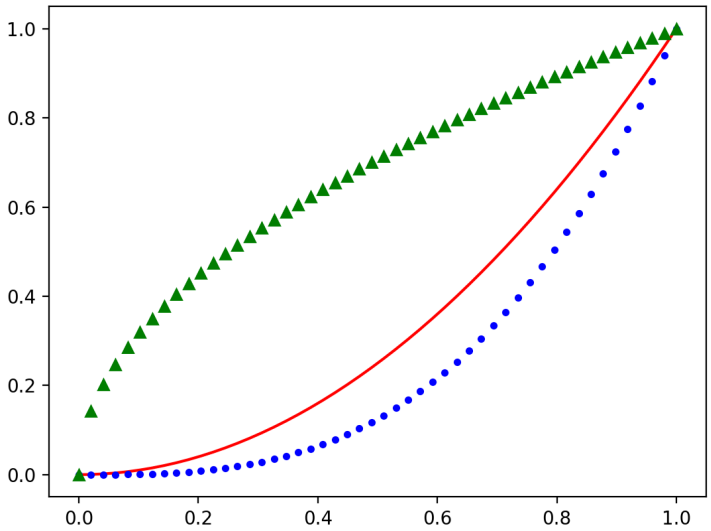
🧩🤔 : Why does the plot look like this?

If one gives just one array `a`, this array is interpreted as y-axis values. By default the x-values are just enumerated 0 to len(a)-1. All those points are connected by line segments.



Example 2: formatting the style of plot

```
1   import numpy as np
2   x = np.linspace(0,1,50)
3   # Plotting x vs. y data  (for multiple functions/ x-y pairs with their own style)
4   plt.plot(x,x**2,'r',x,x**3,'b.',x,np.sqrt(x),'g^')        # See next slide for explanations
5   plt.show()
```



Format String Quick Reference

Format: `[color][marker][line]`

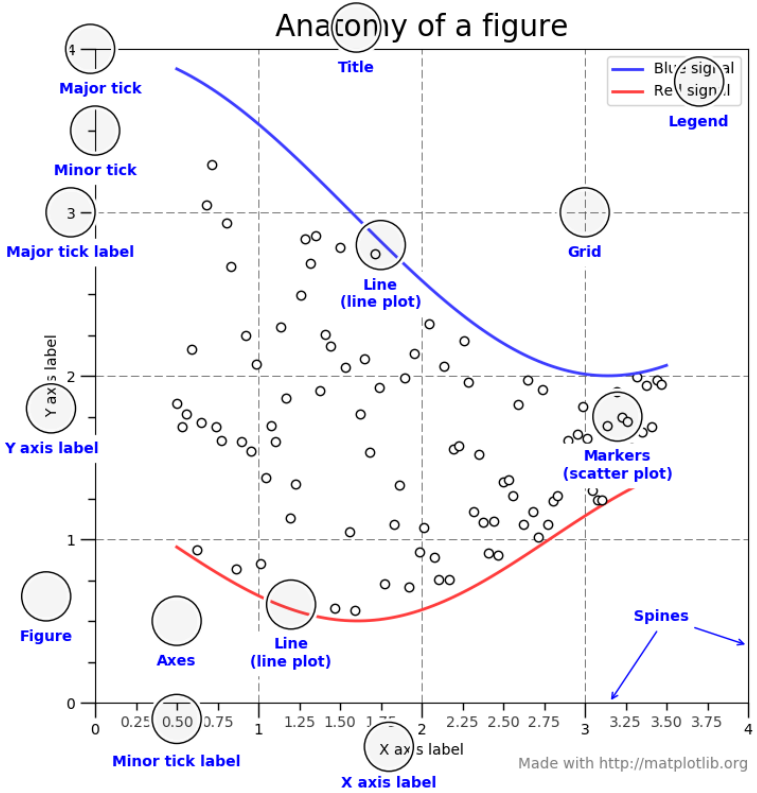
Example: `'ro-'` = Red + Circle markers + Solid line

Component	Options	Example
Color	r, g, b, c, m, y, k	'r' = red
Marker	., o, ^, s, *, +	'o' = circle
Line	-, --, -., :	'–' = dashed

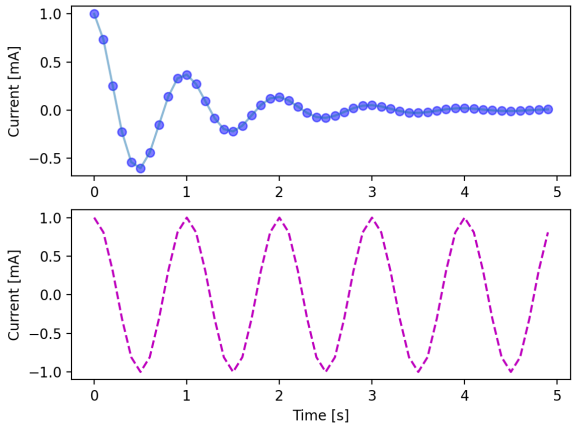
Try it: What does `'k*--'` produce? 🤔

Instead of format strings, you can be explicit: `plt.plot(x, y, color='red', marker='o', linestyle='-', linewidth=2, markersize=8)`

Anatomy of a matplotlib figure

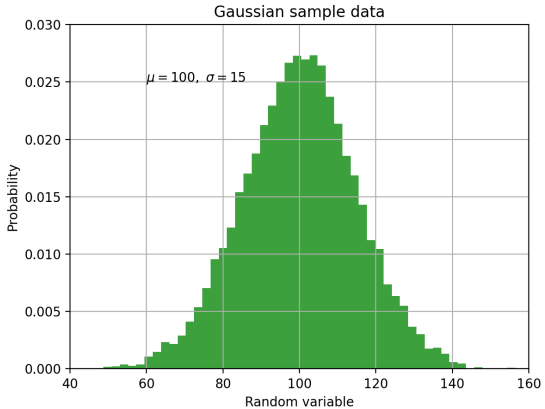


```
# Creating figures with subplots (here nrows = 2, ncols = 1)
T = np.arange(0.0, 5.0, 0.1)
Y = np.exp(-T) * np.cos(2*np.pi*T)    # Vectorized computation!
plt.figure()                          # Creating a new figure (or activate existing)
plt.subplot(211)                      # subplot(nrows, ncols, index)
plt.plot(T, Y, 'bo',T,Y,'-',alpha=.5)
plt.ylabel("Current [mA]")            # y-label
plt.subplot(212)                      # creating index = 2
plt.plot(T, np.cos(2*np.pi*T), 'm--')
plt.xlabel("Time [s]")
plt.ylabel("Current [mA]")
plt.show()
```



Histograms and working with text

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)     # creating 10k samples with mu 100 and std 15
# Creating histogram of the data
n, bins, patches = plt.hist(x, 50, density=1, facecolor='g', alpha=0.75)
plt.xlabel('Random variable')
plt.ylabel('Probability')
plt.title('Gaussian sample data')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')      # putting text at location (60,0.025)
plt.axis([40, 160, 0, 0.03])                      # setting the axis limits
plt.grid(True)                                    # making grid
```



Remember, use docstrings to get help!

```
In [15]: plt.axis?
Signature: plt.axis(*args, emit=True, **kwargs)
Docstring:
Convenience method to get or set some axis properties.

Call signatures::

  xmin, xmax, ymin, ymax = axis()
  xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
  xmin, xmax, ymin, ymax = axis(option)
  xmin, xmax, ymin, ymax = axis(**kwargs)

Parameters
----------
xmin, xmax, ymin, ymax : float, optional
    The axis limits to be set.  This can also be achieved using ::

        ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))

option : bool or str
    If a bool, turns axis lines and labels on or off. If a string,
    possible values are:

    ======== ==========================================================
```

Questions?

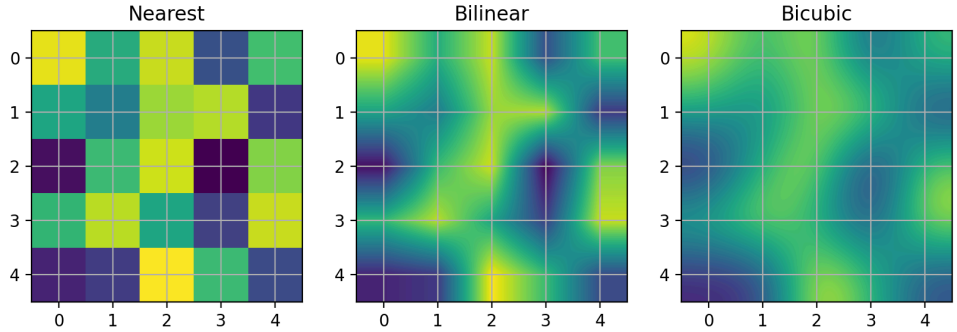
Matplotlib has two interfaces

- `matplotlib.pyplot` is a state-based interface to matplotlib
  - this is what we saw so far
  - Pyplot tutorial
- it also has an object-oriented (OO) interface. In this case, we utilize an instance of `axes.Axes` in order to render visualizations on an instance of `figure.Figure`.
  - more details what that means with a nice example plotting financial data
  - all plots we saw so far, you can all also do this way
  - lots of examples

Plotting images `imshow`

```
A = np.random.rand(5, 5)           # creating a random 5 x 5 array (uniform)
fig, axs = plt.subplots(1, 3, figsize=(10, 3))      # creating a figure object
for ax, interp in zip(axs, ['nearest', 'bilinear', 'bicubic']):
    ax.imshow(A, interpolation=interp)              # plotting `image` A
    ax.set_title(interp.capitalize())
    ax.grid(True)

plt.show()
```



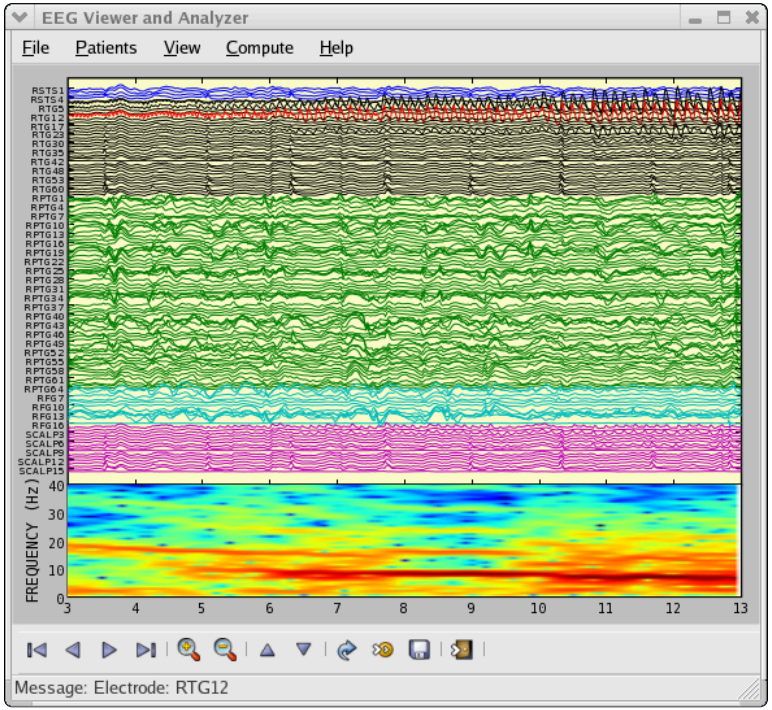
Source / also works for images (loaded as arrays)

Check out the Matplotlib gallery

Tons of visual examples with code, e.g. matlab-unchained

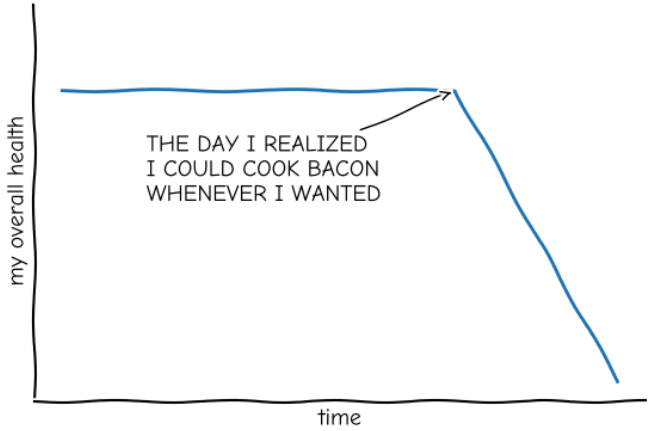
Fun stuff I:

Matplotlib can be integrated in GUIs and make complex figures, e.g., here is a screenshot from pbrain



Fun stuff II:

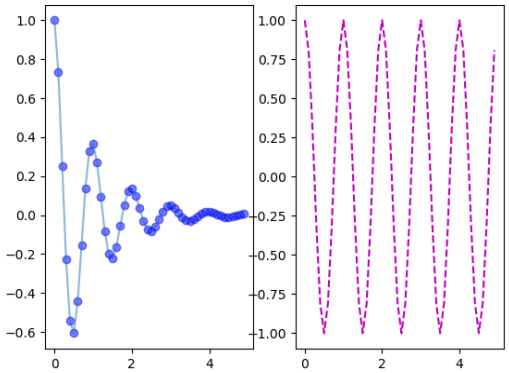
You can plot xkcd-comic style…



"Stove Ownership" from xkcd by Randall Munroe

🧩🤔: How do you make a plot with 2 columns and 1 row?

```
T = np.arange(0.0, 5.0, 0.1)
Y = np.exp(-T) * np.cos(2*np.pi*T)    # Vectorized computation!
plt.figure()                          # Creating a new figure (or activate existing)
plt.subplot(121)                      # subplot(nrows, ncols, index)
plt.plot(T, Y, 'bo',T,Y,'-',alpha=.5)
plt.subplot(122)                      # creating index = 2
plt.plot(T, np.cos(2*np.pi*T), 'm--')
plt.show()
```



Common matplotlib mistakes

- Forgetting plt.show() in scripts
- Not closing figures. This creates memory leaks.

What is a memory leak? They occur when a program allocates memory but fails to release it back to the system when it's no longer needed. This consumes memory, which ultimately might slow down the system, or crash the program (as you run out of memory), ...

```
for i in range(1000):
    plt.figure()  # Creates a new figure
    plt.plot(np.random.rand(100))
    plt.savefig(f'plot_{i}.png')
    # Missing: plt.close() - figure stays in memory!
```

Additional references

Remember, check out the Matplotlib gallery

- Matplotlib tutorial
- Excellent additional matplotlib resources
- Ten Simple Rules for Better Figures
- Review on Visualization of Biomedical Data

Questions?

Today's summary

- Write effective GitHub issues and respond to code reviews
- Document Python functions with proper docstrings
- Create plots with matplotlib

As always, try out the commands in the python shell/notebooks!

In the exercises you will add docstrings and visualizations to your project.

After lunch:

- This week we will add visualizations and docstrings.
- Stay tuned for your code review, release v3 by Monday at 10 am.
- Monday 16:15 - 17: my office hours at SV 2811