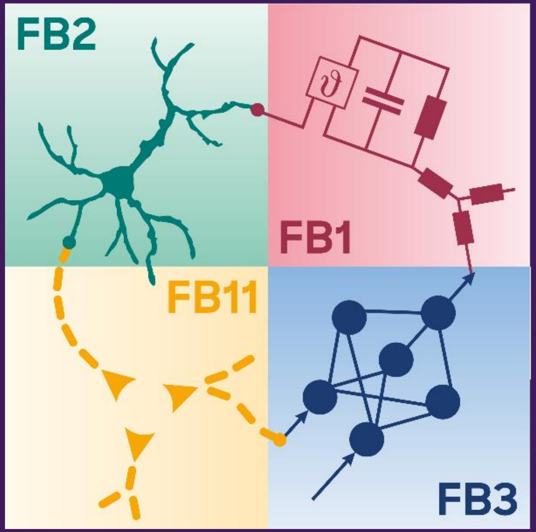


MN-F3
Programming
WS2024/25

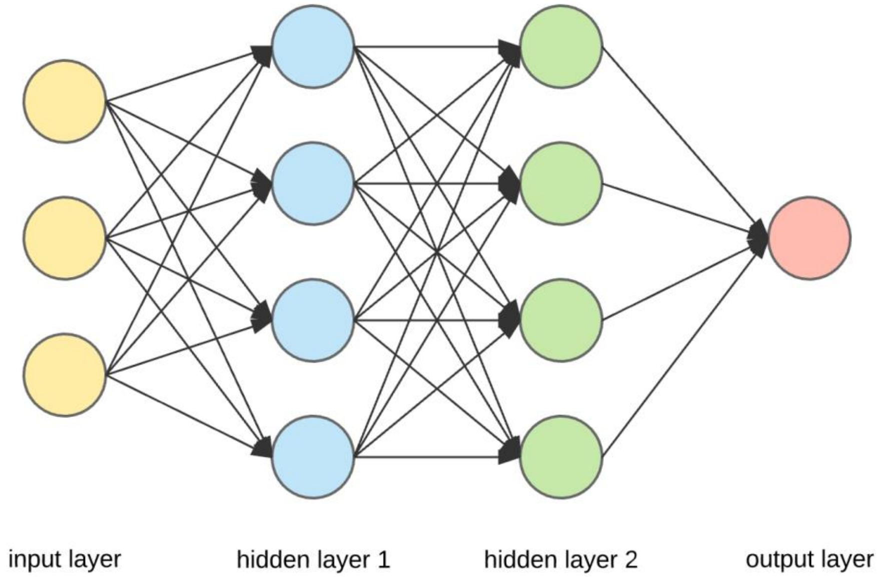
Contents

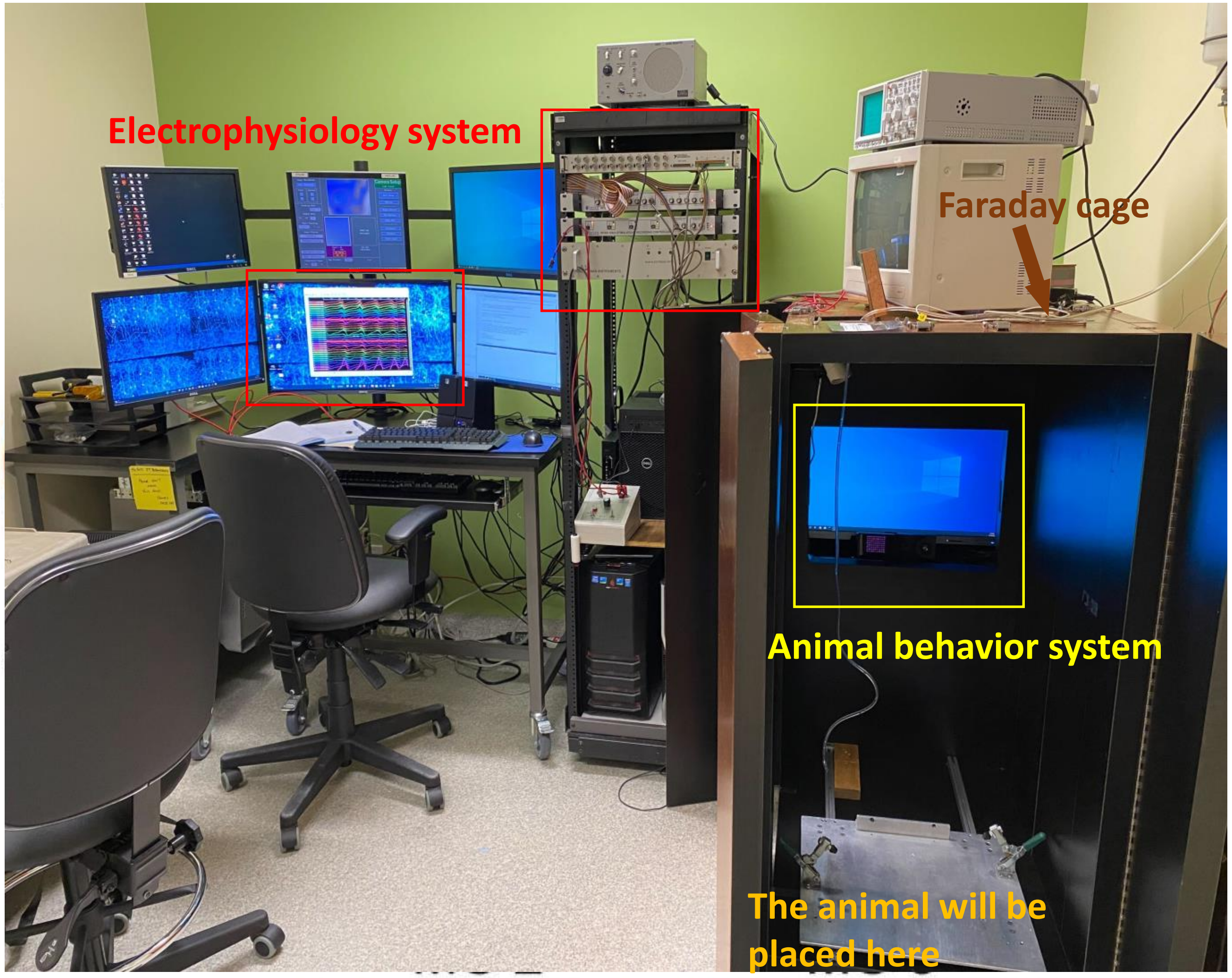
Motivation

Why programming?
Why Python?

Why to learn programming as a neuroscientist?

- Modelling and simulation
- Data analysis
- **Designing and setting up experimental hardware and paradigms**

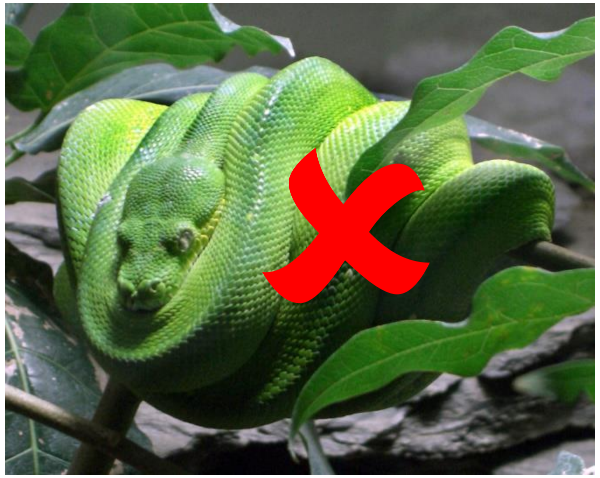


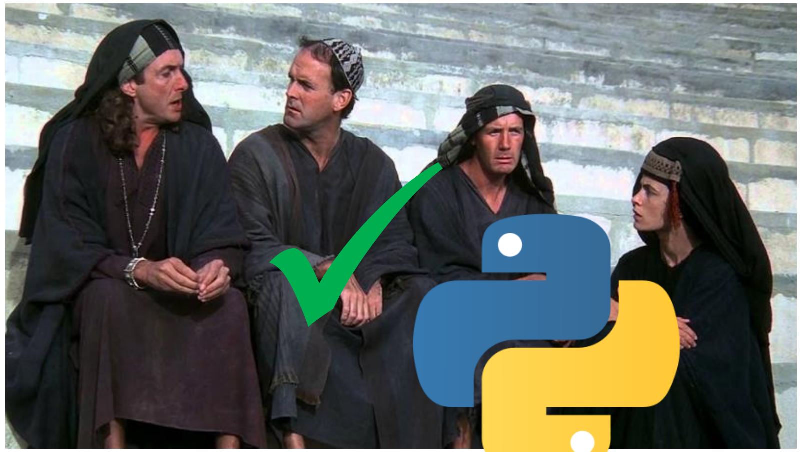


Why to learn Python?

- Freely available (no costs & open source)
- Large community (most popular, about 30% "market share")
- Lots of tools for neuroscientists available (numpy, scipy, psychopy, deeplabcut, etc. etc. etc.)
- Vectorized computations as in Matlab
- Links to different kinds of hardware easily
- Standard tool for machine learning (PyTorch)
- Easy to learn and fast to develop code

It's not named Python to motivate you to write snake-like code where a single computation wraps over multiple lines…

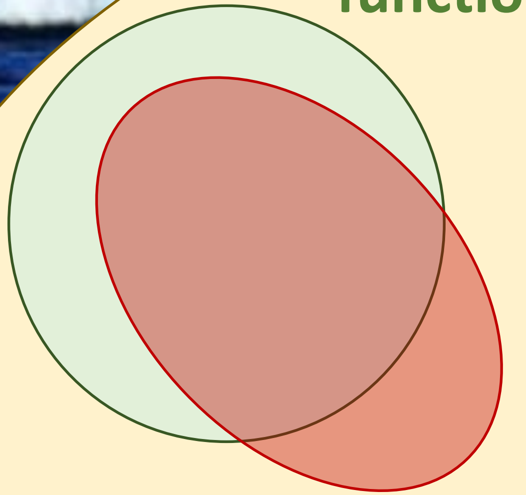




…it's named Python since it's fun to code with and because there are sometimes silly changes in new versions!

Scope of this course

Python core functionality



What we teach you:

concepts to get started in a programming language, methods useful in neuroscientific context (for experimental design, simulation and modelling, data analysis)…

Examples:

programming logics and flow control, working with large amounts of data (numpy), displaying and visualizing data (matplotlib)…

Python and all of its extensions (modules)

Organization of Lecture and Tutorials

- **Lecture:** introduce topics and concepts
- → …train to use these concepts in **Exercises** (on your own, with colleagues!)
- → …present solutions/approaches in **Tutorials** and show you've mastered it!
- **Material provided:** these slides, example code, exercise sheets, the Rotermund Python compendium!
- **Additional material:** Socratica YouTube channel, online ressources!

Some remarks…

Programming is like learning to **play a musical instrument**…

A programming language **has a grammar and a vocabulary**…

Initially: **separation of techniques and content**, link to neuroscience in 2nd block of lecture and subsequent Theoretical Neurosciences lecture.

QUESTIONS
and
OTHER REMARKS?

L1: Basics

What a computer does – entering & executing a program – variables and lists, indexing and slicing – simple input and output – fundamental computations – not getting lost.

What does a computer do (basically…)?

What a computer **is good at**:

- it performs mathematical operation very rapidly
- it shifts around large amounts of data very quickly

Typical operations:

- fetch/store data
- perform computation on data
- test condition on data & determine what to do next

keyboard, harddisk, camera, EEG, Utah array…

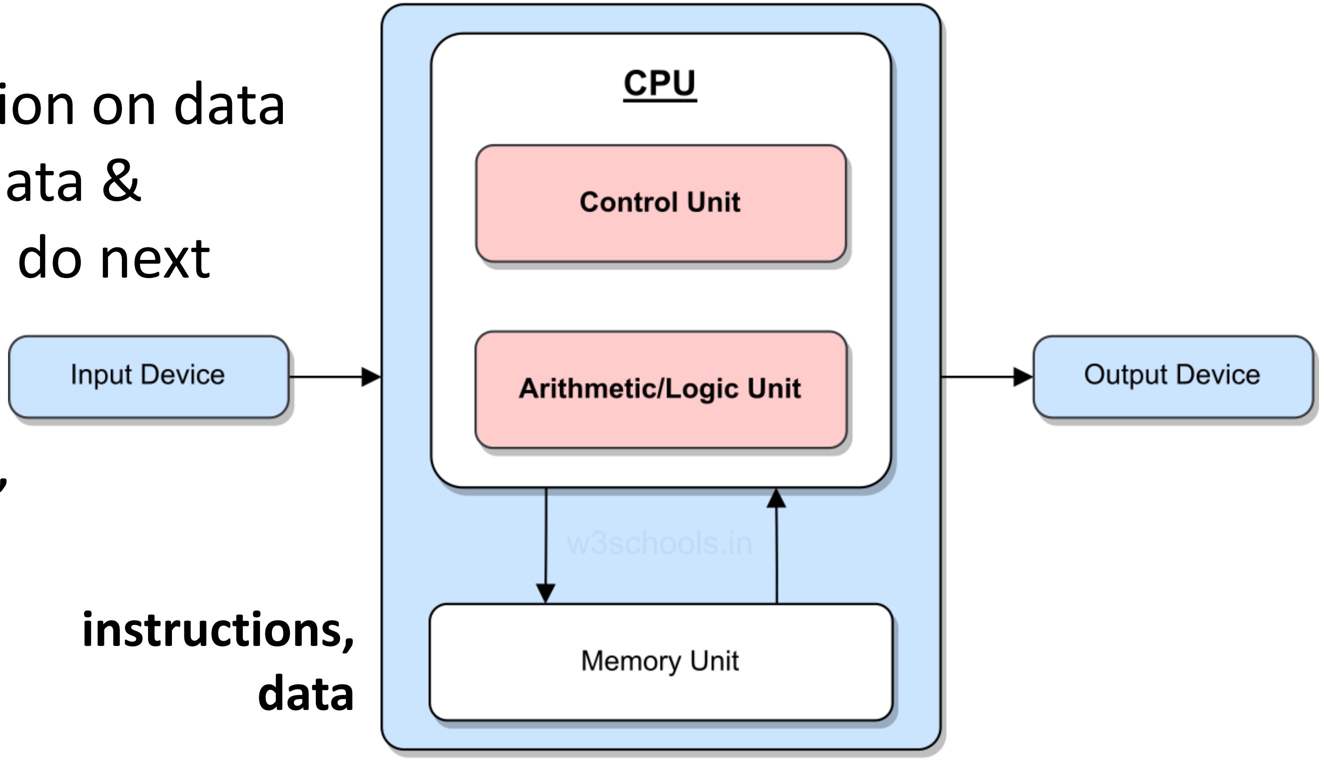


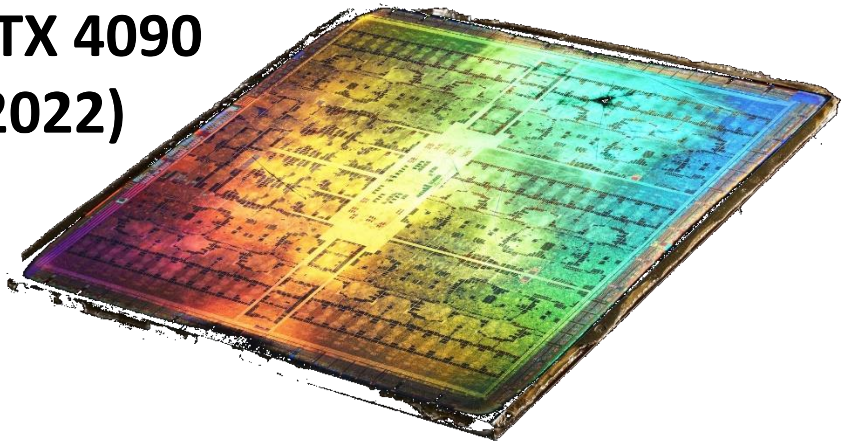
Fig: Von-Neumann Architecture

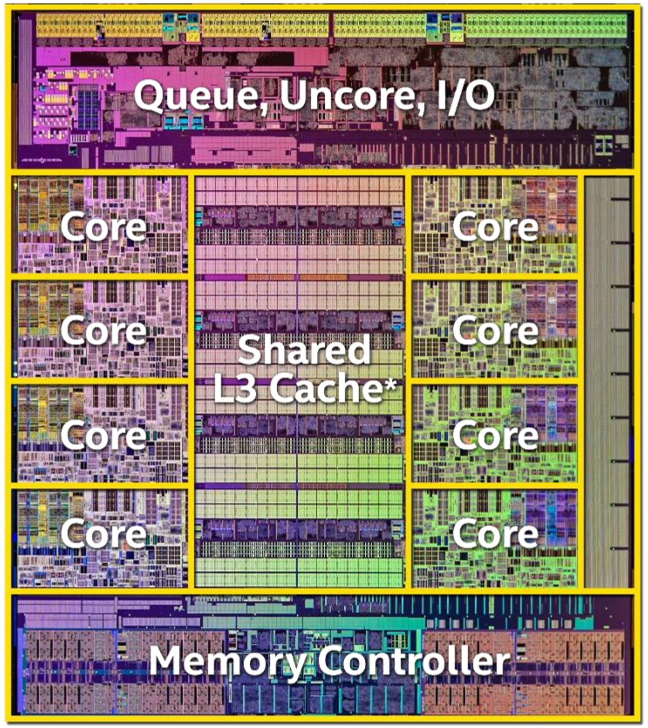
instructions, data

monitor, harddisk, robot arm, stimulation electrode

Zuse Z1 (1937, mechanical)

NVidia GPU
RTX 4090
(2022)





Intel Core
i7-10700
(2020)

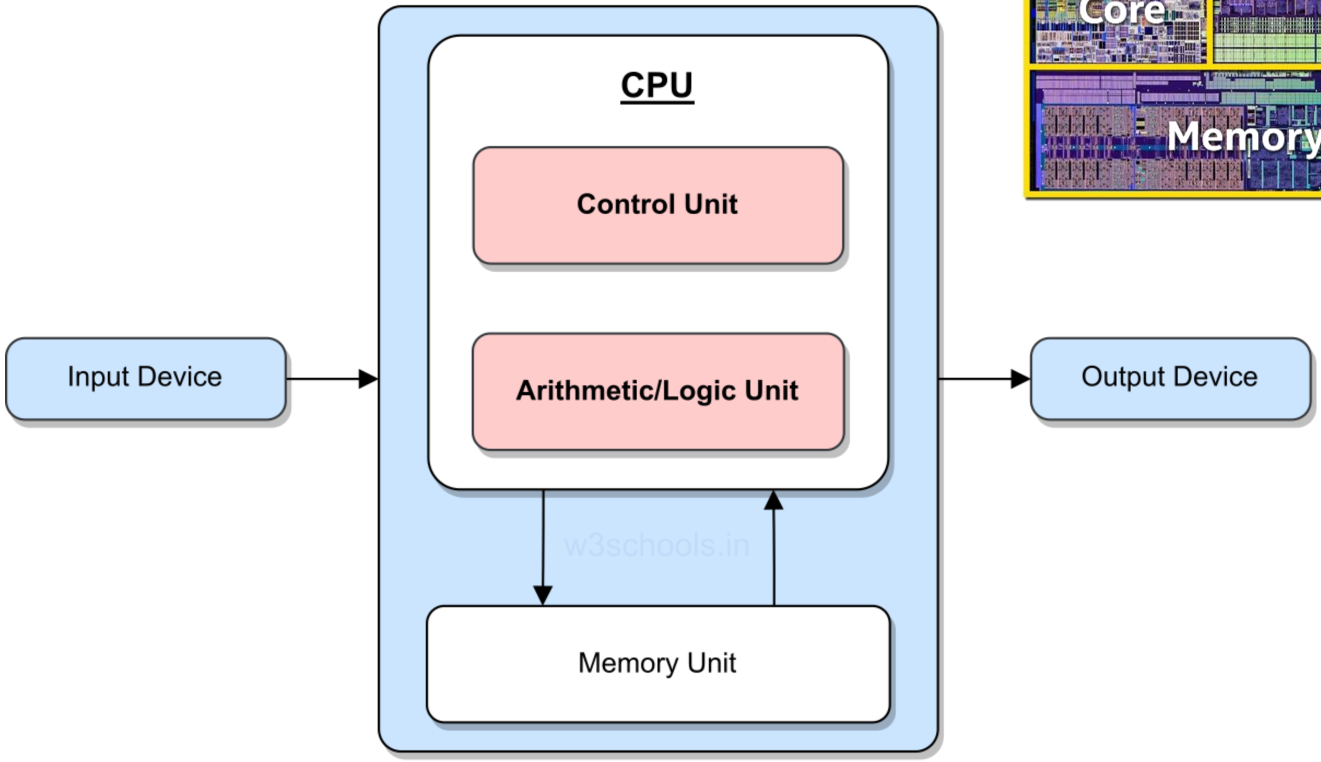


Fig: Von-Neumann Architecture

What is a program?

A computer program is **a sequence or set of instructions in a programming language for a computer to execute** (Wikipedia).

Examples: pocket calculator, sowing seeds…

Example →

Writing and executing code using VSCode and Python

VSCode: a powerful Python 'editor'
- code window (left)
- output window (bottom)
- edit/save/load (files should have .py extension)

Some nice VSCode Commands:
- TAB, SHIFT-TAB level of indent
- SHIFT+ENTER, CTRL+ENTER
- F2
- #%% [markdown]

Editor/Code Window:
- enter programme line by line, Python has no line numbers like 'Basic'
- # → Comment; \ → line extension
- indentation: spaces not tabs, autoformatting (e.g. Black) cares for it…
- formatting rules enhancing readability: PEP8: https://peps.python.org/pep-0008/

Interactive execution: needs 'ipykernel' package
- interactive window (right)
- allows to execute parts of code (#%% cells)

…in Tutorials! →

Assignments

Get data into your program:

- **Assignments** assign the output of an expression on the r.h.s. of an equation to a **variable name** on the l.h.s.
- Assignments are NOT a mathematical equation or equality!
- The r.h.s. of an assignment may only contain **variable names that have been defined before**

```
a_number = 0
print(type(a_number))
```

```
a_second_number = 3.33
print(type(a_second_number))
```

```
a_second_number = 3.33
a_second_number = int(a_second_number)
print(type(a_second_number))
```

Variables have types!

int: integer
float: floating-point number
bool: Boolean truth value
str: string, series of characters
... (there are more)

int(), float(), bool(), str() can also be used as keywords for type conversion!

Input and output

- Display messages and results with **print**
- Get data from the user with **input**
- If you're unsure how a Python function or keyword works, get help: **help(name)** for a function; or **help("name")** for a keyword

```
a = 42.42
print("Message")
print(a)
```

```
something = input()
```

← is a string, you have to type-convert if you need an int or float!

```
help(print)
```

```
help("for")
```

Example →

Formatting

f-strings: allow you to mix text with contents of variables/expressions

```
a = 42
print(f"Variable a={a}...")
print(f"Variable a={a} of type {type(a)}...")
print(f"Variable a={a} of type '{type(a)}'...")
```

...plus **formatting specs** (have a look at script for more...):

```
a = 42.42
print(f"a has the value {a:.03f}")
```

...you want to print a "{"? Just double it! There's also tab \t and newline \n .

...terms in {...} could also be expressions:

```
a = 42
s='fortytwo'
print(f"{a}={s}\treally {s}?\nreally {s}!")
```

https://docs.python.org/3/library/string.html#formatspec

Example →

Arithmetics and 'math' module

a) **Expressions (e.g. r.h.s. of assignment) can combine complex mathematical operations.** Some examples:

elementary maths: +, -, *, /

potentiation, modulo: **, %

b) Important **mathematical functions** (sin, cos, exp, …) are´defined in the module 'math':

```
import math
s = math.sin(42.42)
z = s*math.exp(2)**(-4)
```

c) Comparisons such as ==, >, <, >=, <=, != yield **logical values** True, False:

```
4 == 5  # False
4 == 4  # True
```

d) Bitwise **logical functions**: &, |, ^, ~ (and, or, xor, not)

Example →

Holding more in a variable than just one item…

…about lists and tuples and ranges!

Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

https://docs.python.org/3/library/stdtypes.html#lists

Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the enumerate() built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a set or dict instance).

https://docs.python.org/3/library/stdtypes.html#tuple

Ranges

The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Examples of lists

Simple list:

```
primes = [2, 3, 5, 7]
planets = [
        "Mercury",
        "Venus",
        "Earth",
        "Mars",
        "Jupiter",
        "Saturn",
        "Uranus",
        "Neptune",
]
```

List of lists:

```
hands = [
        ["J", "Q", "K"],
        ["2", "2", "2"],
        ["6", "A", "K"],  # (Comma after the last element is optional)
]
```

A list can contain a mix of different types of variables:

```
def my_function(a):
        return a

my_favourite_things = [32, "sleep", my_function]
```

Indexing and slicing

```
planets = ["Mercury", "Venus", "Earth", "Mars", \
        "Jupiter", "Saturn", "Uranus", "Neptune",]
```

```
print(planets[0])
print(planets[1])
print(planets[-2])
print(planets[-1])
```

```
Mercury
Venus
Uranus
Neptune
```

```
print(planets[0:3])
print(planets[:3])
print(planets[3:])
print(planets[1:-1])
print(planets[-3:])
```

```
['Mercury', 'Venus', 'Earth']
['Mercury', 'Venus', 'Earth']
['Mars', 'Jupiter', 'Saturn', 'Uranus',
'Neptune']
['Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus']
['Saturn', 'Uranus', 'Neptune']
```

Assignments

```
planets = ["Mercury", "Venus", "Earth", "Mars", \
        "Jupiter", "Saturn", "Uranus", "Neptune",]
```

```
planets[3] = "Malacandra"
print(planets)

planets[:3] = ["Mur", "Vee", "Ur"]
print(planets)

planets[:4] = ["Mercury", "Venus", \
        "Earth", "Mars",]
print(planets)
```

```
['Mercury', 'Venus', 'Earth',
'Malacandra', 'Jupiter', 'Saturn',
'Uranus', 'Neptune']
```

```
['Mur', 'Vee', 'Ur', 'Malacandra',
'Jupiter', 'Saturn', 'Uranus',
'Neptune']
```

```
['Mercury', 'Venus', 'Earth',
'Mars', 'Jupiter', 'Saturn',
'Uranus', 'Neptune']
```

Required: same number of elements in source (right-hand side) and target expression (left-hand side)!

Other functions: length, sort, remove, append, pop…

```
planets = ["Mercury", "Venus", "Earth", "Mars", \
        "Jupiter", "Saturn", "Uranus", "Neptune",]
```

```
print(planets.index("Earth"))
print(planets.index("Pluto"))
```

```
2
ValueError: 'Pluto' is not in list
```

```
print(len(planets))
print(sorted(planets))
```

```
8
['Earth', 'Jupiter', 'Mars', 'Mercury',
'Neptune', 'Saturn', 'Uranus', 'Venus']
```

```
planets.append("Pluto")
print(planets)
print(planets.pop())
print(planets)
planets.remove("Earth")
print(planets)
del planets[1:-1]
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars',
'Jupiter', 'Saturn', 'Uranus',
'Neptune', 'Pluto']
Pluto
['Mercury', 'Venus', 'Earth', 'Mars',
'Jupiter', 'Saturn', 'Uranus', 'Neptune']
['Mercury', 'Venus', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
['Mercury', 'Neptune']
```

Tuples

Tuples are almost exactly the same as lists. They differ in just two ways.

a) The syntax for creating them uses parentheses instead of square brackets

```
t = (1, 2, 3)
```

b) They cannot be modified (they are immutable).

```
t = (1, 2, 3)
t[0] = 100
```

→ `TypeError: 'tuple' object does not support item assignment`

Tuples are often used for functions that have multiple return values.

Summary / Further Reading

Concepts:

- form of a programme
- variables and assignments
- mathematical operations and expressions
- input and output
- lists and tuples

Socratica Channel:
https://www.youtube.com/watch?v=bY6m6_IIN94&list=PLi01XoE8jYohWFPpC17Z-wWhPOSuh8Er-

The Rotermund Python Compendium:

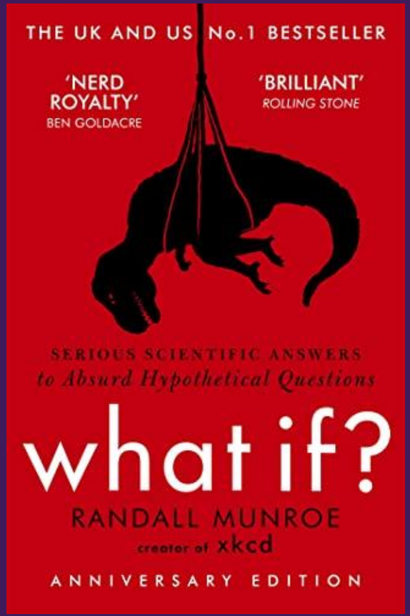
Uploaded on StudIP!

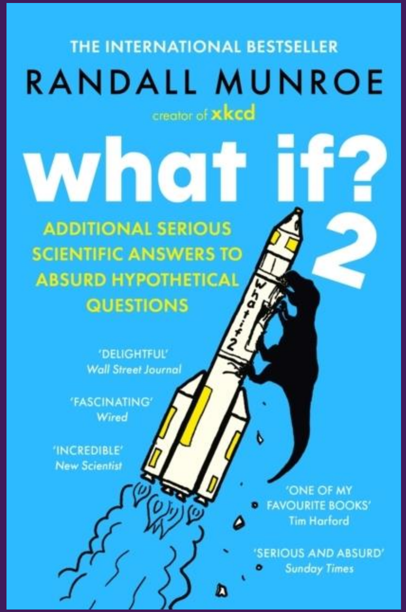
L2: Flow control

for, while – if, elif, else, match case – break, continue, pass

```
# Bremen Freimarkt simulation
for [ever]:
    sim.eat(food='chips', quantity=1e7)
    sim.enter(type='rollercoaster')
    sim.puke()
```

…if not knowing how if,
why not reading what if?
https://xkcd.com





Flow control

a) Iteration statements (Loops):

`for` loop

`while` loop

b) Selection statements:

if, **elif**, **else**

match case (>= Python 3.10)

c) Jump statements:

break

continue

pass

d) Functions (later lecture):

def

return

lambda

Loops allow to execute code multiple times with different parameters

```
value = 0
print(f"Squaring {value} gives us {value**2}")
value = 1
print(f"Squaring {value} gives us {value**2}")
value = 2
print(f"Squaring {value} gives us {value**2}")
value = 3
print(f"Squaring {value} gives us {value**2}")
value = 4
print(f"Squaaeraring {value} gives us {value**2}")
```

```
Squaring 0 gives us 0
Squaring 1 gives us 1
Squaring 2 gives us 4
Squaring 3 gives us 9
Squaaeraring 4 gives us 16
```

Loops allow to execute code multiple times with different parameters

For loops have a **value(s)** that change with every iteration.

For loops have an **iterable** that provides these values.

```
for value in range(10):
    print(f"Squaring {value} gives us {value**2}")
```

```
Squaring 0 gives us 0
Squaring 1 gives us 1
Squaring 2 gives us 4
Squaring 3 gives us 9
Squaring 4 gives us 16
Squaring 5 gives us 25
Squaring 6 gives us 36
Squaring 7 gives us 49
Squaring 8 gives us 64
Squaring 9 gives us 81
```

Indentation defines scope,
i.e. what belongs into the loop.

…but what about this?

```
for item in [42, "teddybear", (42, 17,)]:
    print(f"The current item is a {item}!")
```

Loops

Logic blocks need to be indented. Preferable with 4 spaces!

a) For

`for` iterates through an iterable, such as a `range` or `list`…

```
for i in range(0, 3):
        print(i)
```

→

```
0
1
2
```

```
range(start, stop[, step])
```

```
for i in [0, "A", 7, "nom num"]:
        print(i)
```

→

```
0
A
7
nom num
```

…it also allows to specify code executed upon successful termination:

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

Very useful: enumerate!

You can use enumerate to iterate through a list and get in each iteration an item and the index where this item is in a list:

```python
some_list = ["duplo", "lego", "fischertechnik"]
for index, item in enumerate(some_list):
    print(f"List item number {index} is '{item}'.")
```

```
List item number 0 is 'duplo'.
List item number 1 is 'lego'.
List item number 2 is 'fischertechnik'.
```

Loops (cont'd)

```
while_stmt ::=  "while" assignment_expression ":" suite
                ["else" ":" suite]
```

b) While

```
i = 0
while i < 3:
      print(i)
      i += 1
```

→

```
0
1
2
```

same as...

```
for i in range(0, 3):
      print(i)
```

→

```
0
1
2
```

Why while? It's useful in situations when the items over which to iterate are not known, or the termination condition is not known when the loop starts.

```
my_number = 4
your_guess = -1
print("Guess which number I'm thinking of.")
while your_guess != my_number:
    your_guess = float(input("Take a guess: "))
print("Yep, that's it!")
```

Flow control: if, elif and else

if allows the execution of a set of commands if an expression evaluates to "True":

```
if i == 1:
        print("if")
elif i == 2:
        print("elif branch A")
elif i == 3:
        print("elif branch B")
else:
        print("else -- default")
```

elif and else are optional.

A common form of logical comparison can test whether an item occurs in a list:

```
A = 2
if A in [0, 2, 4, 6, 8]:
        print("found")
else:
        print("NOT found")
```

→ found

Flow control: match

For multiple case comparisons, the new match command is available from Python 3.10 on…

```
for i in range(0, 4):
        match (i):
                case 0:
                    print("This is a 0")
                case 0:
                    print("This is a 0 too.")
                case 1:
                    print("This is a 1.")
                case 2:
                print("This is a 2.")
            case _:
                print(f"I don't know what to do with a {i}!")
```

```
This is a 0
This is a 1.
This is a 2.
I don't know what to do with a 3!
```

Flow control: pass, break, and continue

a) pass

pass is a null operation.

Nothing happens when executed:

```
if A == B:
    pass
```

```
def A(x):
    pass
```

```
for A in [1, 2]:
    pass
```

Since Python uses indents as definition for a functional block it needs pass for signaling an empty functional block...

b) break

break terminates nearest enclosing loop, skipping the optional else clause. The loop control target keeps its current value.

```
for i in range(0, 5):
    if i == 2:
        break
print(i)
```

→ 2

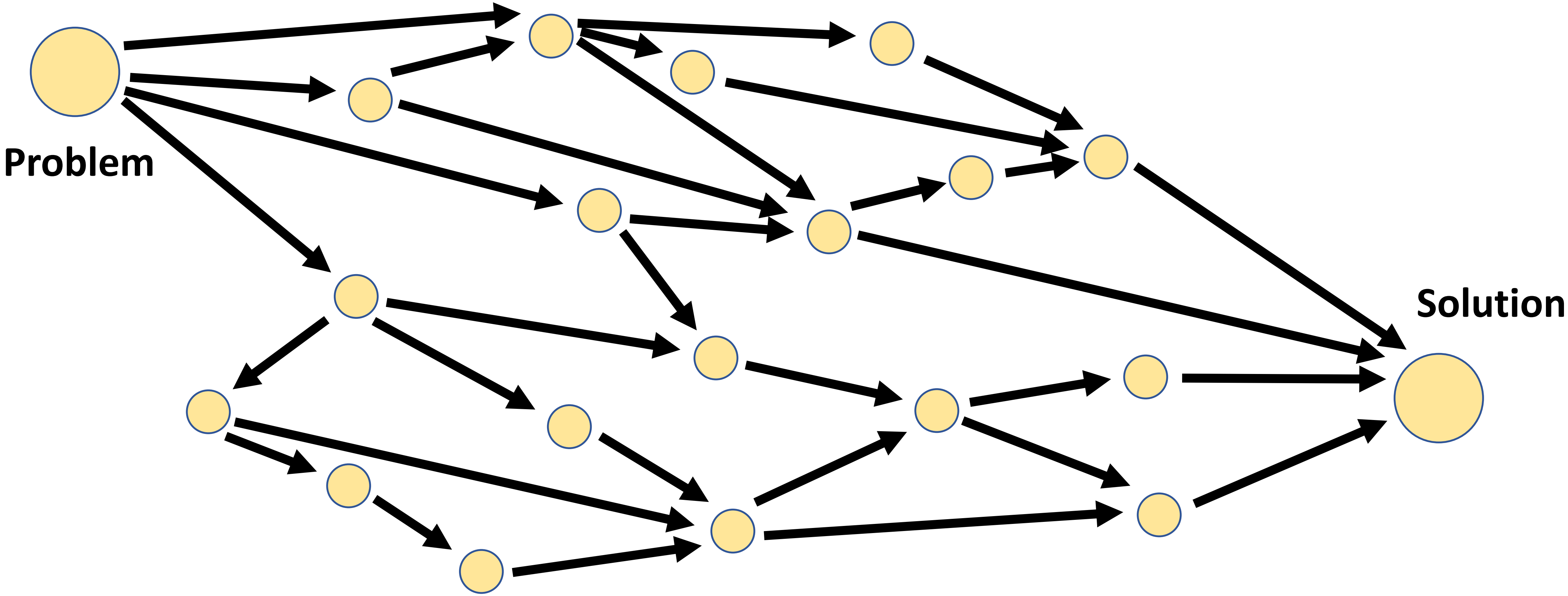
c) continue

continue is used in for or while, for continuing with the next cycle of the nearest enclosing loop.

```
for i in range(0, 5):
    if i == 2:
        continue
    print(i)
```

→
0
1
3
4

Zu guter Letzt: there's exactly one way in Python to solve a problem!



L3-A: Functions and Modules

def, import

Code NOT using functions and modules....

...code using functions and modules!

Functions

Functions serve to "encapsulate" non-trivial parts of your code and make them accessible under a new (function) name:

```
def my_function():
    pass
```

empty function...

```
def my_function():
    return 2
```

function returning an integer...

Functions can have input argument(s) and output value(s), but they don't have to. Some examples you know:

```
print("chnoergl…")          # input only
inp = input()               # output only
exit()                      # neither in- nor output
s = np.sin(42)              # input and output
```

```
def my_function():
    return
```

==

```
def my_function():
    return None
```

Functions: Input arguments and return values

a) Input arguments:

Can be given as positional arguments or keywords with a value:

```
def my_function(a, b):
        return a * b

print(my_function(a=2, b=3))
print(my_function(b=3, a=2))
print(my_function(2, b=3))
print(my_function(a=2, 3))
```

print(my_function(a=2, b=3)) → ...okay!

print(my_function(b=3, a=2)) → ...okay!

print(my_function(2, b=3)) → ...okay!

b) Return values:

Multiple values can be specified; they will be returned as a tuple:

```
def my_function():
        return 2, "A", 79, 3.1314

print(my_function())
```

```
(2, 'A', 79, 3.1314)
```

Formal definition:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

Functions: Default values and documentation strings

c) Default values:

Can be specified by assigning a value in the definition of a function:

```
def my_function(a=2, b=3):
      return a * b

print(my_function())
print(my_function(a=4))
print(my_function(b=5))
print(my_function(b=6, a=7))
```

```
6
12
10
42
```

d) Documentation strings

Very useful to provide help to other users (or to you if you forgot how to use)

```
def my_function():
      """This is a universal
      function. It does nothing."""
      pass

help(my_function)
```

```
Help on function my_function in module __main__:

my_function()
    This is a universal function.
    It does nothing.
```

Modules: Basics

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable __name__.

```
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

file fibo.py…

contains two functions named fib and fib2…

Modules: Import

`import fibo` does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there.

Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Different ways of importing (parts of) modules:

```
import math                    # Usage of sin:  result = math.sin(42)
from math import sin           # Usage of sin:  result = sin(42)
import math as m               # Usage of sin:  result = m.sin(42)
from math import sin as s      # Usage of sin:  result = s(42)
from math import *             # Usage of sin:  result = sin(42)
```

Pleaaaase, don't do that!!! Keep names separate!

Modules executed as scripts

When you run a Python module with `python fibo.py <arguments>`
the code in the module will be executed, just as if you imported it, but with __name__ set to "__main__". That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
        import sys
        fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

Modules in subdirectories

Modules can be placed into (sub)directories and imported by preceding the module's name by the name of a (sub)directory and a dot:

Example directory structure:

```
maindir/
  +--- myprog.py
  +--- subdir/
           +--- fibo.py
           +--- subsubdir/
                    +--- fibo.py
```

Usage in myprog.py:

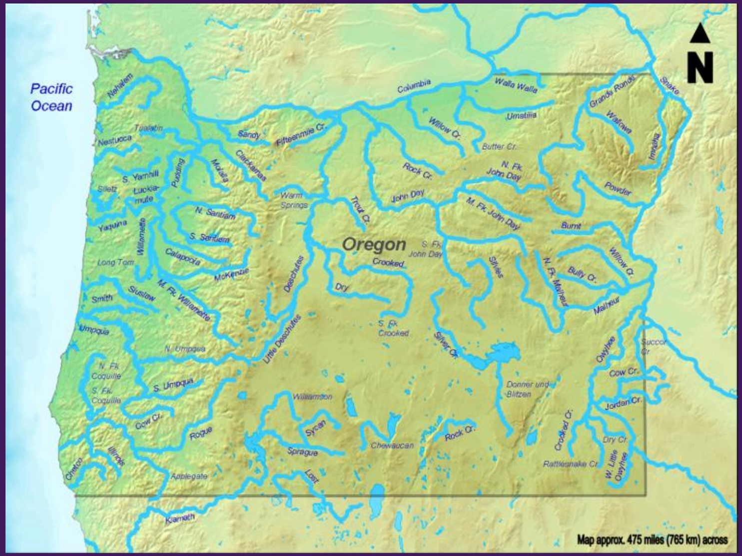
```python
import subdir.fibo as phebo
import subdir.subsubdir.fibo as feepo

phebo.fib(15)
print(phebo.fib2(15))
feepo.fib(42)
print(feepo.fib2(42))
```

```
1 1 2 3 5 8 13
[0, 1, 1, 2, 3, 5, 8, 13]
1 1 2 3 5 8 13 21 34
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Note: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use importlib.reload(), e.g. import importlib; importlib.reload(modulename).

L3-B: Systematic Programming and Good Programming Practice

Flow Charts – Do's and Don't's



Systematic programming

1. **Problem definition**
2. **Algorithmic solution**
3. **Division into simple(r) steps**
4. **Creating a flow chart (optional, might be useful!)**
5. **Translation of steps into commands / control instructions**
6. **Testing and debugging /refinement**

knowledge about what a computer can in principle do…

knowledge about a particular programming language, its syntax, capabilities and extensions…

Problem definition and algorithmic solution

- Precise formulation of a problem in exact language and/or mathematics…
- …what goes in out, what comes out? (variables, parameters)
- What has to be done to the input to get to the desired output? (the algorithm)

In our tutorials, these remarks give contextual information to connect programming to neuroscience. Note that understanding this information is actually not required to solve the problem!

Example:

Simulate a so-called 'random walk' of a particle. A random walk describes a movement of a particle (e.g. molecule, protein) that is determined by a random process. The particle moves in every time step by a distance of dx, either to the left or to the right. Both possibilities have the same probability of happening. The task for the computer is to simulate n = 1000 trials with the particle starting at location x=0, and stop the simulation each time the particle crosses a barrier at x = −1 or x = 1. The output of our program shall be the mean number of steps needed to reach one barrier, calculated from the n iterations.

Division into simpler steps (can be done together with flow chart)

- Which steps have to be executed first, which later? (sequencing)
- Which steps have to be iterated several times resp. applied in a similar manner on different quantities?
- Which steps have to be carried out dependent on a specific condition?
- Which steps demand user interaction or an informative feedback?
- Can a complex problem be partitioned into simpler blocks? (functions)
- Make sure that the solution can be found in finite steps (if possible…)

Creating a flow chart (get your thoughts organized!)

- Use different symbols for normal steps, repeated execution, conditional execution etc. and connect them by arrows indicating sequence
- Down: normal flow, Up: repeated execution, Horizontal: different levels of functions/subfunctions

Translation of steps into commands / control instructions

- Determine suitable data structures for holding your data, parameters, and temporary results (scalar or vector/array? text or number, integer or float? homogeneous data or dictionary combining diverse data types? …)
- Replace each step by one or few instructions
- Further subdivide unexpectedly complex steps
- Adhere to the syntax / grammar, use help, look at examples…
- Use functions to modularize your code, making it leaner and making its logical structure transparent

(*) except in cases we want you to do your own wheel because we think you might learn something really important!

Systematic programming

Some rules which we think are very important (in our field):

Do not reinvent the wheel (if a good wheel exists, please re-use it (*), but please please do also understand how it works and how you mount it!)

Document your code (your own knowledge of what you cooked up will decay exponentially at a fast rate)

Expect your code to be used by the DAU (use assert often and early, think actively about error conditions that might occur, avoid giving your code to other users –they will only fuck it up…)

Use new variable names for everything that you define or compute on at least two places (want to change value? There's only one place, not to overlook!)

For each (collection of) function(s) supply ample test code and prototypical examples on how to use it

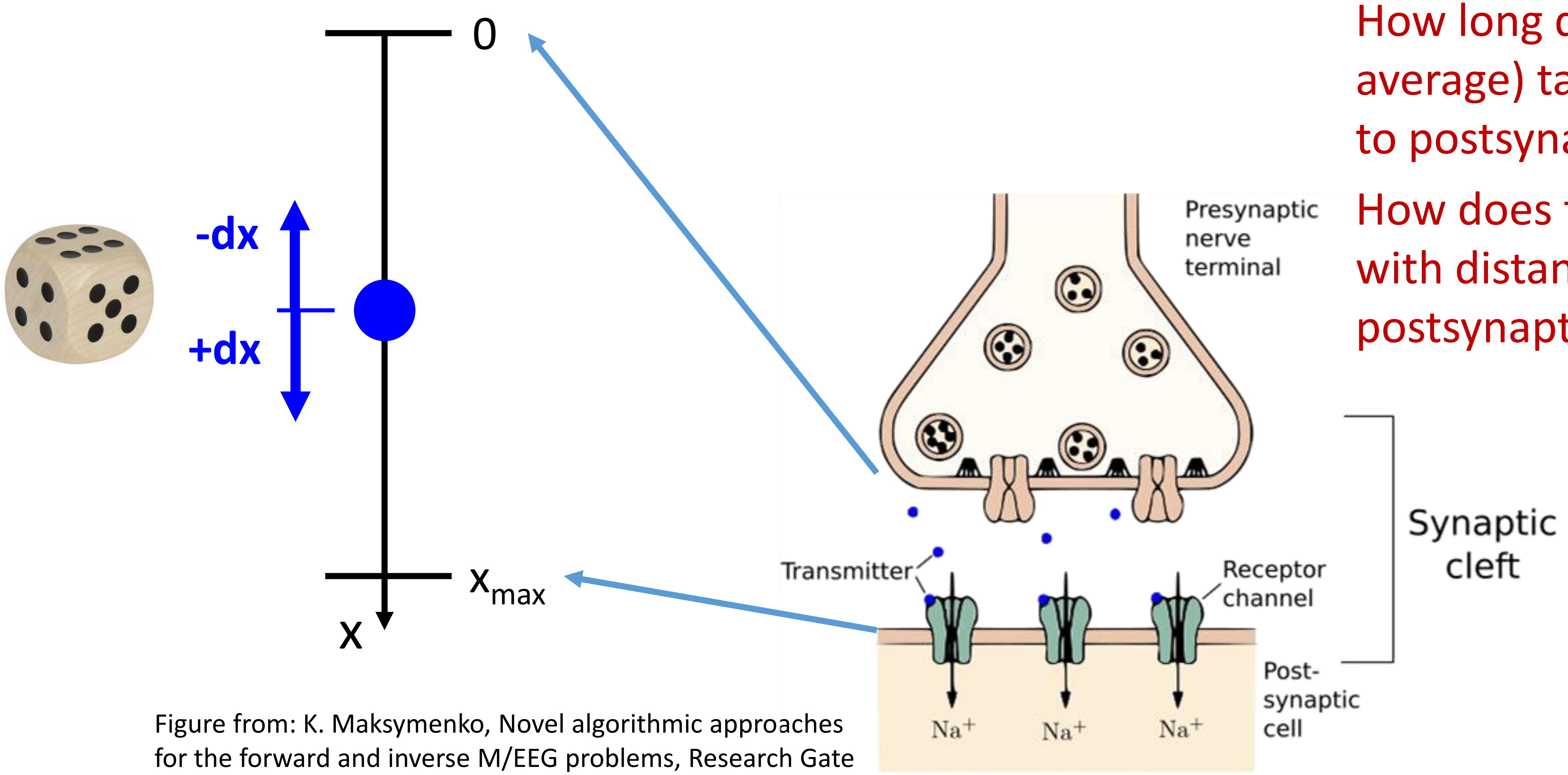
Avoid lengthy expressions, break down into simpler chunks and use temp vars

Keep your resources in mind ("640k of memory should be enough for everybody")

In larger projects, use proper version control (e.g. Github, ask Joscha!)

Example: Systematic Programming and Functions

Simulation of diffusion across synaptic cleft



How long does it (on average) take to diffuse to postsynaptic cell?

How does this time scale with distance to postsynaptic cell?

Figure from: K. Maksymenko, Novel algorithmic approaches for the forward and inverse M/EEG problems, Research Gate

End of first Block

Die Hälfte ist geschafft – jetzt braucht's Übung!

L4: Numpy & Matplotlib

Vectors, Matrices and Arrays – Axes and Functions – Plotting and Labeling

Njam-Pie: delicious, but not capable of handling large data!

Numpy and Matplotlib

NumPy
Base N-dimensional array package

Matplotlib
Comprehensive 2-D plotting

- inspired by Matlab (more consistent implementation in torch)
- process large data sets with few instructions
- avoid going through single elements "by hand"
- display results nicely

Making it available to your code:

import matplotlib.pyplot as plt DOCS: https://matplotlib.org/stable/api/index.html

import numpy as np DOCS: https://numpy.org/doc/stable/reference/index.html

How to represent data?

Basic data container: type ndarray **(more precisely,** numpy.ndarray**)**

(0D), 1D, 2D, 3D, …, nD

a) Define array directly…

…and fill with a specific value: zeros, ones, empty…

```python
import numpy as np

M = np.zeros((2, 3, 4))
print(M)

N = np.zeros_like(M)
print(N)
```

→

```
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
```

→

```
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
```

b) Create array from a range

arange: return evenly spaced values within a given interval:

```
numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None)
```

WARNING!!!

When using a non-integer step, such as 0.1, it is often better to use numpy.linspace.

linspace: return evenly spaced numbers over a specified interval:

```
numpy.linspace(start, stop, num=50, endpoint=True,
retstep=False, dtype=None, axis=0)
```

Returns num evenly spaced samples, calculated over the interval [start, stop].

The endpoint of the interval can optionally be excluded.

c) Define array by hand, or convert from list / tuple

Nesting the values into square brackets (i.e. conversion from lists of lists of lists)

```
a_1d = np.array([4, 5, 6])
a_2d = np.array([[4, 5], [6, 7]])
a_3d = np.array([[[4, 5], [6, 7]], [[8, 9], [10, 11]]])
```

Other properties good to know…

Query number of values in array 'data': `data.size`

Query size of dimensions in array 'data': `data.shape`

Query (and also specify, as optional argument on creation) data type: `data.dtype`

How do we 'print' what's inside the data?

…that's what we need matplotlib.pyplot for, e.g.:

```
plt.plot(x, y)
plt.show()
```

Indexing and slicing on numpy ndarrays

- A valid (single) index starts at 0 and runs until N-1 (we assume N as the size of the dimension.
- [start:stop:step]
  start = 1, stop=N, step=1
  results in the sequence
  1,2,3,...,(N-1)
- [start:stop:1] can be shortened to [start:stop]
- [0:stop] can be shortened to [:stop]
- [start:N] can be shortened to [start:]
- B = A[:] gives you a view of A.
  B has the same shape and size of A.
- Indexing can also be used on the **left-hand-side** of an assignment for filling up part of an ndarray!

Examples for:

```
A = numpy.arange(0,10)
```

```
B = A[1:10:1]   →   [1 2 3 4 5 6 7 8 9]
B = A[3:7:2]    →   [3 5]
B = A[3:6]      →   [3 4 5]
B = A[:6]       →   [0 1 2 3 4 5]
B = A[5:]       →   [5 6 7 8 9]
```

Examples #1 →

Computing with ndarrays

Mathematical operations either defined as **methods**…: s = a.sum()
…or defined as **functions**: s = np.sum(a)

Examples: sum, std, mean, var, …

Functions defined in math are usually also defined in numpy, and they perform the corresponding operation **element-wise**!

Examples: sin, cos, exp, log, …

Often **optional arguments** can change the behaviour of numpy-functions/meths: s_first = np.sum(a, axis=0)

Examples: axis, dtype, keepdims, …

Many fcts for linear algebra such as matrix multiplication: c = np.matmul(a, b)

Examples: matmul, dot, .T (transpose!)

If you want to perform numerical stuff which usually requires the math-module and/or performs operation on non-scalar data, please use numpy from scratch!

Basic functions in matplotlib.pyplot:

```
plt.plot(x, y)            # plot y-vector against x-vector
plt.show()                # terminate drawing and show the graph(s)
plt.figure(nr)            # open figure number nr
plt.title(a_title)        # give the graph a title
plt.xlim([a, b])          # set limits of horizontal axis
plt.legend(['abra', 'kadabra'])      # legends for multiple funcs in one graph
plt.ylabel(a_label)       # give the vertical axis a label
plt.grid()                # add a grid to the graph
plt.text(x, y, a_text)    # print a text a_text to coordinates x, y
```

Warning: all these examples can take different optional/named arguments which control their behavior! Use help() to have a closer look…

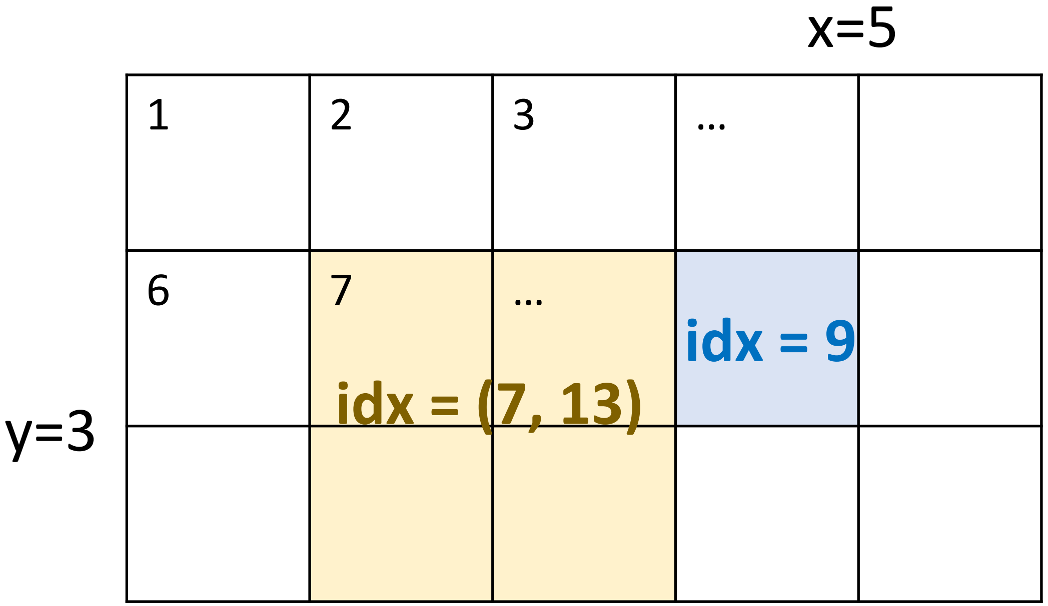
Some more plotting stuff, just for the show!

```
plt.imshow(array_2d)                       # show 2D-array or 3D RGB image
plt.colorbar()                             # show colorbar, e.g. for imshow
plt.savefig("schrotty.png", dpi=100)       # save figure as bitmap to "schrotty.png"
plt.savefig("pretty.pdf")                  # save figure as lineart to "pretty.pdf"
```

WARNING: Do a savefig before you do plt.show() !!!

plt.subplot(y, x, idx)

establishes a grid for having y by x subplots and addresses subplotindexed by idx with subsequent plot commands…



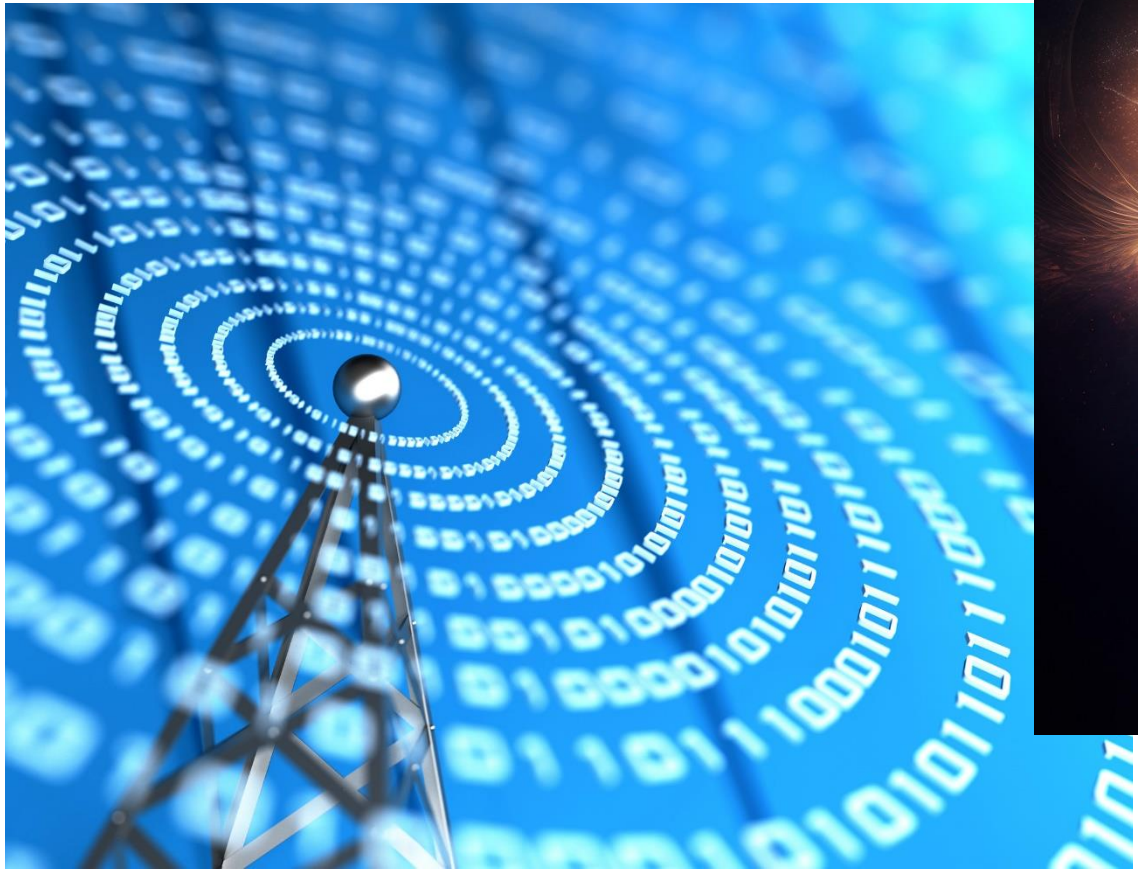
Examples #2 →

L5: More Numpy and Files

Broadcasting, Slicing – Save, Load, Paths

A 4 by 5 Njam-Pie!

Broadcasting and Working with Multidimensional Entities

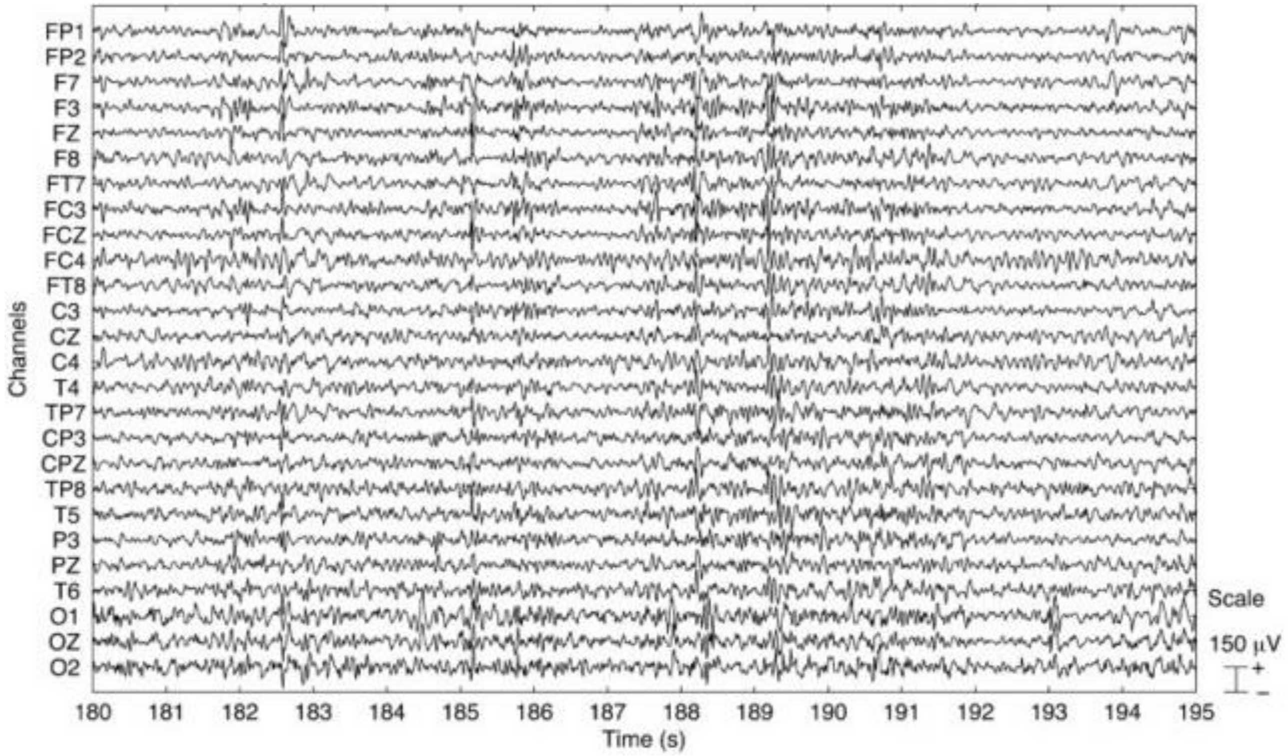


Source:
[r/aiArt]

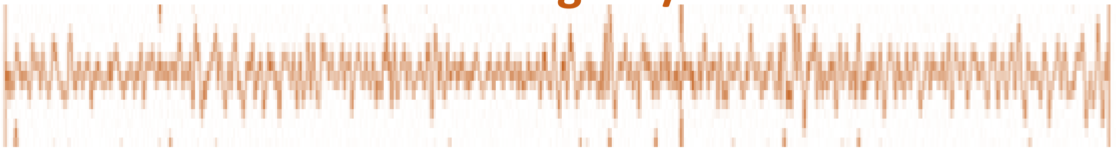
https://www.itu.int/

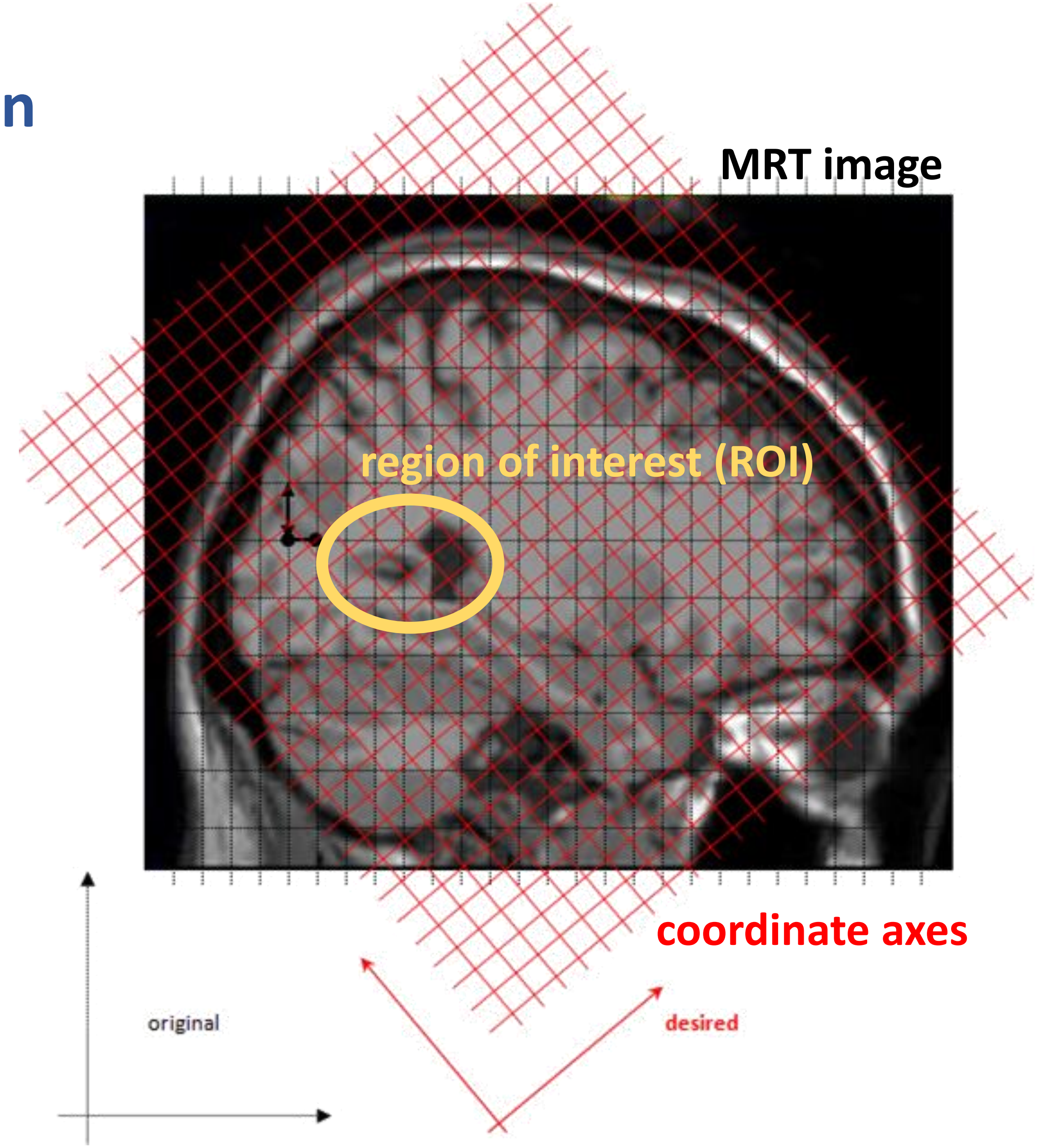
Broadcasting and Slicing: Motivation

raw EEG recording



common source signal / external noise

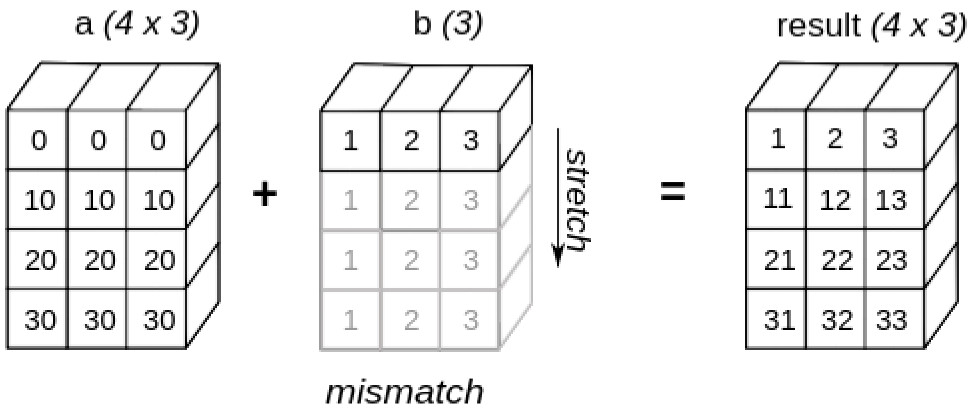




→ Whiteboard

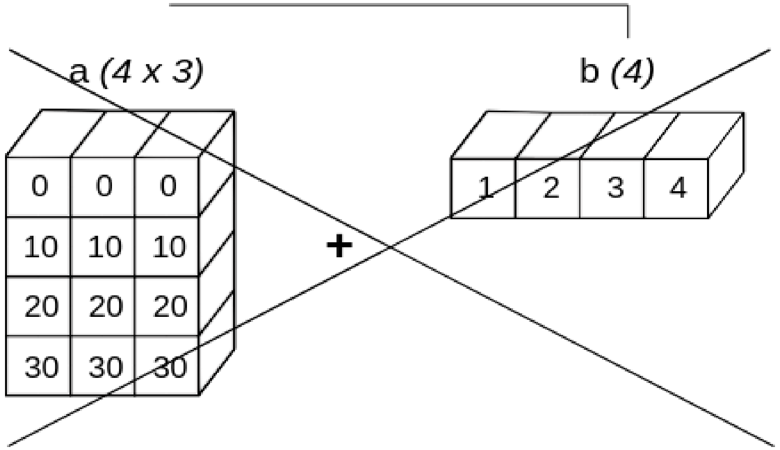
Broadcasting: Basics

Broadcasting extends arrays for mathematical and indexing operations by **replicating their contents across 'missing' dimensions (or dims with size 1)**:



Matching trailing dimensions:

array b gets 'broadcasted' along 'missing' dimension

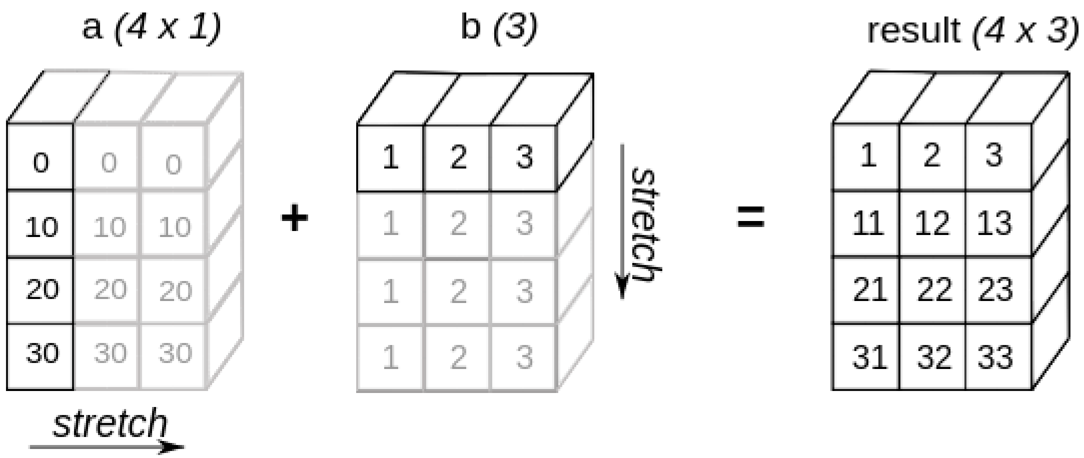


Non-matching trailing dimensions:

Broadcasting not possible

Broadcasting: Multiple extensions

Extensions along different dimensions in both operands possible:



Is broadcasting possible?

a) write down the shapes of the two operands (arrays) below each other, flush right

b) start from the right, and check the numbers below each other:
   - if one number is one (or none), this dim gets broadcasted
     to the max of the two numbers
   - if both numbers are above 1, they have to be equal (otherwise: error)

Broadcasting: Examples

The rule: start from the right, and check the numbers below each other:

- if one number is one (or none), this dim gets broadcasted to the max of the two numbers
- if both numbers are above 1, they have to be equal (otherwise: error)

Good:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
```

```
A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
```

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
```

```
A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
```

```
A      (2d array):  5 x 4
B      (1d array):      1
```

```
A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
```

Broken:

```
A      (1d array):  3
B      (1d array):  4
```

```
A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3
```

Reshape and Flatten

reshape allows **changing dimensions without changing the contents**.

For example, useful when reading a multidimensional array from a linear stream such as a file or external device…

```python
import numpy
A = numpy.arange(0,15)
B2D = numpy.reshape(A, (5,3))
print(B2D)
print("View: " + str(numpy.may_share_memory(A,B2D)))
```

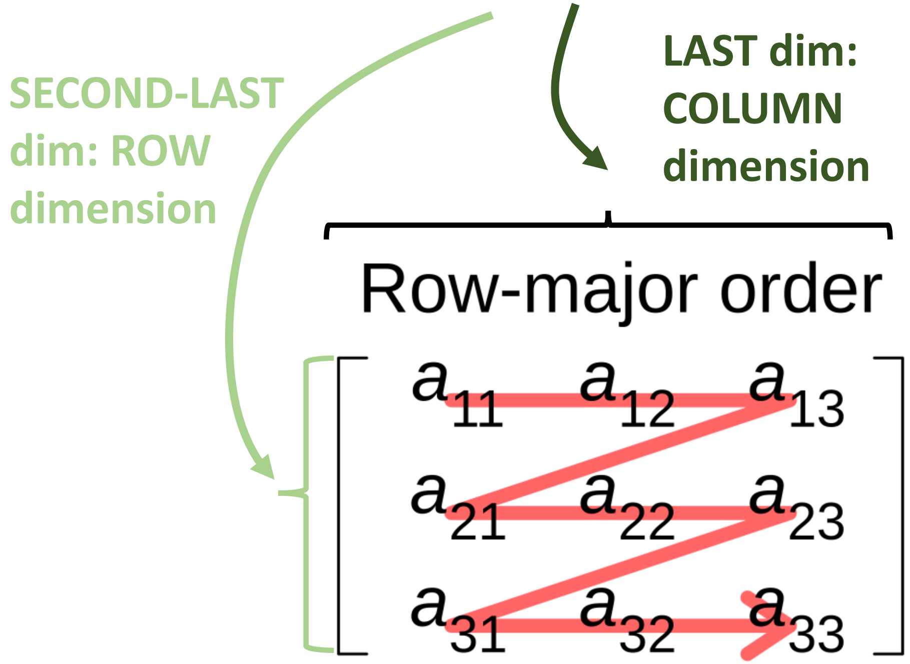
```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
View: True
```

flatten makes a one-dimensional vector!

```python
a_again = b2d.flatten()
print(a_again)
```

```
✓ 0.0s

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

```python
a = np.zeros((3, 3))
```



Slicing in N dimensions

N-dim is like 1-dim-slicing, but applied to several dimensions in parallel...

```
A = numpy.arange(0,30)
B_3D = numpy.reshape(A, (5,3,2))
```

```
C = B_3D[2:,:2,0]
```

```
[[12 14]
 [18 20]
 [24 26]]
View: True
```

```
C = B_3D[2:0:-1,2,:]
```

```
[[16 17]
 [10 11]]
View: True
```

If one or more dimensions are not specified, all of their elements are addressed:

```
C = B_3D[1:2]
```

```
[[[ 6  7]
  [ 8  9]
  [10 11]]]
```

better, recommended since it reminds us that there's more than one dim: `C = B_3D[1:2, ...]`

C.shape is 1, 3, 2. Do you want to get rid of dims that are 1? Try this: `C = B_3D[1]`

Adding and removing array dimensions/axes

numpy.newaxis inserts new dimensions, and numpy.squeeze removes all axes with size 1. numpy.newaxis can be replaced by numpy.reshape.

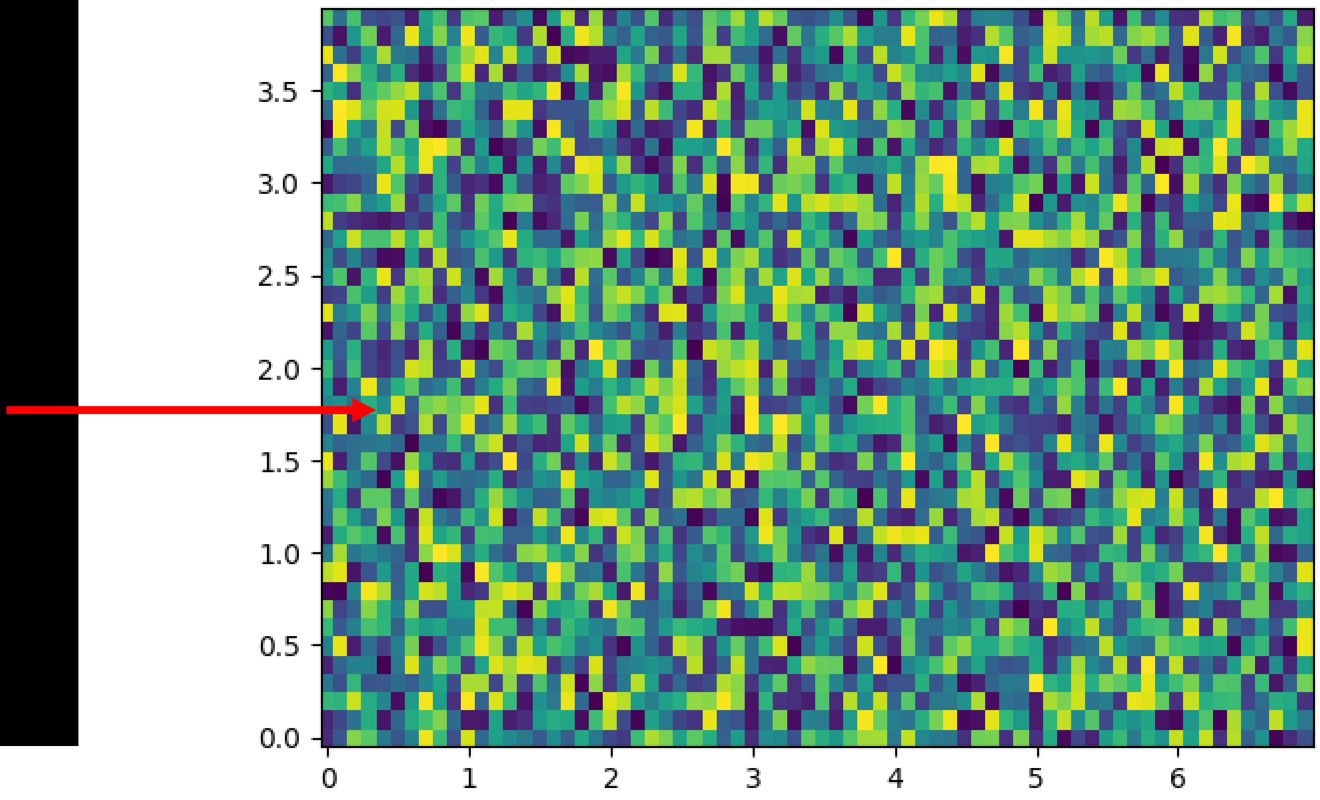
```
import numpy as np

a3 = np.ones((5, 4, 3))
print(a3.shape)

a5r = np.reshape(a3, (1, 5, 1, 4, 3))
print(a5r.shape)

a5n = a3[np.newaxis, :, np.newaxis, ...]
print(a5n.shape)

a3s = a5n.squeeze()
print(a3s.shape)

a3alone = a3s[..., 2:3]
print(a3alone.shape)

a2 = a3alone.squeeze()
print(a2.shape)

a1 = a3[3:4, 1:3, 0:1].squeeze()
print(a1.shape)
```

(5, 4, 3)
(1, 5, 1, 4, 3)
(1, 5, 1, 4, 3)
(5, 4, 3)
(5, 4, 1)
(5, 4)
(2,)

Coordinates in multiple (array) dimensions

For displaying or working with n-dimensional arrays, it is convenient to define proper axes:

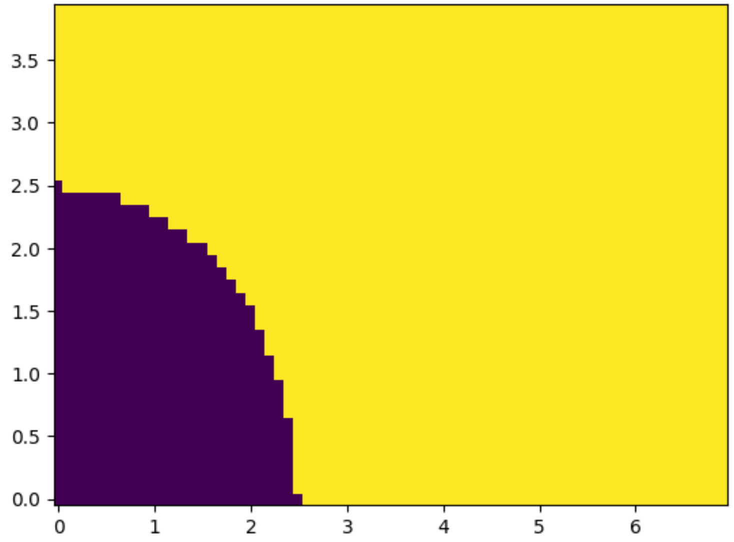
```
import numpy as np
import matplotlib.pyplot as plt

nx = 70
ny = 40

x = 0.1 * np.arange(nx)
y = 0.1 * np.arange(ny)

a = np.random.rand(ny, nx)

plt.pcolor(x, y, a)
plt.show()
```

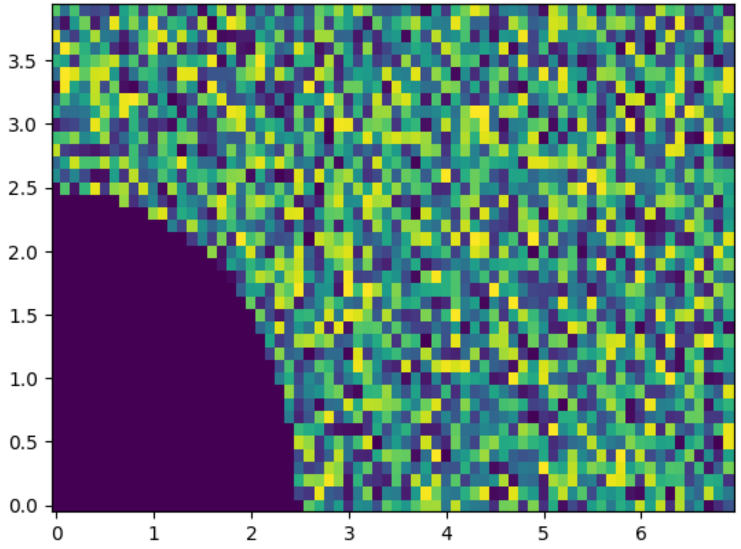


Coordinates in multiple (array) dimensions, continued!

If you put the coordinates into the 'right' dimension (i.e., ...z, y, x!), you can use broadcasting to do computations over n-dims with n vectors, avoiding to create full n-dim arrays with numpy.meshgrid:

```
y = y[:, np.newaxis]

r = np.sqrt(x**2 + y**2)
plt.pcolor(x, y, r > 2.5)
plt.show()

plt.pcolor(x, y, a * (r > 2.5))
plt.show()
```





Views and Copies

What the f***?

```
import numpy as np

z = np.zeros((4, 4))

o = z[1:3, 1:3]
o[...] = 1

print(f"z={z}")

print(f"o and z may share memory: { \
    np.may_share_memory(o, z)}")
```

```
z=[[0. 0. 0. 0.]
 [0. 1. 1. 0.]
 [0. 1. 1. 0.]
 [0. 0. 0. 0.]]
o and z may share memory: True
```

Many slicing operations return views. Changing contents of a variable that contains a view **changes also the 'original', source array**!

Views and Copies, continued...

To avoid this problem, do a .copy(). Normally mathematical operations also provide a copy instead of just a view.

```
import numpy as np

z = np.zeros((4, 4))

o = z[1:3, 1:3]
o[...] = 1

print(f"z={z}")
print(f"o and z may share memory:
{np.may_share_memory(o, z)}")

p = z[1:3, 1:3].copy()
p[...] = 42
q = z[1:3, 1:3]**2
p[...] = 17

print(f"z={z}")
print(f"p and z may share memory:
{np.may_share_memory(p, z)}")
print(f"q and z may share memory:
{np.may_share_memory(q, z)}")
```

```
z=[[0. 0. 0. 0.]
 [0. 1. 1. 0.]
 [0. 1. 1. 0.]
 [0. 0. 0. 0.]]
p and z may share memory: False
q and z may share memory: False
```

Storing and Retrieving Data

Magnetic drum store storage device, from Deuce computer, 1955-1964 England, English Electric Company Limited

Storing and Retrieving Data: a Map for the Zoo…

Different ways of storing and retrieving data

- The 'classic' way: open, read, write, close (or better, use open together with with!)
  - …for texts
  - …for binary formats
  - …for configurations/parameter files:
    json.dump(s), json.load(s)

→You will probably never need it if you have these other, simpler methods – but they are available as a fallback option!

→ David will show you!

- The unsafe way: pickle

→ Never ever!

- The Numpy way: numpy.load, numpy.save, numpy.savez

→ Today!

- The Matlab way: scipy.io, h5py

→ David will show you!

- The many other proprietary ways…

→ In the tutorials, you will learn how to help yourselves!

Storing and retrieving data: Numpy arrays

Storing data:

Single numpy vars are saved with numpy.save, multiple vars with numpy.savez. You can specify under which name each variable is saved with savez, and you can compress (losslessly) your file to save storage space with numpy.savez_compressed:

```
import numpy as np

a = "Some string"
b = np.eye(4000, 4000)
c = 42

file_single = "save_single"  # .npy gets added...
file_multi = "save_multi"  # .npz gets added...
file_multi_named = "save_multi_named"
file_multi_named_compressed = "save_multi_named_compressed"

np.save(file_single, b)
np.savez(file_multi, a, b, c)
np.savez(file_multi_named, x=a, y=b, z=c)  # better, specify names
np.savez_compressed(file_multi_named_compressed, a=a, b=b, c=c)
```

Storing and retrieving data: Numpy arrays

Retrieving data:

You use numpy.load to load numpy vars. For **single vars** (stored with numpy.save) the contents are directly provided as the return value. For **multiple vars**, the return value is a handle from which the contents can be retrieved by **indexing with their var names** (get a list by assessing handle.files), like in **dictionaries**…

```
b_load = np.load("save_single.npy")  # needs extension in name

handle = np.load("save_multi.npz")  …better also use option 'allow_pickle=True'
print(handle.files)  # -> ['arr_0', 'arr_1', 'arr_2']
print(handle["arr_1"].shape)
                                     …better also use option 'allow_pickle=True'
handle = np.load("save_multi_named_compressed.npz")
print(handle.files)  # -> ['x', 'y', 'z']
print(handle["b"].shape)
```

Storing and retrieving data: Taming a flood of files

Directory structures and filenames

Data on a computer is organized in a tree-like hierarchy with folders and subfolders. This makes it possibly to easily find and access data and program code:

```
C:\Users\Udo\Desktop\Aktuell\Lehre\Programming_WS23>tree
Auflistung der Ordnerpfade
Volumeseriennummer : F4FC-F241
C:.
├───Code
│   ├───OLD1
│   ├───OLD2
│   ├───programming_lecture_05
│   │   └───subdir
│   │       ├───subsubdir
│   │       │   └───__pycache__
│   │       └───__pycache__
│   └───__pycache__
├───Compendium
│   └───OLD1
├───Exercises
├───Material
├───OLD1
```

…except if you keep way too many old versions and obsolete data!

Storing and retrieving data: Taming a flood of files

Handling dirs and files and finding good names!

The modules glob, os.path, and datetime are your friends:

glob.glob(pattern)	- gives a list with filenames matching pattern
os.path.join(a, b, c, …)	- joins directory/file names to a full path
os.path.basename(fullpath)	- gets the filename(+suffix) from a full path…
os.path.dirname(fullpath)	-… gets the other part, i.e. the directory hierarchy
os.path.splitext(basename)	- splits a file basename into filename and suffix
os.path.exists(fullpath)	- tests if file or path exists
os.getsize(fullpath)	- gets size of file
datetime.datetime.now()	- gets current time, as prerequisite for constructing paths

Storing and retrieving data: Taming a flood of files

Examples:

```
import os
import glob
import datetime

everything = glob.glob("./*")
for item in everything:
    print(item)

d = "dir"
s = "subdir"
f = "filename"
x = ".py"
fullname = os.path.join(d, s, f)+x
print(f"Full filename: {fullname}")

basename = os.path.basename(fullname)
print(f"Basename: {basename}")
print(f"Dirname: {os.path.dirname(fullname)}")
print(f"Filename, Suffix: {os.path.splitext(basename)}")

file = "programming_lecture_05.py"
print(f"File {file} exists? {os.path.exists(file)}!")
print(f"Its size is {os.path.getsize(file)} bytes...")
```

Examples continued:

```python
import datetime

d = datetime.datetime.now()
for animal in ['Botox', 'Versace', 'Fritz', 'DonkeyKong']:
    for electrode in range(17):
        recdate = f"{d:%Y-%h-%d_%H%M%S}"
        file = f"Monkey{animal}_Elec{electrode:03d}_RecDate-{recdate}"
        print(f"Saving data under {file}...")
```

```
Saving data under MonkeyBotox_Elec007_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec008_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec009_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec010_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec011_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec012_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec013_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec014_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec015_RecDate-2023-Dec-03_133618...
Saving data under MonkeyBotox_Elec016_RecDate-2023-Dec-03_133618...
Saving data under MonkeyVersace_Elec000_RecDate-2023-Dec-03_133618...
Saving data under MonkeyVersace_Elec001_RecDate-2023-Dec-03_133618...
Saving data under MonkeyVersace_Elec002_RecDate-2023-Dec-03_133618...
Saving data under MonkeyVersace_Elec003_RecDate-2023-Dec-03_133618...
Saving data under MonkeyVersace_Elec004_RecDate-2023-Dec-03_133618...
```

- %a Weekday as locale's abbre
- %A Weekday as locale's full na
- %w Weekday as a decimal num
- %d Day of the month as a zero
- %b Month as locale's abbrevia
- %B Month as locale's full name
- %m Month as a zero-padded d
- %y Year without century as a z
- %Y Year with century as a deci
- %H Hour (24-hour clock) as a z
- %I Hour (12-hour clock) as a z
- %p Locale's equivalent of eithe

, 999999

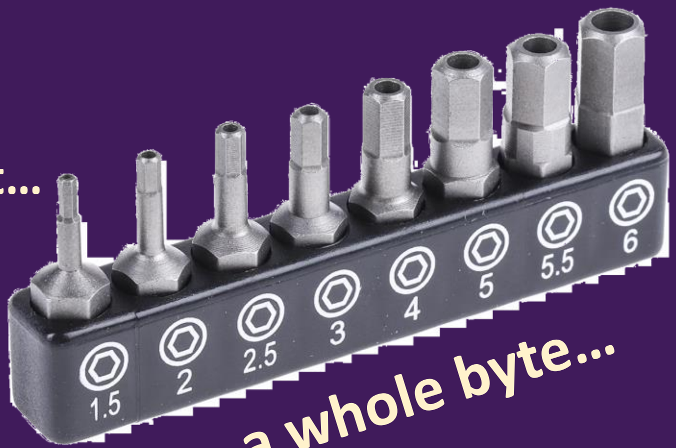
00, +1030

l number.

L6: Missing bits and bytes

Enjoy the Sammelsurium

One bit…



…a whole byte…

…bite!

Missing bits and bytes...

a) Indexing with arrays

Numpy arrays (...or lists) with integer values can be used inside rectangular brackets [...] to address a subset of elements. Here a few simple examples:

```
a = np.arange(10, 0, -1)
indices = [3, 5, 3]
print(a[indices])
```

→

```
[7 5 7]
```

numpy.where (with one arg) gives index list in a tuple where ndarray contents meet a logical expression (see also the more elaborate usage with three args):

```
spiketrain = np.array([0, 1, 0, 0, 0, 4, 1, 0, 0])
dt = 0.010
# attention, returns tuple!
spikes = np.where(spiketrain > 0)
print(f"Neuron fired at time: {spikes[0]*dt}")
print(f"Observed activities were:
{spiketrain[spikes]}")
```

→

```
Neuron fired at time:
[0.01 0.05 0.06]
Observed activities were:
[1 4 1]
```

Missing bits and bytes…

If **two (or more)** index arrays are used to select from two (or more) different dimensions, the resulting selection **pairs the broadcasted indices** in these dimension (i.e., the result is not every combination of the elements in the two index sets, aka Matlab style).

```
    0 1 2
A  0, [[0 1 2]
   1,  [3 4 5]
   2,  [6 7 8]]
```

paired indices

(0,0) (1,1) (2,2)

```
B = A[[0,1,2],[0,1,2]]
```

→ `[0 4 8]` selected values

```
[[First dim],[Second dim] , [… more dims if your array has them]]
```

```
B = A[[0,1,3],[0,1,2]]
```

→ …errors are punished via exceptions

```
IndexError: index 4 is out of bounds for axis 0 with size 3
```

Missing bits and bytes...

So, what about this example?

```
a = np.reshape(np.arange(100), [10, 10])

i = np.arange(10)
j = np.arange(10)

print(a[i, j])

i = np.arange(10)
j = np.arange(10)[:, np.newaxis]

print(a[i, j])
```

```
[ 0 11 22 33 44 55 66 77 88 99]
```

```
[[ 0 10 20 30 40 50 60 70 80 90]
 [ 1 11 21 31 41 51 61 71 81 91]
 [ 2 12 22 32 42 52 62 72 82 92]
 [ 3 13 23 33 43 53 63 73 83 93]
 [ 4 14 24 34 44 54 64 74 84 94]
 [ 5 15 25 35 45 55 65 75 85 95]
 [ 6 16 26 36 46 56 66 76 86 96]
 [ 7 17 27 37 47 57 67 77 87 97]
 [ 8 18 28 38 48 58 68 78 88 98]
 [ 9 19 29 39 49 59 69 79 89 99]]
```

If **n-dimensional index arrays** in combination with slices/colon are used, the resulting selection and dimensions of the result can be quite difficult to imagine (see David's Compendium for examples...)

Missing bits and bytes...

b) Preventing reinventing

If you have a particular, well-defined mathematical/algorithmic task for a numpy ndarray, check out the documentation!

For example, you might find useful.

numpy.flip	- reverses order(s) of ndarray dimension(s)
numpy.roll	- roll ndarray with periodic boundary condition(s)
numpy.tile	- take an ndarray and stitch it into a tile pattern along several dims
numpy.pad	- pad your ndarray from the left and right (or up and down)
numpy.concatenate	- concatenate ndarray along some dimension

Missing bits and bytes...

c) Dictionaries

Dictionaries are **extremely useful to hold data collections which consist of items of different sizes and/or types**. Every member of a dictionary has a name called **key**. These members can be addressed by giving the key in quotes as an index, **similar to accessing the data in a numpy .npz** file:

```
{"one": 1, "two": 2, "three": 3}
```

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'three': 3, 'one': 1, 'two': 2})
f = dict({'one': 1, 'three': 3}, two=2)
```

Missing bits and bytes…

For working with dictionaries, plenty methods are available (see Compendium). We mention only two here, one for **getting the keys**, and the other for **retrieving the corresponding values** (useful for looping over dicts):

```
dishes = {"eggs": 2, "sausage": 1, "bacon": 1, "spam": 500}
keys = dishes.keys()
values = dishes.values()

print(list(keys))
print(list(values))
print(dishes["eggs"])
```

```
['eggs', 'sausage', 'bacon', 'spam']
[2, 1, 1, 500]
2
```

```
for key in dishes.keys():
    print(f"{key} {dishes[key]}")
```

```
eggs 2
sausage 1
bacon 1
spam 500
```

Missing bits and bytes…

d) Module json

json is a wonderful module for **reading/writing dictionaries holding a collection of parameters** (e.g. for a simulation, a configuration file, etc.). The output is human-readable and can be interpreted by many other programs or programming tools!

dump

```
import json

a = dict(one=1, two=2, three=3)

with open("data_out.json", "w") as file:
    json.dump(a, file)
```

Content of data_out.json:

```
{"one": 1, "two": 2, "three": 3}
```

load

```
import json

with open("data_out.json", "r") as file:
    b = json.load(file)

print(b)
```

Output:

```
{'one': 1, 'two': 2, 'three': 3}
```

Missing bits and bytes…

e) When hardware matters…

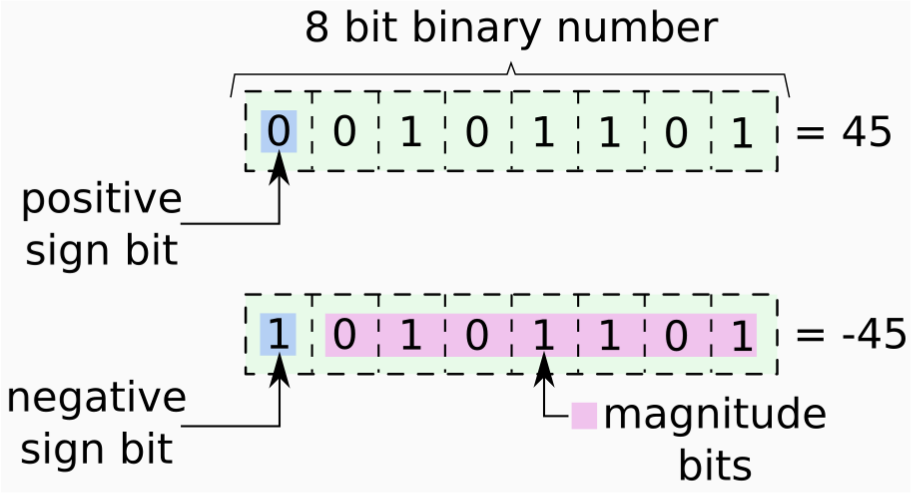
Binary data

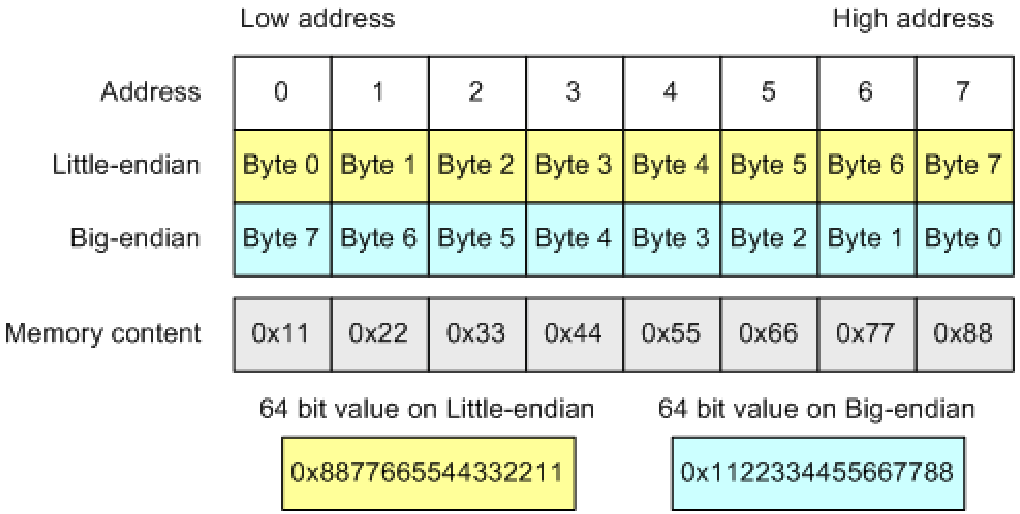
Some information in case you have to handle binary data – also useful if you are working with specific hardware in the lab (e.g. recording devices, controllers such as Arduino):

Everything is a **sequence of 0's and 1's** (i.e. True/False)

8 bits is a **byte** (int8: -128…127). Large integers need more bytes.

Order matters, both within numbers (**endianness**)…





Missing bits and bytes...

Binary data (continued)

...or across numbers (arrays)

Floats represented as **mantissa** and **exponent**:
e.g. 42.42 = **4242***10^**-2**

Numbers have a certain **range**: e.g. uint8 between 0 and 255

Numbers have a certain **precision**: e.g. uint8 has precision 1

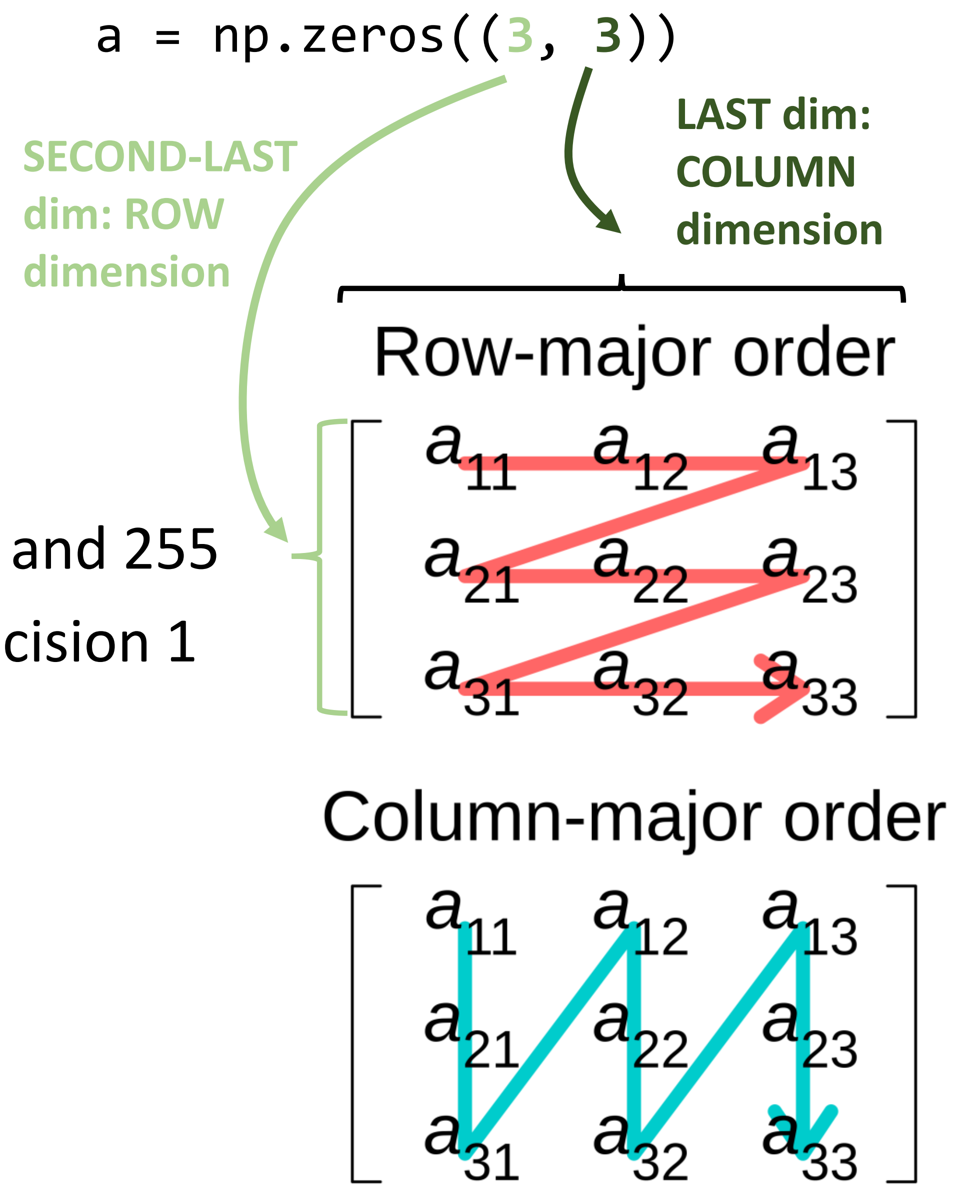
→ **rounding and range errors** might occur!

Floating-point arithmetic:
https://en.wikipedia.org/wiki/Floating-point_arithmetic

Endianness:
https://en.wikipedia.org/wiki/Endianness

Binary numbers:
https://en.wikipedia.org/wiki/Binary_number

Row- and column major ordering:
https://en.wikipedia.org/wiki/Row-_and_column-major_order



Missing bits and bytes…

f) Storing and retrieving data: Matlab

Matlab files occur in different versions. You might have to use different tools:

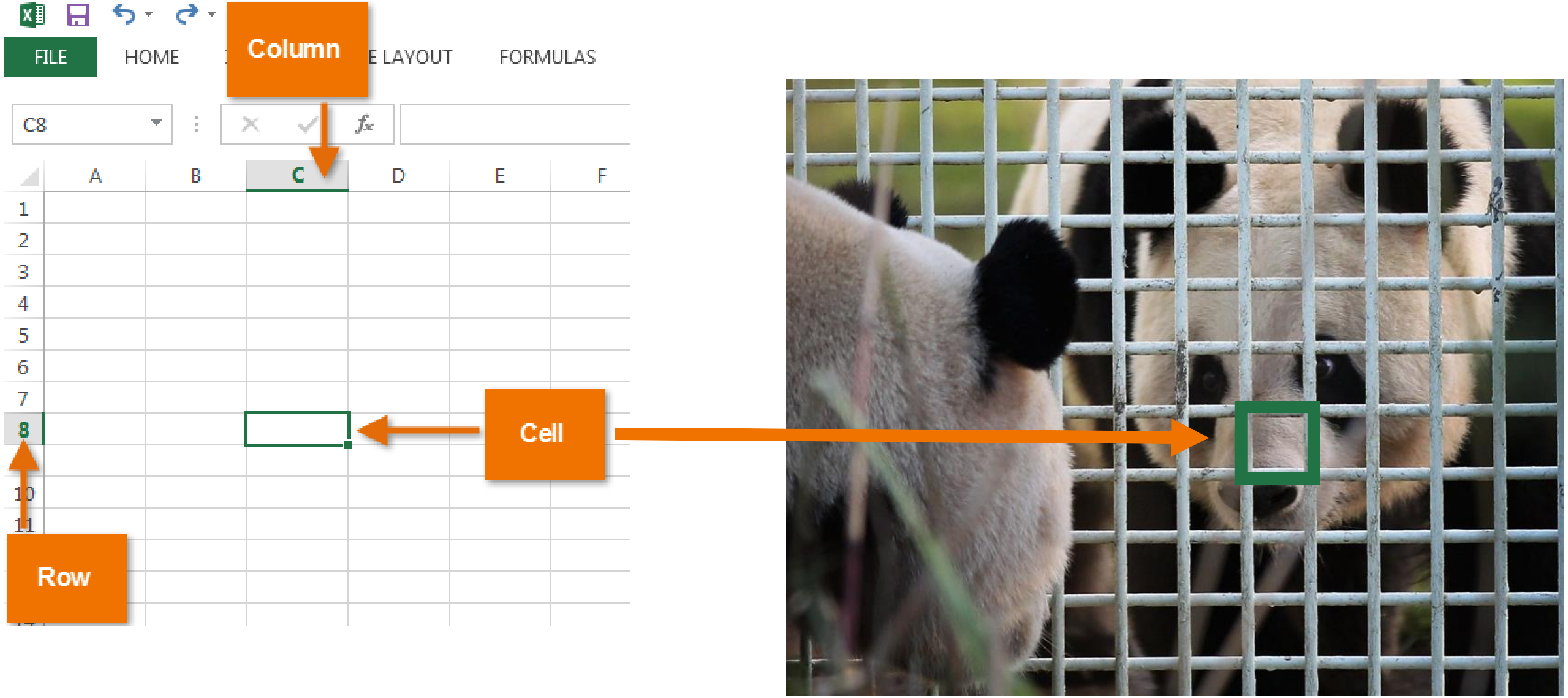
- for Matlab <= 7.2 use scipy.io: loadmat, savemat, whosmat
- from Matlab 7.3 on, data is conveniently stored in the HDF format, https://www.hdfgroup.org/solutions/hdf5/
  Here you can use the HDF module h5py.

MATLAB® files

`loadmat`(file_name[, mdict, appendmat])	Load MATLAB file.
`savemat`(file_name, mdict[, appendmat, …])	Save a dictionary of names and arrays into a MATLAB-style .mat file.
`whosmat`(file_name[, appendmat])	List variables inside a MATLAB file.

Missing bits and bytes…

f) Need a fancy Excel? import pandas!



Missing bits and bytes…

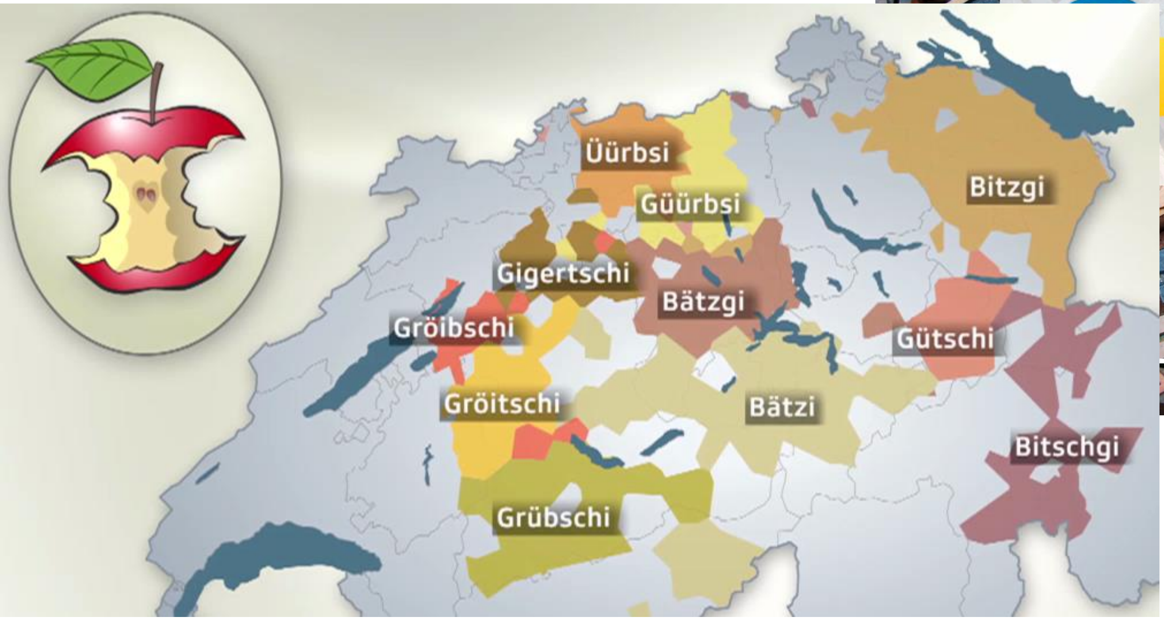
g) Error handling and debugging (try/except/finally)

h) Type annotations

i) Variable scope and namespaces

j) Classes

Namespaces…



Classes…


End of
second Block
You're now a master
of nested loops!