Introduction to Python

Verjinia Metodieva and Daniel Parthier

2025-01-21

Why would you code?

Motivation

- Saving time
- Reproducible workflow
- Flexibility
- Unlimited creativity

Goal of today

```python
# This script exports metadata information to a JSON file.
# The metadata includes the author, date, and the average resting
# membrane potential, units and the sweep count.
# The output is saved to 'data/data_info.json'.
import json
import numpy as np
path = 'data/'
resting_membrane = [-70.1, -73.3, -69.8, -68.5, -71.2]
resting_membrane_avg = np.mean(resting_membrane)
sweeps = [1, 2, 3, 4, 5]
voltage_unit = 'mV'
sweep_count = len(resting_membrane)
output_file = path + 'data_info.json'
output_data = {
    'author': 'Doe, John',
    'date': '2025-01-10',
    'resting_membrane_avg': resting_membrane_avg,
    'unit': voltage_unit,
    'sweep_count': sweep_count
}
with open(output_file, 'w') as f:
    json.dump(output_data, f)
```

This is an example to showcase what we will achieve today.

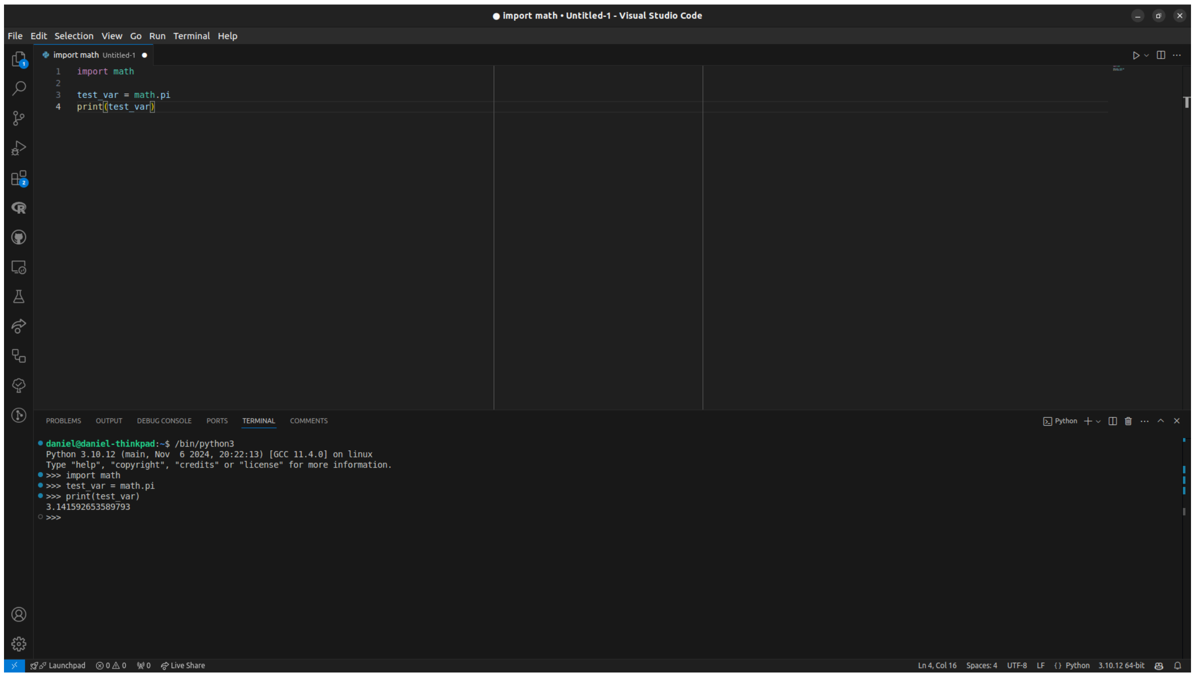
Basic usage

- Can be run from the terminal/console as well
  - Start python by typing `python`/`python3` into the console
  - You can quit by typing `quit()` into the console
- Most of the time with a GUI (graphical user interface)
- Sometimes code is run in document style (*Jupyter Notebook*)
- Run scripts

The console is mainly used for quick testing of commands you will use once and won't need to track. If you want to save your workflow or keep track of what you did, a script is required. In principle, a script is nothing else than a text file with a language-specific extension (.py). The code is saved in such a file and can be used by activating the script as a whole or running single code lines.

Visual Studio Code

- 1 of multiple options (PyCharm, Spyder)
- VS Code offers multifunctionality and integration of useful extensions (Git, Remote Explorer, Jupyter, GitHub Copilot)


import math • Untitled-1 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
import math Untitled-1
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL COMMENTS

```python
import math

test_var = math.pi
print(test_var)
```

```
daniel@daniel-thinkpad:~$ /bin/python3
Python 3.10.12 (main, Nov  6 2024, 20:22:13) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> test_var = math.pi
>>> print(test_var)
3.141592653589793
>>>
```

A graphical user interface, like VS Code, provides code highlighting, formatting, and completion. At the same time, it gives a structured overview of a project. In the case of VS Code, you can use multiple languages by just adding the appropriate extension.

Visual Studio Code (Features)

- Multi-language support (*Python*, *R*, *Matlab*, *Julia*, *C++*, etc.)
- Set up your project (make environment, create files and folders)
- Provide visual notation (code highlighting)
- Auto-complete code snippets
- Show documentation of functions
- Find and fix errors in code (debugging)
- Synchronise code with GitHub
- And much more...

Environments

- Only bring the tools you need
  - Less bloated
  - Fewer conflicts
- Only project-specific packages

Make environment

- Setting up an empty environment (get a drawer)

- Can be done via terminal

```
python -m venv .venv
```

- Make one in VS Code
  - `CTRL + SHIFT + P` → type: *env*
  - Select: *Create Environment*

Start environment

Windows

```
.venv\Scripts\activate
```

Unix/macOS

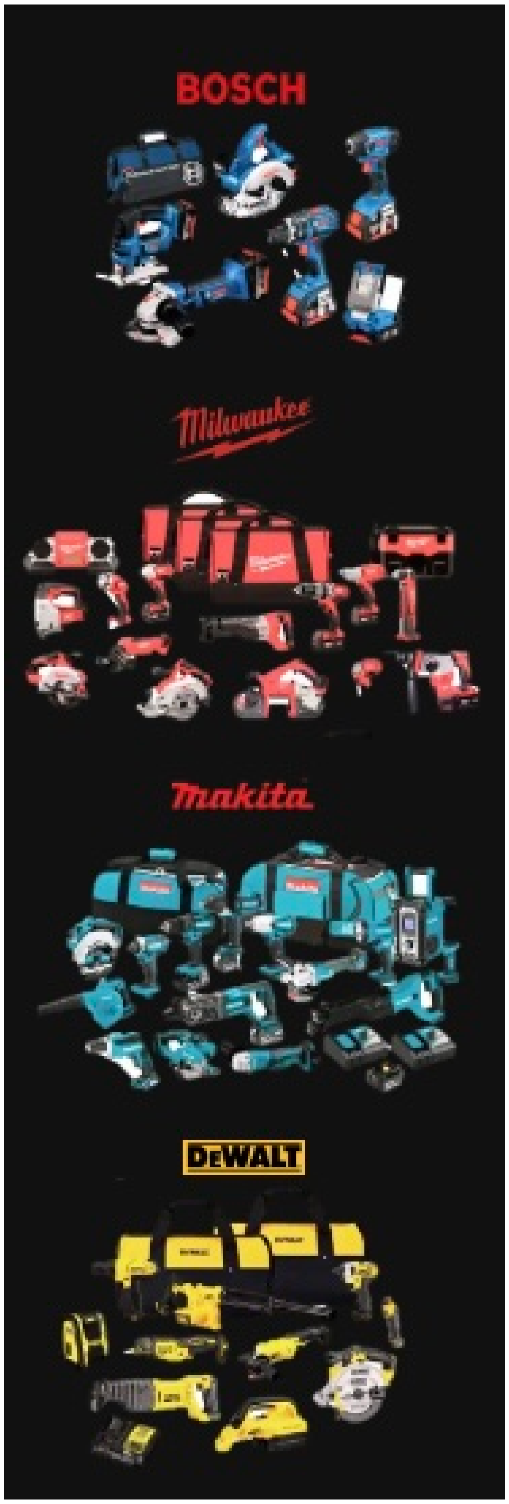
```
source .venv/bin/activate
```

- VS Code will start the environment for you

Quit environment

```
deactivate
```

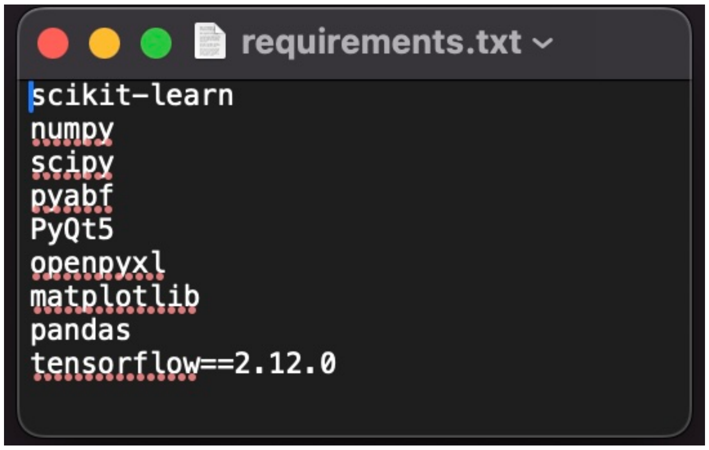
Install packages



Package managers

- *What are they?*
- pip (recommended)
  conda-forge (if you really have to)

Usage



```
scikit-learn
numpy
scipy
pyabf
PyQt5
openpyxl
matplotlib
pandas
tensorflow==2.12.0
```

```
pip install --upgrade pip
pip install -r requirements.txt
```

# installing packages from
requirements.txt file

```
pip uninstall tensorflow
pip install tensorflow==2.12
```

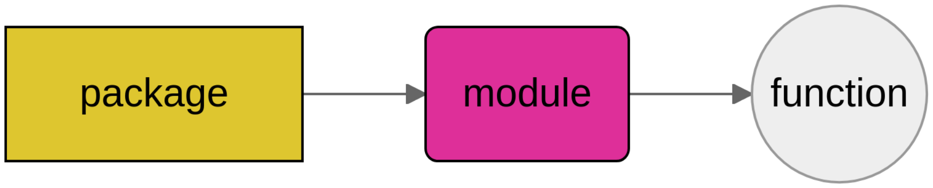
# uninstalling a package
# package with specified version

- a package is a colection of functions
- package manager = a collection of tools that automates the process of installing, upgrading, configuring, and removing computer programs
- helps the user to easily and consistently work with packages - installing, updating, etc.
- provides some security, in the sense that packages that are provided by package managers are already checked for malfunctions

Import

When do you have to import?

- Only some functions are available by default
- Other functions are available through external packages
- A package can have smaller 'packages' inside called modules



Import packages

Let's open the 'numpy' drawer!

```python
import numpy
```

This will open our nummpy *toolbox* drawer

- Pull out a tool to use it with the dot notation: toolbox.tool

```python
numpy.pi
```

```
3.141592653589793
```

```python
numpy.sin(1)
```

```
np.float64(0.8414709848078965)
```

Importing a package only has to happen once. By using `import package` everything from the package will be made available.

Import (abbreviated)

- Some packages can have long names or you want an abbreviation (ie. `numpy` to `np`)
- Assign a new name during the import (`as`)

```python
import numpy as np
```

Generally abbreviating longer packages will make writing, but also reading code easier. At the same time you should check whether your abbreviation makes sense and is understandable for others. There are also common ways to abbreviate well known packages. Some other examples would be `numpy` as `np`, `pandas` as `pd`.

Import single functions

Load only parts of the package

- This could mean single functions or multiple functions/objects
- Avoid clutter and only import what you need

```python
from numpy import sin, cos

sin(1)
cos(1)
```

- Now everything after `import` is available

This way of import comes in handy if you just need a function or part of a large package. However, be careful with conflicting names which can arise from loading the function directly.

Word of advice

- Some packages use the same function names
- Avoid conflicts: reference package or use alias

```python
import math
import numpy as np

math.sin(1)
np.sin(1)
```

Import modules

- Also a module as part of a package can be loaded using `.` or with `from`

```python
from numpy import random
import matplotlib.pyplot as plt

random_number = random.normal(size=2)
plt.plot(random_number)
```

Programming building blocks

Photo by JAQUES London

Building blocks are small objects which can be changed, combined and used in many different ways to build something more complex. We always start with small blocks first. Such building blocks exist in all the programming languages. They might differ slightly but in principle will be very similar. We can also think in the same way about tasks we want to solve. Most of the time we can make a big task into smaller tasks and a small task into even smaller blocks.

Variables

- Variables are objects we want to keep
- We assign them and can use them in the future

```
cell_count = 1

print(cell_count)
```

```
1
```

- Now `cell_count` will be `1` until changed or deleted
- We can assign anything we want

In this example `print()` will print the content of `cell_count` to the terminal.

Assign multiple variables

- Multiple variables can be assigned at the same time

```
cell_count, cell_density, cell_size = 1, 0.3, 4.1
```

- Consider when this is useful (readability)
- It is equivalent to writing 3 lines of code

Operators

Kind of like functions[1] but different

- Have elements on both sides: `a operator b`

Basic operators

1. `+` add two elements together
2. `-` subtract
3. `*` multiply
4. `**` power
5. `/` division, `//` integer[2] division
6. `%` modulus

---

[1]more on functions later
[2]whole numbers

Comparison operators

Let's compare things!

- Comparison operators will tell you if something is:
    - `True` or `False`[3]

1. `==` equal
2. `!=` not equal
3. `<` smaller and `>` larger
4. `<=` smaller or equal and `>=` larger or equal

Logical operators

Logical operators check for conditions and returns `True` or `False`

1. `and` checks if both side are `True`

```
1==1 and 2==2
1==1 and 2>3
```

```
True
```

```
False
```

2. `or` checks if at least one of the sides is `True`

```
1==1 or 2!=2
1==1 or 2==2
1<0 or 1>4
```

```
True
```

```
True
```

```
False
```

[3]`True` and `False` are referred to as boolean/bool

(data-) types

There are many…

- But you only need to know a few
- They can do very different things and might have different properties

---

Strings

- Strings are simply text
- Very important for loading files
  - Directory is a string
- Typically made with `'` or `"` as in `'text'` or `"text"`

```
path = 'data/'
```

What to do with strings

Combine:

```
path = 'data/'
path + 'data_info.json'
```

```
'data/data_info.json'
```

Split string into multiple strings:

```
output_file = path + 'data_info.json'
output_file.split('/')
```

```
['data', 'data_info.json']
```

You are curious about other things?

- Type the name of your string into the python console with a `.` and press `tab` twice
  - `output_file.`

Other things to do with strings

- They have a length
  - Number of characters including spaces etc.

```
test_string = 'four'
len(test_string)
```

```
4
```

- You can access parts of a string (indexing)

```
test_string[1]
```

```
'o'
```

What did you expect?

- In python we start counting positions from `0`
  - 0: `'f'`, 1: `'o'`, 2: `'u'`, 3: `'r'`

What is your intuition for:

```
test_string[-1]
```

```
'r'
```

Strings, as you can see, are characters strung together to form a long chain of single elements. At the same time, you can still access and use single elements from your chain of characters. Should you want to access elements from the end of the chain, you can use negative indices and count backward.

How to break things[4] 101

Be careful how you write text or name folders

```
folder = 'path/sub_path\'
```

```
  File "<stdin>", line 1
    folder = 'path/sub_path''
             ^
SyntaxError: unterminated string literal (detected at line 1)
```

- Avoid escape characters
- Some functions cannot handle special characters

Having complicated folder structures or names with special characters can break you code or functions which want to access this directory. Keep in mind to avoid such characters which could lead to errors (`*`, `\`, `/`). For example when using a folder `data/subpath/` with a file inside called `strange/file/name.txt` python will assume that `strange` and `file` are directories where a file called `name.txt` exists.

There are numbers and there are numbers

- Whole numbers: Integers `int`

```
type(1)
```

```
int
```

- Real numbers: Floats `float`

```
type(1.0)
```

```
float
```

- Most of the time it might not matter[5]

```
1 == 1.0
```

[4]strings
[5]In python

```
True
```

- Sometimes there is a difference and we will see later why

Most of the time python handles the integer vs. float automatically. You will not have to worry about assigning.

None

- A variable which exists but has no content can be `None`

```
var_a = None
type(var_a)
```

```
NoneType
```

- Your program then knows that `var_a` exists
- You can change it later to another value

```
var_a = 1
type(var_a)
```

```
int
```

Tuple

You can combine single elements into one

- Can be different types (`strings`, `int`, `float`, or other objects)
- Chain elements and combine them with `(` and `)`
- Tuples cannot be changed after creation ('immutable')

```
resting_membrane = (-70.1, -73.3, -69.8, -68.5, -71.2)
```

- Single elements can be accessed by their location

```
resting_membrane[1]
```

```
-73.3
```

- Indexing start in python at 0[6]

Tuples cannot be changed

- You cannot modify elements inside the tuple or add any after creation

```
resting_membrane[1] = -72.2
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

---

[6] more on indexing later

Lists

- Multiple items in a list
- Similar to `tuple` but more powerful
- Lists can be changed after creation ('mutable')

Let's list things!

Make lists

- Lists are made by using `[` and `]`
- Elements are inside

```
author_list = ['Verjinia', 'Daniel']
author_list
```

```
['Verjinia', 'Daniel']
```

- Access elements inside a list[7]

```
author_list[1]
author_list.index('Daniel')
```

```
'Daniel'
```

```
1
```

The crucial difference between `lists` and `tuples` is that a `list` allows you to modify the results. Depending on how your code generates the output you might have to decide to take the `list`. If you want that the chain cannot be modified by your code (intended or by accident) a `tuple` is a safe option.

---

[7]remember we start counting at 0

Features of lists

- Lists can do special things

```
author_list.sort()
author_list
```

```
['Daniel', 'Verjinia']
```

```
author_list.append('Dietmar')
author_list
```

```
['Daniel', 'Verjinia', 'Dietmar']
```

```
author_list.remove('Daniel')
author_list
```

```
['Verjinia', 'Dietmar']
```

```
author_list[1] = 'Daniel'
author_list
```

```
['Verjinia', 'Daniel']
```

You can try to sort different lists with numbers and strings or even mixed lists. You will see that there is an order. Also try to use capitalised letters vs. small letters.

Slicing

Sometimes you want to access multiple elements in a chain

- You can use `:` to access a slice between to indices

```
resting_membrane = [-70.1, -73.3, -69.8, -68.5, -71.2]
resting_membrane[1:3]
```

```
[-73.3, -69.8]
```

- Specify 'slice' properties `slice(start, stop, step)`

```
resting_membrane[slice(1,5,2)]
```

```
[-73.3, -68.5]
```

- Alternative using `:stop:step` or `start::step`

```
resting_membrane[:5:2]
resting_membrane[1::2]
```

```
[-70.1, -69.8, -71.2]
```

```
[-73.3, -68.5]
```

Slicing is an important part of accessing data. This allows you to select a subset of the whole data set without going through all the elements separately and without copying the data. When using `:` or `::` without having an integer in front, Python will start at the 0 index. If you use a negative integer as a step, also referred to as stride, you will access items backwards.

Dictionary

You want to to have different things in different things and maybe add some things?

Hello `dictionary`!

- Very flexible
- Still very structured
- Easily accessible

Make dictionary

- Made using { and }
- Contains key and values

```
new_dict = {
    "project": 'course',
    "date": '2025-01-21',
    "authors": author_list
}
new_dict
```

```
{'project': 'course', 'date': '2025-01-21', 'authors': ['Verjinia', 'Daniel']}
```

```
new_dict["authors"]
new_dict["authors"].append('Dietmar')
new_dict
```

```
['Verjinia', 'Daniel']
```

```
{'project': 'course',
 'date': '2025-01-21',
 'authors': ['Verjinia', 'Daniel', 'Dietmar']}
```

Properties of dictionary

- List all the keys

```
new_dict.keys()
```

```
dict_keys(['project', 'date', 'authors'])
```

- Get all values

```
new_dict.values()
```

```
dict_values(['course', '2025-01-21', ['Verjinia', 'Daniel', 'Dietmar']])
```

- Get all values from specific key

```
new_dict['authors']
new_dict.get('authors')
```

```
['Verjinia', 'Daniel', 'Dietmar']
```

```
['Verjinia', 'Daniel', 'Dietmar']
```

- In case you need an output even if key does not exist use `.get`

```
new_dict.get('experimenter', 'unknown')
```

```
'unknown'
```

Simple functions

```
print('This function is boring')
```

```
This function is boring
```

```
# combine variable types
var_a = 'a variable'
value_a = 42
print('more interesting when we include \n', var_a, 'with value', value_a)

print('the varaible type of var_a is', type(var_a))
print('the varaible type of value_a is', type(value_a))
```

```
more interesting when we include
 a variable with value 42
the varaible type of var_a is <class 'str'>
the varaible type of value_a is <class 'int'>
```

```
# sort a list
animal_list = ['SNA 0254581', 'DSC 035576', 'SNA 0954581','SNA 0856662','DSC 024504']
sorted_animal_list = sorted(animal_list)

print(sorted_animal_list)
```

```
['DSC 024504', 'DSC 035576', 'SNA 0254581', 'SNA 0856662', 'SNA 0954581']
```

- call a function
- function
    - A named group of instructions that is executed when the function's name is used in the code.
    - may process input arguments and return a result back
    - logically grouping together pieces of code
- print() - display information on the screen
- type() - outputs the variable type
- can look up functions while typing code

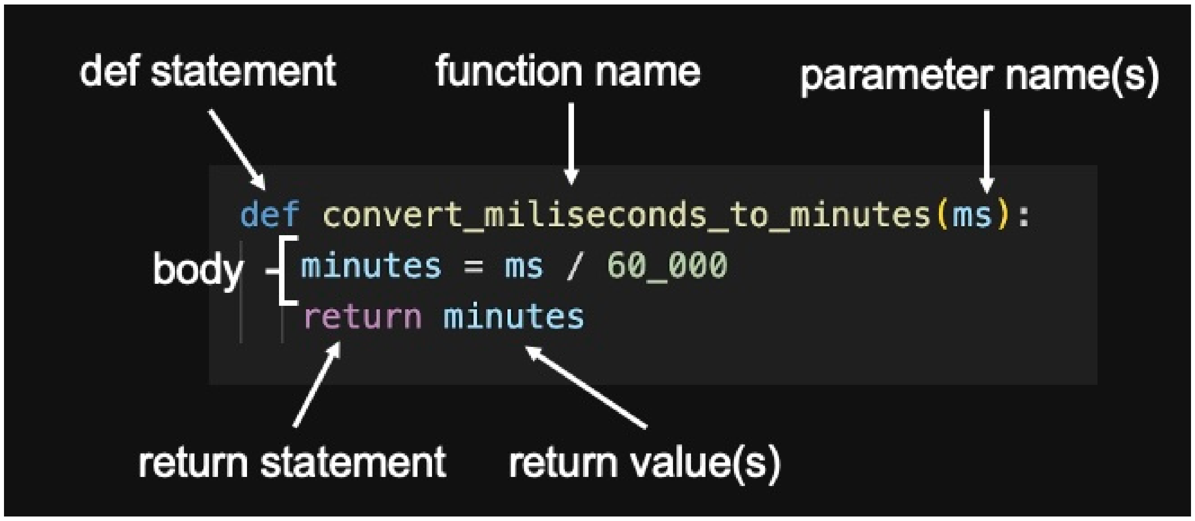
Methods

```python
# sort a list with a method
animal_list = ['SNA 0254581', 'DSC 035576', 'SNA 0954581','SNA 0856662','DSC 024504']

sorted_animal_list = sorted(animal_list)

# same thing but using a method

animal_list.sort() # sorting in ascending order
print(animal_list)

animal_list.sort(reverse = True) # sorting in descending order
print(animal_list)


num_animals = len(animal_list)
print("I've analyzed the date of", num_animals, "animals.")
```

```
['DSC 024504', 'DSC 035576', 'SNA 0254581', 'SNA 0856662', 'SNA 0954581']
['SNA 0954581', 'SNA 0856662', 'SNA 0254581', 'DSC 035576', 'DSC 024504']
I've analyzed the date of 5 animals.
```

- methods
    - .sort()
- Function — a set of instructions that perform a task
- Method — a set of instructions that are associated with an object.
- method is called with an object and has the possibility to modify data of an object.
- can define your own functions –> live coding example

Define your own functinos



```python
def convert_miliseconds_to_minutes(ms):
    minutes = ms / 60_000
    return minutes
```

```python
import json
import numpy as np

path = 'data/'
resting_membrane = [-70.1, -73.3, -69.8, -68.5, -71.2]

resting_membrane_avg = np.mean(resting_membrane)
sweeps = [1, 2, 3, 4, 5]
voltage_unit = 'mV'
sweep_count = len(resting_membrane)

output_file = path + 'data_info.json'


output_data = {
    'author': 'Doe, John',
    'date': '2025-01-10',
    'resting_membrane_avg': resting_membrane_avg,
    'unit': voltage_unit,
    'sweep_count': sweep_count
}

with open(output_file, 'w') as f:
    json.dump(output_data, f)

import json
import numpy as np

def create_meta_data_json(patcher, date_of_rec, RMPs, save_path, save_filename):
  avg_RMP = np.mean(RMPs)
  num_sweeps = len(RMPs)

  output_data = {
    'author': patcher,
    'date': date_of_rec,
    'resting_membrane_avg': avg_RMP,
    'unit': 'mV',
    'sweep_count': num_sweeps
  }

  print('saving the file ', save_filename, 'in', save_path)
  with open(save_path + save_filename, 'w') as f:
    json.dump(output_data, f)

  return output_data

patcher = 'Verji'
date_of_rec = '2025-01-15'
save_path = 'data/'
save_fn  = 'verji_s_first_recording.json'
resting_membrane = [-70.1, -73.3, -69.8, -68.5, -71.2]

# ways to call the function
create_meta_data_json(patcher = 'Verji', date_of_rec = date_of_rec , RMPs = resting_membrane
                                 save_path = save_path, save_filename = save_fn)

out_data = create_meta_data_json(patcher, date_of_rec, resting_membrane, save_path, save_fn)

out_data = create_meta_data_json('Verji', '2025-01-15', [-70.1, -73.3, -69.8, -68.5, -71.2],
                                 'verji_s_first_recording.json')
```

- The function definition opens with the keyword *def* followed by the name of the function (convert_miliseconds_to_minutes) and a parenthesized list of parameter names (ms). The body of the function — the statements that are executed when it runs — is indented below the definition line. The body concludes with a **return** keyword followed by the return value.

  - can set default parameters in a function. this means that a function will work with those parameters, if one doensn't pass other ones

- when passing parameters to a function, it's important that one passes them in the correct order
- local variables
  - variables that live only inside of functions, minutes in our example
  - they no longer exist once the function is done executing show it by running minutes
  - If we try to access their values outside of the function, we will encounter an error
- global variables
  - variables defined outside any function

Documentation

# comments describe the code

- audience - you or other developers
- functionality
- 'why?'

' ''docstrings give explanations' ' '

- how to use?
- for single functions, modules, scripts
- accessed through *help()*

README.md

- project documentation

- commenting
  - use #
  - when something is not intuitive or has to be paid attention to
- docstrings
  - ' ' ' ' ' '
  - from single functions, to modules, to whole scripts

    - different formal formats. see resources.docx for complete overview documenting projects
    - appear (if existing) when you wigle the mouse above a function either in VSC or in jupyter notebook

- readme files
    - prupose of the project, use case example

Documentation example

```python
import json
import numpy as np

def create_meta_data_json(patcher, date_of_rec, RMPs, save_path, save_filename):
    '''
    Returns and saves a dictionary with metadata.

    Args:       patcher (str): name of experimenter
                date_of_rec (str): date of experiment
                RMPs (list): list of recorded resting membrane potentials
                save_path (str): destination folder
                save_filename (str): name of the file

    Returns:    output_data (dictionary): containing the metadata
    '''
  avg_RMP = np.mean(RMPs)
  num_sweeps = len(RMPs) # have as many sweeps as values

  # define metadata dictionary
  output_data = {
    'author': patcher,
    'date': date_of_rec,
    'resting_membrane_avg': avg_RMP,
    'unit': 'mV', # always mV for RMP
    'sweep_count': num_sweeps
    }

  # confirmation to user
  print('saving the file ', save_filename, 'in', save_path)

  with open(save_path + save_filename, 'w') as f:
    json.dump(output_data, f)
    return output_data

resting_membrane = [-70.1, -73.3, -69.8, -68.5, -71.2]
# create a out_dict containing the metadata
out_dict = create_meta_data_json('Verji', '2025-01-15', resting_membrane, 'data/', 'verji_s_
```

Homework

- explaining how to use jupyter notebooks for the homework
- important to mention that the correct environment (kernel) needs to appear in the top right