ECMWF — University of Reading — Deutscher Wetterdienst, Wetter und Klima aus einer Hand, DWD

Python and AI/ML for Weather, Climate and Environmental Applications

Let us enjoy 🚀

playing 🤖 with

Python 🐍 and AI/ML!

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	**Lab:** Python Startup: Basics	**Lab:** Feed-forward and Graph NNs	**Lab:** Graph Learning with PyTorch	**Lab:** Containers and Reproducibility	**Lab:** Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	**Lab:** Work environments, Python everywhere	**Lab:** Simple Transformer and LLM Use	**Lab:** Agent Frameworks	**Lab:** CI/CD Pipelines	**Lab:** Graph-based Assimilation
12:45–13:30	**Lunch Break**				
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	**Lab:** GRIB, NetCDF and Obs Visualisation	**Lab:** RAG Pipeline	**Lab:** DAWID Exploration	**Lab:** Anemoi Training Pipeline	**Lab:** Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	**The AI Transformation**	Learning from Observations Only
16:15–17:00	**Lab:** Torch Tensors and First Neural Net	**Lab:** Radar, SAT and Multimodal Data	**Lab:** MLflow Hands-on	**Lab: How work style could change**	**Lab:** ORIGEN and Open Discussion
17:00–20:00	Joint Dinner				

Lecture 2: Jupyter Notebooks, APIs and Servers

Goal of this Lecture

- Work productively with Jupyter Notebooks
- Understand Notebooks as part of a scientific workflow
- Prepare the ground for reproducible ML experiments

Focus

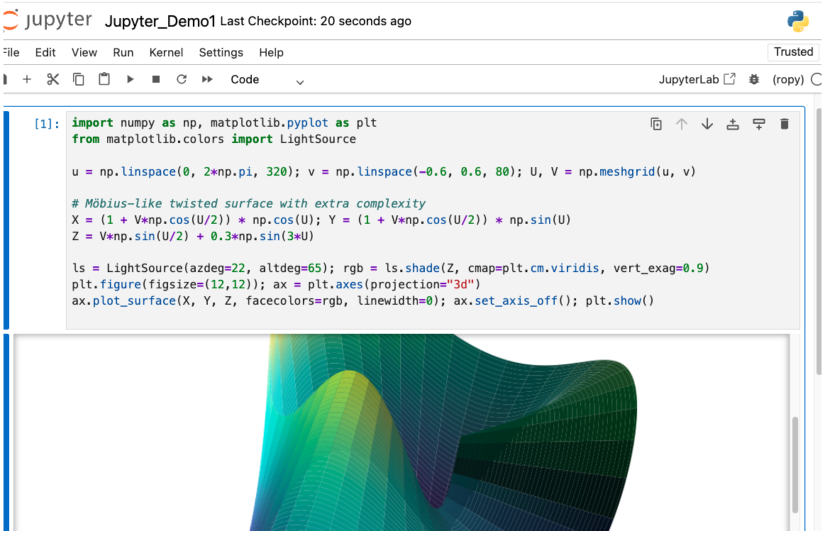
- Not UI details, but how things fit together
- From exploration to engineering

Topics Overview

- Jupyter Notebooks: Kernel, Server, Browser
- Environments and package management
- Markdown, magic commands, shell integration
- Data visualization as quality control
- APIs as a structuring principle
- Local, library and web APIs
- Native code integration (Fortran / C++)

Jupyter Notebook

- Cells with comments
- Cells with code
- Cells with output


jupyter
Jupyter_Demo1 Last Checkpoint: 20 seconds ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
(ropy)
[1]:

```
import numpy as np, matplotlib.pyplot as plt
from matplotlib.colors import LightSource

u = np.linspace(0, 2*np.pi, 320); v = np.linspace(-0.6, 0.6, 80); U, V = np.meshgrid(u, v)

# Möbius-like twisted surface with extra complexity
X = (1 + V*np.cos(U/2)) * np.cos(U); Y = (1 + V*np.cos(U/2)) * np.sin(U)
Z = V*np.sin(U/2) + 0.3*np.sin(3*U)

ls = LightSource(azdeg=22, altdeg=65); rgb = ls.shade(Z, cmap=plt.cm.viridis, vert_exag=0.9)
plt.figure(figsize=(12,12)); ax = plt.axes(projection="3d")
ax.plot_surface(X, Y, Z, facecolors=rgb, linewidth=0); ax.set_axis_off(); plt.show()
```

Why Jupyter Notebooks in Science and ML?

Why Notebooks Matter

- Rapid exploration of data and ideas
- Immediate feedback via plots and diagnostics
- Combine code, results and explanation

In Research Contexts

- Hypothesis testing and prototyping
- Understanding data and model behavior
- Bridging theory and implementation

Why They Scale Beyond Prototyping

- Documentation of decisions and assumptions
- Reproducible experiments (when done right)
- Natural interface to libraries and APIs
- Gateway to larger workflows and services

Key Message

- Notebooks are tools, not products
- Value comes from disciplined usage

Jupyter Architecture: Browser – Server – Kernel

Three Core Components

- **Browser** User interface: notebooks, plots, interaction
- **Jupyter Server** Manages files, sessions, security
- **Kernel** Executes Python code, holds state

Key Idea

- UI and computation are decoupled

Why This Matters

- Code runs in the kernel, not in the browser
- Kernel state persists across cells
- Server and kernel may run remotely
- Multiple notebooks can share one kernel

Strength and Risk

- Powerful interactive workflow
- Risk of hidden state and irreproducibility

Reproducibility in Jupyter Notebooks

Why Reproducibility Matters

- Results must be repeatable
- Experiments must be explainable
- Others (and you later) must trust them

Typical Problems

- Hidden kernel state
- Cells executed out of order
- Undocumented dependencies

Four Simple Rules

- Restart kernel and Run All
- Clear, explicit import and parameter cells
- Save outputs (plots, files, artefacts)
- Document dependencies and environment

Key Message

- A notebook is an executable document

Installing Packages in Jupyter Notebooks

Recommended Workflow

- Install packages either outside in the right virtual environment or directly in the notebook
- Use `pip install` or `!pip install`
- Packages are installed into the running kernel

Why This Works

- Jupyter uses the kernel's Python environment
- `pip` is typically bound to this Python

Good Practice: Check Once

Check active environment

```python
import sys
print(f"Python executable: {
    sys.executable}")
```

Key Message

- Use `pip` where you run your code
- Verify the environment if something looks wrong

Markdown and Narrative Computing

Why Markdown Matters

- Makes notebooks readable
- Explains intent, not just results
- Turns experiments into documents

Narrative Computing

- Code, text and results in one place
- Reasoning becomes explicit
- Supports review and reuse

Minimum You Should Use

- Headings (`#`, `##`)
- Bullet lists (`-`)
- Inline formulas (`$x^2$`)
- Short explanations

Markdown Cells

- Change cell type: `Esc` $\rightarrow$ `M`

Key Message

- A notebook should read like a lab notebook

Magic Commands and Shell Access

Why Magic Commands

- Speed up interactive work
- Reduce boilerplate code
- Support exploration and debugging

Line vs Cell Magics

- % for single-line commands
- %% for whole cells

The Few You Really Need

- `%timeit` (runtime)
- `%%writefile` (save code)
- `%%bash` (run shell)
- `!pip` (install packages)
- `!ls` (files)

Key Message

- Use magic and shell commands sparingly

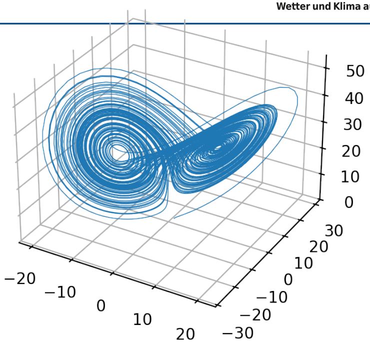
Visualization as Quality Control

Why Visualization Matters

- See patterns and anomalies
- Detect bugs early
- Build intuition about data and models

Typical Questions

- Does this look reasonable?
- Are scales and units correct?
- Are outliers expected?



Static vs Interactive

- Static: documentation, papers
- Interactive: exploration, debugging

Key Message

- Plot early, plot often!

Matplotlib

- Low-level, explicit control
- Publication-ready figures
- Default for scientific Python

Matplotlib (static)

```python
plt.plot(x, y)
plt.xlabel("x");
plt.ylabel("y")
plt.savefig("plot.png")
```

When to use

- Final figures
- Full control over layout

Seaborn

- Built on Matplotlib
- Statistical defaults
- Fast insight into distributions

Seaborn (statistical)

```python
sns.kdeplot(x=data)
```

Plotly

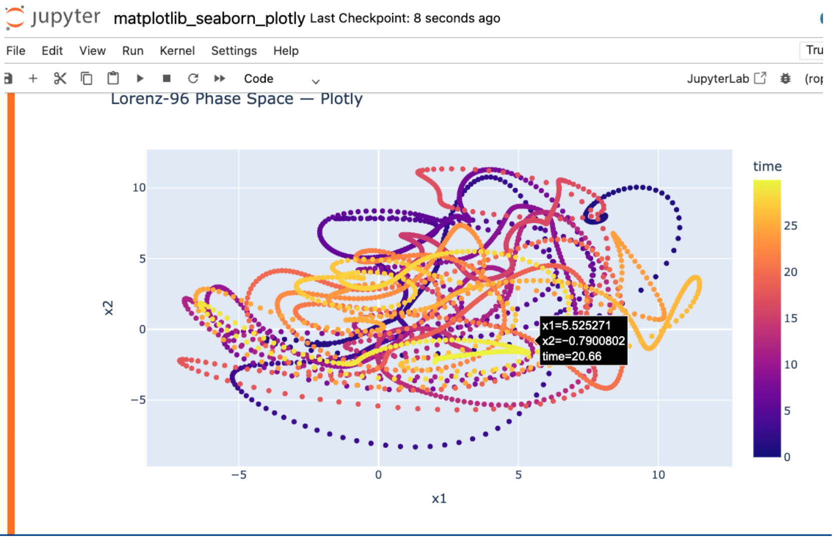
- Interactive plots
- Zoom, hover, selection

Plotly (interactive)

```python
fig = px.scatter(df, x="x",
    y="y"); fig.show()
```

Interactive Plot via Plotly

- Explore your data
- Learn, check!
-



What Is an API — and What Is Not

What an API Is

- A defined interface to call functionality
- Clear inputs, outputs and behavior
- Stable contract between components

Typical API Examples

- Python functions and classes
- Library interfaces (e.g. NumPy)
- Web endpoints (HTTP/REST)

What an API Is *Not*

- Just installable code
- A script without defined entry points
- A collection of loosely coupled functions

Important Distinction

- Installation is distribution
- An API is how you call the code

Key Message

- Packages may expose APIs
- Installation alone does not define one

APIs as a Scaling Principle

Core Idea

- APIs defines a contract on how functionality is accessed
- Clear separation of interface and implementation
- Same principle from notebooks to services

Why This Matters

- Logic becomes reusable and testable
- Notebooks stay thin and readable
- Systems can grow without rewrites

Example 1: Local API (project code)

Function API usage

```python
from model import forecast
y = forecast(x, params)
```

Example 2: Web API (service)

HTTP request

```python
response = requests.get(
  "/weather",
  params={"city": "Berlin", "
    date": "2025-01-20"})
data = response.json()
```

Key Insight

- APIs shift complexity behind a stable boundary

REST Essentials — Tasks as Resources / Agent Framework Elements

Resources

- A task is a resource
- Identified by a URL
- Has a state and metadata

- `/tasks`
- `/tasks/<id>`

HTTP Methods

- `POST` — create a task
- `GET` — inspect task or list
- `PUT` — change task state
- `DELETE` — remove a task

Status Codes (Minimum)

- 200 OK
- 201 Created
- 400 Bad Request
- 404 Not Found

Task Server Setup — Minimal Flask Example

Goal

- Tasks as REST resources
- Explicit task state
- Minimal server logic

States

- `created`
- `checked`
- `executing`
- `completed / failed`

task_server.py

```python
from flask import Flask, request, jsonify
app = Flask(__name__)
tasks = {}; i = 1

@app.post("/tasks")
def create():
    global i
    if not request.json: abort(400)
    t = {"id": i, "state": "created", "data
    ": request.json}
    tasks[i] = t; i += 1
    return jsonify(t), 201

app.run()
```

Testing the Task REST API

What We Test

- Send JSON to the server (`POST /tasks`)
- Server creates a new task
- Server assigns ID and initial state

Expected Result

- HTTP status 201 `Created`
- JSON response with:
  - task id
  - state = `created`
  - echoed input data

Test via `curl`

```
curl -X POST http://127.0.0.1:5000/tasks -H "Content-Type: application/json" -d '{"type":"demo","params":{"x":1}}'
```

Minimal Python Task Creation

```python
import requests
r = requests.post(
  "http://127.0.0.1:5000/tasks",
  json={"type":"demo","params":{"x":1}})
print(r.status_code); print(r.json())
```

```
{'id': 1, 'state': 'created',
 'data': {'type': 'demo',
          'params': {'x': 1}}}
```

Task API — Server Endpoints

Purpose

- Expose tasks via REST
- Read-only inspection
- Server owns all state

Server Resource Endpoints

- `/tasks` — task collection
- `/tasks/<id>` — single task

Semantics

- Stateless client
- Explicit URLs
- JSON responses

task_endpoints.py

```python
from flask import Flask, jsonify, abort
app = Flask(__name__)
tasks = {}

@app.get("/tasks")
def list_tasks():
    return jsonify(list(tasks.values()))

@app.get("/tasks/<int:i>")
def get_task(i):
    return jsonify(tasks[i]) if i in tasks else abort(404)

app.run()
```

Querying Tasks — Inspecting Server State

Goal

- Inspect existing tasks
- Read state and metadata
- No client-side state

REST Principle

- Tasks are resources
- Identified by URL
- Read via `GET`

Endpoints

- `GET /tasks`
- `GET /tasks/<id>`

List all Tasks

```python
import requests
r = requests.get("http://127.0.0.1:5000/tasks")
print(r.status_code)
print(r.json())
```

Describe one task

```python
import sys, requests
tid = int(sys.argv[1])
r = requests.get(f"http://127.0.0.1:5000/tasks/{tid}")
print(r.status_code)
print(r.json())
```

Native Code Integration: Fortran and C++ in Python & ML

Why Native Code Matters

- Decades of validated Fortran in NWP
- High-performance kernels in C++
- Tight control over memory and execution
- Reuse of trusted implementations

Typical Use Cases

- Physical parameterizations
- Linear operators, solvers, kernels
- Observation operators
- Legacy model components

Python as the Orchestration Layer

- Python controls the workflow
- Native code provides compute kernels
- Clean separation via APIs

Integration Options (Overview)

- `ctypes` – explicit C-compatible interfaces
- `f2py` – automatic Fortran bindings
- `pybind11` – modern C++ bindings
- Shared libraries: `.so` / `.dylib`

Fortran with C Bindings: A Stable Interface

Why C Bindings?

- Fortran and Python do not talk directly
- The common denominator is the C ABI
- Stable, explicit, language-independent

Key Concept

- Fortran exposes functions as C-compatible symbols
- No name mangling
- Well-defined data types

Minimal Fortran Example

```fortran
function f_sin_cos(x) result(f) bind(C)
  use iso_c_binding
  real(c_double), intent(in) :: x
  real(c_double) :: f
  f = sin(x) * cos(x)
end function
```

What This Ensures

- Symbol name is predictable
- Argument layout follows C rules
- Callable from Python, C, C++

Calling Fortran from Python with `ctypes`

Role of Python

- Python loads the shared library
- Defines the function signature
- Manages data exchange

Important Detail

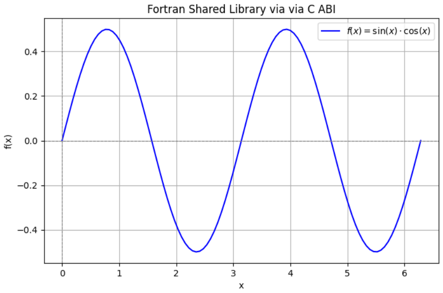
- Fortran arguments are passed by reference
- Python must pass pointers

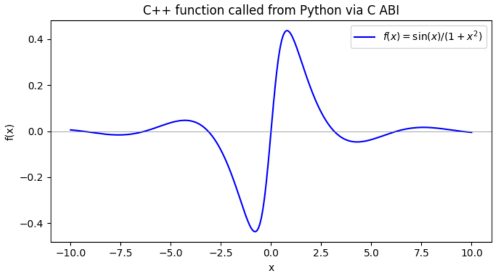
Minimal Python Interface

```python
import ctypes

lib = ctypes.CDLL("./fortran_interface.so")
lib.f_sin_cos.argtypes = [
    ctypes.POINTER(ctypes.c_double) ]
lib.f_sin_cos.restype = ctypes.c_double

x = ctypes.c_double(1.0)
y = lib.f_sin_cos(ctypes.byref(x))
```

- Native Fortran computation
- Controlled Python interface
- No performance-critical Python loop

Native Code Integration — Fortran vs. C++ from Python





Fortran via `ISO_C_BINDING` + `ctypes`
Numerics compiled, called from Python

C++ via C-compatible ABI + `ctypes`
Explicit interface, manual symbol control

- Python orchestrates, native code computes
- Identical workflow: compile → load → call → plot

Exposing a C++ Function via the C ABI

Role of C++

- Implements numerical logic
- Compiled into a shared library
- Exposes a stable C ABI

Key Requirement

- Use `extern "C"`
- Avoid C++ name mangling

Minimal C++ Interface

```cpp
#include <cmath>

extern "C" double f_cpp(double x)
{
    return std::sin(x) / (1.0 + x*x);
}
```

- Plain C-compatible symbol
- No templates, no classes
- ABI-safe function signature

Calling C++ from Python with `ctypes`

Role of Python

- Python loads the shared library
- Defines the binary interface
- Controls execution and visualization

Important Detail

- C++ function uses C ABI
- Arguments passed by value

Minimal Python Interface

```python
import ctypes

lib = ctypes.CDLL("./cpp_interface.so")
lib.f_cpp.argtypes = [ctypes.c_double]
lib.f_cpp.restype  = ctypes.c_double

x = 1.0
y = lib.f_cpp(x)
```

- Native C++ computation
- Explicit ABI contract
- Minimal Python overhead

Remote Jupyter — Principle

Core Idea

- Compute: remote (server, HPC, cloud)
- Visualization: local in the browser
- Browser setup remains unchanged

Typical Scenarios

- HPC login or compute nodes
- Cloud virtual machines
- Office workstation / bastion host

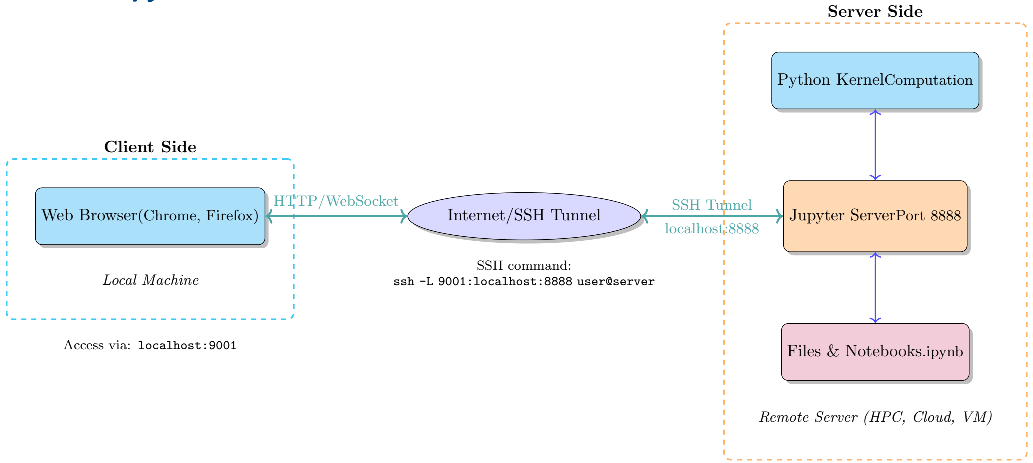
Security Principle

- **No** open notebook ports to the network
- Never expose :8888 on a public IP
- Access exclusively via SSH tunnels

Key Takeaway

- Jupyter only listens locally
- SSH provides secure transport

Remote Jupyter — Network Architecture



SSH command:
`ssh -L 9001:localhost:8888 user@server`

Access via: `localhost:9001`

Remote Jupyter — Port Forwarding Recipe

Step 1: Start Jupyter Remotely

Remote (Linux)

```
jupyter notebook \
  --no-browser \
  --port=8888
```

- Access token appears in the terminal
- Port is local only on the remote server

Step 2: Create SSH Tunnel

Local (Bash / PowerShell)

```
ssh -N -L 9001:localhost:8888 \
    user@remote-host
```

Step 3: Open in Browser

- `http://localhost:9001`
- Use token from the remote log
- Adjust port if needed

```
To access the server, open this file in a browser:
  file:///hpc/uhome/rpotthas/.local/share/jupyter/runtime/jpserver-1662241-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/tree?token=369b469c83e8eaa369ee02ea443d994865ca93e4185bb385
  http://127.0.0.1:8888/tree?token=369b469c83e8eaa369ee02ea443d994865ca93e4185bb385
```

Remote Jupyter — Firewalls & Troubleshooting

Typical Problems

- Local port already in use
- Remote port 8888 blocked by firewall
- Unstable SSH connection
- Multi-hop access (Jump / Bastion host)

Important Note

- A blocked 8888 is not an error
- SSH tunnels do not require open inbound ports

Best Practices

- Remote:
  - `-ip=127.0.0.1`
  - no public binding
- Local:
  - choose a free port (9001, 9002, . . .)
- Infrastructure:
  - Bastion / jump host via `-J`

Recommendation

- Never expose Jupyter notebook ports publicly