

Introduction to Computer-based Physical Modeling

Frank Cichos

2024-08-14

This book is a collection of lecture notes for the course “Introduction to Computer-based Physical Modeling” at the University of Leipzig. The course is part of the Bachelor’s program in Physics and is intended for all students. The course covers the basics of computer-based physical modeling, including numerical methods, simulation techniques, and data analysis. The book is designed to be used as a reference for students taking the course, as well as for anyone interested in learning more about computer-based physical modeling.

Table of contents

1	This Book	7
I	Course Info	8
2	Course Information	9
3	Vorlesender	10
4	Diese Webseiten	11
5	Zeitplan für den Kurs	14
6	Ressourcen	15
6.1	Molecular Nanophotonics Group	15
6.2	Additional Advanced Courses	15
6.3	Python Documentation	15
6.4	Python Tutorials	15
6.5	Julia Tutorial	15
6.6	Pluto NoteBook	16
7	Anleitung für das Quiz	17
7.1	Quiz	17
7.1.1	Student Information	17
II	Exam	18
8	Exam	19
III	Lecture 1	20
9	Jupyter Notebooks	21

10 Jupyter Notebooks	22
10.1 What is Jupyter Notebook?	22
10.1.1 Key Components of a Notebook	22
10.1.2 Kernels	23
10.1.3 Notebook Documents	24
10.2 Using the Notebook Editor	27
10.2.1 Edit mode	27
10.2.2 Command mode	28
10.2.3 Keyboard navigation	28
10.2.4 Running code	29
10.3 Managing the kernel	29
10.4 Markdown in Notebooks	29
10.4.1 Markdown basics	30
10.4.2 Headings	31
10.4.3 Embedded code	31
10.4.4 LaTeX equations	31
10.4.5 Images	32
10.4.6 Videos	32
11 Python Overview	34
12 Python Overview	35
12.1 Variables in Python	35
12.1.1 Symbol Names	35
12.1.2 Variable Assignment	35
12.2 Number Types	36
12.2.1 Comparison of Number Types	36
12.2.2 Integers	37
12.2.3 Floating Point Numbers	37
12.2.4 Complex Numbers	37
12.3 Type Conversion	38
12.3.1 Implicit Type Conversion	38
12.3.2 Explicit Type Conversion	38
IV Lecture 2	40
13 Python Overview	41
13.1 Functions	41
13.1.1 Defining a Function	41
13.1.2 Calling a Function	41
13.2 Loops	42
13.2.1 For Loop	42

13.2.2	While Loop	42
13.3	Conditional Statements	42
13.3.1	If Statement	43
13.3.2	Else Statement	43
13.3.3	Elif Statement	43
14	Modules	45
14.1	Modules	45
14.1.1	Namespaces	46
14.1.2	Directory of a module	46
14.1.3	Advanced topics	47
15	Plotting	50
16	Plotting	51
16.1	Simple Plotting - Implicit Version	52
16.1.1	Line Plot	52
16.1.2	Scatter plot	55
16.1.3	Histograms	55
16.1.4	Combined plots	57
16.2	Saving figures	58
16.3	Plots with error bars	58
16.3.1	Setting plotting limits and excluding data	59
16.4	Logarithmic plots	60
16.4.1	Semi-log plots	60
16.4.2	Log-log plots	61
16.5	Arranging multiple plots	62
16.6	Contour and Density Plots	63
16.6.1	Simple contour plot	63
16.6.2	Color contour plot	65
16.6.3	Image plot	65
16.7	Advanced Plotting - Explicit Version	66
16.7.1	Plots with Multiple Spines	67
16.7.2	Insets	68
16.7.3	Spine axis	69
16.7.4	Polar plot	70
16.7.5	Text annotation	70
16.7.6	3D Plotting	70
V	Lecture 3	74
17	Lecture 3	75

18 Dynamics and Simple Motion	76
18.1 Creating Numpy Arrays	76
18.1.1 From lists	76
18.1.2 Using array-generating functions	77
18.2 Manipulating NumPy arrays	81
18.2.1 Slicing	81
18.2.2 Reshaping	83
18.2.3 Adding a new dimension: newaxis	83
18.2.4 Stacking and repeating arrays	84
18.3 Applying mathematical functions	86
18.3.1 Operation involving one array	86
18.3.2 Operations involving multiple arrays	87
18.4 Application	88
 VI Seminar Contents	 89
 19 Seminar 1	 90

1 This Book

Part I

Course Info

2 Course Information

3 Vorlesender

Email: *lastname@physik.uni-leipzig.de*

- Prof. Dr. Frank Cichos
 - Linnéstr. 5, 04103 Leipzig
 - Office: 322
 - Phone: +0341 97 32571

4 Diese Webseiten

Diese Website enthält alle Informationen, die für unseren Kurs **Einführung in die Modellierung Physikalischer Prozesse** erforderlich sind. Sie werden hier jede Woche eine neue Vorlesung und eine neue Aufgabe finden. Die Vorlesungshefte werden von Videos begleitet, die den Inhalt der Vorlesung **auf Englisch** erklären, aber Sie können auch mit dem Lesen auskommen. Die Vorlesungen in Person, werden auf Deutsch stattfinden. Von diesen Webseiten aus werden Sie zu verschiedenen Ressourcen geführt, die Sie nutzen können, um das Programmieren in Python zu lernen. Dabei werden wir einige großartige Tools aus dem Internet nutzen, wie

1. **Google Colab** Dienst, um auch Jupyter Notebooks (<https://colab.research.google.com>) zu hosten. Das Google Colab-Projekt bietet eine nützliche Umgebung zur gemeinsamen Nutzung von Notebooks.

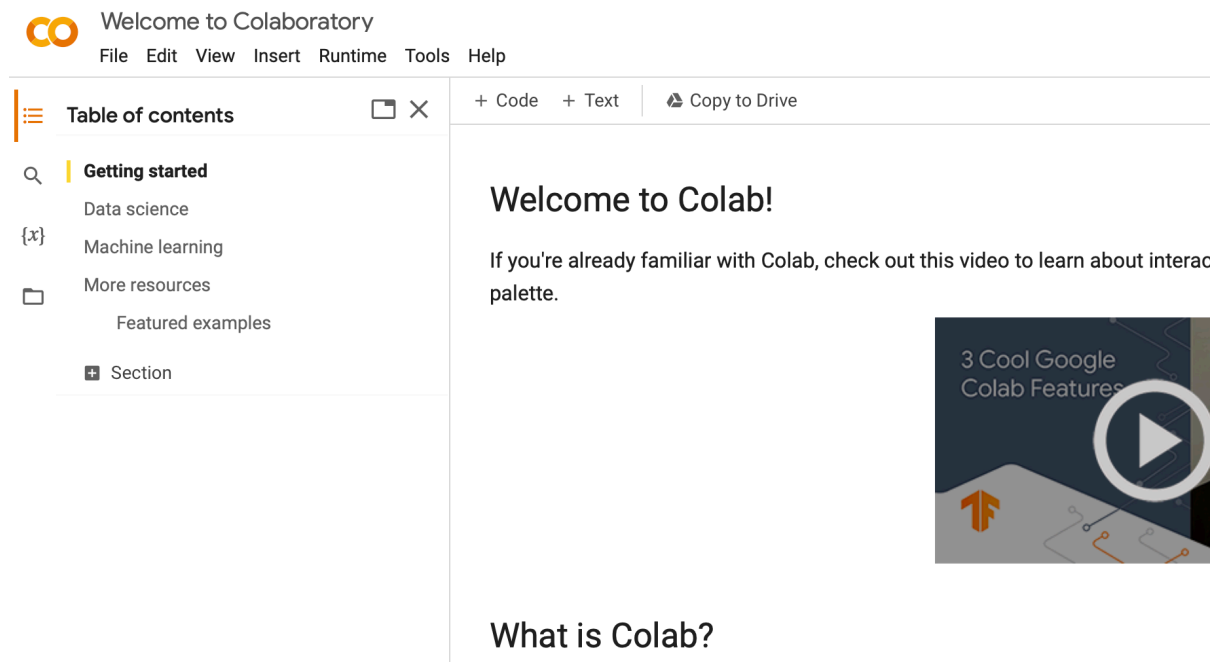


Figure 4.1: google colab screen

Wenn Sie die folgende Website besuchen, werden Sie an mehreren Stellen das folgende Symbol sehen.

![Substitution Name1]

Dieses Symbol zeigt an, dass diese Webseite auf einem Jupyter Notebook basiert. Anstatt nur die Website zu betrachten, können Sie auf das Symbol klicken und der Google Colab-Dienst wird geöffnet, damit Sie das Notizbuch interaktiv nutzen können. Google Colab öffnet sich viel schneller als myBinder, aber die Notizbücher sind für die Arbeit mit myBinder gemacht und nicht alle Funktionen funktionieren mit Colab. Ich arbeite jedoch an der Kompatibilität.

2. **GitHub and GitHub Pages** Dienst zum Hosting von Websites (<https://github.com>). GitHub ist ein großartiger Ort, um Ihre kollaborativen Coding-Projekte einschließlich Versionskontrolle zu hosten. In der oberen rechten Ecke finden Sie auch einen Link zum GitHub-Repository, in dem die Notebooks gehostet werden.

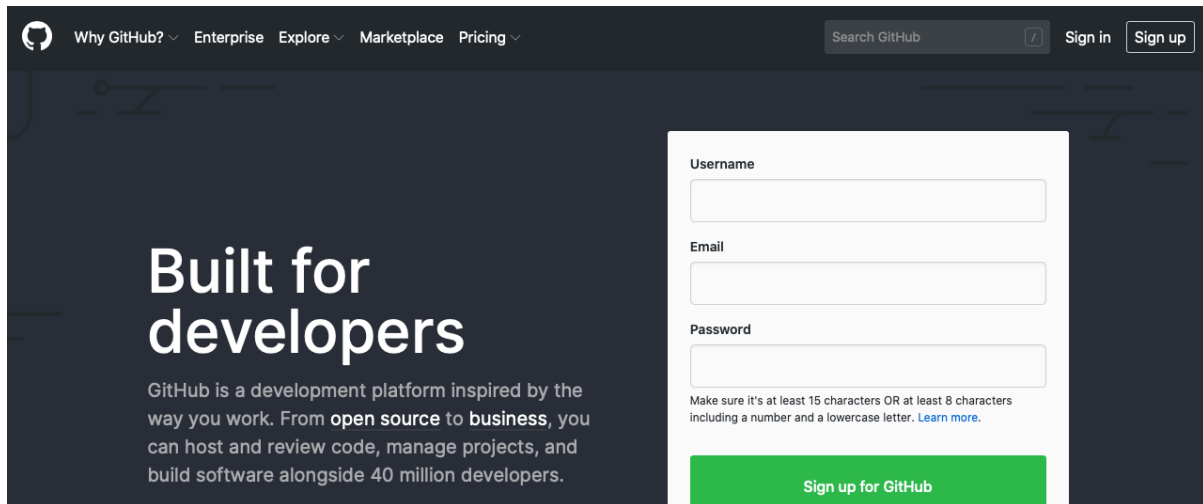


Figure 4.2: github screen

3. **Anaconda Jupyter package** für Notebooks auf dem eigenen Computer (<https://www.anaconda.com/dis>). Das Paket anaconda stellt Ihnen die Jupyter Notebook-Umgebung einschließlich Python zur Verfügung. Wenn Sie Jupyter zu Hause ohne Online-Zugang verwenden möchten, ist dies ein gutes Paket zur Installation.

GARTNER REPORT: ORGANIZATIONAL BEST PRACTICES FOR SUCCESSFUL AI & ML INITIATIVES

The scaling and operationalization of AI and ML are often hindered by nontechnical hurdles. This report aims to help data and analytics leaders achieve better business outcomes.

[Get Report](#)

Figure 4.3: anaconda screen

5 Zeitplan für den Kurs

Der Kurs wird wöchentlich mit dem Zeitplan der Vorlesungen aktualisiert. Erwarten Sie also jede

Dienstag ab XX. Oktober 2024, jeweils um 11:15

eine neue Vorlesung und eine neue Aufgabe. Bitte nutzen Sie den ersten Dienstag, um sich die Website und die Einführungsvideos durchzulesen. Bitte benutzen Sie auch das myBinder/Collab-Symbol am oben auf der Seite und versuchen Sie, sich mit den Notizbüchern vertraut zu machen. Die Vorlesung 1 und die Aufgabenstellung 1 folgen am **XX. Oktober 2024**.

Erfahrungsgemäß werden die besten Ergebnisse erzielt, wenn Sie bei den Vorlesungen im Hörsaal anwesend sind. Das gesamte Material wird jedoch auch online zur Verfügung stehen, so dass Sie jederzeit wiederkommen können, um zu lernen, wann immer es Ihnen passt. Wir werden auch jede Woche Videos (Englisch) hinzufügen, die die Details der Notebooks erklären.

6 Ressourcen

Es gibt eine Menge weiterer gut strukturierter Ressourcen zu Python im Netz. Nachfolgend finden Sie nur eine sehr kleine Auswahl.

6.1 Molecular Nanophotonics Group

- [Molecular Nanophotonics Group Website](#)
- [Computer-based Physical Modeling Website @ MONA](#)
- [Discord Channel Invitation Link for Discussions](#)

6.2 Additional Advanced Courses

- Rosenow Group (Theory), Master Course on [Statistical Mechanics of Deep Learning](#)

6.3 Python Documentation

- [Python](#)
- [Matplotlib](#)
- [Pandas](#)

6.4 Python Tutorials

- [Introduction to Python for Science](#)
- [Nice Matplotlib tutorial](#)

6.5 Julia Tutorial

- [Julia Programming Language](#)

6.6 Pluto NoteBook

- [Pluto GitHub Webpage](#)

In this conversion:

1. I've changed the RST headers to Markdown headers, using `#` for the main title and `##` for sub-headers.
2. The links have been converted from RST format to Markdown format.
3. The overall structure remains the same, but it's now in Quarto-compatible Markdown syntax.

```
`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImNvdXJzZWluZm8ifQ== -->`{=html}
```

```
````{=html}
```

```
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImNvdXJzZWluZm8iLCJib29rSXRlbVR5cGUiOiJjaGFwd
```



# 7 Anleitung für das Quiz

## 7.1 Quiz

### 7.1.1 Student Information

Please enter your details before starting the quiz.

Name: Email: Start Quiz

**Question:** Write a python code that generates the number 1 to 10 and calculates the mean value without loading any additional modules.

```
#| exercise: ex1
Replace this comment with your code
For example add the lines:
n = range(1, 10)
sum(n)/len(n)
#
The last line will define the result and will be checked.
```

```
#| exercise: ex1
#| check: true
#| solution: true

n = range(1, 11)
mean = sum(n)/len(n)
feedback = None
if (result == mean):
 feedback = { "correct": True, "message": "Nice work!" }
else:
 feedback = { "correct": False, "message": "That's incorrect, sorry." }
feedback
```

#### Hint

Use the `range`, `sum` and `len` functions.

# **Part II**

# **Exam**

## 8 Exam

**Part III**

**Lecture 1**

## 9 Jupyter Notebooks

# 10 Jupyter Notebooks

## 10.1 What is Jupyter Notebook?

A Jupyter Notebook is a web browser based **interactive computing environment** that enables users to create documents that include code to be executed, results from the executed code such as plots and images, and finally also an additional documentation in form of markdown text including equations in LaTeX.

These documents provide a **complete and self-contained record of a computation** that can be converted to various formats and shared with others using email, version control systems (like git/[GitHub](#)) or [nbviewer.jupyter.org](#).

### 10.1.1 Key Components of a Notebook

The Jupyter Notebook ecosystem consists of three main components:

1. Notebook Editor
2. Kernels
3. Notebook Documents

Let's explore each of these components in detail:

#### 10.1.1.1 Notebook Editor

The Notebook editor is an interactive web-based application for creating and editing notebook documents. It enables users to write and run code, add rich text, and multimedia content. When running Jupyter on a server, users typically use either the classic Jupyter Notebook interface or JupyterLab, an advanced version with more features.

Key features of the Notebook editor include:

- **Code Editing:** Write and edit code in individual cells.
- **Code Execution:** Run code cells in any order and display computation results in various formats (HTML, LaTeX, PNG, SVG, PDF).
- **Interactive Widgets:** Create and use JavaScript widgets that connect user interface controls to kernel-side computations.

- **Rich Text:** Add documentation using [Markdown](#) markup language, including LaTeX equations.

#### Advance Notebook Editor Info

The Notebook editor in Jupyter offers several advanced features:

- **Cell Metadata:** Each cell has associated metadata that can be used to control its behavior. This includes tags for slideshows, hiding code cells, and controlling cell execution.
- **Magic Commands:** Special commands prefixed with `%` (line magics) or `%%` (cell magics) that provide additional functionality, such as timing code execution or displaying plots inline.
- **Auto-completion:** The editor provides context-aware auto-completion for Python code, helping users write code more efficiently.
- **Code Folding:** Users can collapse long code blocks for better readability.
- **Multiple Cursors:** Advanced editing with multiple cursors for simultaneous editing at different locations.
- **Split View:** The ability to split the notebook view, allowing users to work on different parts of the notebook simultaneously.
- **Variable Inspector:** A tool to inspect and manage variables in the kernel's memory.
- **Integrated Debugger:** Some Jupyter environments offer an integrated debugger for step-by-step code execution and inspection.

### 10.1.2 Kernels

Kernels are the computational engines that execute the code contained in a notebook. They are separate processes that run independently of the notebook editor.

Key responsibilities of kernels include:

- \* Executing user code
- \* Returning computation results to the notebook editor
- \* Handling computations for interactive widgets
- \* Providing features like tab completion and introspection

### Advanced Kernel Info

Jupyter notebooks are language-agnostic. Different kernels can be installed to support various programming languages such as Python, R, Julia, and many others. The default kernel runs Python code, but users can select different kernels for each notebook via the Kernel menu.

Kernels communicate with the notebook editor using a JSON-based protocol over ZeroMQ/WebSockets. For more technical details, see the [messaging specification](#).

Each kernel runs in its own environment, which can be customized to include specific libraries and dependencies. This allows users to create isolated environments for different projects, ensuring that dependencies do not conflict.

Kernels also support interactive features such as:

- **Tab Completion:** Provides suggestions for variable names, functions, and methods as you type, improving coding efficiency.
- **Introspection:** Allows users to inspect objects, view documentation, and understand the structure of code elements.
- **Rich Output:** Supports various output formats, including text, images, videos, and interactive widgets, enhancing the interactivity of notebooks.

Advanced users can create custom kernels to support additional languages or specialized computing environments. This involves writing a kernel specification and implementing the necessary communication protocols.

For managing kernels, Jupyter provides several commands and options:

- **Starting a Kernel:** Automatically starts when a notebook is opened.
- **Interrupting a Kernel:** Stops the execution of the current code cell, useful for halting long-running computations.
- **Restarting a Kernel:** Clears the kernel's memory and restarts it, useful for resetting the environment or recovering from errors.
- **Shutting Down a Kernel:** Stops the kernel and frees up system resources.

Users can also monitor kernel activity and resource usage through the Jupyter interface, ensuring efficient and effective use of computational resources.

### 10.1.3 Notebook Documents

Notebook documents are self-contained files that encapsulate all content created in the notebook editor. They include code inputs/outputs, Markdown text, equations, images, and other media. Each document is associated with a specific kernel and serves as both a human-readable record of analysis and an executable script to reproduce the work.



Characteristics of notebook documents:

- **File Extension:** Notebooks are stored as files with a `.ipynb` extension.
- **Structure:** Notebooks consist of a linear sequence of cells, which can be one of three types:
  - **Code cells:** Contain executable code and its output.
  - **Markdown cells:** Contain formatted text, including LaTeX equations.
  - **Raw cells:** Contain unformatted text, preserved when converting notebooks to other formats.

#### Advanced Notebook Documents Info

- **Version Control:** Notebook documents can be version controlled using systems like Git. This allows users to track changes, collaborate with others, and revert to previous versions if needed. Tools like `nbdime` provide diff and merge capabilities specifically designed for Jupyter Notebooks.
- **Cell Tags:** Cells in a notebook can be tagged with metadata to control their behavior during execution, export, or presentation. For example, tags can be used to hide input or output, skip execution, or designate cells as slides in a presentation.
- **Interactive Widgets:** Notebook documents can include interactive widgets that allow users to manipulate parameters and visualize changes in real-time. This is particularly useful for data exploration and interactive simulations.
- **Extensions:** The Jupyter ecosystem supports a wide range of extensions that enhance the functionality of notebook documents. These extensions can add features like spell checking, code formatting, and integration with external tools and services.
- **Security:** Notebook documents can include code that executes on the user's machine, which poses security risks. Jupyter provides mechanisms to sanitize notebooks and prevent the execution of untrusted code. Users should be cautious when opening notebooks from unknown sources.
- **Collaboration:** Jupyter Notebooks can be shared and collaboratively edited in real-time using platforms like Google Colab, Microsoft Azure Notebooks, and Jupyter-Hub. These platforms provide cloud-based environments where multiple users can work on the same notebook simultaneously.
- **Customization:** Users can customize the appearance and behavior of notebook documents using CSS and JavaScript. This allows for the creation of tailored interfaces and enhanced user experiences.

- **Export Options:** In addition to static formats, notebooks can be exported to interactive formats like dashboards and web applications. Tools like **Voila** convert notebooks into standalone web applications that can be shared and deployed.
- **Provenance:** Notebooks can include provenance information that tracks the origin and history of data and computations. This is important for reproducibility and transparency in scientific research.
- **Documentation:** Notebook documents can serve as comprehensive documentation for projects, combining code, results, and narrative text. This makes them valuable for teaching, tutorials, and sharing research findings.
- **Performance:** Large notebooks with many cells and outputs can become slow and unwieldy. Techniques like cell output clearing, using lightweight data formats, and splitting notebooks into smaller parts can help maintain performance.
- **Integration:** Jupyter Notebooks can integrate with a wide range of data sources, libraries, and tools. This includes databases, cloud storage, machine learning frameworks, and visualization libraries, making them a versatile tool for data science and research.
- **Internal Format:** Notebook files are **JSON** text files with binary data encoded in **base64**, making them easy to manipulate programmatically.
- **Exportability:** Notebooks can be exported to various static formats (HTML, reStructuredText, LaTeX, PDF, slide shows) using Jupyter's **nbconvert** utility.
- **Sharing:** Notebooks can be shared via **nbviewer**, which renders notebooks from public URLs or GitHub as static web pages, allowing others to view the content without installing Jupyter.

This integrated system of editor, kernels, and documents makes Jupyter Notebooks a powerful tool for interactive computing, data analysis, and sharing of computational narratives.

## 10.2 Using the Notebook Editor

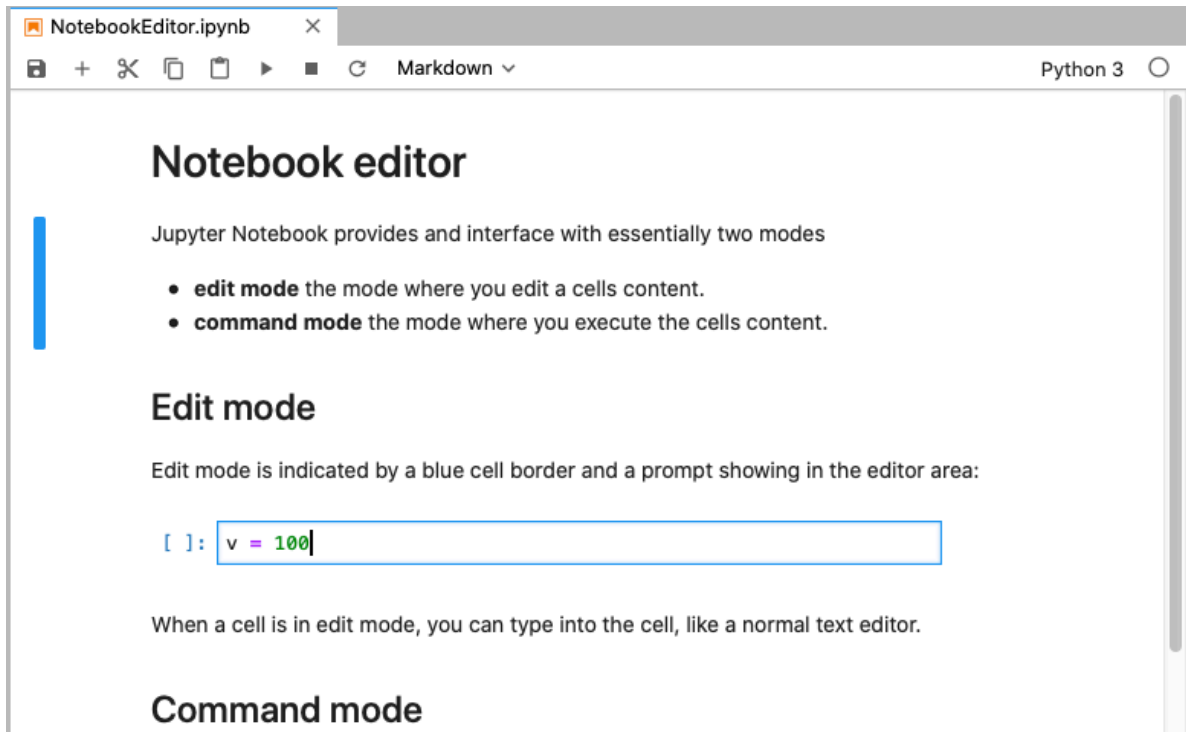


Figure 10.1: Jupyter Notebook Editor

The Jupyter Notebook editor provides an interactive environment for writing code, creating visualizations, and documenting computational workflows. It consists of a web-based interface that allows users to create and edit notebook documents containing code, text, equations, images, and interactive elements. A Jupyter Notebook provides an interface with essentially two modes of operation:

- **edit mode** the mode where you edit a cells content.
- **command mode** the mode where you execute the cells content.

In the more advanced version of JupyterLab you can also have a **presentation mode** where you can present your notebook as a slideshow.

### 10.2.1 Edit mode

Edit mode is indicated by a blue cell border and a prompt showing in the editor area when a cell is selected. You can enter edit mode by pressing **Enter** or using the mouse to click on a cell's editor area.

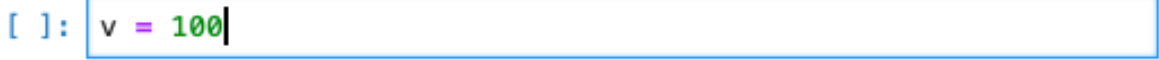
A screenshot of a Jupyter Notebook cell in Edit Mode. The cell has a blue border and contains the text `[ ]: v = 100|`, where the cursor is at the end of the line.

Figure 10.2: Edit Mode

When a cell is in edit mode, you can type into the cell, like a normal text editor

### 10.2.2 Command mode

Command mode is indicated by a grey cell border with a blue left margin. When you are in command mode, you are able to edit the notebook as a whole, but not type into individual cells. Most importantly, in command mode, the keyboard is mapped to a set of shortcuts that let you perform notebook and cell actions efficiently.

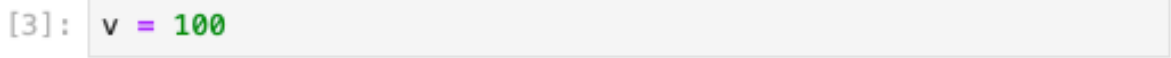
A screenshot of a Jupyter Notebook cell in Command Mode. The cell has a grey border and contains the text `[3]: v = 100`.

Figure 10.3: Command Mode


If you have a hardware keyboard connected to your iOS device, you can use Jupyter keyboard shortcuts. The modal user interface of the Jupyter Notebook has been optimized for efficient keyboard usage. This is made possible by having two different sets of keyboard shortcuts: one set that is active in edit mode and another in command mode.

### 10.2.3 Keyboard navigation

In edit mode, most of the keyboard is dedicated to typing into the cell's editor area. Thus, in edit mode there are relatively few shortcuts available. In command mode, the entire keyboard is available for shortcuts, so there are many more. Most important ones are:

1. Switch command and edit mods: **Enter** for edit mode, and **Esc** or **Control** for command mode.
2. Basic navigation: **↑/k**, **↓/j**
3. Run or render currently selected cell: **Shift+Enter** or **Control+Enter**
4. Saving the notebook: **s**
5. Change Cell types: **y** to make it a **code** cell, **m** for **markdown** and **r** for **raw**
6. Inserting new cells: **a** to **insert above**, **b** to **insert below**
7. Manipulating cells using pasteboard: **x** for **cut**, **c** for **copy**, **v** for **paste**, **d** for **delete** and **z** for **undo delete**
8. Kernel operations: **i** to **interrupt** and **0** to **restart**

### 10.2.4 Running code

Code cells allow you to enter and run code. Run a code cell by pressing the  button in the bottom-right panel, or **Control+Enter** on your hardware keyboard.



```
v = 23752636
print(v)
```



23752636

There are a couple of keyboard shortcuts for running code:

- **Control+Enter** run the current cell and enters command mode.
- **Shift+Enter** runs the current cell and moves selection to the one below.
- **Option+Enter** runs the current cell and inserts a new one below.

## 10.3 Managing the kernel

Code is run in a separate process called the **kernel**, which can be interrupted or restarted. You can see kernel indicator in the top-right corner reporting current kernel state:  means kernel is **ready** to execute code, and  means kernel is currently **busy**. Tapping kernel indicator will open **kernel menu**, where you can reconnect, interrupt or restart kernel.

Try running the following cell — kernel indicator will switch from  to , i.e. reporting kernel as “busy”. This means that you won’t be able to run any new cells until current execution finishes, or until kernel is interrupted. You can then go to kernel menu by tapping the kernel indicator and select “Interrupt”.


## 10.4 Markdown in Notebooks

Text can be added to Jupyter Notebooks using Markdown cells. This is extremely useful providing a complete documentation of your calculations or simulations. In this way, everything really becomes an notebook. You can change the cell type to Markdown by using the “Cell Actions” menu, or with a hardware keyboard shortcut **m**. Markdown is a popular markup language that is a superset of HTML. Its specification can be found here:

<https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>

Markdown cells can either be **rendered** or **unrendered**.

When they are rendered, you will see a nice formatted representation of the cell’s contents.

When they are unrendered, you will see the raw text source of the cell. To render the selected cell, click the  button or **shift+ enter**. To unrender, select the markdown cell, and press **enter** or just double click.

### 10.4.1 Markdown basics

Below are some basic markdown examples, in its rendered form. If you wish to access how to create specific appearances, double click the individual cells to put them into an unrendered edit mode.

You can make text *italic* or **bold**. You can build nested itemized or enumerated lists:

- First item
  - First subitem
    - \* First sub-subitem
  - Second subitem
    - \* First subitem of second subitem
    - \* Second subitem of second subitem
- Second item
  - First subitem
- Third item
  - First subitem

Now another list:

1. Here we go
  1. Sublist
  2. Sublist
2. There we go
3. Now this

Here is a blockquote:

Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Namespaces are one honking great idea – let's do more of those!

And Web links:

[Jupyter's website](#)

## 10.4.2 Headings

You can add headings by starting a line with one (or multiple) `#` followed by a space and the title of your section. The number of `#` you use will determine the size of the heading

```
Heading 1
Heading 2
Heading 2.1
Heading 2.2
Heading 2.2.1
```

## 10.4.3 Embedded code

You can embed code meant for illustration instead of execution in Python:

```
def f(x):
 """a docstring"""
 return x**2
```

## 10.4.4 LaTeX equations

Courtesy of MathJax, you can include mathematical expressions both inline:  $e^{i\pi} + 1 = 0$  and displayed:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!} x^i$$

Inline expressions can be added by surrounding the latex code with `$`:

`$e^{i\pi} + 1 = 0$`

Expressions on their own line are surrounded by `$$`:

```
$$e^x=\sum_{i=0}^{\infty} \frac{1}{i!}x^i$$
```

### 10.4.5 Images

Images may be also directly integrated into a Markdown block.

To include images use

```
![alternative text](url)
```

for example

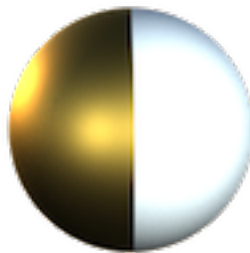


Figure 10.4: alternative text

### 10.4.6 Videos

To include videos, we use HTML code like

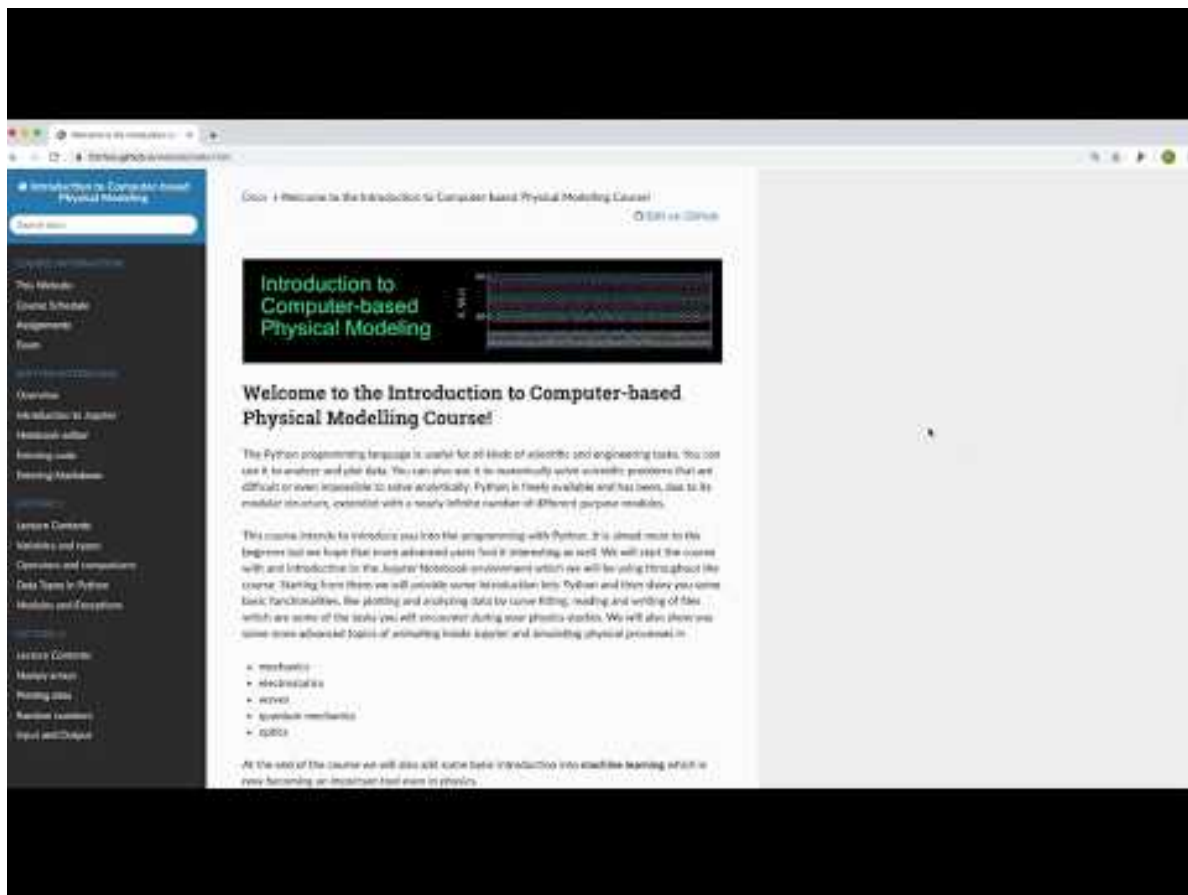
```
<video src="mov/movie.mp4" width="320" height="200" controls preload></video>
```

in the Markdown cell. This works with videos stored locally.

You can embed YouTube Videos as well by using the IPython module.

```
from IPython.display import YouTubeVideo
YouTubeVideo('Q1Lx32juGzI',width=600)
```





# 11 Python Overview

# 12 Python Overview

## 12.1 Variables in Python

### 12.1.1 Symbol Names

Variable names in Python can include alphanumerical characters `a-z`, `A-Z`, `0-9`, and the special character `_`. Normal variable names must start with a letter or an underscore. By convention, variable names typically start with a lower-case letter, while Class names start with a capital letter and internal variables start with an underscore.

#### Reserved Keywords

There are a number of Python keywords that cannot be used as variable names because Python uses them for other things. These keywords are:

`and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `with`, `yield`

Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. However, as a reserved keyword, it cannot be used as a variable name.

### 12.1.2 Variable Assignment

The assignment operator in Python is `=`. Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one.

Assigning a value to a new variable creates the variable:

```
variable assignments
x = 1.0
my_favorite_variable = 12.2
x
```

Although not explicitly specified, a variable does have a type associated with it (e.g., integer, float, string). The type is derived from the value that was assigned to it. To determine the type of a variable, we can use the `type` function.

```
type(x)
```

If we assign a new value to a variable, its type can change.

```
x = 1
```

```
type(x)
```

If we try to use a variable that has not yet been defined, we get a `NameError` error.

```
#print(g)
```

## 12.2 Number Types

Python supports various number types, including integers, floating-point numbers, and complex numbers. These are some of the basic building blocks of doing arithmetic in any programming language. We will discuss each of these types in more detail.

### 12.2.1 Comparison of Number Types

Type	Example	Description	Limits	Use Cases
int	42	Whole numbers	Unlimited precision (bounded by available memory)	Counting, indexing
float	3.14159	Decimal numbers	Typically $\pm 1.8e308$ with 15-17 digits of precision (64-bit)	Scientific calculations, prices
complex	$2 + 3j$	Numbers with real and imaginary parts	Same as float for both real and imaginary parts	Signal processing, electrical engineering
bool	True / False	Logical values	Only two values: True (1) and False (0)	Conditional operations, flags

## Examples for Number Types

### 12.2.2 Integers

**Integer Representation:** Integers are whole numbers without a decimal point.

```
x = 1
type(x)
```

**Binary, Octal, and Hexadecimal:** Integers can be represented in different bases:

```
0b1010111110 # Binary
0x0F # Hexadecimal
```

### 12.2.3 Floating Point Numbers

**Floating Point Representation:** Numbers with a decimal point are treated as floating-point values.

```
x = 3.141
type(x)
```

**Maximum Float Value:** Python handles large floats, converting them to infinity if they exceed the maximum representable value.

```
1.7976931348623157e+308 * 2 # Output: inf
```

### 12.2.4 Complex Numbers

**Complex Number Representation:** Complex numbers have a real and an imaginary part.

```
c = 2 + 4j
type(c)
```

- **Accessors for Complex Numbers:**

- `c.real`: Real part of the complex number.
- `c.imag`: Imaginary part of the complex number.

```
print(c.real)
print(c.imag)
```

**Complex Conjugate:** Use the `.conjugate()` method to get the complex conjugate.

```
c = c.conjugate()
print(c)
```

## 12.3 Type Conversion

### 12.3.1 Implicit Type Conversion

Python automatically converts types during operations involving mixed types. The result is a type that can accommodate all the values involved in the operation. This is known as implicit type conversion. For example, adding an integer and a float results in a float.

```
integer_number = 123
float_number = 1.23
new_number = integer_number + float_number
type(new_number)
```

### 12.3.2 Explicit Type Conversion

Use functions like `int()`, `float()`, and `str()` to explicitly convert types from one to another. Explicit type conversion is also known as type casting. For example, converting a string to an integer.

```
num_string = "12"
num_string = int(num_string)
type(num_string)
```

Converting a float to an integer truncates the decimal part.

```
x = 5 / 2
x = int(x)
z = complex(x)
```

#### Handling Complex Numbers

Complex numbers cannot be directly converted to floats or integers; extract the real or imaginary part first.

```
y = bool(z.real)
print(z.real, " -> ", y, type(y))
```

**Part IV**

**Lecture 2**



# 13 Python Overview

## 13.1 Functions

Functions are reusable blocks of code that can be executed multiple times from different parts of your program. They help in organizing code, making it more readable, and reducing redundancy. Functions can take input arguments and return output values.

### 13.1.1 Defining a Function

A function in Python is defined using the `def` keyword followed by the name of the function, which is usually descriptive and indicates what the function does. The parameters inside the parentheses indicate what data the function expects to receive. The `->` symbol is used to specify the return type of the function.

Here's an example:

```
#| autorun: false
Define a function that takes two numbers as input and returns their sum
def add_numbers(a: int, b: int) -> int:
 return a + b
```

### 13.1.2 Calling a Function

Functions can be called by specifying the name of the function followed by parentheses containing the arguments. The arguments passed to the function should match the number and type of parameters defined in the function. Here's an example:

```
#| autorun: false
Call the function with two numbers as input
result = add_numbers(2, 3)
print(result) # prints 5
```

## 13.2 Loops

Loops are used to execute a block of code repeatedly. There are two main types of loops in Python: `for` loops and `while` loops.

### 13.2.1 For Loop

A `for` loop in Python is used to iterate over a sequence (such as a list or string) and execute a block of code for each item in the sequence. Here's an example:

```
#| autorun: false
Define a function that prints numbers from 1 to 10
def print_numbers():
 for i in range(1, 11):
 print(i)

print_numbers()
```

### 13.2.2 While Loop

A `while` loop in Python is used to execute a block of code while a certain condition is met. The loop continues as long as the condition is true. Here's an example:

```
#| autorun: false
Define a function that prints numbers from 1 to 10 using a while loop
def print_numbers_while():
 i = 1
 while i <= 10:
 print(i)
 i += 1

print_numbers_while()
```

## 13.3 Conditional Statements

Conditional statements are used to control the flow of your program based on conditions. The main conditional statements in Python are `if`, `else`, and `elif`.

### 13.3.1 If Statement

An if statement in Python is used to execute a block of code if a certain condition is met. Here's an example:

```
#| autorun: false
Define a function that prints "hello" or "goodbye" depending on the hour of day
def print_hello():
 current_hour = 12
 if current_hour < 18:
 print("hello")

print_hello()
```

### 13.3.2 Else Statement

An else statement in Python is used to execute a block of code if the condition in an if statement is not met. Here's an example:

```
#| autorun: false
Define a function that prints "hello" or "goodbye" depending on the hour of day
def print_hello_or_goodbye():
 current_hour = 12
 if current_hour < 18:
 print("hello")
 else:
 print("goodbye")

print_hello_or_goodbye()
```

### 13.3.3 Elif Statement

An elif statement in Python is used to execute a block of code if the condition in an if statement is not met but under an extra condition. Here's an example:

```
#| autorun: false
Define a function that prints "hello","goodbye" or "good night" depending on the hour of day
def print_hello_or_goodbye():
 current_hour = 12
 if current_hour < 18:
```

```
 print("hello")
elif <20:
 print("goodbye")
else:
 print("good night")
print_hello_or_goodbye()
```

# 14 Modules

## 14.1 Modules

Most of the functionality in Python is provided by *modules*. The Python Standard Library is a large collection of modules that provides *cross-platform* implementations of common facilities such as access to the operating system, file I/O, string management, network communication, math, web-scraping, text manipulation, machine learning and much more.

To use a module in a Python module it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

```
#| autorun: false
import math
import numpy

x = math.sqrt(2 * math.pi)
x = numpy.sqrt(2 * numpy.pi)

print(x)
```

This includes the whole module and makes it available for use later in the program. Alternatively, we can choose to import all symbols (functions and variables) in a module so that we don't need to use the prefix "`math.`" every time we use something from the `math` module:

```
#| autorun: false
from math import *

x = cos(2 * pi)

print(x)
```

This pattern can be very convenient, but in large programs that include many modules it is often a good idea to keep the symbols from each module in their own namespaces, by using the `import math` pattern. This would eliminate potentially confusing problems.

### 14.1.1 Namespaces

#### Namespaces

A namespace is an identifier used to organize objects, e.g. the methods and variables of a module. The prefix `math.` we have used in the previous section is such a namespace. You may also create your own namespace for a module. This is done by using the `import math as mymath` pattern.

```
#| autorun: false
import math as m

x = m.sqrt(2)

print(x)
```

You may also only import specific functions of a module.

```
#| autorun: false
from math import sinh as mysinh
```

### 14.1.2 Directory of a module

Once a module is imported, we can list the symbols it provides using the `dir` function:

```
#| autorun: false
import math

print(dir(math))
```

And using the function `help` we can get a description of each function (almost .. not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

```
#| autorun: false

help(math.log)
```

```
#| slideshow: {slide_type: fragment}
#| tags: []
math.log(10)
```

```
#| slideshow: {slide_type: fragment}
math.log(8, 2)
```

We can also use the `help` function directly on modules: Try

```
help(math)
```

Some very useful modules from the Python standard library are `os`, `sys`, `math`, `shutil`, `re`, `subprocess`, `multiprocessing`, `threading`.

A complete lists of standard modules for Python 3 is available at <http://docs.python.org/3/library/>

### 14.1.3 Advanced topics

#### Create Your Own Modules

Creating your own modules in Python is a great way to organize your code and make it reusable. A module is simply a file containing Python definitions and statements. Here's how you can create and use your own module:

##### 14.1.3.1 Creating a Module

To create a module, you just need to save your Python code in a file with a `.py` extension. For example, let's create a module named `mymodule.py` with the following content:

```
mymodule.py

def greet(name: str) -> str:
 return f"Hello, {name}!"

def add(a: int, b: int) -> int:
 return a + b
```

### 14.1.3.2 Using Your Module

Once you have created your module, you can import it into other Python scripts using the `import` statement. Here's an example of how to use the `mymodule` we just created:

```
main.py

import mymodule

Use the functions from mymodule
print(mymodule.greet("Alice"))
print(mymodule.add(5, 3))
```

### 14.1.3.3 Importing Specific Functions

You can also import specific functions from a module using the `from ... import ...` syntax:

```
main.py

from mymodule import greet, add

Use the imported functions directly
print(greet("Bob"))
print(add(10, 7))
```

### 14.1.3.4 Module Search Path

When you import a module, Python searches for the module in the following locations:

1. The directory containing the input script (or the current directory if no script is specified).
2. The directories listed in the `PYTHONPATH` environment variable.
3. The default directory where Python is installed.

You can view the module search path by printing the `sys.path` variable:

```
import sys
print(sys.path)
```

### 14.1.3.5 Creating Packages

A package is a way of organizing related modules into a directory hierarchy. A package is simply a directory that contains a special file named `__init__.py`, which can be empty. Here's an example of how to create a package:



```
mypackage/
 __init__.py
 module1.py
 module2.py
```

You can then import modules from the package using the dot notation:

```
main.py

from mypackage import module1, module2

Use the functions from the modules
print(module1.some_function())
print(module2.another_function())
```

Creating and using modules and packages in Python helps you organize your code better and makes it easier to maintain and reuse.

## **i** Namespaces in Your Modules

### **14.1.3.6 Namespaces in Packages**

You can also create sub-packages by adding more directories with `__init__.py` files. This allows you to create a hierarchical structure for your modules:

```
mypackage/
 __init__.py
 subpackage/
 __init__.py
 submodule.py
```

You can then import submodules using the full package name:

```
main.py

from mypackage.subpackage import submodule

Use the functions from the submodule
print(submodule.some_sub_function())
```

## 15 Plotting

# 16 Plotting

Data visualization through plotting is a crucial tool for analyzing and interpreting scientific data and theoretical predictions. While plotting capabilities are not built into Python’s core, they are available through various external library modules. [Matplotlib](#) is widely recognized as the de facto standard for plotting in Python. However, several other powerful plotting libraries exist, including [PlotLy](#), [Seaborn](#), and [Bokeh](#), each offering unique features and capabilities for data visualization.

As Matplotlib is an external library (actually a collection of libraries), it must be imported into any script that uses it. While Matplotlib relies heavily on NumPy, importing NumPy separately is not always necessary for basic plotting. However, for most scientific applications, you’ll likely use both. To create 2D plots, you typically start by importing Matplotlib’s pyplot module:

```
import matplotlib.pyplot as plt
```

This import introduces the implicit interface of `pyplot` for creating figures and plots. Matplotlib offers two main interfaces:

- An **implicit** “pyplot” interface that maintains the state of the current Figure and Axes, automatically adding plot elements (Artists) as it interprets the user’s intentions.
- An **explicit** “Axes” interface, also known as the “object-oriented” interface, which allows users to methodically construct visualizations by calling methods on Figure or Axes objects to create other Artists.

We will use most of the the the `pyplot` interface as in the examples below. The section *Additional Plotting* will refer to the explicit programming of figures.

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

We can set some of the parameters for the appearance of graphs globally. In case you still want to modify a part of it, you can set individual parameters later during plotting. The command used here is the

```
plt.rcParams.update()
```

function, which takes a dictionary with the specific parameters as key.

```
plt.rcParams.update({'font.size': 12,
 'lines.linewidth': 1,
 'lines.markersize': 10,
 'axes.labelsize': 11,
 'xtick.labelsize' : 10,
 'ytick.labelsize' : 10,
 'xtick.top' : True,
 'xtick.direction' : 'in',
 'ytick.right' : True,
 'ytick.direction' : 'in',})
```

## 16.1 Simple Plotting - Implicit Version

Matplotlib offers multiple levels of functionality for creating plots. Throughout this section, we'll primarily focus on using commands that leverage default settings. This approach simplifies the process, as Matplotlib automatically handles much of the graph layout. These high-level commands are ideal for quickly creating effective visualizations without delving into intricate details. At the end of this section, we'll briefly touch upon more advanced techniques that provide greater control over plot elements and layout.

### 16.1.1 Line Plot

To create a basic line plot, use the following command:

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

By default, this generates a line plot. However, you can customize the appearance by adjusting various parameters within the `plot()` function. For instance, you can modify it to resemble a scatter plot by changing certain arguments. The versatility of this command allows for a range of visual representations beyond simple line plots.

Let's create a simple line plot of the sine function over the interval  $[0, 4]$ . We'll use NumPy to generate the x-values and calculate the corresponding y-values. The following code snippet demonstrates this process:

```
x = np.linspace(0, 4.*np.pi, 100) ①
y = np.sin(x) ②

plt.figure(figsize=(4,3)) ③
plt.plot(x, y) ④
plt.tight_layout() ⑤
plt.show() ⑥
```

- ① Create an array of 100 values between 0 and 4 .
- ② Calculate the sine of each value in the array.
- ③ create a new figure
- ④ automatically adjust the layout
- ⑤ show the figure

Here is the code in a Python cell:

```
x = np.linspace(0, 4.*np.pi, 100)
y = np.sin(x)

plt.figure(figsize=(4,3))
plt.plot(x, y)
plt.tight_layout()
plt.show()
```

Try to change the values of the `x` and `y` arrays and see how the plot changes.

#### 💡 Why use `plt.tight_layout()`

`plt.tight_layout()` is a very useful function in Matplotlib that automatically adjusts the spacing between plot elements to prevent overlapping and ensure that all elements fit within the figure area. Here's what it does:

1. **Padding Adjustment:** It adjusts the padding between and around subplots to prevent overlapping of axis labels, titles, and other elements.
2. **Subplot Spacing:** It optimizes the space between multiple subplots in a figure.
3. **Text Accommodation:** It ensures that all text elements (like titles, labels, and legends) fit within the figure without being cut off.
4. **Margin Adjustment:** It adjusts the margins around the entire figure to make sure everything fits neatly.
5. **Automatic Resizing:** If necessary, it can slightly resize subplot areas to accommodate all elements.

6. Legend Positioning: It takes into account the presence and position of legends when adjusting layouts.

Key benefits of using `plt.tight_layout()`:

- It saves time in manual adjustment of plot elements.
- It helps create more professional-looking and readable plots.
- It's particularly useful when creating figures with multiple subplots or when saving figures to files.

You typically call `plt.tight_layout()` just before `plt.show()` or `plt.savefig()`. For example:

```
plt.figure()
... (your plotting code here)
plt.tight_layout()
plt.show()
```

#### 16.1.1.1 Axis Labels

To enhance the clarity and interpretability of our plots, it's crucial to provide context through proper labeling. Let's add descriptive axis labels to our diagram, a practice that significantly improves the readability and comprehension of the data being presented.

```
plt.xlabel('x-label')
plt.ylabel('y-label')
```

```
plt.figure(figsize=(4,3))
plt.plot(x,np.sin(x)) # using x-axis
plt.xlabel('t') # set the x-axis label
plt.ylabel('sin(t)') # set the y-axis label
plt.tight_layout()
plt.show()
```

#### 16.1.1.2 Legends

```
plt.plot(..., label=r'$\sin(x)$')
plt.legend(loc='lower left')
```

```
plt.figure(figsize=(4,3))
plt.plot(x,np.sin(x),"ko",markersize=5,label=r"$\delta(t)$") #define an additional label
plt.xlabel('t')
plt.ylabel(r'$\sin(t)$')

plt.legend(loc='lower left') #add the legend
plt.tight_layout()
plt.show()
```

### 16.1.2 Scatter plot

If you prefer to use symbols for plotting just use the

```
plt.scatter(x,y)
```

command of pylab. Note that the scatter command requires a  $x$  and  $y$  values and you can set the marker symbol (see an overview of the [marker symbols](#)).

```
plt.figure(figsize=(4,3))
plt.scatter(x,np.cos(x),marker='o')
plt.xlabel('t') # set the x-axis label
plt.ylabel('y') # set the y-axis label
plt.tight_layout()
plt.show()
```

### 16.1.3 Histograms

A very useful plotting command is also the *hist* command. It generates a histogram of the data provided. If only the data is given, bins are calculated automatically. If you supply an array of intervals with `hist(data,bins=b)`, where `b` is an array, the `hist` command calculates the histogram for the supplied bins. `density=True` normalizes the area below the histogram to 1. The `hist` command not only returns the graph, but also the occurrences and bins.

#### **i** Physics Interlude- Probability density for finding an oscillating particle

Let's take this opportunity to integrate histogram plotting with a physics concept. We'll examine the simple harmonic oscillator in one dimension, a fundamental model in physics. As you may recall, this system is described by a specific equation of motion, which we'll explore in the context of data visualization using histograms.

$$\ddot{x}(t) = -\omega^2 x(t) \quad (16.1)$$

The solution of that equation of motion for an initial elongation  $\Delta x$  at  $t = 0$  is given by

$$x(t) = \Delta x \cos(\omega t) \quad (16.2)$$

If you now need to calculate the probability to find the spring at a certain elongation you need to calculate the time the oscillator spends at different positions. The time  $dt$  spend in the interval  $[x(t), x(t) + dx]$  depends on the speed, i.e.

$$v(t) = \frac{dx}{dt} = -\omega \Delta x \sin(\omega t) \quad (16.3)$$

The probability to find the oscillator at a certain intervall then is the fraction of time residing in this intervall normalized by the half the oscillation period  $T/2$ .

$$\frac{dt}{T/2} = \frac{1}{T/2} \frac{dx}{v(t)} = \frac{1}{T/2} \frac{-dx}{\omega \Delta x \sin(\omega t)} \quad (16.4)$$

As the frequency of the oscillator is  $\omega = 2\pi/T$  we can replace  $T$  by  $T = 2\pi/\omega$  which yields

$$p(x)dx = \frac{1}{\pi \Delta x} \frac{dx}{\sqrt{1 - \left(\frac{x(t)}{\Delta x}\right)^2}} \quad (16.5)$$

This is the probability density of finding an oscillating spring at a certain elongation  $x(t)$ . If you look at the example more closely, it tells you, that you find an elongation more likely when the speed of the mass is low. This is even a more general issue in non-equilibrium physics. If cells or cars are moving with variable speed, they are more likely to be found at places where they are slow.

This can be also addressed with the histogram function. If we use the solution of the equation of motion and evaluate the position at equidistant times, the histogram over the corresponding positions will tell us the probability of finding a certain position value if normalized properly.



```

#| ExecuteTime: {end_time: '2019-04-23T08:38:43.064728Z', start_time: '2019-04-23T08:38:42
#| slideshow: {slide_type: fragment}
t=np.linspace(0,np.pi,10000)
y=np.cos(t)

#ymin=np.mean(y)-2*np.std(y)
#ymax=np.mean(y)+2*np.std(y)
b=np.linspace(-1,1,10)

plot the histogram
n, bins, _ =plt.hist(y,bins=b,density=True,color='lightgreen',label='histogram')

#plot the analytical solution
x=np.linspace(-0.99,0.99,100)
plt.plot(x,1/(np.pi*np.sqrt(1-x**2)), 'k--',label='analytic')

plt.xlabel('y') # set the x-axis label
plt.ylabel('occurrence') # set the y-axis label
plt.legend()
plt.show()

```

### 16.1.4 Combined plots

You can combine multiple data with the same axes by stacking multiple plots.

```

#| ExecuteTime: {end_time: '2019-04-23T12:32:54.303226Z', start_time: '2019-04-23T12:32:53.7
#| slideshow: {slide_type: subslide}
create x and y arrays for theory
x = np.linspace(-10., 10., 100)
y = np.cos(x) * np.exp(-(x/4.47)**2)

create plot
plt.figure(figsize = (7,5))

plt.plot(x, y, 'b-', label='theory')
plt.scatter(x, y, label="data")

plt.axhline(color = 'gray', linewidth=5,zorder=-1)
plt.axvline(ls='--',color = 'gray', zorder=-1)

```

```
plt.xlabel('x')
plt.ylabel('transverse displacement')
plt.legend(loc='upper right')

plt.tight_layout()
plt.show()
```

## 16.2 Saving figures

To save a figure to a file we can use the `savefig` method in the `Figure` class. Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. For scientific papers, I recommend using PDF whenever possible. (LaTeX documents compiled with `pdflatex` can include PDFs using the `includegraphics` command). In some cases, PGF can also be good alternative.

```
#| slideshow: {slide_type: subslide}
theta = np.arange(0.01, 10., 0.04)
ytan = np.sin(2*theta)+np.cos(3.1*theta)

plt.figure(figsize=(8,6))
plt.plot(theta, ytan)
plt.xlabel(r'θ')
plt.ylabel('y')
plt.savefig('filename.pdf')
plt.show()
```

## 16.3 Plots with error bars

When plotting experimental data it is customary to include error bars that indicate graphically the degree of uncertainty that exists in the measurement of each data point. The `Matplotlib` function `errorbar` plots data with error bars attached. It can be used in a way that either replaces or augments the `plot` function. Both vertical and horizontal error bars can be displayed. The figure below illustrates the use of error bars.

```
#| ExecuteTime: {end_time: '2019-04-23T08:41:04.097326Z', start_time: '2019-04-23T08:41:03.6'}
#| slideshow: {slide_type: subslide}
import numpy as np
import matplotlib.pyplot as plt
```

```
create plot
plt.figure(1, figsize = (8,6))
xdata=np.arange(0.5,3.5,0.5)
ydata=210-40/xdata

yerror=2e3/ydata

plt.errorbar(xdata, ydata, fmt="ro", label="data",
 xerr=0.15, yerr=yerror, ecolor="black")

plt.xlabel("x")
plt.ylabel("transverse displacement")
plt.legend(loc="lower right")

plt.show()
```

### 16.3.1 Setting plotting limits and excluding data

If you want to zoom in to a specific region of a plot you can set the limits of the individual axes.

```
#| ExecuteTime: {end_time: '2019-04-23T08:41:06.146706Z', start_time: '2019-04-23T08:41:05.79...}
#| slideshow: {slide_type: subslide}
import numpy as np
import matplotlib.pyplot as plt
theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)
plt.figure(figsize=(8,6))
plt.plot(theta, ytan)
plt.xlim(0, 4)
plt.ylim(-8, 8) # restricts range of y axis from -8 to +8
plt.axhline(color="gray", zorder=-1)
plt.show()
```

#### 16.3.1.1 Masked arrays

Sometimes you encounter situations, when you wish to mask some of the data of your plot, because they are not showing real data as the vertical lines in the plot above. For this purpose, you can mask the data arrays in various ways to not show up. The example below uses the

```
np.ma.masked_where()
```

function of NumPy, which takes a condition as the first argument and what should be returned if that condition is fulfilled.

```
#| ExecuteTime: {end_time: '2017-05-13T22:58:57.167887+02:00', start_time: '2017-05-13T22:58:57.167887+02:00'}
#| slideshow: {slide_type: subslide}
import numpy as np
import matplotlib.pyplot as plt
theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)

#exclude specific data from plotting
ytanM = np.ma.masked_where(np.abs(ytan)>20., ytan)

plt.figure(figsize=(8,6))
plt.plot(theta, ytanM)
plt.ylim(-8, 8)
#plt.axhline(color="gray", zorder=-1)
plt.show()
```

If you look at the resulting array, you will find, that the entries have not been removed but replaced by --, so the values are not existent and therefore not plotted.

## 16.4 Logarithmic plots

Data sets can span many orders of magnitude from fractional quantities much smaller than unity to values much larger than unity. In such cases it is often useful to plot the data on logarithmic axes.

### 16.4.1 Semi-log plots

For data sets that vary exponentially in the independent variable, it is often useful to use one or more logarithmic axes. Radioactive decay of unstable nuclei, for example, exhibits an exponential decrease in the number of particles emitted from the nuclei as a function of time.

Matplotlib provides two functions for making semi-logarithmic plots, `semilogx` and `semilogy`, for creating plots with logarithmic x and y axes, with linear y and x axes, respectively. We illustrate their use in the program below, which made the above plots.

```

#| ExecuteTime: {end_time: '2019-04-23T12:42:51.135605Z', start_time: '2019-04-23T12:42:50.3
#| slideshow: {slide_type: subslide}
create theoretical curve
tau = 20.2 # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
N0 = 8200. # Initial count rate (per second)
t = np.linspace(0, 180, 30)
N = N0 * np.exp(-t/tau)

create plot
plt.figure(1, figsize = (6,5))

plt.semilogy(t, N, 'bo', label="theory")
plt.xlabel('time (days)')
plt.ylabel('counts per second')
plt.legend(loc='upper right')

plt.tight_layout()
plt.show()

```

### 16.4.2 Log-log plots

Matplotlib can also make log-log or double-logarithmic plots using the function `loglog`. It is useful when both the  $x$  and  $y$  data span many orders of magnitude. Data that are described by a power law  $y = Ax^b$ , where  $A$  and  $b$  are constants, appear as straight lines when plotted on a log-log plot. Again, the `loglog` function works just like the `plot` function but with logarithmic axes.

```

#| slideshow: {slide_type: subslide}
create theoretical fitting curve
tau = 20.2 # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
N0 = 8200. # Initial count rate (per second)
t = np.linspace(0.1, 180, 128)
N = N0 * t**(-3)
N1 = N0 * t**(-2)

create plot
plt.figure(1, figsize = (3,3),dpi=150)
plt.loglog(t, N, 'b-', label="theory")
plt.loglog(t, N1, 'r-', label="theory")
#plt.plot(time, counts, 'ro', label="data")
plt.xlabel('time (days)')

```

```
plt.ylabel('counts per second')
plt.legend(loc='upper right')
plt.show()
```

## 16.5 Arranging multiple plots

Often you want to create two or more graphs and place them next to one another, generally because they are related to each other in some way.

```
#| tags: []
theta = np.arange(0.01, 8., 0.04)
y = np.sqrt((8./theta)**2-1.)
ytan = np.tan(theta)
ytan = np.ma.masked_where(np.abs(ytan)>20., ytan)
ycot = 1./np.tan(theta)

ycot = np.ma.masked_where(np.abs(ycot)>20., ycot)
```

```
#| ExecuteTime: {end_time: '2019-04-23T12:39:11.304135Z', start_time: '2019-04-23T12:39:10.6'}
#| slideshow: {slide_type: subslide}
plt.figure(1,figsize=(8,8))
plt.subplot(2, 2, 2)

plt.plot(theta, ytan)
plt.ylim(-8, 8)
plt.xlabel("theta")
plt.ylabel("tan(theta)")

plt.subplot(2, 2, 3)
plt.plot(theta, -y)
plt.plot(theta, ycot)
plt.ylim(-8, 8)
plt.xlabel(r'θ')
plt.ylabel(r'$\cot(\theta)$')

plt.tight_layout()
plt.show()
```

## 16.6 Contour and Density Plots

A contour plots are useful tools to study two dimensional data, meaning  $Z(X, Y)$ . A contour plots isolines of the function  $Z$ .

### 16.6.1 Simple contour plot

**Note:** Physics Interlude

Interference of spherical waves

We want to use contour plots to explore the interference of two spherical waves. A spherical wave can be given by a complex spatial amplitude

$$U(r) = U_0 \frac{e^{-i k r}}{r} \quad (16.6)$$

where  $U_0$  is the amplitude,  $k$  the magnitude of the k-vector, i.e.  $k = 2\pi/\lambda$  and  $r = \sqrt{x^2 + y^2}$  the distance, here in 2 dimensions. Note that the total wavefunction also contains a temporal part such that

$$U(r, t) = U_0 \frac{e^{-i k r}}{r} e^{i \omega t} \quad (16.7)$$

We will however ignore the temporal factor, as we are interested in the spatial interference pattern with a time-averaged intensity.

To show interference, we just use two of those monochromatic waves located at  $\vec{r}_1$  and  $\vec{r}_2$ .

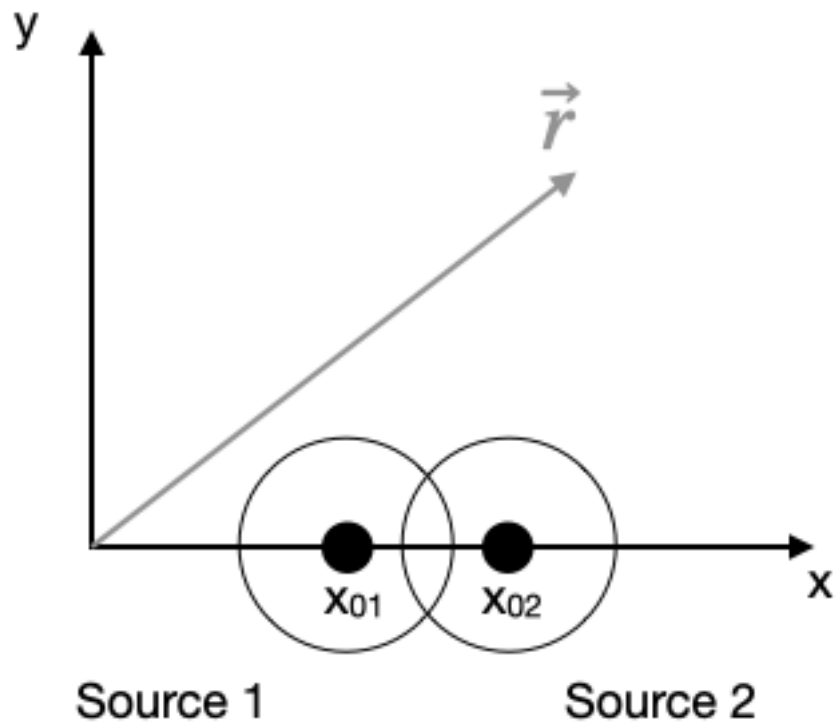


Figure 16.1: interference

To obtain the total amplitude we have to sum up the wavefunctions of the two sources.

$$U(\vec{r}) = U_{01} \frac{e^{-i k |\vec{r} - \vec{r}_1|}}{|\vec{r} - \vec{r}_1|} + U_{02} \frac{e^{-i k |\vec{r} - \vec{r}_2|}}{|\vec{r} - \vec{r}_2|} \quad (16.8)$$

The intensity of the wave at a position is then related to the magnitude square of the wavefunction

$$I(r) \propto |U(r)|^2 \quad (16.9)$$

To keep it simple we will skip the  $1/r$  amplitude decay.

```
#| ExecuteTime: {end_time: '2019-04-23T12:47:28.112164Z', start_time: '2019-04-23T12:47:28.112164Z'}
#| slideshow: {slide_type: subslide}
lmda = 2 # defines the wavelength
x01=1.5*np.pi # location of the first source, y01=0
```



```

x02=2.5*np.pi # location of the second source, y02=0
x = np.linspace(0, 4*np.pi, 100)
y = np.linspace(0, 4*np.pi, 100)

X, Y = np.meshgrid(x, y)
I= np.abs(np.exp(-1j*np.sqrt((X-x01)**2+Y**2)*2*np.pi/lmda)
 +np.exp(-1j*np.sqrt((X-x02)**2+Y**2)*2*np.pi/lmda))**2

```

```

#| slideshow: {slide_type: subslide}
plt.figure(1,figsize=(6,6))

#contour plot
plt.contour(X, Y, I, 20)
plt.colorbar()
plt.show()

```

### 16.6.2 Color contour plot

```

#| ExecuteTime: {end_time: '2019-04-23T12:49:50.378580Z', start_time: '2019-04-23T12:49:49.6
#| slideshow: {slide_type: fragment}
plt.figure(1,figsize=(7,6))
plt.contourf(X, Y, I, 10, cmap='gray')
plt.colorbar()
plt.show()

```

### 16.6.3 Image plot

```

#| ExecuteTime: {end_time: '2019-04-23T12:51:02.316610Z', start_time: '2019-04-23T12:51:01.5
#| slideshow: {slide_type: fragment}
plt.figure(1,figsize=(7,6))
plt.imshow(I,extent=[10,0,0,10],cmap='gray');
plt.ylim(10,0)
plt.colorbar()
plt.show()

```

## 16.7 Advanced Plotting - Explicit Version

While we have so far largely relied on the default setting and the automatic arrangement of plots, there is also a way to precisely design your plot. Python provides the tools of object oriented programming and thus modules provide classes which can be instanced into objects. This explicit interfaces allows you to control all details without the automatisms of `pyplot`.

The figure below, which is taken from the matplotlib documentation website shows the sets of commands and the objects in the figure, the commands refer to. It is a nice reference, when creating a figure.



```

fig, ax = plt.subplots()

fig.subplots_adjust(right=0.75)
fig.subplots_adjust(right=0.75)

twin1 = ax.twinx()
twin2 = ax.twinx()

Offset the right spine of twin2. The ticks and label have already been
placed on the right by twinx above.
twin2.spines.right.set_position(("axes", 1.2))

time=np.linspace(0,10,100)
omega=2
p1, = ax.plot(time,np.cos(omega*time), "C0", label="position")
p2, = twin1.plot(time,-omega*np.sin(omega*time), "C1", label="velocity")
p3, = twin2.plot(time, -omega**2*np.cos(omega*time), "C2", label="acceleration")

ax.set(xlim=(0, 10), ylim=(-2, 2), xlabel="time", ylabel="position")
twin1.set(ylim=(-4, 4), ylabel="velocity")
twin2.set(ylim=(-6, 6), ylabel="acceleration")

ax.yaxis.label.set_color(p1.get_color())
twin1.yaxis.label.set_color(p2.get_color())
twin2.yaxis.label.set_color(p3.get_color())

ax.tick_params(axis='y', colors=p1.get_color())
twin1.tick_params(axis='y', colors=p2.get_color())
twin2.tick_params(axis='y', colors=p3.get_color())

#ax.legend(handles=[p1, p2, p3])

plt.show()

```

### 16.7.2 Insets

Insets are plots within plots using their own axes. We therefore need to create two axes systems, if we want to have a main plot and an inset.

```

#| slideshow: {slide_type: skip}
x = np.linspace(0, 5, 10)
y = x ** 2

#| slideshow: {slide_type: skip}
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.9, 0.9]) # main axes
axes2 = fig.add_axes([0.35, 0.2, 0.4, 0.3]) # inset axes

main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

insert
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('insert title')
plt.show()

```

### 16.7.3 Spine axis

```

#| slideshow: {slide_type: skip}
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0)) # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);

```

### 16.7.4 Polar plot

```
#| slideshow: {slide_type: skip}
polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = np.linspace(0, 2 * np.pi, 100)
ax.plot(t, t, color='blue', lw=3);
```

### 16.7.5 Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

```
#| slideshow: {slide_type: skip}
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="C0")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="C1");
```

### 16.7.6 3D Plotting

Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization. Three-dimensional plots are enabled by importing the `mplot3d` toolkit, included with the main Matplotlib installation:

```
#| ExecuteTime: {end_time: '2019-04-23T12:51:20.097941Z', start_time: '2019-04-23T12:51:20.097941Z'}
#| slideshow: {slide_type: fragment}
from mpl_toolkits import mplot3d
```

Once this submodule is imported, a three-dimensional axes can be created by passing the keyword `projection='3d'` to any of the normal axes creation routines:

### 16.7.6.1 Projection Science

```
#| ExecuteTime: {end_time: '2019-04-23T12:52:00.234327Z', start_time: '2019-04-23T12:51:59.000000Z'}
#| slideshow: {slide_type: fragment}
fig=plt.figure(figsize=(6,6))
ax = plt.axes(projection='3d')
#ax.set_proj_type('ortho')
ax.set_proj_type('persp', focal_length=0.2)
ax.view_init(elev=40., azimuth=10)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```

With this three-dimensional axes enabled, we can now plot a variety of three-dimensional plot types. Three-dimensional plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically in the notebook; recall that to use interactive figures, you can use `%matplotlib notebook` rather than `%matplotlib inline` when running this code.

### 16.7.6.2 Line Plotting in 3D

from sets of  $(x, y, z)$  triples. In analogy with the more common two-dimensional plots discussed earlier, these can be created using the `ax.plot3D` and `ax.scatter3D` functions. The call signature for these is nearly identical to that of their two-dimensional counterparts, so you can refer to Simple Line Plots and Simple Scatter Plots for more information on controlling the output. Here we'll plot a trigonometric spiral, along with some points drawn randomly near the line:

```
plt.figure(figsize=(10,6))

#create the axes
ax = plt.axes(projection='3d')

Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'blue')

Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens')
plt.show()
```

Notice that by default, the scatter points have their transparency adjusted to give a sense of depth on the page. While the three-dimensional effect is sometimes difficult to see within a static image, an interactive view can lead to some nice intuition about the layout of the points. Use the `scatter3D` or the `plot3D` method to plot a random walk in 3-dimensions in your exercise.

### 16.7.6.3 Surface Plotting

A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized:

```
#| ExecuteTime: {end_time: '2018-06-17T13:30:18.509589Z', start_time: '2018-06-17T13:30:18.509589Z'}
#| slideshow: {slide_type: fragment}
x = np.linspace(-6, 6, 50)
y = np.linspace(-6, 6, 60)

X, Y = np.meshgrid(x, y)
Z=np.sin(X)*np.sin(Y)

#| ExecuteTime: {end_time: '2018-06-17T13:30:37.677530Z', start_time: '2018-06-17T13:30:37.677530Z'}
#| slideshow: {slide_type: fragment}
np.shape(Z)
```



```
#| ExecuteTime: {end_time: '2018-06-17T13:29:49.631101Z', start_time: '2018-06-17T13:29:49.631101Z'}
#| slideshow: {slide_type: subslide}
plt.figure(figsize=(10,6))
ax = plt.axes(projection='3d')
ax.view_init(30, 50)
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
 cmap='viridis', edgecolor='none')
ax.set_title('surface')
plt.show()
```

**Part V**

**Lecture 3**

## 17 Lecture 3

# 18 Dynamics and Simple Motion

- Lists and arrays (introduction to `numpy` for numerical operations).
- Basic vector operations using `numpy`.

## Numpy Array

The NumPy array, formally called `ndarray` in NumPy documentation, is the real workhorse of data structures for scientific and engineering applications. The NumPy array is similar to a list but where all the elements of the list are of the same type. The elements of a **NumPy array** are usually numbers, but can also be booleans, strings, or other objects. When the elements are numbers, they must all be of the same type. For example, they might be all integers or all floating point numbers. NumPy arrays are more efficient than Python lists for storing and manipulating data.

```
#| slideshow: {slide_type: fragment}
import numpy as np
```

## 18.1 Creating Numpy Arrays

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples using `np.array`
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files which will be covered in the files section of this course.

### 18.1.1 From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function. This is demonstrated in the following cells:

```
#this is a list
a = [0, 0, 1, 4, 7, 16, 31, 64, 127]

type(a)
```

list

```
b=np.array(a,dtype=float)
type(b)
```

numpy.ndarray

### 18.1.2 Using array-generating functions

For larger arrays it is impractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in **numpy** that generate arrays of different forms and shapes. Some of the more common are:

```
#| slideshow: {slide_type: fragment}
create a range

x = np.arange(0, 10, 1) # arguments: start, stop, step
x
```

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

```
#| slideshow: {slide_type: fragment}
x = np.arange(-5, -2, 0.1)
x
```

array([-5. , -4.9, -4.8, -4.7, -4.6, -4.5, -4.4, -4.3, -4.2, -4.1, -4. ,  
-3.9, -3.8, -3.7, -3.6, -3.5, -3.4, -3.3, -3.2, -3.1, -3. , -2.9,  
-2.8, -2.7, -2.6, -2.5, -2.4, -2.3, -2.2, -2.1])

#### **i** linspace

The **linspace** function creates an array of N evenly spaced points between a starting point and an ending point. The form of the function is `linspace(start, stop, N)`. If the

third argument N is omitted, then N=50. The function `linspace` always includes the end points.

```
#| slideshow: {slide_type: fragment}
using linspace, both end points ARE included
np.linspace(0,10,25)
```

```
array([0. , 0.41666667, 0.83333333, 1.25 , 1.66666667,
 2.08333333, 2.5 , 2.91666667, 3.33333333, 3.75 ,
 4.16666667, 4.58333333, 5. , 5.41666667, 5.83333333,
 6.25 , 6.66666667, 7.08333333, 7.5 , 7.91666667,
 8.33333333, 8.75 , 9.16666667, 9.58333333, 10.])
```

### **i** `logspace`

`logspace` is doing equivalent things with logarithmic spacing. The function `logspace` generates an array of N points between decades  $10^{\text{start}}$  and  $10^{\text{stop}}$ . The form of the function is `logspace(start, stop, N)`. If the third argument N is omitted, then N=50. The function `logspace` always includes the end points.

```
#| slideshow: {slide_type: fragment}
np.logspace(0, 10, 10, base=np.e)
```

```
array([1.00000000e+00, 3.03773178e+00, 9.22781435e+00, 2.80316249e+01,
 8.51525577e+01, 2.58670631e+02, 7.85771994e+02, 2.38696456e+03,
 7.25095809e+03, 2.20264658e+04])
```

Other types of array creation techniques are listed below. Try around with these commands to get a feeling what they do.

### **i** `mgrid`

`mgrid` generates a multi-dimensional matrix with increasing value entries, for example in columns and rows. The arguments are similar to `arange` and `linspace`.

```
x, y = np.mgrid[0:1:0.1, 0:5] # similar to meshgrid in MATLAB
```

x

```
array([[0. , 0. , 0. , 0. , 0.],
 [0.1, 0.1, 0.1, 0.1, 0.1],
 [0.2, 0.2, 0.2, 0.2, 0.2],
 [0.3, 0.3, 0.3, 0.3, 0.3],
 [0.4, 0.4, 0.4, 0.4, 0.4],
 [0.5, 0.5, 0.5, 0.5, 0.5],
 [0.6, 0.6, 0.6, 0.6, 0.6],
 [0.7, 0.7, 0.7, 0.7, 0.7],
 [0.8, 0.8, 0.8, 0.8, 0.8],
 [0.9, 0.9, 0.9, 0.9, 0.9]])
```

y

```
array([[0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.],
 [0., 1., 2., 3., 4.]])
```

```
np.array([[1,2,3],[4,5,6],[7,8,9]])
```

```
array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])
```

#### **i** diag

**diag** generates a diagonal matrix with the list supplied to it as the diagonal values. The values can be also offset from the main diagonal by using the optional argument **k**. If **k** is positive, the diagonal is above the main diagonal, if negative, below the main diagonal.

```
a diagonal matrix
np.diag([1,2,3])
```

```
array([[1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]])
```

```
diagonal with offset from the main diagonal
np.diag([1,2,3], k=-1)
```

```
array([[0, 0, 0, 0],
 [1, 0, 0, 0],
 [0, 2, 0, 0],
 [0, 0, 3, 0]])
```

#### **i** zeros and ones

**zeros** and **ones** creates a matrix with the dimensions given in the argument and filled with 0 or 1. The argument is a tuple with the dimensions of the matrix. For example, **np.zeros((3,3))** creates a 3x3 matrix filled with zeros.

```
np.zeros((3,3))
```

```
array([[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]])
```

```
np.ones((3,3))
```

```
array([[1., 1., 1.],
 [1., 1., 1.],
 [1., 1., 1.]])
```



## 18.2 Manipulating NumPy arrays

### 18.2.1 Slicing

Slicing is the name for extracting part of an array by the syntax `M[lower:upper:step]`. When any of these are unspecified, they default to the values `lower=0`, `upper=size of dimension`, `step=1`. We can also use negative indices to count from the end of the array. Here are some examples:

```
A = np.array([1,2,3,4,5])
A
```

```
array([1, 2, 3, 4, 5])
```

```
A[1:4]
```

```
array([2, 3, 4])
```

Any of the three parameters in `M[lower:upper:step]` can be omitted.

```
A[:] # lower, upper, step all take the default values
```

```
array([1, 2, 3, 4, 5])
```

```
A[:,2] # step is 2, lower and upper defaults to the beginning and end of the array
```

```
array([1, 3, 5])
```

Negative indices counts from the end of the array (positive index from the beginning) and can be used in any of the three slicing parameters. Here are some examples:

```
A = np.array([1,2,3,4,5])
```

```
A[-1] # the last element in the array
```

```
np.int64(5)
```

```
A[2:] # the last three elements
```

```
array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays. We can slice along each axis independently. Here are some examples:

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
```

```
array([[0, 1, 2, 3, 4],
 [10, 11, 12, 13, 14],
 [20, 21, 22, 23, 24],
 [30, 31, 32, 33, 34],
 [40, 41, 42, 43, 44]])
```

```
a block from the original array
A[1:3, 1:4]
```

```
array([[11, 12, 13],
 [21, 22, 23]])
```

### **i** Differences

**Slicing** can be effectively used to calculate differences for example for the calculation of derivatives. Here the position  $y_i$  of an object has been measured at times  $t_i$  and stored in an array each. We wish to calculate the average velocity at the times  $t_i$  from the arrays by the formula

$$v_i = \frac{y_i - y_{i-1}}{t_i - t_{i-1}} \quad (18.1)$$

```
y = np.array([0. , 1.3, 5. , 10.9, 18.9, 28.7, 40.])
t = np.array([0. , 0.49, 1. , 1.5 , 2.08, 2.55, 3.2])
```

```
v = (y[1:]-y[:-1])/(t[1:]-t[:-1])
v
```

```
array([2.65306122, 7.25490196, 11.8 , 13.79310345, 20.85106383,
 17.38461538])
```

### 18.2.2 Reshaping

Arrays can be reshaped into any form, which contains the same number of elements. For example, a 4-element array can be reshaped into a 2x2 array, or a 2x2 array can be reshaped into a 4-element array. Here are some examples:

```
a=np.zeros(4)
a
```

```
array([0., 0., 0., 0.])
```

```
np.reshape(a,(2,2))
```

```
array([[0., 0.],
 [0., 0.]])
```

### 18.2.3 Adding a new dimension: newaxis

With `newaxis`, we can insert new dimensions in an array, for example converting a vector to a column or row matrix. Here are some examples:

```
#| slideshow: {slide_type: fragment}
v = np.array([1,2,3])
v
```

```
array([1, 2, 3])
```

```
#| slideshow: {slide_type: fragment}
v.shape
```

```
(3,)
```

```
#| slideshow: {slide_type: fragment}
make a column matrix of the vector v
v[:, np.newaxis]
```

```
array([[1],
 [2],
 [3]])
```

```
#| slideshow: {slide_type: fragment}
column matrix
v[:,np.newaxis].shape
```

(3, 1)

```
#| slideshow: {slide_type: fragment}
row matrix
v[np.newaxis,:].shape
```

(1, 3)

## 18.2.4 Stacking and repeating arrays

Using function `repeat`, `tile`, `vstack`, `hstack`, and `concatenate` we can create larger vectors and matrices from smaller ones by repeating or stacking. Please try the individual functions yourself in your notebook. We won't discuss them in detail here.

### 18.2.4.1 Tile and repeat

```
#| slideshow: {slide_type: skip}
a = np.array([[1, 2], [3, 4]])
a
```

```
array([[1, 2],
 [3, 4]])
```

```
#| slideshow: {slide_type: skip}
repeat each element 3 times
np.repeat(a, 3)
```

```
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

```
#| slideshow: {slide_type: skip}
tile the matrix 3 times
np.tile(a, 3)
```

```
array([[1, 2, 1, 2, 1, 2],
 [3, 4, 3, 4, 3, 4]])
```

#### 18.2.4.2 Concatenate

Concatenate joins arrays along an existing axis. Here are some examples:

```
#| slideshow: {slide_type: skip}
a = np.array([[1, 2], [3, 4]])
a
```

```
array([[1, 2],
 [3, 4]])
```

```
#| slideshow: {slide_type: skip}
b = np.array([[5, 6]])
```

```
#| slideshow: {slide_type: skip}
np.concatenate((a, b), axis=0)
```

```
array([[1, 2],
 [3, 4],
 [5, 6]])
```

```
#| slideshow: {slide_type: skip}
np.concatenate((a, b.T), axis=1)
```

```
array([[1, 2, 5],
 [3, 4, 6]])
```

#### 18.2.4.3 Hstack and vstack

hstack and vstack stack arrays horizontally and vertically. Here are some examples:

```
#| slideshow: {slide_type: skip}
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
```

```
#| slideshow: {slide_type: skip}
np.vstack((a,b))
```

```
array([[1, 2, 3],
 [2, 3, 4]])
```

```
#| slideshow: {slide_type: skip}
np.hstack((a,b.T))
```

```
array([1, 2, 3, 2, 3, 4])
```

## 18.3 Applying mathematical functions

All kinds of mathematical operations can be carried out on arrays. Typically these operation act element wise as seen from the examples below where `a` is an array of numbers from 0 to 9.

### 18.3.1 Operation involving one array

```
#| slideshow: {slide_type: fragment}
a=np.arange(0, 10, 1.5)
a
```

```
array([0. , 1.5, 3. , 4.5, 6. , 7.5, 9.])
```

```
#| slideshow: {slide_type: fragment}
a/2
```

```
array([0. , 0.75, 1.5 , 2.25, 3. , 3.75, 4.5])
```

```
#| slideshow: {slide_type: fragment}
a**2
```

```
array([0. , 2.25, 9. , 20.25, 36. , 56.25, 81.])
```

```
#| slideshow: {slide_type: fragment}
np.sin(a)
```

```
array([0. , 0.99749499, 0.14112001, -0.97753012, -0.2794155 ,
 0.93799998, 0.41211849])
```

```
#| slideshow: {slide_type: fragment}
np.exp(-a)
```

```
array([1.00000000e+00, 2.23130160e-01, 4.97870684e-02, 1.11089965e-02,
 2.47875218e-03, 5.53084370e-04, 1.23409804e-04])
```

```
#| slideshow: {slide_type: fragment}
(a+2)/3
```

```
array([0.66666667, 1.16666667, 1.66666667, 2.16666667, 2.66666667,
 3.16666667, 3.66666667])
```

### 18.3.2 Operations involving multiple arrays

Operation between multiple vectors allow in particular very quick operations. The operations address then elements of the same index. These operations are called vector operations since they concern the whole array at the same time. The product between two vectors results therefore not in a dot product, which gives one number but in an array of multiplied elements.

```
#| ExecuteTime: {end_time: '2017-04-20T21:16:34.978152+02:00', start_time: '2017-04-20T21:
#| slideshow: {slide_type: fragment}
a = np.array([34., -12, 5.,1.2])
b = np.array([68., 5.0, 20.,40.])
```

```
#| ExecuteTime: {end_time: '2017-04-20T21:16:41.810170+02:00', start_time: '2017-04-20T21:
#| slideshow: {slide_type: fragment}
a + b
```

```
array([102. , -7. , 25. , 41.2])
```

```
#| ExecuteTime: {end_time: '2017-04-20T21:16:48.002366+02:00', start_time: '2017-04-20T21:
#| slideshow: {slide_type: fragment}
#| tags: []
2*b
```

```
array([136., 10., 40., 80.])
```

xxw

```
#| ExecuteTime: {end_time: '2017-04-20T21:20:19.373091+02:00', start_time: '2017-04-20T21:20:19.373091+02:00'}
#| slideshow: {slide_type: fragment}
a*np.exp(-b)
```

```
array([9.98743918e-29, -8.08553640e-02, 1.03057681e-08, 5.09802511e-18])
```

```
#| tags: []
v1=np.array([1,2,3])
v2=np.array([4,2,3])
```

## 18.4 Application

- Simulating and plotting the trajectory of a projectile under the influence of gravity (2D motion).
- Introduction to vector addition and resolving vectors into components.
- Visualization: Plotting the path of the projectile and velocity vectors.
- Homework: Simulate projectile motion with air resistance (optional for advanced students).



**Part VI**

**Seminar Contents**

# 19 Seminar 1

Calculate the average of all of the integers from 1 to 10.

```
#| exercise: ex_1_py
n = range(1, 11)

```

use the `sum()` and `len()` functions

```
n = range(1, 11)
```

*Solution.*

```
n = range(1, 11)
sum(n)/len(n)
```

①  
②

- ① create the range of numbers from 1 to 10.
- ② calculate the sum of the numbers and divide by the number of elements in the list.

```
#| exercise: ex_1_py
#| check: true
n = range(1, 11)
mean = sum(n)/len(n)
feedback = None

if (result == mean):
 feedback = { "correct": True, "message": "Nice work!" }
else:
 feedback = { "correct": False, "message": "That's incorrect, sorry." }
feedback
```