Welcome to BIO-210

Applied software engineering for life sciences

November 10th 2025 – 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.
- Monday 16:15 - 17: my office hours at SV 2811

	Date	Topic	Software version	Software releases	Feedback
0	08/09/2025	Python introduction I			
1	15/09/2025	Python introduction II			
2	22/09/2025	Public holiday			
3	29/09/2025	Git & GitHub (+ installation)			
4	06/10/2025	Project introduction	v1		
5	13/10/2025	Functionify	v2	v1	
6	20/10/2025	EPFL fall break			
7	27/10/2025	Visualization & documentation	v3	**v2**	code review (API)
8	03/11/2025	Unit-tests, functional tests	v4	v3	
9	10/11/2025	Code refactoring	v5	v4	graded (tests)
10	17/11/2025	Profiling & code optimization	v6	v5	code review
11	24/11/2025	Object oriented programming	v7	v6	graded (speed)
12	01/12/2025	Model analysis & project report	v8	v7	code review (OO)
13	08/12/2025	Work on project (no class)			
14	15/12/2025	Wrap up		v8	graded (project)

🧩🤔 : Is there something wrong with this doctest?

```
def get_dna_complement(seq):
    """
    Return the complement of a DNA sequence.

    Example
    -------
    >>> get_dna_complement('ATCG')
    TAGC
    """
    complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return ''.join(complement[base] for base in seq)
```

🧩🤔 : Yes, missing quotes!

```
def get_dna_complement(seq):
    """
    Return the complement of a DNA sequence (adenine (A) pairs with thymine (T),
    and cytosine (C) pairs with guanine (G)).

    Example
    -------
    >>> get_dna_complement('ATCG')
    'TAGC'                                  # Missing quotes. It expects a string!
    """
    complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return ''.join(complement[base] for base in seq)
```

🧩🤔 : How do we test floating-point results?

```
import numpy as np

def calculate_std(data):
    """Calculate standard deviation of data."""
    return np.std(data)
```

Test that calculate_std([1, 2, 3, 4, 5]) returns approximately 1.4142135623730951.

Option 1: Doctest with rounding

```
def calculate_std(data):
    """
    Calculate standard deviation.

    >>> round(calculate_std([1, 2, 3, 4, 5]), 2)
    1.41
    """
    return np.std(data)
```

Option 2: Pytest with approx

```
import pytest

def test_calculate_std():
    """Test standard deviation calculation"""
    result = calculate_std([1, 2, 3, 4, 5])
    assert result == pytest.approx(1.414, rel=1e-3)

    # Or with absolute tolerance
    assert result == pytest.approx(1.414, abs=0.001)
```

Pytest's approx() is cleaner for floating-point comparisons!

Summary: Doctest vs Pytest

Feature	Doctest	Pytest
Purpose	Documentation + basic testing	Comprehensive testing
Location	Inside docstrings	Separate test files
Best for	Simple examples, user-facing docs	Edge cases, complex scenarios
Syntax	`>>> function(input)`	`assert function(input) == expected`
Floating-point	Use `round()`	Use `pytest.approx()`
Exceptions	`Traceback...`	`pytest.raises()`
Multiple cases	Repeat examples	`@pytest.mark.parametrize (see later!)

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; i.e. if one goes to your repository, the README is rendered below the file list.

Typical content:

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

A brief overview of licenses

When you create a project, you should also choose a license for your code. A license tells others what they can and cannot do with your code. Unlicensed code is considered "all rights reserved" by default, meaning others cannot use, modify, or distribute it without your permission.

Some common open-source licenses:

- MIT License: very permissive, allows reuse with minimal restrictions
- GNU General Public License (GPL): requires derivative works to also be open-source under the same license
- Apache License 2.0: allows use, modification, and distribution with some conditions regarding patents and trademarks
- Check out this page for comparing licenses

Naturally for your (private) BIO-210 project, the license is not a big issue, but for real-world projects, it's important to choose an appropriate license to protect your work and clarify how others can use it.

Examples of good READMEs

Check out some example projects. Firstly, typical research codebases:

- two recent machine learning projects from my group: Kinesis and hBehaveMAE

Secondly, popular open-source projects with well-structured READMEs:

- Scipy
- MMore
- Awesome Python
- Python The Algorithms
- Strix

Also check out the Demo project.

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 Markdown Guide 📝🎨👷🤹🛠️…
- Check out this readme generator
- You can also use Markdown live preview to see how your markdown looks like while writing it.

Git management: .gitignore file

Use a **.gitignore** file to exclude files you do not want to version control (e.g. notebook checkpoints, pycache, files from your IDE, ...). This keeps your repository clean and avoids committing unnecessary files. The .gitignore file should be placed in the root directory of your repository.

- 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 (software) 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. If needed, use configuration files or command-line arguments to set parameters.
- Leverage scripts to run experiments, which can help in automating and reproducing results.
- 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. It also means that you can run the same code in the future, even if libraries have changed.
- Different programs might also require different libraries, which can be supported by `environments`. This avoids dependency conflicts between different projects on your system.
- Common python environments include MiniForge, Anaconda (conda) and virtualenv. You are already familiar with MiniForge from the course setup

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

-> also see hBehaveMAE requirements

Discussion: How should your code be organized?

What components should your project have? How should files and folders be organized?

How should 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`)
- Remark 1: you could also add jupyter notebooks that integrate documentation (description + figures)
- Remark 2: 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?

🧩🤔: How can we make this faster?

```
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
```

🧩🤔: Are these good docstrings?

```python
def generate_pattern(pattern_number,pattern_size):
    '''
    Generates an array of (pattern_number) patterns of size (pattern_size)

    Parameters
    ----------
    pattern_number : int
        Number of patterns to generate.
    pattern_size : int
        Size (number of neurons) in each pattern.

    Returns
    -------
    np.ndarray
        Array of shape (pattern_number, pattern_size) with entries of -1 or 1.

    Examples
    --------
    >>> generate_pattern(3,4).shape
    (3,4)
    '''
    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]`

Reminder: Argument-matching modes

Python functions allow highly flexible calling patterns for functions

- *Positionals*: matched left to right
- *Keywords*: matched by argument name; `name = value` syntax
- *Defaults*: specify values for optional arguments (that do not need to be passed)
- ***Varargs collecting*: pass arbitrarily many positional or keyword arguments**

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.

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

Read more in the docs

🧩🤔: How can we calculate the mean of each column ignoring missing data?

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

🧩🤔: How can we calculate the mean of each column ignoring missing data?

```
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  ])
```

Let's verify: `(2+3+4)/3 = 3` and `(0+1)/2 = 0.5`. The 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.

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.

Expression vs. Statement

Aspect	Expression	Statement
Returns	Evaluates to a value	Performs an action
Can assign?	✓ Yes: `x = expression`	❌ No: `x = statement`
Use inline?	✓ Yes: `print(expression)`	❌ No: `print(statement)`
Examples	`5 + 3`, `lambda x: x**2`, `x > 5`	`x = 5`, `def f():`, `if x > 5:`

Key insight: `Lambda` is powerful because it's an *expression* - unlike `def` you can use it anywhere a value is needed! 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
```

🧩🤔: What does this reduce return?

```python
from functools import reduce
numbers = [2, 3, 2]

result = reduce(lambda x, y: x ** y, numbers)
print(result)
```

Let's trace it step by step:

1. `x=2, y=3` → `2 ** 3 = 8`
2. `x=8, y=2` → `8 ** 2 = 64`

Thus, the answer is: `64`

What is a Decorator?

A **decorator** is a function that modifies or enhances another function without changing its code.

Decorators wrap a function to add extra behavior. We'll see an example next!

Parametrized Testing with Pytest

`@pytest.mark.parametrize` is a decorator. It alllows you to run the same test function multiple times with different inputs.

Benefits:

- Avoid repetitive code (no need to write `assert` for each case)
- Each parameter set runs as a separate test
- Easy to add new test cases
- Clear reporting of which specific inputs fail

Generic syntax:

```
@pytest.mark.parametrize("argument_names", [
    (value1_arg1, value1_arg2, ...),
    (value2_arg1, value2_arg2, ...),
])
def test_function(argument_names):
    # test code using the arguments
```

Pytest with multiple test cases (parametrize)

```python
import pytest
import numpy as np

@pytest.mark.parametrize("data,expected", [
    ([1, 2, 3, 4, 5], 1.4142135623730951),
    ([10, 10, 10, 10], 0.0),
    ([0, 0, 0, 0], 0.0),
    ([1, 5], 2.0),
    ([-1, 1], 1.0),
    ([100, 200, 300], 81.64965809277260),
])
def test_calculate_std(data, expected):
    """Test standard deviation calculation with multiple inputs"""
    result = calculate_std(data)
    assert result == pytest.approx(expected, rel=1e-6)
```

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
```

Imperative programming example:

```
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)    # Numpy implementation
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.

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.

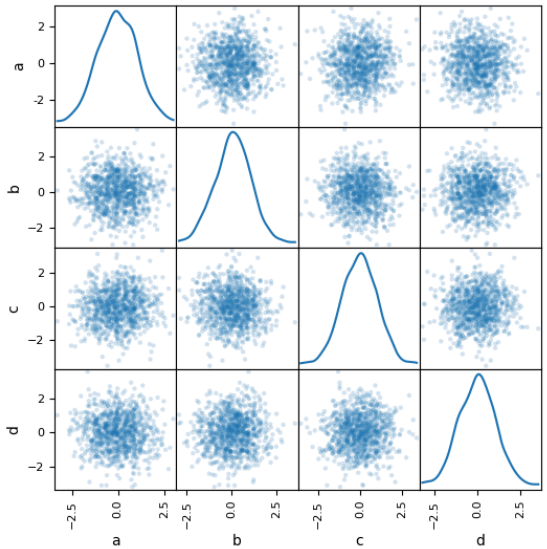
Pandas: data structures and analysis

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

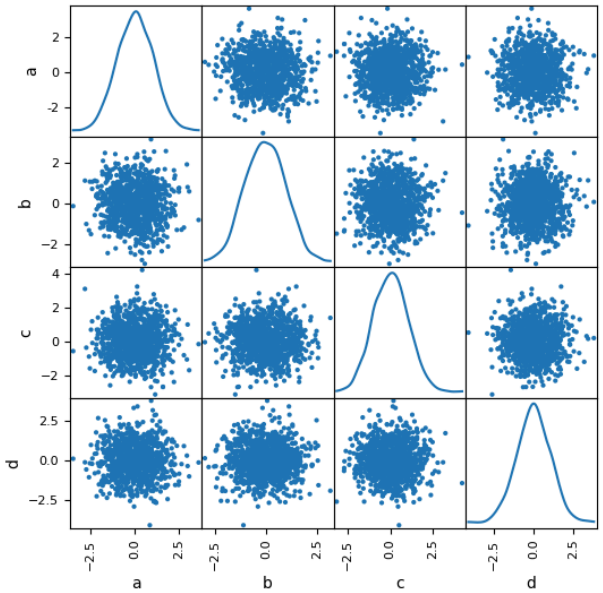
```
>>> from pandas.plotting import scatter_matrix
>>> scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal="kde");
```



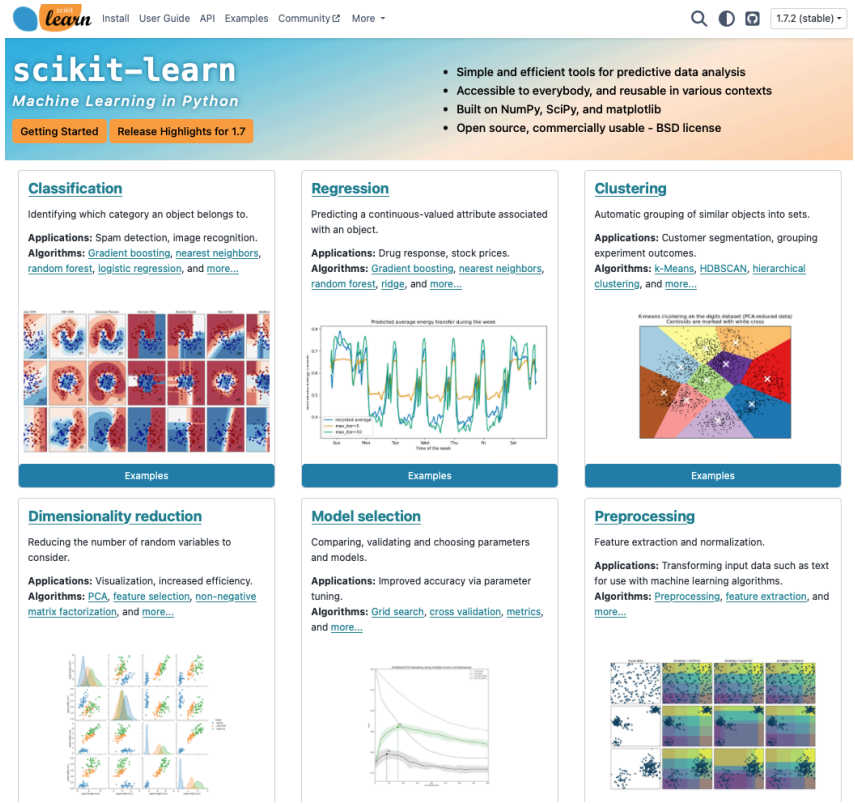
Note: Matplotlib (and thus pandas) allows you to regulate the transparency of a graph plot using the alpha attribute. By default, alpha=1 (not transparent).

Same code without transparency

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



Scikit learn: Machine learning in python



Questions?

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:

- Monday 13 - 15: exercises working on v5 of your project
- Monday 16:15 - 17: my office hours at SV 2811