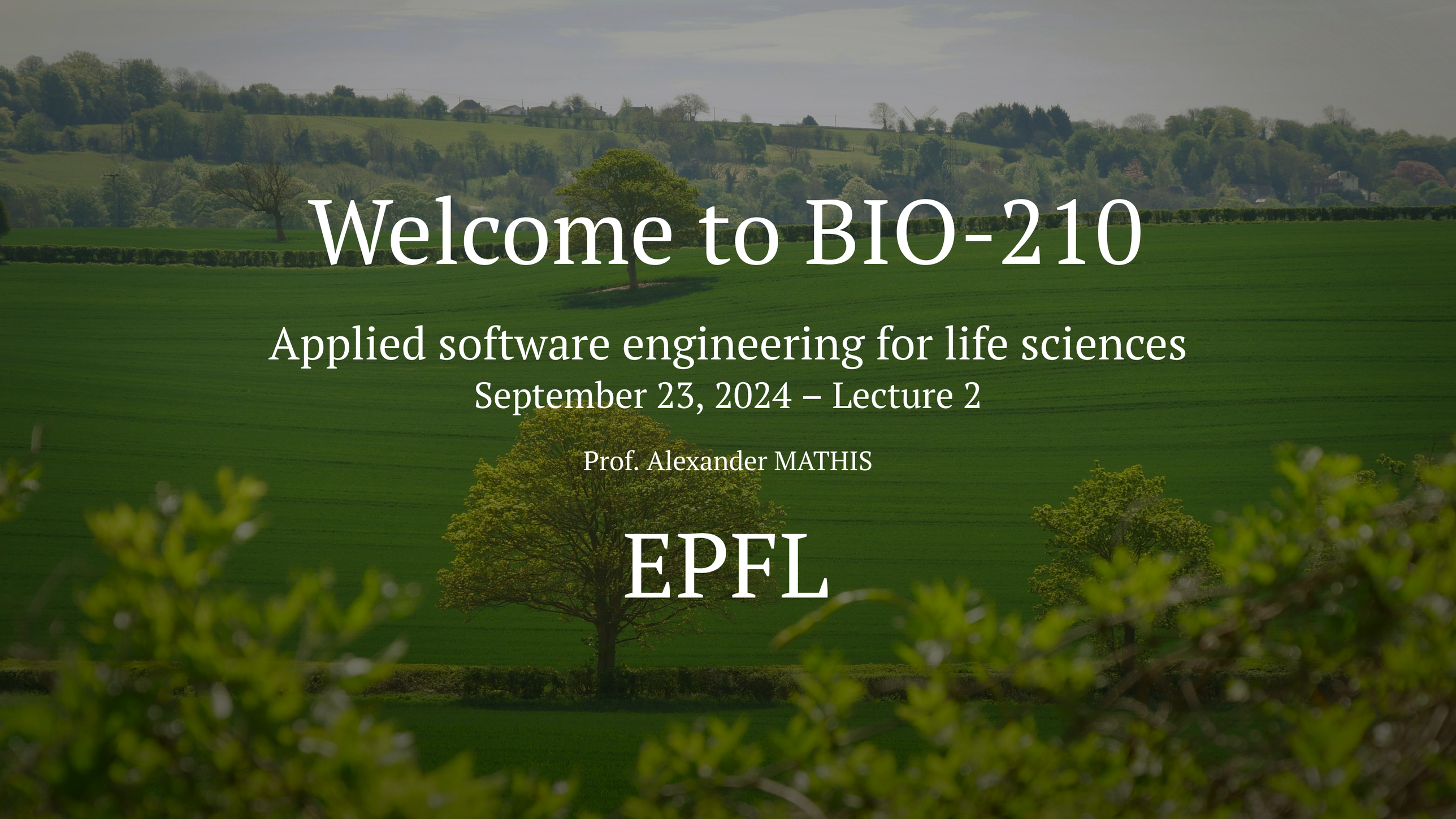


Welcome to BIO-210

Applied software engineering for life sciences

September 23, 2024 – Lecture 2

Prof. Alexander MATHIS

EPFL

	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)

Announcements

- We posted solutions to the notebook from last week
- All clear about the quizzes?

Reminder:

- quizzes in week 3 (next week), 5, 7, 9 and 11 (online). Pythonic counting…
- for quiz 3 and 11 you have from Monday to Friday to fill it out; quizzes 5, 7 and 9 are in person.
- Monday 15:15 - 16: my office hours at SV 2811

Preparation for projects:

- create a personal GitHub account.
- find two teammates and sign up here. You will need your GitHub names.

Please sign up by the end of the week (we will also announce this via Moodle).

Python’s conceptual hierarchy

1. Programs are composed of modules
2. Modules contain statements
3. Statements contain expressions
4. Expressions create and process objects

Python’s core built-in objects

Object type:	Examples:
Numbers	123, 3.14, math.pi, …
Strings	'abc', 'EPFL', "Geneva", …
Lists	[1, [2, 'troi'],4], list(range(99)
Dictionaries	{'Apples': 200, 'Pears': 123.5}, dict(hours=10)
Tuples	(x,y,z), (1, [2, 'troi'],4)
Sets	set('abc'), {'E','P','F','L'}
Other core types	Booleans, types, None
Files	open('data.txt'), open(r('/home/alex/abc.bin'),'wb')
Program unit types	Functions, modules, classes

Learning by trying it out!

slidev — IPython: alex/slidev — ipython — 123×29

```
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev %
(base) alex@alexs-MacBook-Air-2 slidev % ipython
Python 3.9.12 (main, Apr  5 2022, 01:52:34)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.15.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 2*4
Out[1]: 8

In [2]: 3**4
Out[2]: 81

In [3]: 33*"exciting"+"python"
Out[3]: 'excitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingex
citingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcitingexcit
ingexcitingexcitingexcitingpython'

In [4]:
```

Open your own session with Ipython.

Review: a simple translation system

```
>>> english2spanish = {}
>>> type(english2spanish)
<class 'dict'>
>>> # defining key-value pairs:
>>> english2spanish['cat'] = 'gato/a'        # here str -> str (but can be mixed)
>>> english2spanish['dog'] = 'perro/a'
>>> english2spanish['fox'] = 'zorro'
>>> print(english2spanish)                   # let's look at our dictionary:
{'cat': 'gato/a', 'dog': 'perro/a', 'fox': 'zorro'}
# Using our translation system:
>>> animal = 'fox'
>>> print("What is "+animal+" in Spanish?", english2spanish[animal],'!')
What is fox in Spanish? zorro!       # great, but bad spelling! Easy to fix....
>>> print("What is "+animal+" in Spanish?", '¡'+english2spanish[animal].capitalize()+'!')
What is fox in Spanish? ¡Zorro!      # fixed
```

Note: Python allows the creation of complex expressions.

A second dictionary example: store inventory

```
# defining a dictionary (in one expression):
>>> inventory = {'apples': 458, 'oranges': 196, 'pears': 644, 'peaches': 409}
>>> print(inventory['apples'])        # Looking up number of apples
458
>>> inventory['oranges']-=22          # 22 oranges were sold
>>> inventory['peaches']+=100         # 100 peaches were delivered
>>> print(inventory)
{'apples': 458, 'oranges': 174, 'pears': 644, 'peaches': 509}
>>> inventory['melons']
Traceback (most recent call last):
  File "", line 1, in <module>
KeyError: 'melons'
>>> inventory.get('melons', 0)        # look up with default value, does not give an err.
0
>>> inventory.get('oranges', 0)       # get returns the correct number, when present!
174
>>> del inventory['oranges']          # Our shop stops having oranges...
>>> print(inventory)
{'apples': 458, 'pears': 644, 'peaches': 509}
```

Note: `a+=1` is shorthand for `a=a+1`

Quiz

```
1   inventory  = {'skis': 15, 'ski boards': 35}    # defining a dictionary
2   statement1 = 'skis' in inventory
3   statement2 = 'boots' in inventory
```

What is the value and type of `statement1` and `statement2`?

Quiz: performing tests

```
1   inventory  = {'skis': 15, 'ski boards': 35}     # defining a dictionary
2   statement1 = 'skis' in inventory
3   statement2 = 'boots' in inventory
```

The statements `statement1` and `statement2` evaluate to booleans.

You can use them as tests (*for and within* your code).

Question: Are 'skis' in our inventory?

Performing tests: comparison operators

- Consider x, y variables names (int, float, strings, lists, …)
- Comparisons output boolean variables, e.g.:

```
x > y
x >= y      # larger or equal to
x < y
x <= y      # less or equal to
x == y      # equals
x != y      # not equal to
```

What happens when you compare strings? … lexicographic order! I.e. `string1 < string2` iff `string1_j < string2_j` for all `j`. Shorter words are padded with blanks, which is considered smaller than every element in `string1`

Combining tests: Logic operators on booleans

A	B	not A	A and B	A or B
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Control flow: branching with if statements

An example program:

```
if x>0:
    print("positive")        # Note: colon & indentation!
elif x<0:                    # else if
    print("negative")
else:                        # alternative case
    print("zero")            # good practice 4 spaces / use tab!
```

```
if a>0 and isinstance(a,int):        # Example for `and` as well as type checking!
  print("a is a natural number")
```

Questions?

Quiz: What will this program print?

```
if not False:
    print('EPFL')
else:
    print('ETHZ')
```

Quiz: What will this program print?

```
if not False:
    print('EPFL')
else:
    print('ETHZ')
```

The expression `not False` is `True`, so 'EPFL'!

If variants

```
if <condition>:    # else is not required!
  <expression>
  <expression>
  ...
```

```
if <condition>:
  <expression>
  <expression>
  ...
else:
  <expression>
  <expression>
  ...
```

```
if <condition>:
  <expression>
  <expression>
  ...
elif <condition>:
  <expression>
  <expression>
  ...
elif ...: # as many else if as you want!
  <expression>
  <expression>
  ...
else:
  <expression>
  <expression>
  ...
```

Control flow: while statements

```
while <condition>:  # while condition is true, carry out indented expr.; then expr._post
    <expression>
    <expression>
    ...

<expression_post>
```

Additional control: break statements

- immediately exits the *current* loop
- thus, skips remaining expressions in this loop

```
while <condition>:
    <expression>
    if <condition>:
      break           # exits while loop!
    <expression>
    ...

<expression_post>
```

Examples

```
1   i=0
2   while i<12:
3       print(i)
4       i+=1            # note: `i+=1` is python shorthand for `i=i+1`
```

A more compact way of writing this is:

```
1   for n in range(12):
2       print(n)
```

Here

- `range(start,stop,step)` creates a list of numbers from start to stop incremented with step.
- By default, `range(12)`, has `start=0` and `step=1`.

Remember: you can ask for help locally

Of course you can also go to the online help, but you don't need to.

```
>>> help(range)        # help in Python console
Help on class range in module builtins:
class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence of integers from start (inclusive)
 |  to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
 |  start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
 |  These are exactly the valid indices for a list of 4 elements.
 |  When step is given, it specifies the increment (or decrement).
... (output truncated)
```

```
In [1]: range?        # help in ipython/Jupyter
Init signature: range(self, /, *args, **kwargs)
Docstring:
range(stop) -> range object
range(start, stop[, step]) -> range object

... (output truncated)
```

Quiz: What is the value of counter at the end?

```
counter=0
for i in range(3,51,5):
  counter+=1
  if counter==5:
    counter+=1
    break
    counter+=1

print(counter)
```

Quiz: What is the value of counter at the end?

```
counter=0
for i in range(3,51,5):
  counter+=1
  if counter==5:       # enters loop with counter=5
    counter+=1
    break              # exit for loop here, thus counter = 6
    counter+=1

print(counter)
```

Quiz II: What is the value of counter at the end?

```
1
2   counter=0
3   while counter<8:
4     counter+=1
5     while counter<11:
6       print("hello!")
7       break
8
9   print(counter)
10
```

Quiz II: What is the value of counter at the end?

```
1
2   counter=0
3   while counter<8:          # We exit at counter = 8
4     counter+=1
5     while counter<11:
6       print("hello!")
7       break                 # This break only exits the *inner loop*
8
9
10  print(counter)
11
```

Simply 8, no tricks here. The first while loop's termination is hit then.

Bonus: How often is "hello" printed?

for loops

- control the number of iterations
- can end early with `break`
- uses a counter
- each `for` loop can be written as a `while` loop!

while loops

- unbounded number of iterations (e.g. `while true:`)
- can end early with `break`
- can use counter, which needs to be *initialized* and *incremented* in the loop
- not every `while` loop can we written as a `for` loop

You can loop over sequential objects

i.e, strings, lists, ...

```
>>> for letter in 'EPFL':
        print(letter)
E
P
F
L
```

```
>>> for element in [22,'99af']:      # can loop over mixed types
        print(element)
22
'99af'
```

```
>>> squares = []
>>> for x in [1,2,3,4,5]:
        squares.append(x**2)        # appending new elements to the list!
>>> print(squares)
[1, 4, 9, 16, 25]
```

Back to the Swiss language greetings...

```
1   swiss_greetings = ['Bonjour', "Grüzi", 'Ciao', 'Allegra']
2   name = 'Seppl'
3   for greeting in swiss_greetings:
4     if greeting == 'Bonjour':
5       print(greeting, name,'!!!')           # spaces will be added between each string
6     else:
7       print(greeting +" "+ name+'!!!')      # manually add spaces as needed, full control!
8
9
10  Bonjour Seppl !!!
11  Grüzi Seppl!!!
12  Ciao Seppl!!!
13  Allegra Seppl!!!
```

Questions?

Tuples

- ordered sequences of objects that *cannot* be changed (immutable)

```
>>> T = (0,1,2,3)      # a 4-item tuple
>>> len(T)
4
>>> T + (5,6)          # concatenation
(0,1,2,3,5,6)
>>> T[0]               # slicing
0
>>> T[0]=3
... error ...
TypeError: 'tuple' object does not support item assignment
```

Why tuples? Their immutability is the point; they are used for control…

```
>>> x, y = 3, 4        # here we define two variables in one line
>>> print(x,y)
(3,4)
>>> (x,y) = (y,x)      # convenient way to swap values!
>>> print(x,y)
(4,3)
```

Sets

- unordered collection of unique and immutable objects

Example definitions and operations:

```
>>> x = set('abcde')
>>> y = set('bdxyz')
>>> print(x)
{'c', 'e', 'd', 'a', 'b'}
>>> x - y                 # difference set
{'a', 'c', 'e'}
>>> x | y                 # union
{'a', 'b', 'c', 'd', 'e', 'x', 'y', 'z'}
>>> 'a' in x              # membership in sets
True
```

Questions?

NumPy: Python's foundation of scientific computing

NumPy is a fundamental package that

- provides multidimensional array objects (e.g. matrix of shape (M x N), but also vectors of shape (N) and tensors of shape (M_1 x M_2 x M_3 ... x M_N))
- with various derived objects and powerful mathematical functions
- numerical problems can often be described with high-level code (vectorized formulas), thus enabling scientific code that is both easy to maintain and read
- vectorization also fuels speed gains (see later classes)

Importing and checking the version:

```
In [1]: import numpy as np      # NumPy is imported like this.

In [2]: np.__version__          # checking the version of NumPy
Out[2]: '1.20.1'                # [version](https://pypi.org/project/numpy/) in my machine
```

NumPy arrays: the ndarray object

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).
- changing the size of an ndarray will create a new array and delete the original.
- arrays can have any dimensionality
- arrays contain numbers of the *same* type (e.g. floats, complex numbers, booleans)
- arrays are sequential objects and can be indexed by the `[]` operator

A first example

```
# Generate the integers from zero to eight and # re-arrange them into a 3 x 3 array
In [1]: x = np.arange(9).reshape((3, 3))
In [2]: x
Out[2]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
In [3]: type(x)
Out[3]: numpy.ndarray
In [4]: x.dtype?
        Type:        dtype[int64]
        String form: int64
        Length:      0
        File:        ~/opt/anaconda3/lib/python3.8/site-packages/numpy/__init__.py
        Docstring:   <no docstring>

```

Shapes of arrays

```
In [1]: x = np.arange(9)
In [2]: x
Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])
In [3]: np.shape(x)
Out[3]: (9,)
In [4]: E = np.eye(3)
In [5]: E
Out [5]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
In [6]: np.shape(E)
Out [6]: (3,3)
In [7]: nix=np.zeros((4,3,2))   # define an array of zeros, also np.ones, etc.
In [8]: np.shape(nix)
Out [8]: (4,3,2)                # next page for looking at this ...
```

A (visually big) array

```
In [7]: nix=np.zeros((4,3,2))
In [8]: np.shape(nix)
Out [8]: (4,3,2)
In [8]: nix                          # let's look at the [[ ... ]]
Out [8]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

```
# Let's illustrate standard slicing: `start:stop:step`
In [1]: x = np.arange(9).reshape((3, 3))
In [2]: x
Out[2]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
In [3]: x[:2, :]      # first two rows of x
Out[3]:
array([[0, 1, 2],
       [3, 4, 5]])
In [4]: x[:,1:]       # columns 1 to the end
Out [4]:
array([[1, 2],
       [4, 5],
       [7, 8]])
In [5]: x[::2,::-1]   # every second row and reverse the columns!
Out[5]:
array([[2, 1, 0],     # reversing as step = -1
      [8, 7, 6]])
In [11]: x            # Note, x is unchanged -- we just used views, but ...
Out[11]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Views of memory - shallow copy

```
In [1]: import numpy as np
   ...: x =  np.arange(9).reshape((3, 3))
   ...: y = x[::2,::-1]        # creating a view of the memory
   ...: print(x)
[[0 1 2]
 [3 4 5]
 [6 7 8]]
In [2]:  print(y)
[[2 1 0]
 [8 7 6]]
In [3]: y[0,2]=33
In [4]: print(y)
[[ 2  1 33]                  # obviously y is changed...
 [ 8  7  6]]
In [5]: print(x)
[[33  1  2]                  # NOTICE: x is changed. x and y point to the same memory.
 [ 3  4  5]                  # You can avoid this behavior, by creating a copy during
 [ 6  7  8]]                 # assignment: `y=x[::2,::-1].copy()` (then y is a new obj.)
```

Here, y is a called a view of x (or a shallow copy).

Note: I’m not displaying "Out [x]: for space reasons here"

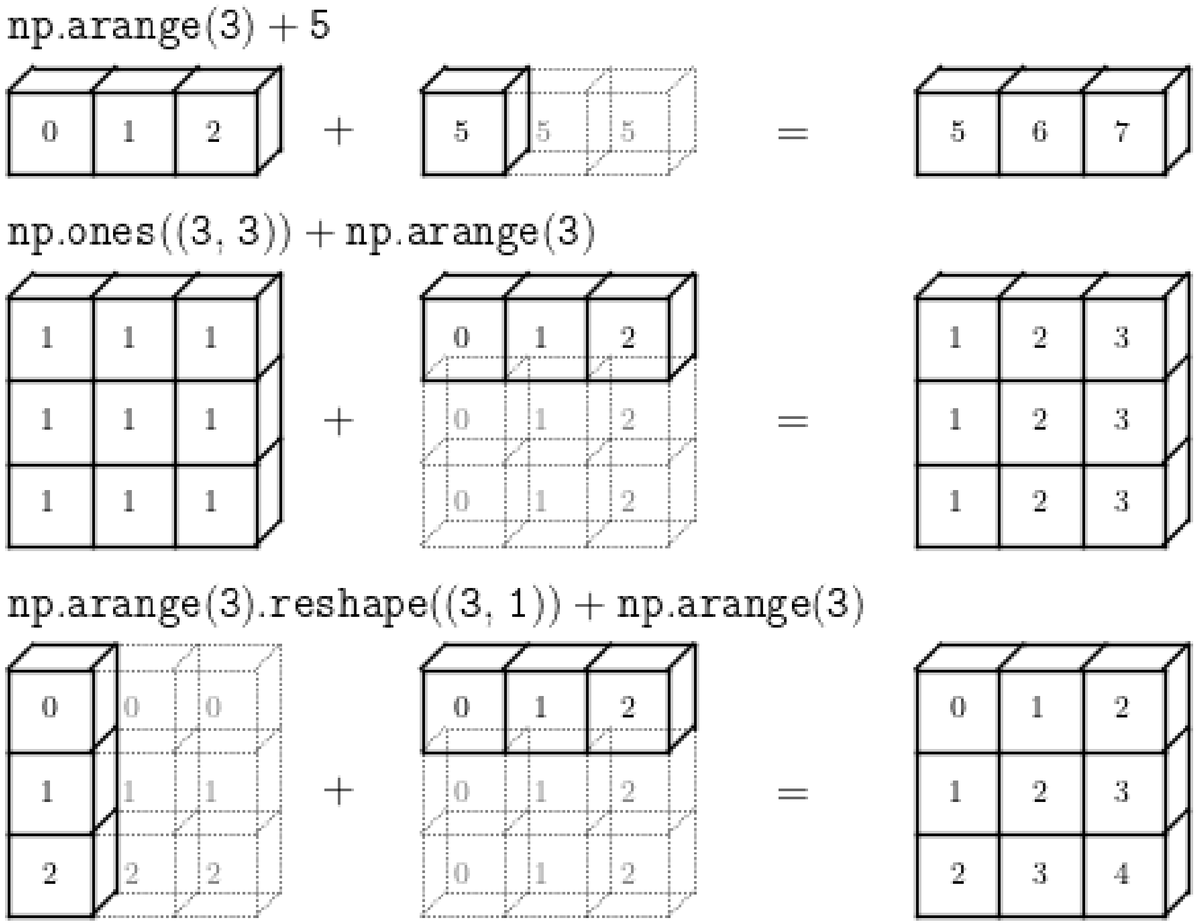
Vectorized operations in NumPy

are implemented in C. Always try to use vectorization.

```
In [1]: import numpy as np
In [2]: x,y=np.arange(10),np.ones(10)
In [3]: x*y
Out[3]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
In [4]: x+y
Out[4]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
In [5]: x-y
Out[5]: array([-1.,  0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
In [6]: x**y
Out[6]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
In [7]: y/x
<ipython-input-7-d2d3fa2acc3f>:1: RuntimeWarning: divide by zero encountered in true_divi
  y/x
Out[7]:
array([       inf, 1.        , 0.5       , 0.33333333, 0.25      ,
       0.2       , 0.16666667, 0.14285714, 0.125     , 0.11111111])
```

Vectorization for mismatched arrays?

When the shapes are different, but share a common shape dimension, operations are *broadcasted!*



Source: astroml.org

Vectorization of mismatched arrays?

```
In [8]: x=np.array([3,18,9.])

In [9]: m=np.arange(6).reshape((2,3))

In [10]: m
Out[10]:
array([[0, 1, 2],
       [3, 4, 5]])

In [11]: m+x
Out[11]:
array([[ 3., 19., 11.],
       [ 6., 22., 14.]])
```

To save memory, broadcasted arrays are never physically constructed!

Click here to learn about the broadcasting rules.

Quiz: What is output computing here?

```
x = np.arange(0,100,3)
y = x**2
output = (y[1:]-y[:-1])/(x[1:]-x[:-1])
```

Quiz: computing finite differences

```
x = np.arange(0,100,3)
y = x**2
dy_over_dx = (y[1:]-y[:-1])/(x[1:]-x[:-1])
```

Subtracting those slices, effectively computes for *all* i:

$$\frac{\Delta y_i}{\Delta x_i} = \frac{y(i+1) - y(i)}{x(i+1) - x(i)}$$

This vector has length `len(x)-1`.

There is also a built-in function, np.diff for taking discrete differences.

Common type-specific operations…

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])
```

Quiz: What is computed here?

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

What is j?

Quiz: 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

s

```
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)
```

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 [6]: 3 + 2*np.random.randn()        # Gen. normal distr. sample with mean 3 and std 2
Out[6]: 4.067920463989419
```

See np.random for more options!

Questions?

Today's summary

We learned about:

- built-in objects: `set`, `tuple`,
- routing mechanisms: `if`, `while`, `for`
- introduction to NumPy

Logistics:

- create a personal GitHub account.
- find two teammates and sign up here. You will need your GitHub names. You also need to use your EPFL login for access to this data sheet!

Please sign up by the end of the week (we will also announce this via Moodle).

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

After lunch:

- Monday 13 - 15: exercises (5 groups)
  - CO4 ← A-B + Y + Z (TA: Albert Dominguez Mantes)
  - CO5 ← C-F+V (TAs: Hale-Seda Radoykova, Shaokai Ye)
  - CO260 ← G-L+W (TA: Oliver Ulrich)
  - CO6 ← M-P (TA: Haoze Qi)
  - CO023 ← Q - U (TA: Andy Bonnetto)
- Monday 15:15 - 16: my office hours at SV 2811