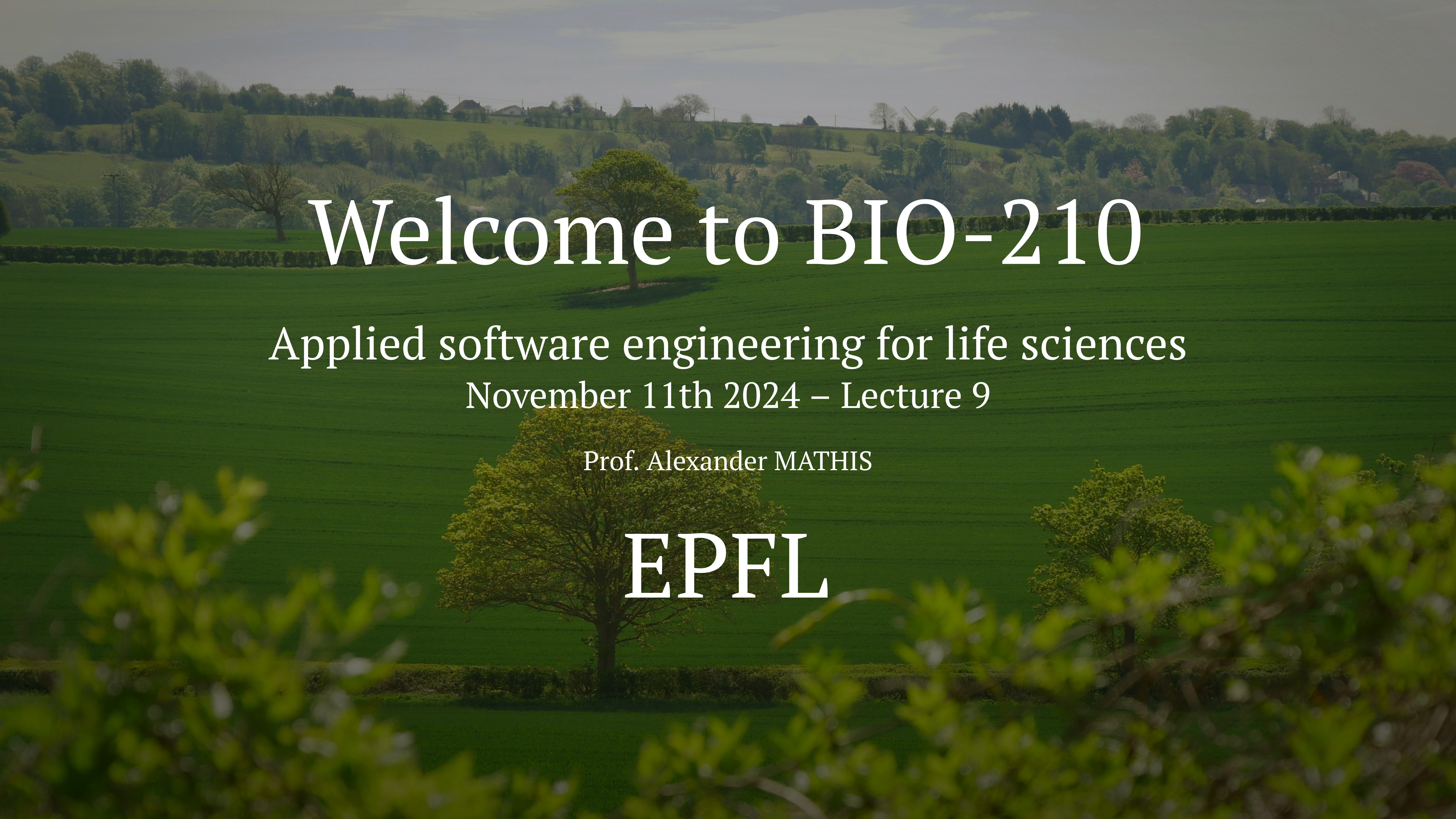


Welcome to BIO-210

Applied software engineering for life sciences

November 11th 2024 – Lecture 9

Prof. Alexander MATHIS

EPFL

Announcements

- Today (Monday at 10am) v4 on testing was due, we will grade it. If you haven’t released it yet, please do so asap.
- Today is the last in person quiz: please come *in time*, there will be no extra time. Submission closes at 13:35. To start, you’ll need to sign in. Bring your Camipro. No notes are allowed. If you switch to a different tab from Moodle’s quiz or communicate with somebody, you’ll receive 0 points.
- Monday 15:15 - 16: my office hours at SV 2811

	Date	Topic	Software version	Software releases	Grading / Feedback
0	09/09/2024	Python introduction I			
1	16/09/2024	Public holiday			
2	23/09/2024	Python introduction II			
3	30/09/2024	Git and GitHub (+installation VS Code)			
4	07/10/2024	Project introduction	v1		
5	14/10/2024	Functionify	v2	v1	
6	21/10/2024	EPFL fall break			
7	28/10/2024	Visualization and documentation	v3	v2	code review (API)
8	04/11/2024	Unit-tests, functional tests	v4	v3	
9	11/11/2024	Code refactoring	v5	v4	graded (tests)
10	18/11/2024	Profiling and code optimization	v6	v5	code review
11	25/11/2024	Object oriented programming	v7	v6	graded (speed)
12	02/12/2024	Model analysis and project report	v8	v7	code review (OO)
13	09/12/2024	Work on project			
14	16/12/2024	Wrap up		v8	graded (project)

Practical tips for addressing code reviews

A few weeks ago, you got code reviews for version v2 and you will also get feedback for v5 (due next week).

The course assistants opened ``GitHub issues`` on your project to give feedback. GitHub Issues allow one to track ideas, feedback, tasks, or bugs for work on GitHub.

Importantly, Issues also allow traceability:

- you can link your pull request (PR) to isssues (opened by the assistants) with the corresponding "#number". E.g. in a commit message you can write "git commit -m "closes #3", which will literally close issue #3 when you push it to GitHub (e.g. see this issue)
- You can also cross-link by putting hyperlinks: example
- These mechanisms support working in teams

Your README.md file

A README is a text file that introduces and explains your GitHub project.

- should be succinct, but detailed
- notice that it’s the face of your project

Typical content:

- Project title + description
- Installation guide/requirements (see below)
- Examples (for using the code)
- License

Examples

Check out some example projects:

- Scipy
- a somewhat recent machine learning project in my group: DMAP
- a recent machine learning project in my group: hBehaveMAE
- ML-From-Scratch
- Awesome Python
- Python The Algorithms
- 100-days of ML
- Awesome DeepVision
- Unity ML-Agents Toolkit

but also the class Demo project.

A readme is indeed the face of your project: Demo project without readme

How to format a Readme.md?

Formatting READMEs is based on the Markdown language, which enables you to write nicely formatted and visually appealing texts!

- it's simple to learn, just skim this Markup Guide 📝🎨👦🤹⚒️…
- Check out this readme generator

Use a .gitignore file

Use a **.gitignore** file to exclude files you don’t want to version control (e.g. notebook checkpoints, pycache, files from your IDE, …).

- Docs for gitignore
- demo-project/.gitignore.

Note: if you already have unwanted files in your repository and create a .gitignore, you need to remove those, as they are already tracked with git.

How to make code reproducible?

- make dependencies and requirements explicit (next slide)
- use version control and store which version produced what results
- use tests / doctests
- do not comment/uncomment sections of your code to control the behavior. This approach makes it error prone, hard to reproduce and difficult to automate. Instead, use if/else statements to control the flow of your program
- You could also consider making different experiment scripts (to reproduce each experiment)

Documenting requirements: Software dependencies

- Software often has many dependencies
- Code might run differently with different library versions!
- Thus, placing dependencies within a contained environment can minimize issues and allow others to run the code just like it runs on your system
- Different programs might also require different libraries, which can be supported by `` `environments` ``
- Common python environments include Anaconda (conda) and virtualenv

-> share an enviroment/requirement file (e.g. see demo-project)

-> also see DMAP

Discussion: How could your code be organized?

How could your code be organized?

Code and Basics:

- README (see above, also demo code!) + .gitignore file
- a main function that is clean and easy to run (additionally `experimentX.py`, `exptY.py`, ...)
- a test function with all our doctests/pytests
- scripts containing functions (named reasonably, e.g. `TuringModel.py`) and a `utils.py` script containing plotting etc.

Results:

- a folder (e.g. called 'results') containing saved experiments/outcomes (could have traceable names: e.g. `experiment17_parametersXYZ.png` are the results for `experiment17.py`)
- Note: you could also add jupyter notebooks that integrate documentation (description + figures)
- NOTE: for BIO-210 just follow the problem sets (to get the highest score!); e.g. main.py in v2 should do whatever you’re asked to have implemented

Further reading

- Good enough practices in scientific computing - a very short article!
- Guide for reproducible research by the Alan Turing Institute

Questions?

Quiz: How can we make this faster?

```python
def generate_patterns(num_patterns, pattern_size):

    patterns = np.zeros([num_patterns, pattern_size])
    for i in range(num_patterns):
        for j in range(pattern_size):
            patterns[i,j]=np.random.choice([-1,1])

    return patterns
```

Quiz: Are these good doctrings?

```python
def generate_pattern(pattern_number,pattern_size):
    '''
    Generate_pattern generates an array of (pattern_number) patterns of
    neuron connections of size (L)

    Parameters
    -------------------
    pattern_number : int
    pattern_size : int
    Returns an array consisting of "pattern_number" patterns
    -------------------
    '''
    return np.random.choice([-1,1],size=(pattern_number,pattern_size))
```

Questions?

Selected function topics

Reminder: Function-related statements and expressions

Statement or expression	Examples
Call expression	myfunc('Seppl',175,age=22,*rest)

`` `def` ``	def printer(message):
	    print('Hello'+message)

`` `return` ``	def adder(a,b=1,*c):
	    return a+b+c[0]

`` `global` ``	x = 'outside'
	def changer():
	    global x; x= 'new'

`` `lambda` ``	`` `func = [lambda x: x**2, lambda x: x**3]` ``

Arbitrary positional arguments collectors (*args)

When this function is called all arguments are assigned to a tuple *args*.

```
In [1]: def f(*args): print(args)   # Note use of *

In [2]: f()
()                                  # returns a tuple!

In [3]: f(1)
(1,)                                # returns a tuple!

In [4]: f(1,2,3,4)
(1, 2, 3, 4)                        # returns a tuple!

In [5]: f("EPFL","is","great!")
('EPFL', 'is', 'great!')            # returns a tuple!
```

Why is *args useful?

For instance, imagine you need to sum all numbers somebody gives you ...

```
In [1]: def summation(a,b,c):
   ...:   return a+b+c
   ...: summation(1,2,3)
Out[1]: 6
In [2]: summation(1,2,3,4,5)        # That's too many for your function!
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-0851c3d51d8a> in <module>
----> 1 summation(1,2,3,4,5)
TypeError: summation() takes 3 positional arguments but 5 were given
In [3]: def summation(*args):
   ...:     Sigma=0
   ...:     for s in args:
   ...:         Sigma+=s
   ...:     return Sigma
   ...:
In [4]: summation(1,2,3,4,5)
Out[4]: 15
In [5]: summation(213.1,445560123,111)
Out[5]: 445560447.1
```

Arbitrary keyword arguments collectors (**kargs)

When this function is called all arguments are assigned to a dictionary *args*.

```
In [1]: def f(**kargs): print(kargs)      # Note use of **

In [2]: f()
{}                                        # returns a dict

In [3]: f(1)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-281ab0a37d7d> in <module>
----> 1 f(1)

TypeError: f() takes 0 positional arguments but 1 was given

In [4]: f(x=1,y=2)
{'x': 1, 'y': 2}                          # returns a dict

```

Argument matching in Python function

Now we have seen all four styles: *Positionals*, *Keywords*, *Defaults*, and *Varargs collecting*

These methods can also be combined, but note positional arguments come before keyword arguments. Just like default variables come after positional variables.

```python
def myfun(a,b,*args,**kwargs):
  pass
```

Read more in the docs

Anonymous functions: lambda

Python has an expression `lambda` to generate function objects (name from lambda calculus).

```
lambda arg1, arg2, ... argN: expression using args
```

Functions returned by running `lambda` expressions are like those created and assigned by `def`.

```
In [1]: def f(x,y,z): return x*y*z
In [2]: f(1,2,3)
Out[2]: 6
In [3]: f=lambda x,y,z: x*y*z        # explicitly assign
In [4]: f(1,2,3)
Out[4]: 6
In [5]: f=lambda x,y=2,z=2: x*y*z    # defaults work similarly
In [6]: f(1)
Out[6]: 4
```

Differences of `` `lambda` `` and `` `def` ``

- `` `lambda` `` is an expression not a statement. Thus, `` `lambda` `` can appear in different places!
- `` `lambda` ``’s body is a single expression not a block of statements.

This is particularly helpful with other tools…

Mapping functions over iterables: map

```
In [1]: data=[0,123,1224,412.23]
In [2]: map?
Init signature: map(self, /, *args, **kwargs)
Docstring:
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from
each of the iterables.  Stops when the shortest iterable is exhausted.
Type:           type
Subclasses:
In [3]: map(lambda x: x**17-13.2,data)          # it is an iterator!
Out[3]: <map at 0x7fa1515be9a0>
In [4]: list(map(lambda x: x**17-13.2,data))    # convert to list for displaying
Out[4]:
[-13.2, 3.3758791744665375e+35, 3.1065911647383047e+52, 2.866639650957827e+44]
# Remember pow(base,exp) = base ** exp
In [5]: list(map(pow,[0,1,2,3,4],[2,2,2,3]))    # Notice *args in use!
Out[5]: [0, 1, 4, 27]
```

Selecting items in iterables: filter

`filter` allows the selection of iterable's items based on a function.

```
In [1]: list(range(-10,10))
Out[1]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

In [2]: list(filter((lambda x: x>3.4), range(-10,10)))
Out[2]: [4, 5, 6, 7, 8, 9]

```

Combining items in iterables: reduce

`reduce` allows cumulatively applying a function!

```
In [1]: from functools import reduce

In [2]: reduce(lambda x,y: x+y, [1,2,3,4,5])
Out[2]: 15

In [3]: reduce(lambda x,y: x*y, [1,2,3,4,5])
Out[3]: 120

In [4]: reduce?
Docstring:
reduce(function, sequence[, initial]) -> value

Apply a function of two arguments cumulatively to the items of a sequence,
from left to right, so as to reduce the sequence to a single value.
For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
of the sequence in the calculation, and serves as a default when the
sequence is empty.
Type:      builtin_function_or_method
```

Imperative vs. functional programming

```
In [1]: input = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
   ...: output = 0
   ...: for i in range(len(input)):
   ...:   if input[i] % 2 == 0:
   ...:     output += input[i]*10
   ...:

In [2]: output
Out[2]: 300
```

```
In [3]: reduce(lambda x,y: x+y, map(lambda i: 10*i, \
                                    filter(lambda i: i%2 == 0, range(1,11))))
Out[3]: 300
```

```
In [4]: np.sum(np.arange(0,11,2)*10)
Out[4]: 300
```

For some theory, check out: Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs by John Backus.

Quiz: How can we test if the input has too many dimensions?

```python
def dX_dt(X, a=1.0, b=0.1, c=1.5, d=0.75):
    ''' Docstrings ommitted!'''
    if np.size(X)>2:
        raise ValueError("X has only two dimensions!")

    return np.array([a * X[0] - b * X[0] * X[1], -c * X[1] + d * b * X[0] * X[1]])

```

Option 1, doctest:

```python
def dX_dt(X, a=1.0, b=0.1, c=1.5, d=0.75):
    ''' Docstrings ommitted!
    >>> dX_dt(np.ones(13))
    Traceback (most recent call last):
      ...
    dX_dt: X has only two dimensions!
    '''
    if np.size(X)>2:
        raise dX_dt("X has only two dimensions!")

    return np.array([a * X[0] - b * X[0] * X[1], -c * X[1] + d * b * X[0] * X[1]])
```

Option 2, pytest:

```python
def test_dX_dt_wronginput():
    """ testing for wrong dimensions (ValueError) """
    import pytest
    with pytest.raises(ValueError):
        LotkaVolterraModel.dX_dt(np.ones(3), 1, 0.1, 1.5, 0.75)
```

Scipy library: scientific computing

Important tools in the scipy library:

- numerical integration scipy.integrate
- optimization scipy.optimize
- Fourier transforms scipy.fft

For a full list, see the docs and the cookbook.

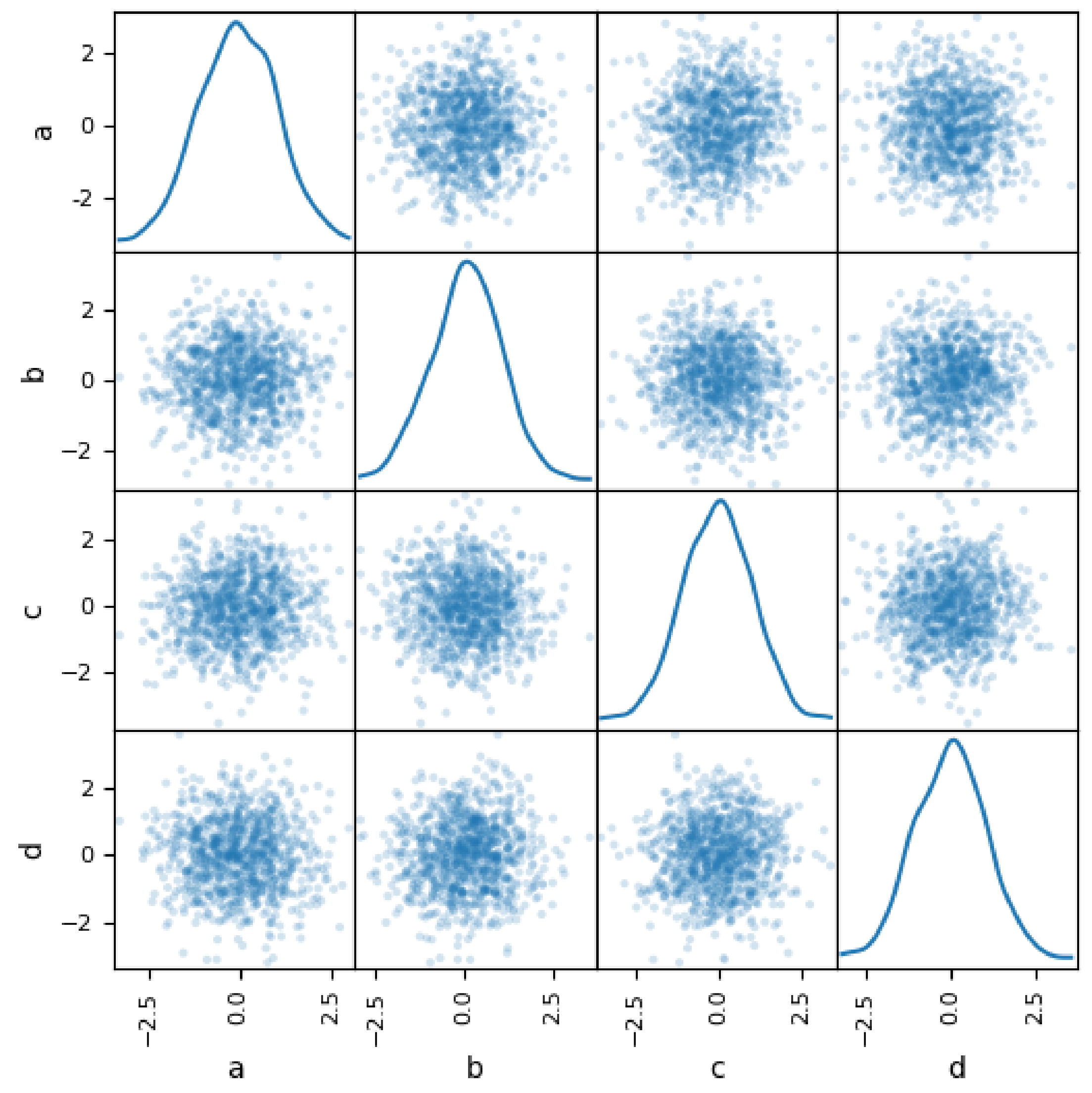
Pandas: data structures and analysis

High-level data analysis package

```
>>> import pandas as pd
>>> import numpy as np
# Defining a dataframe (df):
>>> df = pd.DataFrame(np.random.randn(1000, 4), columns=["a", "b", "c", "d"])
>>> print(df.head())             # prints the first 5 rows
a          b          c          d
0 -0.182791 -0.768629 -1.381591  0.035229
1  1.210803  0.487254 -0.087025  0.253478
2 -0.371740  1.092439 -0.829110  0.518891
3 -1.364988 -2.046488  0.172973 -2.117577
4  1.369287 -0.413473 -2.047923 -1.240338
>>> print(df.mean(axis=0))
a   -0.016737
b    0.000806
c   -0.049374
d   -0.052093
dtype: float64
```

Pandas: data structures and analysis

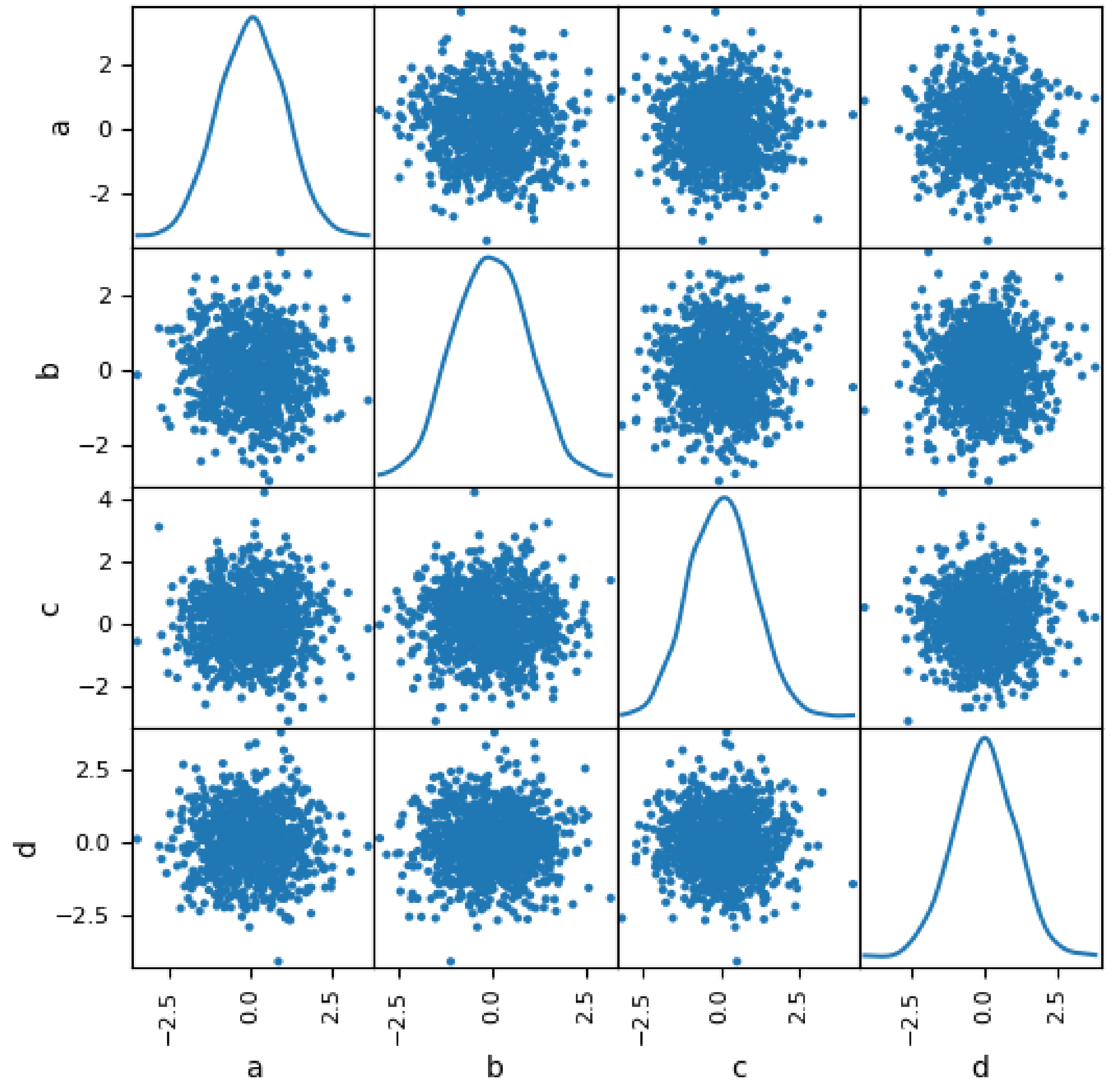
```
>>> from pandas.plotting import scatter_matrix
>>> scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal="kde");
```



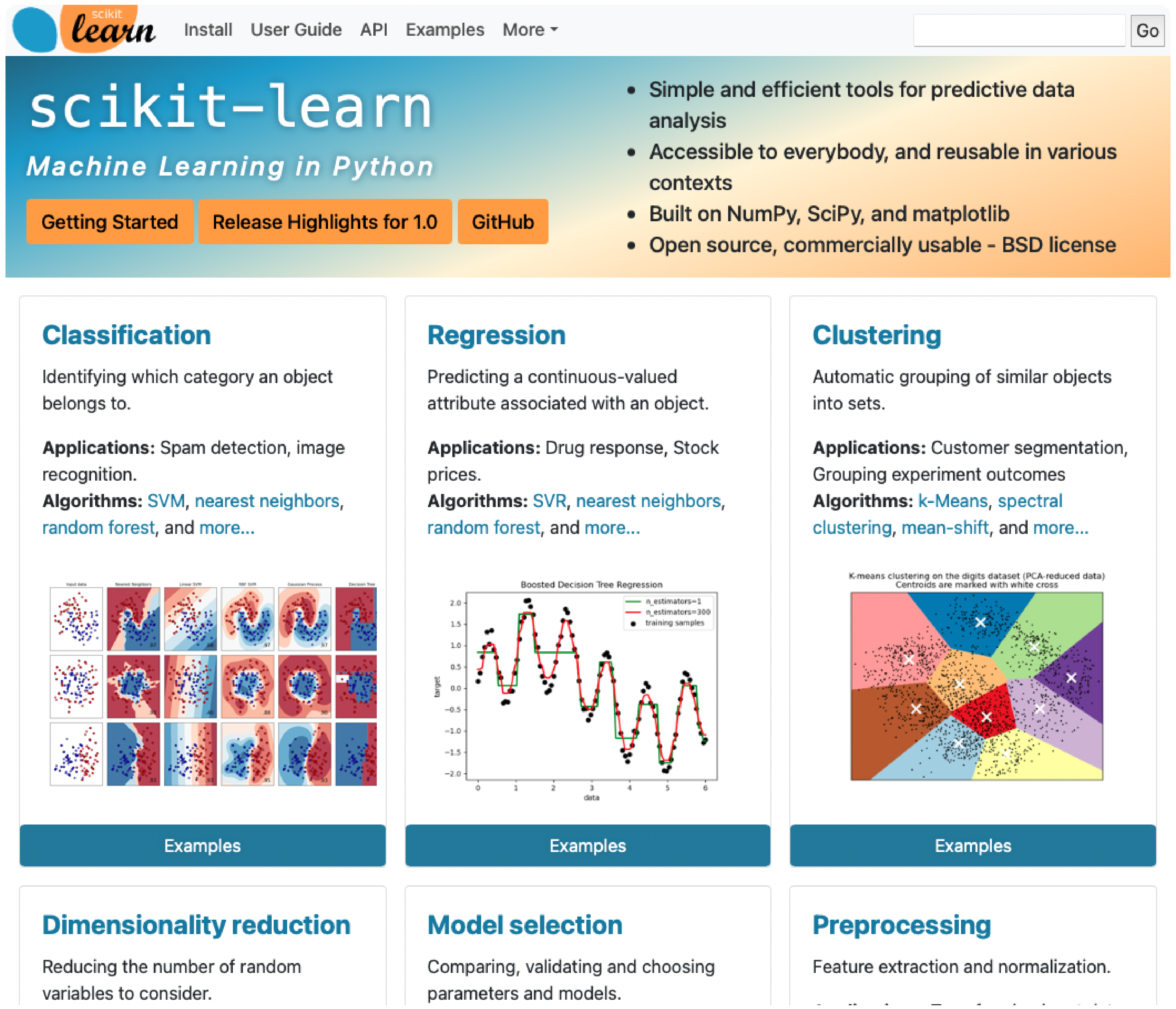
Note: Matplotlib (and thus pandas) allows you to regulate the transparency of a graph plot using the alpha attribute. By default, alpha=1 (not transparent).

Same code without transparency

```
>>> scatter_matrix(df, alpha=1, figsize=(6, 6), diagonal="kde");
```



Scikit learn: Machine learning in python


scikit learn
Install
User Guide
API
Examples
More
Go
Getting Started
Release Highlights for 1.0
GitHub
Boosted Decision Tree Regression
K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white cross
Examples

scikit-learn

Machine Learning in Python

- Simple and efficient tools for predictive data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying which category an object belongs to.

Applications: Spam detection, image recognition.
Algorithms: SVM, nearest neighbors, random forest, and more...

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, nearest neighbors, random forest, and more...

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, and more...

Dimensionality reduction

Reducing the number of random variables to consider.

Model selection

Comparing, validating and choosing parameters and models.

Preprocessing

Feature extraction and normalization.

Questions?

Quiz: What is computed here and what is `j`?

```python
X = np.linspace(1,2,1000)
Y = np.log10(X)
j = np.argmin(Y)
```

Quiz: Extracting the location of the minimum

```
In [1]: import numpy as np
In [2]: X = np.linspace(1,2,1000)
In [3]: X
Out[3]:
array([1.        , 1.001001  , 1.002002  , 1.003003  , 1.004004  ,
       1.00500501, 1.00600601, ...

In [3]: Y = np.log10(X)       # Vectorized logarithm of base 10.
   ...:  j = np.argmin(Y)      # Index of the minimum value
In [4]: j
Out[4]: 0
```

Quiz: What is the result?

```
In [1]: data = np.array([[2,np.nan],[3, 0],[4,1]])
In [2]: np.nanmean(data,axis=0)
```

Mean along columns, while ommiting nans

```
In [1]: data = np.array([[2,np.nan],[3, 0],[4,1]])       # Notice: missing data
In [2]: np.nanmean(data,axis=0)
Out[2]: array([3.0,    0.5   ])
```

Note: `` `(2+3+4)/3 = 3` `` and `` `(0+1)/2 = 0.5` ``. Denominator is automatically adjusted to the number of non-nan items!

Note:

```
In [3]: data.mean(axis=0)
Out[3]: array([3.,      nan])
```

Entries of `` `np.nan` `` is very useful if you have missing data in some table (dataframe/array). The `` `np.nanX` `` functions then make sure that missing data are handled correctly.

Today’s summary

- .gitignore, readme.md, markup, reproducible environments, project structure
- functional tools: `` `lambda` ``, `` `map` ``, `` `reduce` ``, `` `filter` ``
- important libraries: scipy, pandas, and scikit-learn

Try out the commands in the python shell/notebooks! Practice is key.

After lunch:

- 3rd in person quiz!
- Monday 13 - 15: exercises working on v5 of your project
- Monday 15:15 - 16: my office hours at SV 2811