Welcome to BIO-210

Applied software engineering for life sciences

October 13th 2025 – Lecture 4

Prof. Alexander MATHIS

EPFL

Announcements I

- Congrats on your great quiz results! Class average 9.1/10!

Announcements II

- Please note that we altered the room assignment for the exercises slightly. Check here and go to the correct room from now on. NOTE: you need your EPFL login to see it!
- Did you release your code, v1?
- v1 of your project was due at 10am today (not graded/checked), check out the release guide.

	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)

Discussion: How should a project's code be organized on GitHub?

Notes on your repository organization:

- We made a release guide
- put the "final" code for each week in main (branch)
- make a tag to label the state. Call it "v1" and make a release note. This is common practice for packages: e.g. numpy v2.3.3, DLC2action
- one can easily checkout specific tags with:
  - listing the tags (`git tag -l`)
  - `git checkout tags/<tag_name>`
- *v1* should reproduce *this* week's problem set!
- we will not check your branches, but try to keep them clean (e.g. remove merged ones that you do no longer need)
- Check out our example demo – Also make pull-requests if you want to suggest changes!

What if you collaborate on a commit?

Important, for learning its better if you collaborate via git (see later).

But, if multiple team members contribute to a commit, add all relevant authors to your commit message:

```
>>> git add .            # staging all files
>>> git commit -m "Adding testing symmetry of Hopfield weights

Some more content ...
Co-authored-by: My awesome teammates name <use_the_email_of_the_github_account@epfl.ch>
```

Be sure that git is linked to the correct email. Otherwise the commits are not correctly displayed.

How can you check if you set up git correctly?

If your local Git email doesn’t match your GitHub/GitLab account email, your commits will show up as coming from an unknown user or won’t be linked to your profile. The commits will still be in the repository, but they won’t appear on your contribution graph or be visually connected to your account.

```
git config user.email
git config --global user.email "your.email@example.com"
```

Best practice: Use the email associated with your GitHub/GitLab account. You can check which emails are associated with your account in Settings → Emails.

How should you collaborate via git?

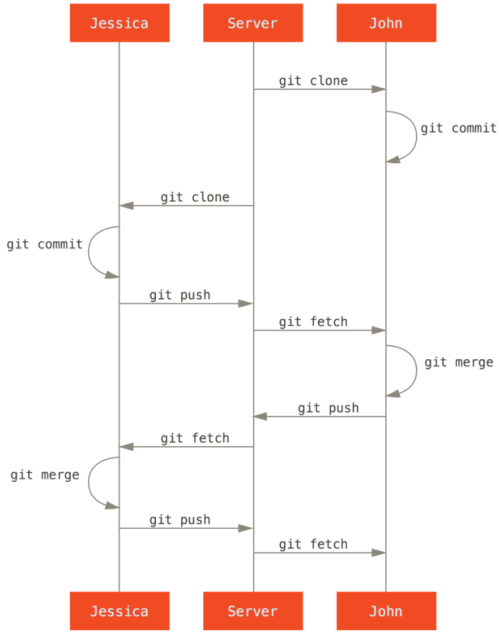


Image source: git docs

Important git references

Jennifer Shan (one of the past SAs) developed a video tutorial on how to use git in Visual Studio Code

Viva Berlenghi (one of the SAs) wrote a Git survival kit

See this git cheat sheet (in French) and English

Common type-specific operations in Numpy

There are tons, check out NumPy's API reference

```
In [1]: X=np.linspace(0,2*np.pi,10)      # create ar. /w 10 linearly sep. pts in [0,2pi]
array([0.        , 0.6981317 , 1.3962634 , 2.0943951 , 2.7925268 ,
       3.4906585 , 4.1887902 , 4.88692191, 5.58505361, 6.28318531])

In [2]: Y=np.sin(X)                      # calculate sine, vectorized of course
array([ 0.00000000e+00,  6.42787610e-01,  9.84807753e-01,  8.66025404e-01,
        3.42020143e-01, -3.42020143e-01, -8.66025404e-01, -9.84807753e-01,
       -6.42787610e-01, -2.44929360e-16])

In [3]: Y>0                              # where is this array >0? Vectorized
Out[3]:
array([False,  True,  True,  True,  True, False, False, False, False,
       False])
```

🧩🤔: What is computed here?

```
1   X = np.logspace(-1,2,7)
2   Y = np.log10(X)
3   j = np.argmin(Y)
```

What is j?

🧩🤔: Extracting the location of the minimum

```
>>> X = np.logspace(-1,2,7)     # Return numbers spaced evenly on a log scale.
array([  0.1       ,   0.31622777,   1.        ,
 3.16227766,        10.        ,  31.6227766 , 100.        ])
>>> Y = np.log10(X)             # Vectorized logarithm of base 10.
array([-1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ])
>>> j = np.argmin(Y)            # Index of the minimum value
0
```

Note `X = 10**np.linspace(-1,2,7)`

A minimalist linear algebra example

```
In [1]: phi = np.pi/3                                # 60 degrees
In [2]: e1,e2=np.array([1,0]),np.array([0,1])        # two basis vectors of R^2
In [3]: rot = np.array([[np.cos(phi), -np.sin(phi)],[np.sin(phi), np.cos(phi)]])
In [4]: print(rot)
Out [4]:
[[ 0.5       -0.8660254]
 [ 0.8660254  0.5      ]]
In [6]: print(np.matmul(rot,e1))                     # matrix multiplication
Out [6]:
[0.5       0.8660254]                                # vector rotated to (cos(phi), sin(phi))
```

A minimalist linear algebra example

```
In [1]: phi = np.pi/3                                # 60 degrees
In [2]: e1,e2=np.array([1,0]),np.array([0,1])        # two basis vectors of R^2
In [3]: rot = np.array([[np.cos(phi), -np.sin(phi)],[np.sin(phi), np.cos(phi)]])
In [4]: print(rot)
Out [4]:
[[ 0.5       -0.8660254]
 [ 0.8660254  0.5      ]]
In [5]: print(np.matmul(rot,e1))                     # matrix multiplication
Out [6]:
[0.5       0.8660254]                                # vector rotated to (cos(phi), sin(phi))
```

np.dot and np.matmul behave the same for matrices, but not 3D (and larger) arrays, check here, for more details.

For more Linear Algebra, check out np.linalg and scipy.linalg for the matrix inverse, Moore-Penrose pseudo-inverse, decompositions (LU, SVD, QR,...), eigenvalues, matrix equation solvers, tensor multiplication...

A minimalist stats example

```
In [1]: data = np.array([[1231,np.nan],[23.123, 0],[0,1]]) # Notice: missing data
In [2]: np.mean(data)                                      # the mean is also nan!
nan
In [3]: np.nanmean(data)                                   # nanmean omits nans
Out[3]: 251.02460000000002
In [4]: np.nanmean(data,axis=0)                            # mean along a specific axis
Out[4]: array([418.041,   0.5  ])

In [5]: np.nanmean(data,axis=1)
Out[5]: array([1.23100e+03, 1.15615e+01, 5.00000e-01])
```

Other stats functions: `np.std`, `np.quantile`, `np.nanmax`, `np.median`, `np.corrcoef`, ...

Creating pseudo random numbers in Numpy

```
In [1]: import numpy as np
In [2]: x = np.random.randint(10)     # Generate a random integer from 0 to 10 (uniform)
In [3]: x
Out[3]: 5

In [4]: np.random.rand()              # Generate a random float from 0 to 1 (uniform)
Out[4]: 0.4150299187382156
In [5]: np.random.rand(2,2)           # Gen. random floats in array of 2 rows & 2 col.
Out[5]:
array([[0.26249554, 0.21950032],
       [0.81072637, 0.01962725]])
In [6]: np.random.randn()             # Generate samples from standard normal
Out[6]: 1.8382156073224407
In [7]: 3 + 2*np.random.randn()       # Gen. normal distr. sample with mean 3 and std 2
Out[7]: 4.067920463989419
```

See np.random for more options!

How can we define our own functions?

We already saw how to use built-in functions or functions from libraries like numpy...

Functions

- so far we have mostly written procedural Python statements/programs
- a `function` can group a set of statements so that they can be run more than once in programs
- `functions` are packaged procedures with a name
- `functions` also can compute a result based on parameters that we can specify
- Coding procedures/operations as `functions` makes them re-usable

Rule of thumb

Every time you copy/paste some statements, make a function!

Functions are one of the most basic Python structures for maximizing code-reuse

Why should you use functions?

- maximize code reuse and minimize redundancy (thus reducing maintenance effort)
- procedural decomposition (splitting programs into well-defined roles)
- it’s easier to implement smaller tasks in isolation (rather than the whole process at once)
- a strategy for minimizing errors

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]`

Function basics

- The keyword `def` is an executable statement
- The keyword `def` creates an object and assigns it a name
- Functions *only exist, once* Python reaches `def`
- Functions really behave like other objects, they can be re-assigned, stored in lists etc.
- The keyword `return` sends a result back to the caller. When a function is called, the caller stops until the function is done and returns control to the caller. Functions that compute a value send it back to caller with a return statement (i.e., the result of the function call).
- Functions without `return` statements, return `None` (upon completion)

Def statements

```
1   def function_name(arg1, arg2, ..., argN):
2     statement_1
3     statement_2
4
5   statement_not_part_of_function    # Added just for illustration
```

- A function's body is indented. This code is run when the function is called.
- For a single statement, one can use `;` and place the code in one line, e.g. `def f(x): return x;`
- The keyword `def` specifies the function name and a list of zero or more `arguments` in parentheses
- The function arguments are assigned to objects passed, when the function is called (not upon definition). I.e., here `arg1` does not need to exist (see Example 0).

Return statements

```
1   def name(arg1, arg2, ..., argN):
2     statements
3     ...
4     return value
```

- A return statement can appear anywhere in the function body (or exist multiple times, e.g. Example 2)
- If there is no return, `None` will be returned

Example 0: defs are not calls

```
In [1]: x
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-6fcf9dfbd479> in <module>
----> 1 x

NameError: name 'x' is not defined

In [2]: def f(x):               # Create and assign function
   ...:     return x            # Body executed when called
   ...:
 In [3]: f(2)                   # Arguments are passed in parentheses
 Out[3]: 2
 In [4]: f(x)
 ---------------------------------------------------------------------------
 NameError                                 Traceback (most recent call last)
 <ipython-input-10-f2d123ee1505> in <module>
 ----> 1 f(x)

 NameError: name 'x' is not defined
```

Note: even though `x` appears in the definition of `f`, `f` is not called.

Example 1: Functions are flexible

```
In [1]: def f1(x):          #
   ...:     return x**2
   ...:
In [2]: def f2(x):
   ...:     return x**4
   ...:
In [3]: f = [f1,f2]           # combine in a list!
In [4]: f(2)                  # f is a list, cannot be called...
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-c510dc86724b> in <module>
----> 1 f(2)

TypeError: 'list' object is not callable
In [5]: f[0](2)               # Index first element, then pass 2, returns 2**2 = 4
Out[5]: 4
In [6]: f[0](3)
Out[6]: 9
In [7]: f[1](3)               # Calling f2, via f[1] (shared object)
Out[7]: 81
```

Example 2: A strange function

```
In [1]: def strange_fun(arg1):
    ...:     if arg1>0:
    ...:         return arg1
    ...:     elif arg1<0:
    ...:         return -1*arg1
    ...:

In [2]: type(strange_fun(1.2))
Out[2]: float
In [3]: type(strange_fun(1))
Out[3]: int
In [4]: type(strange_fun(0))      # returns None, as no return exists for arg1=0
Out[4]: NoneType
```

Functions have dynamic typing behavior in Python. It different types depending on the input.

🧩🤔: What is the result?

```
x=2
if x>3:
  def func(x):
    return 3*x
elif x<3:
  def func(x):
    return 2*x, 0

result = func(2)
```

The code will set result to: `(4,0)`.

Note `def` executes at run-time. You do not need to define func like in C.

🧩🤔: What is the result?

```
In [1]: x=5
   ...: if x>3:
   ...:   print(x)
   ...:
   ...: elif x<3:
   ...:   def func(x):
   ...:     return 2*x, 0
   ...:
   ...: result = func(2)
```

```
5
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 9
      6   def func(x):
      7     return 2*x, 0
----> 9 result = func(2)

NameError: name 'func' is not defined
```

🧩🤔: what is printed when this program runs?

```
1   x = 'abc'
2   def func():
3       x = 'xyz'
4
5   func()
6   print(x)
```

It prints `abc`, as `x` inside func() is a local variable.

This local variable, thus does not affect the global variable x= `abc` (see scoping, later in this lecture)

Functions are typeless, and general!

Defintion:

```
In [1]: def times(x,y):
   ...:     return x*y
   ...:
```

Calls:

```
In [2]: times(2,3)                  # arguments in parentheses
Out[2]: 6
In [3]: times(1.0,4)
Out[3]: 4.0                         # results are casted (type converted)
In [4]: times("La",3)               # functions are "typeless"!
Out[4]: 'LaLaLa'                    # polymorphism
In [5]: times([1,2],4)
Out[5]: [1, 2, 1, 2, 1, 2, 1, 2]
In [6]: my_list = times([1,2],4)    # save the result object
```

Scope

- when you use a name in a program, Python creates, or looks up the name in the `namespace`
- scope refers to a `namespace`. Where you assign a name, determines the scope of a name's visibility
- apart from packaging code for reuse, functions add an extra namespace layer to your programs
- Python has four levels of namespaces:
  - builtins
  - global
  - enclosing (or enclosed)
  - local

The built-in namespace

The built-in namespace contains the names of all of Python's built-in objects.

```
In [1]: print(dir(__builtins__))      # you can list them like this!
   ...:
   ['ArithmeticError', 'AssertionError', 'AttributeError',
    'BaseException','BlockingIOError', 'BrokenPipeError', 'BufferError',
    'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
    'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
    'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
    'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
    'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
    'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
    'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
    'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
    'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
    'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
    'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
    'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
    'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
    'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
    'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
    'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
    'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
    '__doc__', '__import__', '__loader__', '__name__', '__package__',
```

The global namespace

- the global namespace contains all names defined at the level of the main program
- it is created when the main program starts (and exists until the interpreter terminates)

```
(base) alex@mac Code % python
Python 3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 11:23:37) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> globals()
{'__name__': '__main__', '__doc__': None, ... }
>>> a=17
>>> globals()
{'__name__': '__main__', '__doc__': None, ..., 'a': 17}
```

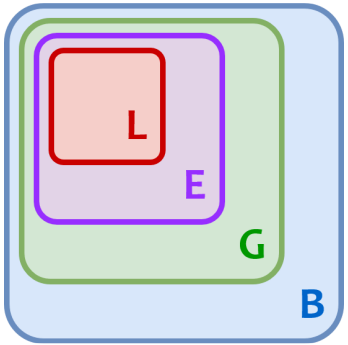
Local and enclosing namespaces

```
In [2]: def f():      # f, the enclosing function of g
   ...:     print('Start f()')
   ...:
   ...:     def g():  # definition of enclosed function g
   ...:         print('Start g()')
   ...:         print('End g()')
   ...:         return
   ...:     g()       # call g()
   ...:
   ...:     print('End f()')
   ...:     return
   ...:
   ...: f()         # Calling f()
Start f()           # Now Python creates a namespace for f()
Start g()           # A new namespace for g() is created
End g()
End f()
```

Here g's namespace is called local namespace, and f's namespace is called enclosing namespace (as f is the enclosing function). Each of these namespaces remains in existence until its respective function terminates.

Variable scope: LEGB rule

- **Local:** If you refer to `x` inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
- **Enclosing:** If `x` is not in the local scope, but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
- **Global:** If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
- **Built-in:** If the interpreter cannot find `x` anywhere else, then the interpreter tries the built-in scope.



Source: Python docs

Questions?

🧩🤔: what does this code print and why?

```
x = 'abc'
def func():
    x = 'xyz'
    print(x)

func()
print(x)
```

It prints `xyz`, then `abc`, as the reference in func() returns the value `xyz` and the reference at the end returns the value of the variable in the global namespace.

Global statement

The `global` statement is one of the only statements, that remotely resembles a declaration statement in Python. However, `global` is not a size-declaration, but a namespace declaration.

```
In [1]: x = 11      # global x
   ...: def func():
   ...:   global x
   ...:   x=99      # global x, within function namescape assignment
   ...:
   ...: func()
   ...: print(x)
99
```

See more, incl. on `nonlocal`, which can be required for enclosed functions, in docs.

🧩🤔: what is printed when this program runs?

```
1  x = 'abc'
2  def func():
3      print(x)
4
5  func()
6  print(x)
```

It prints `abc`, then `abc`, as the reference in func() looks up x in the global scope and the last print refers to the global `x`.

🧩🤔: what does this code print and why?

```
x = 'abc'
def func():
    global x
    x = 'xyz'
    print(x)

func()
print(x)
```

It prints `xyz`, then `xyz`.

As the reference in func() returns the value `xyz` and when func is called, the value of `x` is overwritten.

🧩🤔: what does this code print and why?

```
x = 'abc'
def func():
    global x
    x = 'xyz'
    print(x)

print(x)
```

It just prints `abc` – nothing will be overwritten, as `func()` is never called.

However, try to avoid globals…

Consider this example:

```
X = 99
def f():
  global X
  X = 77

def g():
  global X
  X = 33
```

Why, should you avoid globals?

Here, the value of X is timing dependent, it depends on which function was called last.

Now imagine you want to modify and reuse this code…

Conclusion of scope

Keep in mind that

where you define a name, determines much of its meaning (in functions, modules, etc.)

Argument passing

- arguments are passed by automatically assigning objects to local variables (because references are implemented as pointers, arguments are passed by pointers)
- assignments to argument names in a function, do not affect the caller
- however, changing a mutable object in a function, may impact the caller

Arguments and shared references

```
In [1]: def f(a):       # a is assigned to (references; i.e. the passed obj.)
   ...:   a = 99        # Changes local variable a only
   ...: b = 88
   ...: f(b)            # a and b both reference 88 (initially)
   ...: print(b)        # b is unchanged
88
```

However, if mutable objects (such as lists, dicts,...) are passed, aliasing can happen!

```
In [1]: def f(a,b):   # arguments assigned references to objects
   ...:   a=99        # changes local name's value only
   ...:   b[0]=22     # changes shared object in place!
   ...: A = 1
   ...: B = ['hello',2]
   ...: f(A,B)        # caller
   ...: print(A,B)    # A is unchanged, B is different
1 [22, 2]
```

Note this is the *canonical* Python behavior!

```
In [1]: x = 1
   ...: a = x      # x and a share the same object
   ...: a = 2      # resets a only, x is still 1.
   ...: print(x)
1
```

```
In [2]: x = [1,2]
   ...: a = x         # x and a share the same object
   ...: a[0] = 2      # in-place change to a; x affected!
   ...: print(x)
   ...:
[2, 2]
```

Avoiding mutable argument changes

Method 1 (pass a copy):

```
In [3]: def f(a,b):
    ...:     a, b[0]=99,22        #  You can assign in parallel
    ...:     print("inside f", a,b)
    ...: A, B = 1, ['hello',2]
    ...: f(A,B.copy())           # Caller (pass a copy), also B[:]
    ...: print("outside",A,B)    # A and B are unchanged
inside f 99 [22, 2]
outside 1 ['hello', 2]
```

Method 2 (copy input):

```
In [4]: def f(a,b):
    ...:     b = b[:]            # copy input to not impact caller
    ...:     a, b[0]=99,22
    ...:     print("inside f", a,b)
    ...: f(A,B)
    ...: print("outside",A,B)
inside f 99 [22, 2]
outside 1 ['hello', 2]
```

Argument-matching modes

Python functions allow highly flexible calling patterns for functions

- *Positionals*: matched left to right (standard mode seen so far)
- *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

Let's look at some examples … and discuss *varargs* collecting later in the course.

Keyword examples

```
>>> def f(x,y,z): print(x,y,z);
>>> f(1,2,3)        # passing by position
(1,2,3)

# Using keywords
>>> f(z=3,y=2,x=1)  # match by name
(1,2,3)

# Mixed type
>>> f(1,z=3,y=2)    # x gets assigned 1 by position, others by name
(1,2,3)
```

Why should one use the keyword mode?

To better document code, which goes hand in hand with better variable names, e.g.
`process_user(name='Xavier',age=22,job='EPFL student')`

gives a good idea what this code might do.

Default examples

```
>>> def f(x,y=2,z=3): print(x,y,z);    # x required, y and z optional!
>>> f(1)         # using defaults
(1,2,3)

>>> f(1,4)       # overwriting defaults by positional variable
(1,4,3)
>>> f(1,4,5)
(1,4,5)
# Mixed keyword and default example:
>>> f(1,z=55)    # x gets 1 by position, others by name
(1,2,55)
>>> f(y=2)       # positional arguments need to be passed
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 f(y=2)

TypeError: f() missing 1 required positional argument: 'x'
```

Default, is a very flexible, core Python feature. We have seen it in use for many functions, e.g.: `range`, `np.arange`, `np.linspace`, ...

🧩🤔: How do you get 11 equidistant numbers from 0 to 1?

Variant 1:

```
In [1]: import numpy as np
In [2]: X = np.linspace(0,1,11)
In [3]: X
Out[3]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Variant 2:

```
In [1]: import numpy as np
In [2]: X = 1./10*np.arange(11)
```

...

🧩🤔: What will be printed?

```
1   def f():
2       x = 20
3
4       def g():
5           global x
6           x = 40
7
8       g()
9       print(x)
10
11
12  f()
```

It will print `20`, as x refers to the enclosing namespace in `f()`, not the global namespace!

A custom module

A module is a file containing Python definitions and statements.

The file name is the module name plus the suffix `.py`.

```
# Heaviside function module

def HeavisideFun(x):
    ''' Heaviside function with half-maximum convention '''
    if x>0:
       return 1.
    elif x<0:
       return 0.
    elif x==0:
       return .5
    else:
       return None
```

Let's save this file as `mymodule.py` in the same directory as the working directory.

Example use of our custom module

```
In [1]: import mymodule              # Import our module
In [2]: mymodule.HeavisideFun(13.)
Out[2]: 1.0
In [3]: mymodule.HeavisideFun(0)
Out[3]: 0.5
In [4]: f=mymodule.HeavisideFun
In [5]: f(13.)                    # one can assign a local name.
Out [5]: 1.0
In [6]: mymodule.HeavisideFun?
Signature: mymodule.HeavisideFun(x)
Docstring: Heaviside function with half-maximum convention
File:      ~/Code/Teaching/BIO-210-materials/slides/mymodule.py
Type:      function
```

Remember, we have already seen built-in modules (e.g., `math`) and packages, which are structured modules (e.g., `numpy`).

Note, here we assume that we are in the same folder as mymodule.py. Learn more about where python looks for modules here.

Questions?

Today's summary

- discussion of how you should collaborate & release your project
- deeper dive into functions: `def`, `return`
- scoping, namespaces, LEGB rule, global, globals(), modules
- discussion of Python's argument matching modes

Try out the commands in the Python shell/notebooks!

After lunch:

- Monday 13:15 - 15: exercises. *Please note that we altered the room assignment slightly!* Check here and go to the correct room from now on. NOTE: you need your EPFL login to see it!

- Monday 16:15 - 17: my office hours at SV 2811