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

Let us enjoy

playing with

Python and AI/ML!

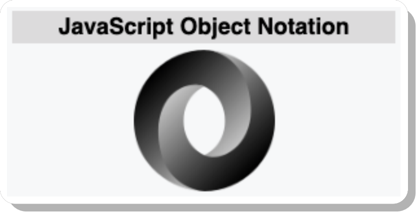
Five-Day Schedule Overview

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 1: Python Setup and Basics

Goal of Lecture

- Setup Python
- Synchronize with your knowledge of programming
- Become aware of the importance of proper package management
- Numpy and Matplotlib
- Get simple plots working
- write and import functions
- Flow Control
- File management
- Dictionaries
- Json Data Handling
- Classes and Dynamics



Python Setup: First Check and Create Virtual Environment

Goal of this step

- Verify that Python is installed
- Work across Linux, macOS, Windows, WSL
- Create isolated environments: aipy
- No dependency conflicts

Basic Installation

```
python --version.    # check version
python3 --version.   # check version
python3.12 --version
cd                   # go to home
python -m venv aipy  # create venv
alias aipy="source ~/aipy/bin/activate"
aipy                 # activate aipy
python               # interactive python
```

Linux/macOS example

Python Package Stack: How Things Build on Each Other

Core idea

- Python is a layered ecosystem
- Packages build on top of each other
- Higher layers assume lower layers exist
- This structure matters for:
  - installation
  - debugging
  - performance

Conceptual Package Stack (ASCII)

```
Python
 |
 +-- NumPy        (arrays, numerics)
 |
 +-- Matplotlib   (plots, figures)
 |
 +-- AI / ML
       |
       +-- PyTorch
       +-- scikit-learn
```

Conceptual view, not an exact dependency graph

Installing Core Scientific Packages

What we need first

- NumPy for numerical arrays
- Matplotlib for visualization
- Foundation for most scientific libraries
- Same tools across weather, climate, AI

Install basic packages

```
# activate virtual environment
aipy

# install core scientific packages
pip install numpy
pip install matplotlib

# verify installation
python -c "import numpy, matplotlib; print
    ('OK')"
```

These packages will be reused throughout the course

Installing Python Packages with `pip`

Why package management matters

- Python itself is minimal
- Most functionality comes from packages
- Packages define your working environment
- Reproducibility depends on exact versions

Key idea:

- One project $\Rightarrow$ one environment

Basic pip workflow

```
# activate aipy, list installed
aipy
pip list

# install core packages
pip install numpy
pip install matplotlib

# verify installation
python -c "import numpy, matplotlib;
    print('ok')"
```

Always install packages inside an activated virtual environment

NumPy Arrays: 1D, 2D, and 3D Data - Standard Programming Skills

Core idea

- One data structure for science
- Same logic for 1D, 2D, 3D data
- Used for time series, maps, fields
- Basis for ML tensors
- Learn the basics yourself, its easy.

NumPy array dimensions

```python
import numpy as np

x = np.array([1, 2, 4])            # 1D
A = np.array([[1, 2], [3, 4]])     # 2D
B = np.zeros((10, 50, 100))        # 3D

print(x.shape, A.shape, B.shape)
# prints (3,) (2, 2) (10, 50, 100)

print(x)
# prints [1 2 4]
```

Dimensions encode structure, not meaning

Vectors, Matrices, and Broadcasting

Core concepts

- Vectors (1D)
- Matrices (2D)
- Matrix–vector product
- Matrix–matrix product
- Broadcasting

Carry out MATLAB like Linear Algebra

Loops only Top Level!

Linear algebra in NumPy

```python
import numpy as np

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

b = A @ x          # matrix-vector
B = A @ A          # matrix-matrix
C = A + 1.0        # broadcasting

print(f"b={b},\nB={B},\nC={C}")
```

Operations follow array shapes

Indexing, Slicing, and Masks - Think Efficiency!

Why this matters

- Access elements efficiently
- Select subdomains in space or time
- Apply conditions to data
- Basis of filtering and diagnostics

Make sure you use an **efficient coding** approach!

Selecting array data

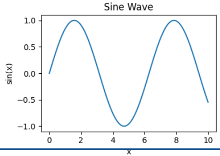
```
import numpy as np

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

B = A[0, 1]              # single element
C = A[:, 0]              # first column
D = A>2                  # boolean mask
E = A[A > 2]             # select elements

print("A=", A, "\nB=", B, "\nC=", C, "\
    nD=", D, "\nE=", E)
```

Masks are vectorized and fast

Visualization Engines: A Sine Wave Example

What happens here

- Generate numerical data with NumPy
- Compute a math function
- Visualize data using Matplotlib
- Save plots for later use



plot-sine-wave.py

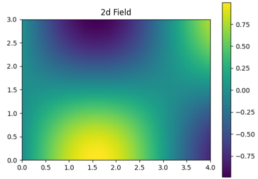
```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.figure(figsize=(4,3))
plt.plot(x, y)
plt.xlabel("x"); plt.ylabel("sin(x)")
plt.title('Sine Wave'); plt.tight_layout()
plt.savefig("plot-sine-wave.png")
plt.close()
```

A minimal but complete scientific plotting workflow

Visualizing 2D Data with Matplotlib

Key idea

- 2D arrays represent spatial fields
- Colors encode values
- Typical for weather and climate data



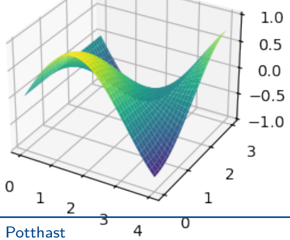
Simple 2D field

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

x = np.linspace(0, 4, 120)
y = np.linspace(0, 3, 90)
X, Y = np.meshgrid(x, y)
Z = np.sin(X) * np.cos(Y)

plt.imshow(Z, origin="lower",
      extent=[0,4,0,3], cmap="viridis")
plt.colorbar(); plt.title("2d Field");
plt.savefig("plot-2d-field.png");
plt.close()
```

3D Data: Fields and Surfaces

Why 3D matters

- Many geophysical fields are spatial
- Height, depth, or phase space
- Visualization helps intuition



plot-3d.py

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

x = np.linspace(0, 4, 80)
y = np.linspace(0, 3, 60)
X, Y = np.meshgrid(x, y)
Z = np.sin(X) * np.cos(Y)

fig = plt.figure(figsize=(4,3))
ax = fig.add_subplot(projection="3d")
ax.plot_surface(X, Y, Z, cmap="viridis")
plt.savefig("plot-3d.png"); plt.close()
```

Example of a smooth 3D field defined on a 2D grid

Functions: Turn Ideas into Reusable Tools

Why functions matter

- Turn a workflow step into a reusable tool
- Make intent explicit: inputs → output
- Easier testing, debugging, and collaboration
- Foundation for pipelines and ML training loops

Mini tool: normalize a signal

```python
import numpy as np

def normalize(x):
    x = np.asarray(x)
    return (x - x.mean()) / x.std()

x = np.array([2., 3., 5., 9.])
print("x =", x)
print("z =", normalize(x))
```

A pattern you will reuse everywhere: prepare data in one function

Control Flow: Quality Checks and Simple Automation

Control flow in practice

- Decision making: reject bad inputs early
- Works on full arrays, makes it fast
- Clearly essential for robust scripts and experiments

Filter missing values

```python
import numpy as np
x = np.array([1.2, 2.0, 1e30, 4.9])
thr = 1e20
# removing values outside of range:
x2 = x[x < thr]; dn = len(x)-len(x2)

fmt = "{:10.3g}"    # width=10, 3 digits
print("".join(fmt.format(v) for v in x))
print("".join(fmt.format(v) for v in x2))
print("mean=", f"{x2.mean():.3f}")
```

Quality control on observations or model fields

Dictionaries and JSON: Configs You Can Share

Why this is exciting

- Store model and experiment settings cleanly
- Pass configs through APIs and workflows
- **Reproducibility**: one file describes the run

Config for an experiment

```python
import json

cfg = {"dt": 0.5, "n": 20, "model": "toy"}
s = json.dumps(cfg, indent=2)
print(s)

cfg2 = json.loads(s)
print("dt =", cfg2["dt"])
```

This is the bridge to automation + deployment later

Loops in Python: Time Stepping and Accumulation

Typical use case

- Discrete time stepping
- Running sums and averages
- Diagnostics over trajectories
- Core pattern in models

Mental model

- Each loop = one time step
- State evolves sequentially

time_loop.py

```python
import numpy as np

dt = 0.1; x = 0.0; traj = [] # initialization

for n in range(20):
    x = x + dt * np.sin(x)
    traj.append(x)

traj = np.array(traj)
print("final x =", x)
print("mean x  =", traj.mean())
```

Sequential updates cannot be vectorized away

Files: Log Results, help yourself, Scientist!

File I/O patterns

- Save key results after every run
- Keep a simple experiment log
- The `with` pattern avoids subtle bugs

Write a tiny log file

```python
from datetime import datetime

myerr = 0.23 # example
msg = f"{datetime.now()}  rmse={myerr}\n"
with open("log-run.log", "a") as f:
    f.write(msg) # a is for append

with open("log-run.log") as f:
    last_line = f.readlines()[-1]
    print(last_line.strip())
```

Minimal logging already gives you insight and transparency

Classes I: Encapsulate State (A Tiny Model Component)

Why classes appear everywhere

- Components have a state (e.g., temperature)
- Methods update that state consistently
- Same structure in NWP, ESMs, and ML modules

A relaxating temperature

```python
class RelaxTemp:
    def __init__(self, T):
        self.T = T
    def step(self, dt, target):
        self.T += 0.1*dt*(target - self.T)

atm = RelaxTemp(288.0)
atm.step(1.0, 290.0)
print("T =", atm.T)
```

A tiny but real modeling pattern: relaxation toward forcing

Classes II: Coupling Two Components (Mini Earth-System Pattern)

The exciting part

- Two components, each with its own state
- A controller coordinates information exchange
- This is the conceptual core of coupled models

Coupling in 10 lines

```
class Box:
        def __init__(s,x): s.x=x
        def step(s,dt,t): s.x+=0.2*dt*(t-s.x)

atm, ocn = Box(288.), Box(290.)
ocn.step(1., atm.x); atm.step(1., ocn.x)
print("atm=", atm.x, "ocn=", ocn.x)
```

Same coupling idea, later: many variables + grids + physics

Lecture 1 — Key Takeaways

Technical Foundations

- Python as a portable workhorse for science
- Virtual environments avoid dependency conflicts
- `pip` and `requirements.txt` ensure reproducibility
- NumPy arrays as the core data structure
- Vectorization replaces explicit loops
- Matplotlib for fast diagnostic visualization

Conceptual Lessons

- Think in terms of arrays, not scalars
- Data selection via slicing and masking
- Broadcasting enables compact math expressions
- Visualization supports scientific intuition
- Clean code beats clever code
- Python skills will transfer directly to AI/ML development