# Welcome to BIO-210

## Applied software engineering for life sciences

September 23, 2024 – Lecture 1

Prof. Alexander MATHIS

EPFL

|  | Date | Topic | Software version | Software releases | Grading / 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 and GitHub (+installation VS Code) |  |  |  |
| 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 and 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 and code optimization | v6 | v5 | code review |
| 11 | 24/11/2025 | Object oriented programming | v7 | v6 | graded (speed) |
| 12 | 01/12/2025 | Model analysis and 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) |

# Announcements

- We posted solutions to the notebook from last week
- next week there is no class (install Python, VS code, git) --> see guide in the course materials
- then: Week 3 - 7.5% of the grade (online available Monday 29.9 - Friday 3.10).
- Monday 16:15 - 17: my office hours at SV 2811 (email: alexander.mathis@epfl.ch)

# Preparation for projects:

- create a personal GitHub account.
- find two teammates and sign up here. You will need your GitHub names. Only fill out the green section, pick any team number. If you do not know any teammates (yet), you can just put your name in the form and state that you are looking for teammates. Please sign up by the end of next 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!

```
Teaching materials — IPython: Code/Teaching materials — python3.12 ~/miniconda/envs/apertus/bin/ipython — 104×32
(apertus) alex@alexs-macbook-air Teaching materials % ipython
Python 3.12.0 | packaged by Anaconda, Inc. | (main, Oct  2 2023, 12:22:05) [Clang 14.0.6 ]
Type 'copyright', 'credits' or 'license' for more information
IPython 9.5.0 -- An enhanced Interactive Python. Type '?' for help.
Tip: Use `object?` to see the help on `object`, `object??` to view its source

In [1]: 3+3
Out[1]: 6

In [2]: 33*"glue"
Out[2]: 'gluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluegluglu
egluegluegluegluegluegluegluegluegluglue'

In [3]: a=3

In [4]: 3*a
Out[4]: 9

In [5]: from mlx_lm import load, generate
   ...:
   ...: model, tokenizer = load("mlx-community/Apertus-8B-Instruct-2509-8bit")
   ...:
   ...: prompt = "How can I learn Python?"
   ...:
   ...: if tokenizer.chat_template is not None:
   ...:     messages = [{"role": "user", "content": prompt}]
   ...:     prompt = tokenizer.apply_chat_template(
   ...:         messages, add_generation_prompt=True
   ...:     )
   ...:
   ...:
   ...: response = generate(model, tokenizer, prompt=prompt, verbose=True)
```

Open your own session with Ipython.

# Dictionaries

- collection of key-value pairs that maps from keys to values.
- the keys can be any immutable type, and the values can be any type.
- like lists they can also be mixed and nested
- a dict is denoted by *curly brackets* {}

# A first example: a simple translation system

```
1   >>> english2spanish = {}
2   >>> type(english2spanish)
3   <class 'dict'>
4   >>> english2spanish['cat'] = 'gato/a'    # defining key-value pairs
5   >>> # Here str -> str (but can be mixed)
6   >>> english2spanish['dog'] = 'perro/a'
7   >>> english2spanish['fox'] = 'zorro'
8
9   # Using our translation system:
10  >>> animal = 'fox'
11  >>> print("What is "+animal+" in spanish?", english2spanish[animal],', tío/a!')
12  What is fox in spanish? zorro, tío/a!
13  >>> # great, but bad spelling! This is easy to fix....
14  >>> print("What is "+animal+" in spanish?", \
15   '¡'+english2spanish[animal].capitalize()+', tío/a!')
16  What is fox in spanish? ¡Zorro, tío/a!
```

Note: Python allows the creation of complex expressions.

# A second dictionary example: store inventory

```
1   # defining a dictionary (in one expression):
2   >>> inventory = {'apples': 458, 'oranges': 196, 'pears': 644, 'peaches': 409}
3   >>> print(inventory['apples'])        # Looking up number of apples
4   458
5   >>> inventory['oranges']-=22          # 22 oranges were sold
6   >>> inventory['peaches']+=100         # 100 peaches were delivered
7   >>> print(inventory)
8   {'apples': 458, 'oranges': 174, 'pears': 644, 'peaches': 509}
9   >>> inventory['melons']
10  Traceback (most recent call last):
11    File "", line 1, in <module>
12  KeyError: 'melons'
13  >>> inventory.get('melons', 0)        # look up with default value, does not give an err.
14  0
15  >>> inventory.get('oranges', 0)       # get returns the correct number, when present!
16  174
17  >>> del inventory['oranges']          # Our shop stops having oranges...
18  >>> print(inventory)
19  {'apples': 458, 'pears': 644, 'peaches': 509}
```

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

# 🧩🤔: What is the value and type of `statement1` and `statement2`?

```
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`?

# 🧩🤔: 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. The first is `True`, second one `False`.

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 (e.g., `` `string1<string2` ``)? Python compares strings character by character from left to right until it finds the first differing character. The string with the smaller character at that position is considered smaller.

# Examples

```
>>> "apple" < "strawberry"
True                    # 'a' < 's' at position 0
>>> "cat" < "car"
False                   # 'c'=='c', 'a'=='a', but 't' > 'r' at position 2
>>> "hello" < "helm"
True                    # 'h'=='h', 'e'=='e', 'l'=='l', but 'l' < 'm' at position 3
#Character Comparison Uses ASCII/Unicode Values
>>> ord('A'), ord('a')  # Check ASCII values of characters.
(65, 97)
>>> 'A' < 'a'           # Capital letters come first in ASCII
True
```

NOTE: ASCII (American Standard Code for Information Interchange) encodes English letters, digits, punctuation, control characters as 7-bit encoding (128 possible characters: 0-127). ASCII is a small subset of Unicode. Unicode is the modern standard that handles all world languages and symbols incl. emojis, while ASCII only handles basic English characters. `ord('🐍') = 128013` (Python snake)

# Combining tests: Logic operators on booleans

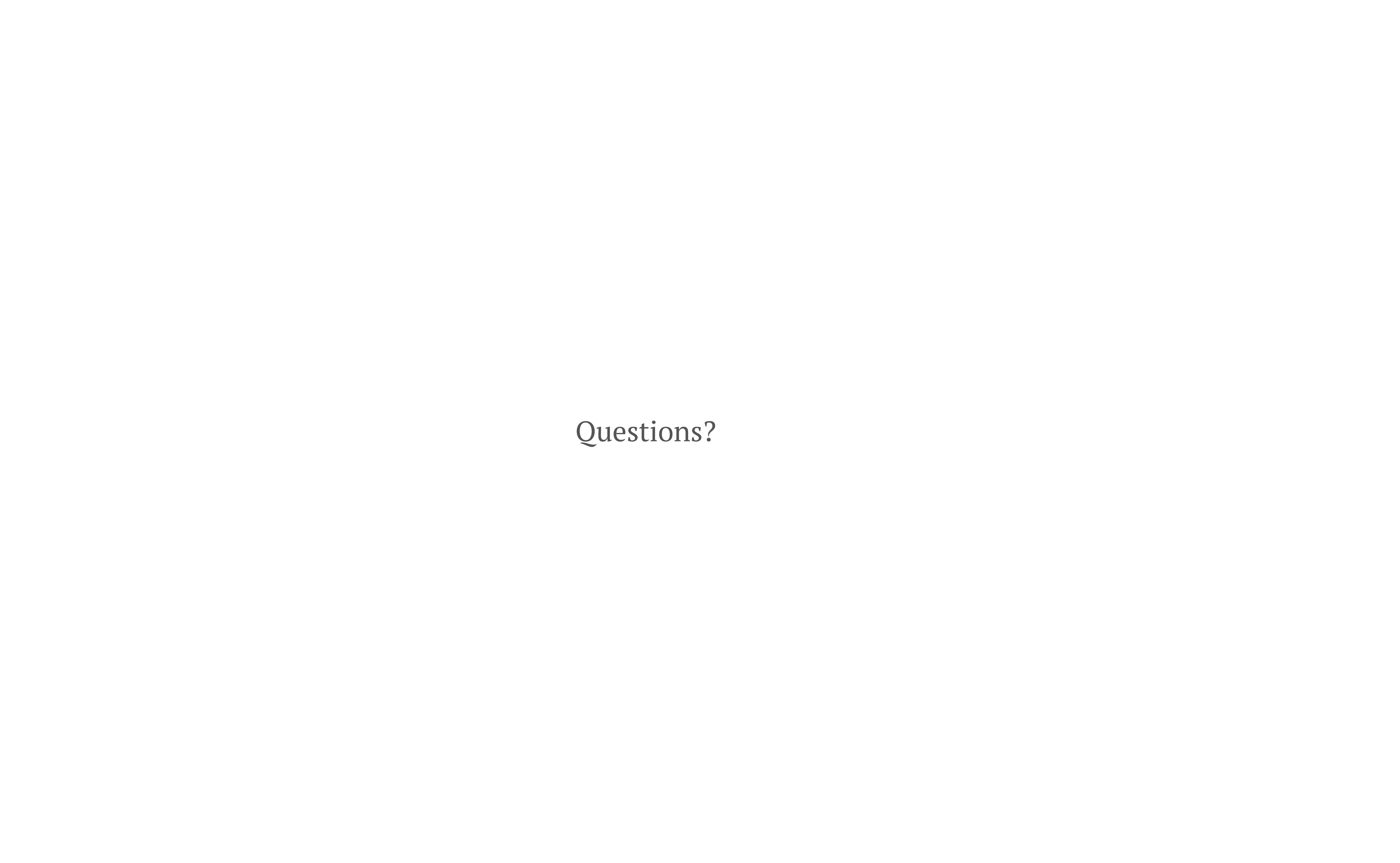
| A | B | not A | A and B | A or B |
|---|---|---|---|---|
| True | True | False | True | True |
| True | False | False | False | True |
| False | True | True | False | True |
| False | False | True | False | False |

# Reminder: branching with if statements

An example program:

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

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


Questions?

# 🧩🤔: What will this program print?

```
1   if not False:
2       print('EPFL')
3   else:
4       print('ETHZ')
```

# 🧩🤔: What will this program print?

```python
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

```
1   while <condition>:  # while condition is true, carry out indented expr.; then expr._post
2       <expression>
3       <expression>
4       ...
5
6   <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

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

A more compact way of writing this is:

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

Here

- `range(start,stop,step)` creates a range object that produces numbers from start to stop incremented with step.
- Its not a list, compare `print(range(10))` and `range(0, 10)`
- By default, `range(12)`, has `start=0` and `step=1`.

# 💡: you can ask for help locally

Of course you can also go to the online help, but you do not need to.

```
1   >>> help(range)        # help in Python console
2   Help on class range in module builtins:
3   class range(object)
4    |  range(stop) -> range object
5    |  range(start, stop[, step]) -> range object
6    |
7    |  Return an object that produces a sequence of integers from start (inclusive)
8    |  to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
9    |  start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
10   |  These are exactly the valid indices for a list of 4 elements.
11   |  When step is given, it specifies the increment (or decrement).
12  ... (output truncated)
```

```
1   In [1]: range?         # help in ipython/Jupyter
2   Init signature: range(self, /, *args, **kwargs)
3   Docstring:
4   range(stop) -> range object
5   range(start, stop[, step]) -> range object
6
7   ... (output truncated)
```

# 🧩🤔: What is the value of counter at the end?

```
1   counter=0
2   for i in range(3,51,5):
3     counter+=1
4     if counter==4:
5       counter+=1
6       break
7       counter+=1
8
9   print(counter)
```

# 🧩🤔: What is the value of counter at the end?

```
1   counter=0
2   for i in range(3,51,5):
3     counter+=1
4     if counter==4:       # enters loop with counter=4
5       counter+=1
6       break              # exit for loop here, thus counter = 5
7       counter+=1
8
9   print(counter)
```

# 🧩🤔 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
```

# 🧩🤔 II: What is the value of counter at the end?

```
1
2   counter = 0
3   while counter < 8:          # Outer loop: continues while counter < 8
4     counter += 1              # counter becomes: 1, 2, 3, 4, 5, 6, 7, 8
5     while counter < 11:       # Inner loop: only runs if counter < 11
6       print("hello!")         # This prints once per outer loop iteration
7       break                   # Immediately exits INNER loop only
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 (good if you know the number)
- 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, ...

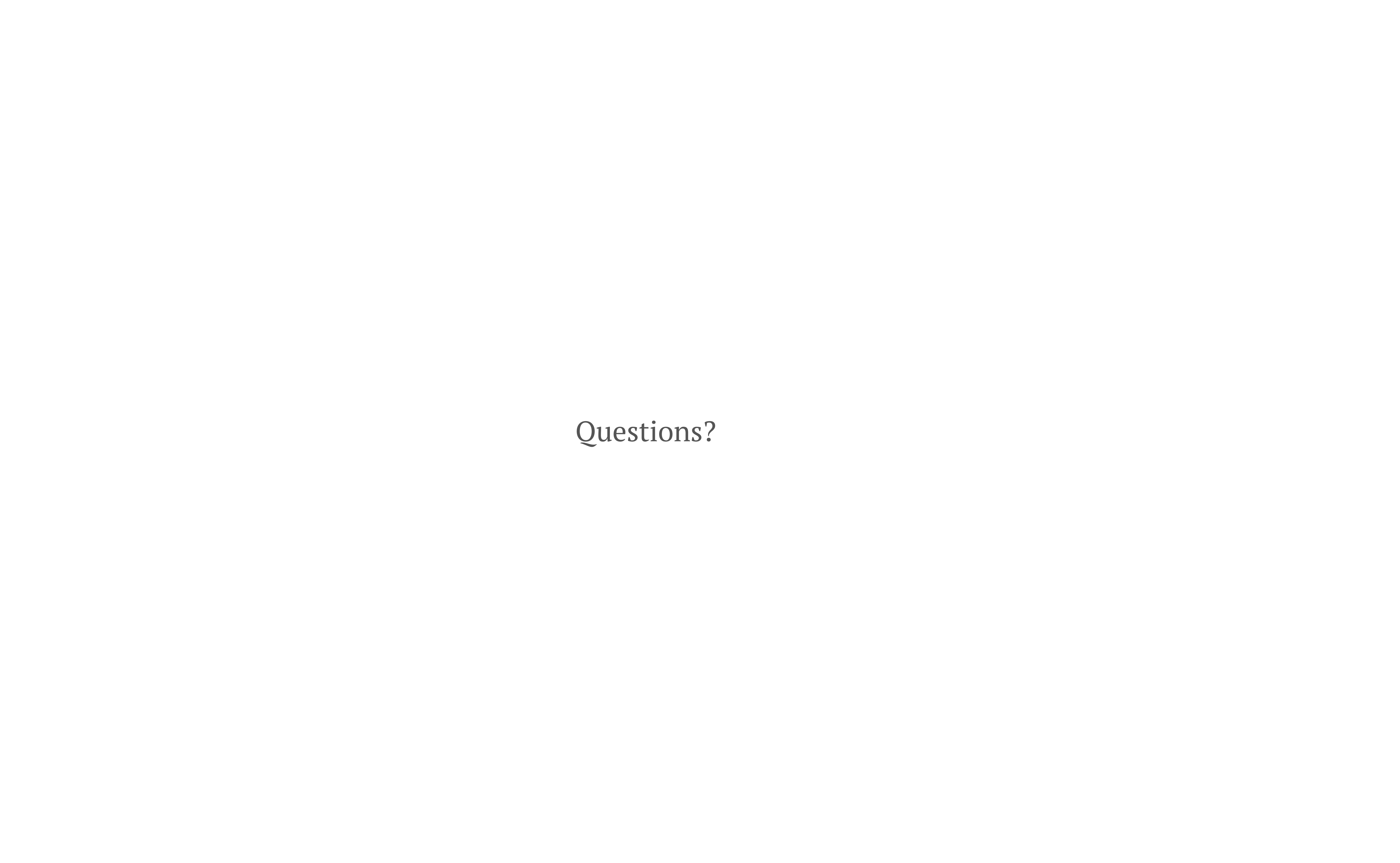
```
1   >>> for letter in 'EPFL':
2           print(letter)
3   E
4   P
5   F
6   L
```

```
1   >>> for element in [22,'99af']:      # can loop over mixed types
2             print(element)
3   22
4   '99af'
```

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

# 🧩🤔: What will be printed?

```
1    >>> "Python" < "python"
2    >>> "123" < "45"
3    >>> "" < "anything"
```

# 🧩🤔: What will be printed?

```
1   >>> "Python" < "python"    # Capital letters come first.
2   True
3   >>> "123" < "45"           # Unicode is used: ord('1')=49, ord('4')=52. "1"<"4": so True.
4   >>> 123 < 45               # Ofc, if we compare numbers we get
5   False
6   >>> "" < "anything"             # An empty string is lexicographically smaller
7   True                       # than any non-empty string
```

# Tuples

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

```
1   >>> T = (0,1,2,3)       # a 4-item tuple
2   >>> len(T)
3   4
4   >>> T + (5,6)           # concatenation
5   (0,1,2,3,5,6)
6   >>> T[0]                # slicing
7   0
8   >>> T[0]=3
9   ... error ...
10  TypeError: 'tuple' object does not support item assignment
```

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

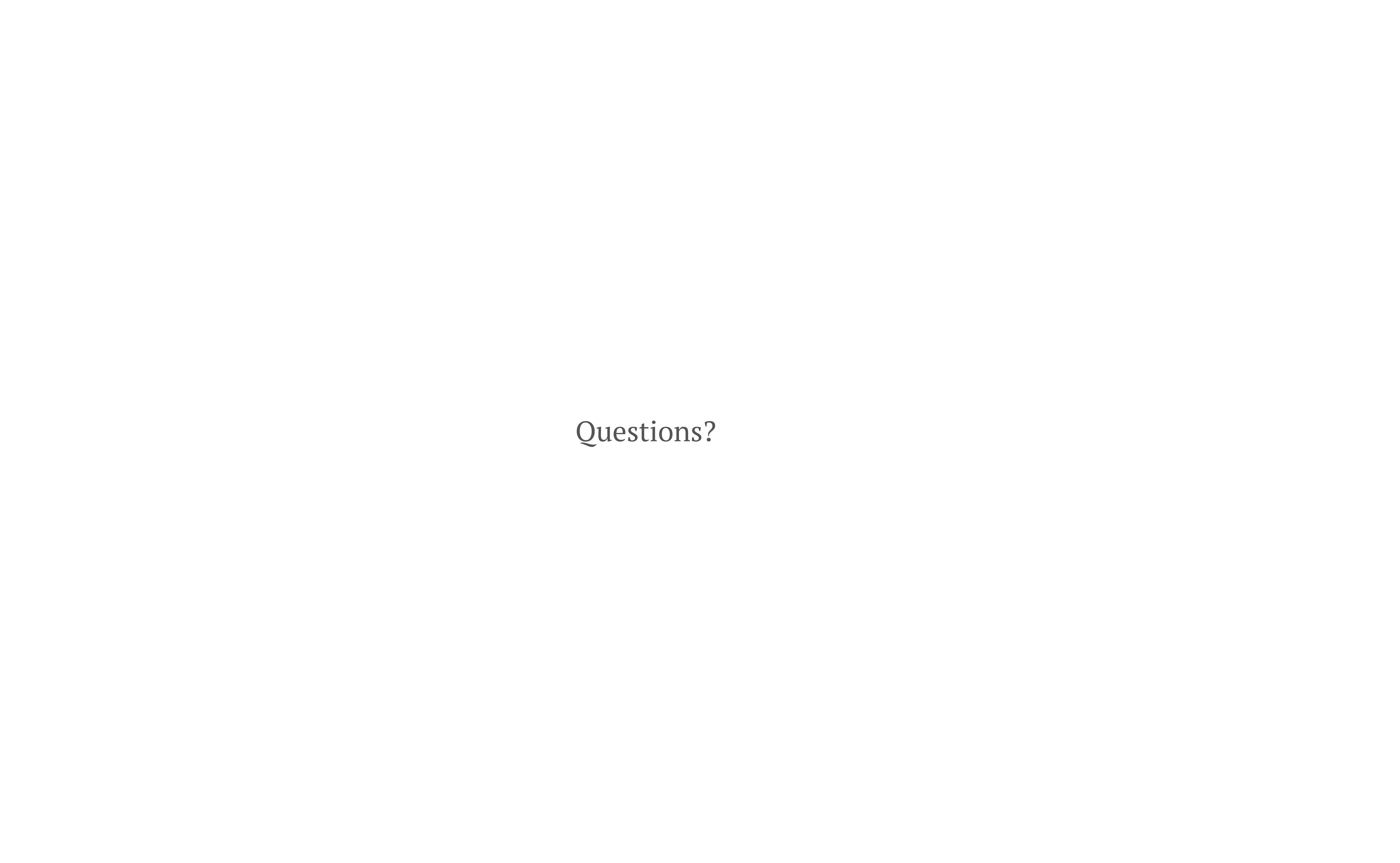
```
1   >>> x, y = 3, 4         # here we define two variables in one line
2   >>> print(x,y)
3   (3,4)
4   >>> (x,y) = (y,x)       # convenient way to swap values!
5   >>> print(x,y)
6   (4,3)
```

# Sets

- unordered collection of unique and immutable objects

Example definitions and operations:

```
1   >>> x = set('abcde')
2   >>> y = set('bdxyz')
3   >>> print(x)
4   {'c', 'e', 'd', 'a', 'b'}
5   >>> x - y                   # difference set
6   {'a', 'c', 'e'}
7   >>> x | y                   # union
8   {'a', 'b', 'c', 'd', 'e', 'x', 'y', 'z'}
9   >>> 'a' in x                # membership in sets
10  True
```


Questions?

# Python lists have limitations for computation

```
1   # Problem 1: Performance with large datasets
2   python_list = [i**2 for i in range(1000000)]  # That's slow (we will profile it later)
3
4   # Problem 2: No built-in mathematical operations
5   numbers = [1, 2, 3, 4, 5]
6   # numbers * 2          # This just repeats the list: [1,2,3,4,5,1,2,3,4,5]
7   # numbers + 10         # ERROR! Cannot add scalar to list
8
9   # Problem 3: Mixed types make math impossible
10  mixed = [1, 2.5, 'hello', 3]  # Fine list, but not good for computation
11
12  # NumPy solves these problems (as we will see in detail!)
13  import numpy as np
14  x = np.array([1, 2, 3, 4, 5])
15  x * 2              # Mathematical operation: [2, 4, 6, 8, 10]
16  x + 10             # Broadcasting: [11, 12, 13, 14, 15]
```

# 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:

```
1   In [1]: import numpy as np     # NumPy is imported like this.
2
3   In [2]: np.__version__         # checking the version of NumPy
4   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

```
1   temps_celsius = np.array([20.1, 22.5, 19.8, 25.2])
2   temps_kelvin = temps_celsius + 273.15          # Scientific standard
3   temps_fahrenheit = temps_celsius * 9/5 + 32  # If you travel to the US
4
5   # With lists you would need a loop:
6   temps_celsius = [20.1, 22.5, 19.8, 25.2]
7   kelvin_list = []
8   for temp in temps_celsius:
9       kelvin_list.append(temp + 273.15)
```

# A second example

```
1   # Generate the integers from zero to eight and # re-arrange them into a 3 x 3 array
2   In [1]: x = np.arange(9).reshape((3, 3))
3   In [2]: x
4   Out[2]:
5   array([[0, 1, 2],
6          [3, 4, 5],
7          [6, 7, 8]])
8   In [3]: type(x)
9   Out[3]: numpy.ndarray
10  In [4]: x.dtype?
11         Type:         dtype[int64]
12         String form:  int64
13         Length:       0
14         File:         ~/opt/anaconda3/lib/python3.8/site-packages/numpy/__init__.py
15         Docstring:    <no docstring>
16
```

# Shapes of arrays

```
1   In [1]: x = np.arange(9)
2   In [2]: x
3   Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])
4   In [3]: np.shape(x)
5   Out[3]: (9,)
6   In [4]: E = np.eye(3)
7   In [5]: E
8   Out [5]:
9   array([[1., 0., 0.],
10         [0., 1., 0.],
11         [0., 0., 1.]])
12  In [6]: np.shape(E)
13  Out [6]: (3,3)
14  In [7]: nix=np.zeros((4,3,2))   # define an array of zeros, also np.ones, etc.
15  In [8]: np.shape(nix)
16  Out [8]: (4,3,2)                # next page for looking at this ...
```

# A (visually big) array

```
1   In [7]: nix=np.zeros((4,3,2))
2   In [8]: np.shape(nix)
3   Out [8]: (4,3,2)
4   In [8]: nix                             # let's look at the [[ ... ]]
5   Out [8]:
6   array([[[0., 0.],
7           [0., 0.],
8           [0., 0.]],
9
10         [[0., 0.],
11          [0., 0.],
12          [0., 0.]],
13
14         [[0., 0.],
15          [0., 0.],
16          [0., 0.]],
17
18         [[0., 0.],
19          [0., 0.],
20          [0., 0.]]])
```

```
1   # Let's illustrate standard slicing: `start:stop:step`
2   In [1]: x = np.arange(9).reshape((3, 3))
3   In [2]: x
4   Out[2]:
5   array([[0, 1, 2],
6          [3, 4, 5],
7          [6, 7, 8]])
8   In [3]: x[:2, :]      # first two rows of x
9   Out[3]:
10  array([[0, 1, 2],
11         [3, 4, 5]])
12  In [4]: x[:,1:]       # columns 1 to the end
13  Out [4]:
14  array([[1, 2],
15         [4, 5],
16         [7, 8]])
17  In [5]: x[::2,::-1]   # every second row and reverse the columns!
18  Out[5]:
19  array([[2, 1, 0],     # reversing as step = -1
20        [8, 7, 6]])
21  In [11]: x            # Note, x is unchanged -- we just used views, but ...
22  Out[11]:
23  array([[0, 1, 2],
24         [3, 4, 5],
25         [6, 7, 8]])
```

# Views of memory - shallow copy

```
1   In [1]: import numpy as np
2      ...: x =  np.arange(9).reshape((3, 3))
3      ...: y = x[::2,::-1]        # creating a view of the memory
4      ...: print(x)
5   [[0 1 2]
6    [3 4 5]
7    [6 7 8]]
8   In [2]:  print(y)
9   [[2 1 0]
10   [8 7 6]]
11  In [3]: y[0,2]=33
12  In [4]: print(y)
13  [[ 2  1 33]                    # obviously y is changed...
14   [ 8  7  6]]
15  In [5]: print(x)
16  [[33  1  2]                    # NOTICE: x is changed. x and y point to the same memory.
17   [ 3  4  5]                    # You can avoid this behavior, by creating a copy during
18   [ 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"

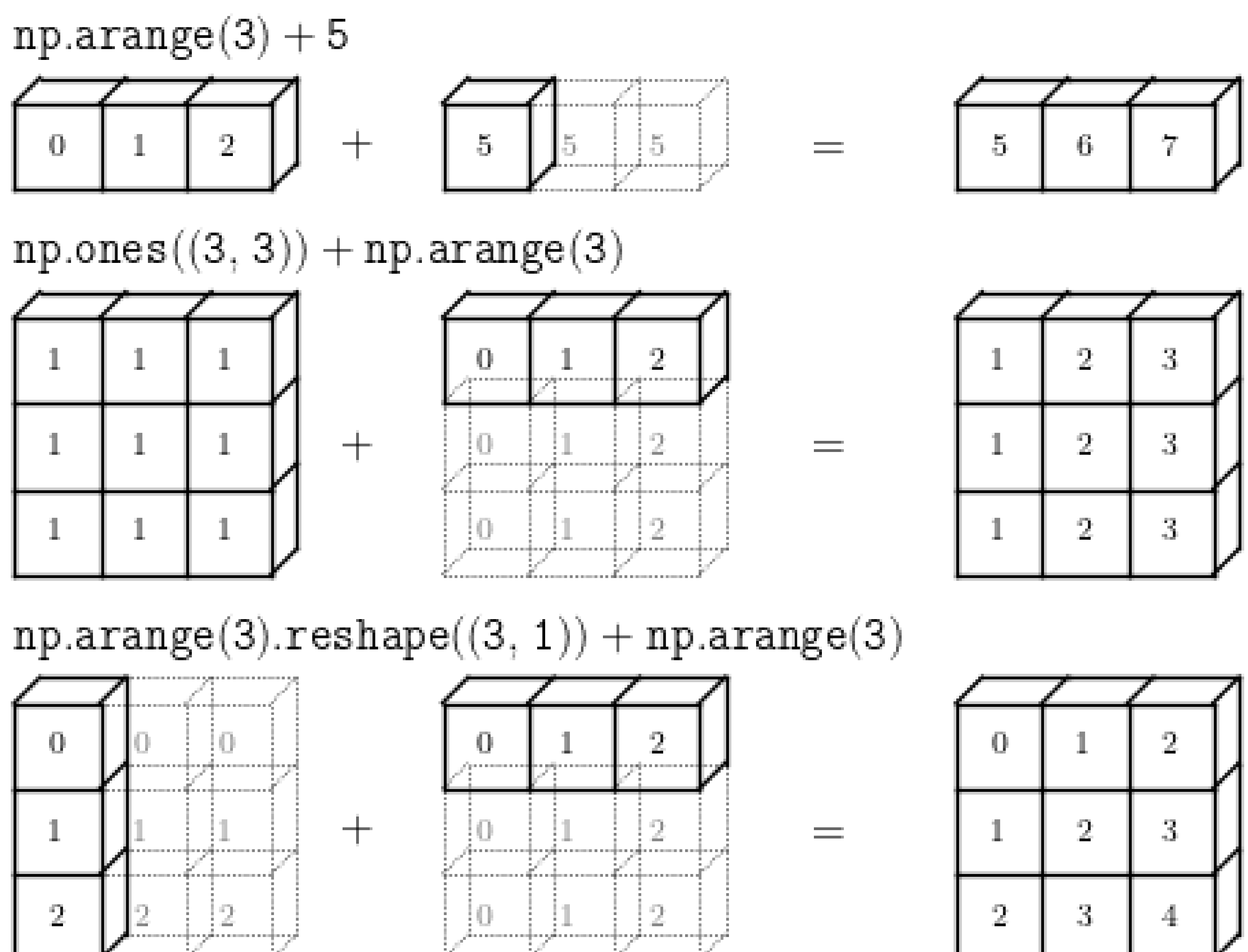
# Vectorized operations in NumPy

are implemented in C. Always try to use vectorization.

```
1   In [1]: import numpy as np
2   In [2]: x,y=np.arange(10),np.ones(10)
3   In [3]: x*y
4   Out[3]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
5   In [4]: x+y
6   Out[4]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
7   In [5]: x-y
8   Out[5]: array([-1.,  0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
9   In [6]: x**y
10  Out[6]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
11  In [7]: y/x
12  <ipython-input-7-d2d3fa2acc3f>:1: RuntimeWarning: divide by zero encountered in true_divi
13    y/x
14  Out[7]:
15  array([       inf, 1.        , 0.5       , 0.33333333, 0.25      ,
16         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?

```
1   In [8]: x=np.array([3,18,9.])
2
3   In [9]: m=np.arange(6).reshape((2,3))
4
5   In [10]: m
6   Out[10]:
7   array([[0, 1, 2],
8          [3, 4, 5]])
9
10  In [11]: m+x
11  Out[11]:
12  array([[ 3., 19., 11.],
13         [ 6., 22., 14.]])
```

To save memory, broadcasted arrays are never physically constructed!

Click here to learn about the broadcasting rules.

# 🧩🤔: What is output computing here?

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

# 🧩🤔: 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

```
1   In [1]: X=np.linspace(0,2*np.pi,10)      # create ar. /w 10 linearly sep. pts in [0,2pi]
2   array([0.        , 0.6981317 , 1.3962634 , 2.0943951 , 2.7925268 ,
3          3.4906585 , 4.1887902 , 4.88692191, 5.58505361, 6.28318531])
4
5   In [2]: Y=np.sin(X)                      # calculate sine, vectorized of course
6   array([ 0.00000000e+00,  6.42787610e-01,  9.84807753e-01,  8.66025404e-01,
7           3.42020143e-01, -3.42020143e-01, -8.66025404e-01, -9.84807753e-01,
8          -6.42787610e-01, -2.44929360e-16])
9
10  In [3]: Y>0                              # where is this array >0? Vectorized
11  Out[3]:
12  array([False,  True,  True,  True,  True, False, False, False, False,
13         False])
```

# 🧩🤔: What is computed here?

```python
X = np.logspace(-1,2,7)
Y = np.log10(X)
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

```
1   In [1]: phi = np.pi/3                                           # 60 degrees
2   In [2]: e1,e2=np.array([1,0]),np.array([0,1])                   # two basis vectors of R^2
3   In [3]: rot = np.array([[np.cos(phi), -np.sin(phi)],[np.sin(phi), np.cos(phi)]])
4   In [4]: print(rot)
5   Out [4]:
6   [[ 0.5        -0.8660254]
7    [ 0.8660254  0.5      ]]
8   In [6]: print(np.matmul(rot,e1))                               # matrix multiplication
9   Out [6]:
10  [0.5       0.8660254]                                         # vector rotated to (cos(phi), sin(phi)
11
```

# A minimalist linear algebra example

```
1    In [1]: phi = np.pi/3                                     # 60 degrees
2    In [2]: e1,e2=np.array([1,0]),np.array([0,1])             # two basis vectors of R^2
3    In [3]: rot = np.array([[np.cos(phi), -np.sin(phi)],[np.sin(phi), np.cos(phi)]])
4    In [4]: print(rot)
5    Out [4]:
6    [[ 0.5        -0.8660254]
7     [ 0.8660254  0.5      ]]
8    In [5]: print(np.matmul(rot,e1))                          # matrix multiplication
9    Out [6]:
10   [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

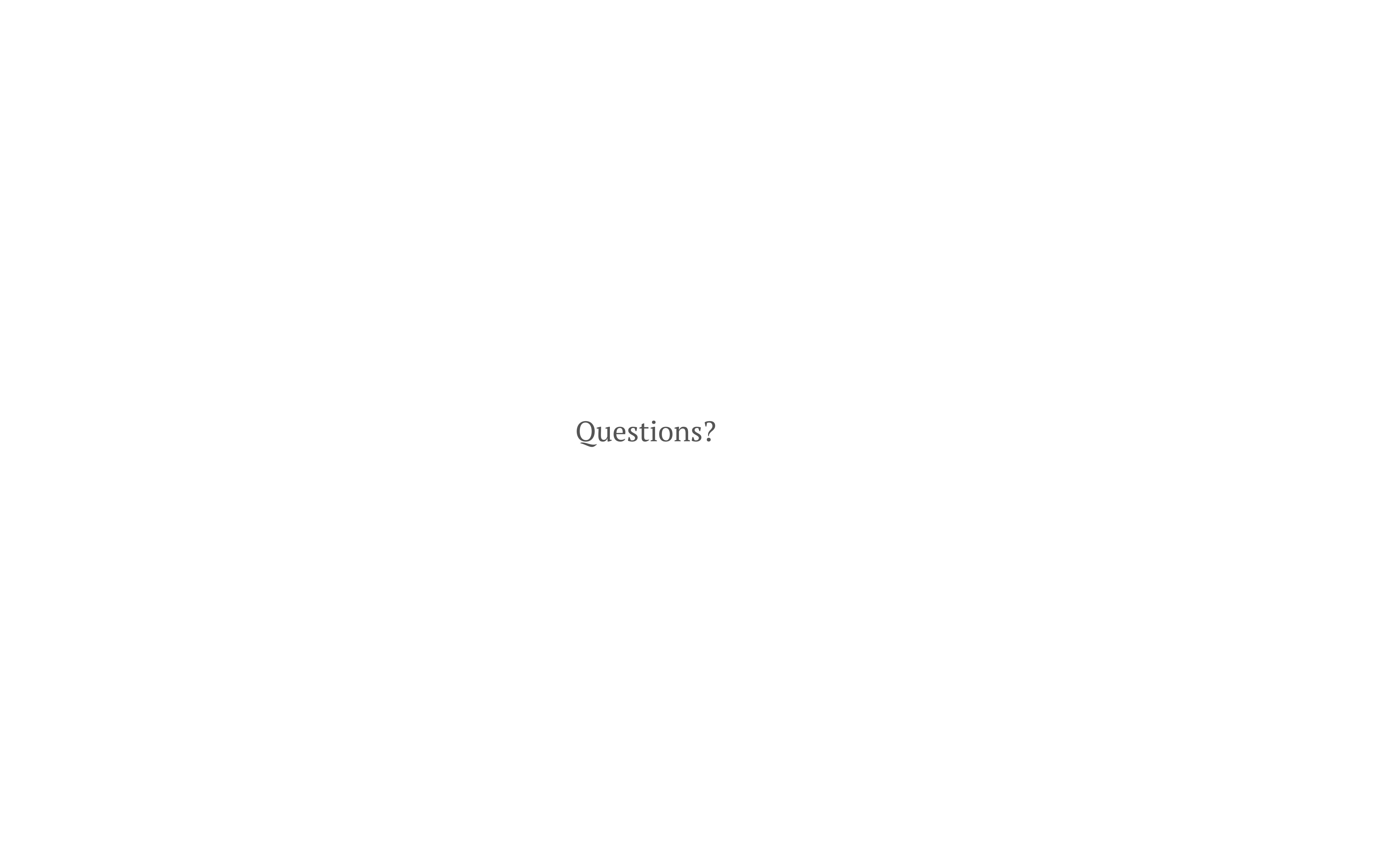
```
1   In [1]: data = np.array([[1231,np.nan],[23.123, 0],[0,1]]) # Notice: missing data
2   In [2]: np.mean(data)                                      # the mean is also nan!
3   nan
4   In [3]: np.nanmean(data)                                   # nanmean omits nans
5   Out[3]: 251.02460000000002
6   In [4]: np.nanmean(data,axis=0)                            # mean along a specific axis
7   Out[4]: array([418.041,   0.5  ])
8
9   In [5]: np.nanmean(data,axis=1)
10  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

```
1    In [1]: import numpy as np
2    In [2]: x = np.random.randint(10)      # Generate a random integer from 0 to 10 (uniform)
3    In [3]: x
4    Out[3]: 5
5
6    In [4]: np.random.rand()               # Generate a random float from 0 to 1 (uniform)
7    Out[4]: 0.4150299187382156
8    In [5]: np.random.rand(2,2)            # Gen. random floats in array of 2 rows & 2 col.
9    Out[5]:
10   array([[0.26249554, 0.21950032],
11          [0.81072637, 0.01962725]])
12   In [6]: np.random.randn()              # Generate samples from standard normal
13   Out[6]: 1.8382156073224407
14   In [7]: 3 + 2*np.random.randn()        # Gen. normal distr. sample with mean 3 and std 2
15   Out[7]: 4.067920463989419
```

See np.random for more options!


Questions?

# Today’s summary

We learned about:

- built-in objects: `dict`, `set`, `tuple`,
- routing mechanisms: `if`, `while`, `for`
- introduction to NumPy (design, usablity and special functions)

Logistics:

- create a personal GitHub account.
- find two teammates and sign up here. You will need your GitHub names. Only fill out the green section, pick any team number. If you do not know any teammates (yet), you can just put your name in the form and state that you are looking for teammates.
- Please sign up by the end of next week (we will also announce this via Moodle).
- You need to use your EPFL login for access to this data sheet!

# After lunch:

- Monday 13 - 15: exercises (6 groups)
  - room CO4 ← A-B (according to your surname)
  - room CO5 ← C-F
  - room CO260 ← G-L
  - room CO6 ← M-R
  - room CO023 ← S-Z
- Monday 16:15 - 17: my office hours at SV 2811 (email: alexander.mathis@epfl.ch)