TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

# A hands-on introduction to Scientific Python


## Markus Wallerberger

### Department of solid state physics, TU Wien
### Co-organised by the Fachschaft Doktorat



FEEL FREE TO SHARE AND REMIX THESE SLIDES UNDER THE
CREATIVE COMMONS-ATTRIBUTION-SHAREALIKE 3.0 LICENSE

| | Tuesday | Thursday |
|---|---|---|
| 14.00 – ~16.00 | Session 1: **Python Basics** | Session 3: ~~**The SciPy ecosystem**~~ Advanced NumPy+Plots |
| ~16.30 – ~18.30 | Session 2: **The NumPy array** | Session 4: **SciPy + ODE + Interfacing** |

# Unleashing numpy arrays

# Reminder: numpy arrays

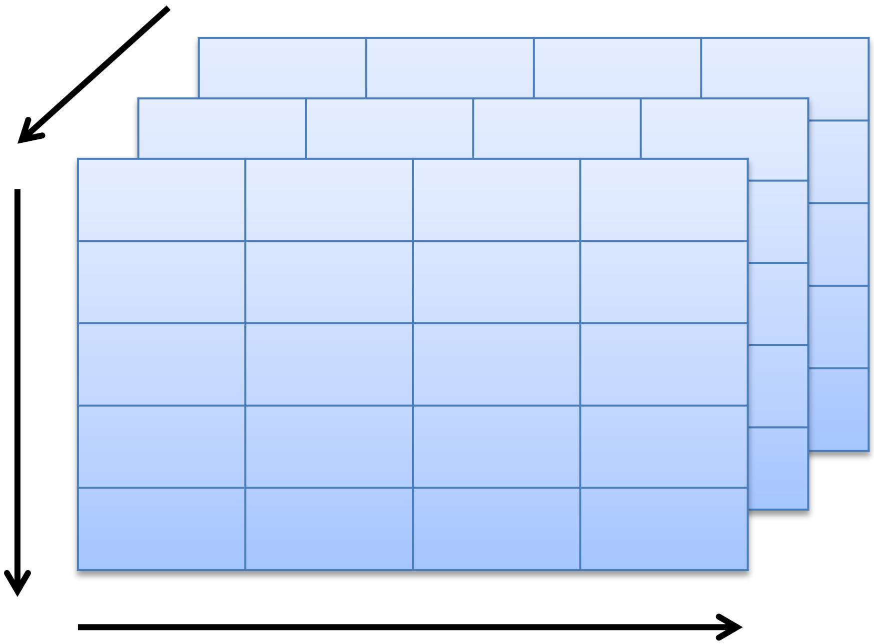
- n-dimensional arrays of one data type

```
import numpy as np
x = np.array([1, 2, 3, 9, 6])
```

- manipulation (like Matlab)
  - broadcasting
  - slicing
  - reduction

# Multiple dimensions

- Numpy array can have *n* dimensions (also n = 0)
- Last dimension varying fastest (row-major)
- Dimensions are consistent
- example:
  3 x 5 x 4 array



# Creating 3 x 3 matrices

- Zeros/Ones
```
O = np.zeros((3, 3))
TEN = 10 * np.ones((3, 3))
```
- Unit matrix
```
I = 2 * np.eye(3)
```
- Diagonal matrix
```
v = np.arange(3)
X = np.diag(v)
```
- Outer product
```
w = np.ones(3)
Y = np.outer(v, w)
```

# Shape

- Shape (tuple of dimensions) of the matrix
```
X = np.eye(6)
X.shape        # => (6, 6)
```
- Number of dimensions („axes“)
```
X.ndim         # => 3
```
- Transposing
```
X.T            # get the transpose (reverse order)
X.transpose(old1, old2, ...)       # transpose
```

# Reshaping

- return new matrix (never works inline!)
- with a changed „layout“ (shape)
- without changing overall number of elements

```
Y = X.reshape(2, 3, 6)
Y = X.reshape(2, 3, 1, 3, 2)
```

- elements are filled
  - row-major (order=‚C‘)
  - column-major (order=‚F‘ - Fortran)

# Reshaping (example)

- ```
  x = np.arange(1, 10)
  C = x.reshape(3, 3, order='C')
  F = x.reshape(3, 3, order='F')
  ```

- x=

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

C=

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

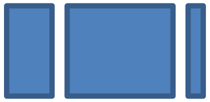
F=

| | | |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |
  ```

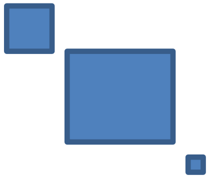
# Joining matrices

- Note the double brackets!
- Vertically (join rows/$0^{th}$ axis)
`np.vstack(( A1, A2, ... ))`

- Horizontally (join columns/$1^{st}$ axis)
`np.hstack(( A1, A2, ... ))`



- In a block-diagonal fashion
`np.dstack(( A1, A2, ... ))`



- Along any axis
`np.concatenate((A1, ...), axis=n)`
- `A.repeat(axis=n)`

# Operation vectorisation

- Same way as for 1D arrays (if dimensions argree!)

```
X = 3 * np.eye(4)
Y = np.ones(4)
X + Y
```

- Element-wise product

```
X * Y
```

- Matrix-matrix product

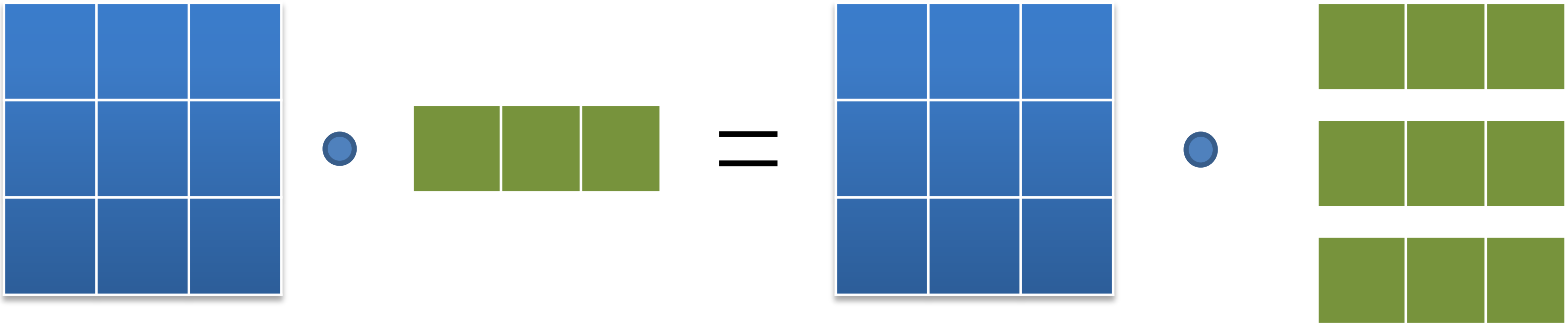
```
X.dot(Y)
```

# Broadcasting

- Mixing arrays of different shapes
- Matrix-times-vector

```
X=3*np.eye(3); Y=np.array([1,2,3])
X * Y == ????
```

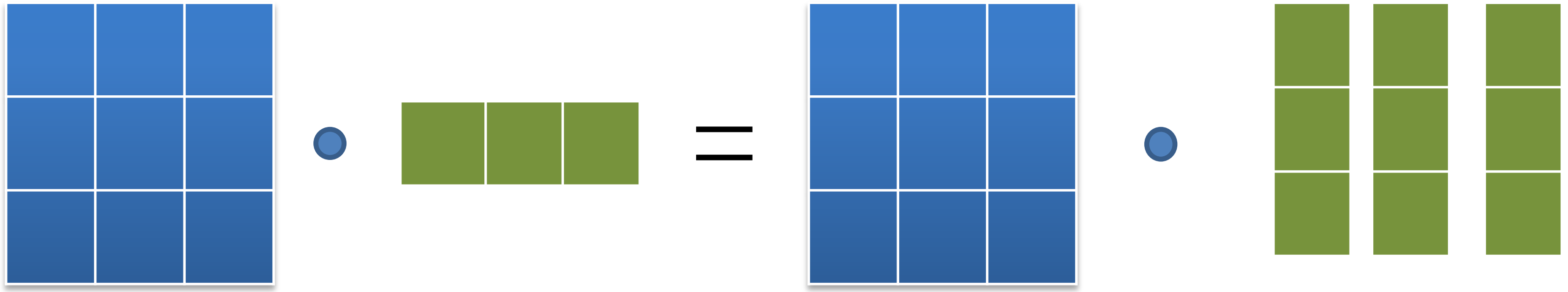
- Vector is **repeated** (= broadcast)



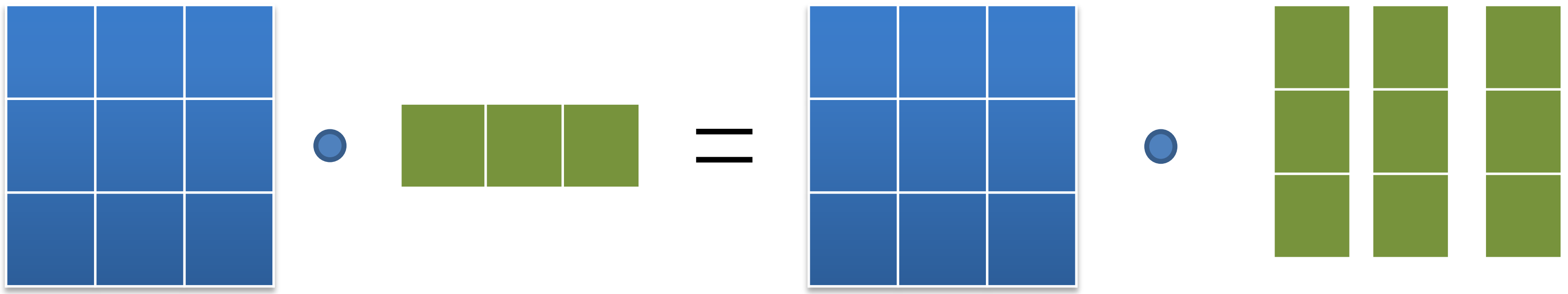
# Broadcasting rules

- add axes of size 1 to the front so that the number of axes matches
- for every axis:
  - either the size agrees
  - or one array has a size 1, in which case it is repeated to match the size of the other array
- X.shape == (4, 5) -> (1, 4, 5)
  Y.shape == (4, 4, 1)
  X*Y -> broadcast first dim of X and last dim of Y
- now clear why row-wise operation!

# How do we do this?



# How do we do this?

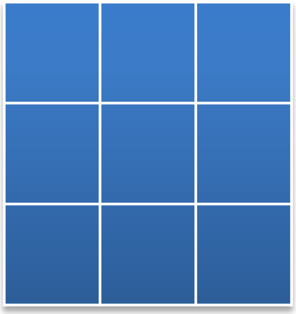


- v = v.reshape(3, 1)
  A * v

- or:
  v = np.atleast_2d(v)
  A * v.T

# Slicing

- for every axis seperately:
  A[1, 1]
  A[:2, 2:]



- get whole row:
  A[1]
- skip axis: „:“ is short for everything
  A[:, 1]
- skip all previous axes:
  A[..., 1, 2]

# Indexed slicing

- Use np.arrays as indices for a dimension

```
A[:, np.array([2, 0])]
```

- Get only specific elements

```
A[np.array([1, 0]),
  np.array([0, 3])]
```
→ A[1,0] and A[0,3]

- e.g., set diagonal to one (cf. `diag_indices`):

```
X = np.arange(10)
A[X, X] = 1
```

# Indexed reshaping

- np.newaxis inserts a new axis at the position with size one

```
A.shape  #3, 3
A[:, np.newaxis].shape  #3, 1, 3
```

- Broadcasting

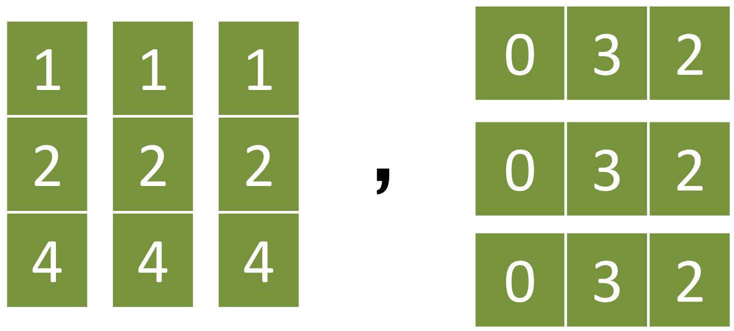
```
A * v[:, np.newaxis]
```

# Index-broadcasted slicing

*“The Hitch-Hiker's Guide to NumPy also mentions indexing.  It says that the best indexing method in existence is the index-broadcasted slicing, the effect of which is like having your brains smashed out with a slice of lemon wrapped round a large gold brick.”*

*Adapted from: Douglas Adams, The Hitchhiker's Guide to the Galaxy*

# Index-broadcasted slicing

- *indices* for different dimensions are also broadcasted to a common shape
- ```
  cols = np.array([1,2,4])[:,np.newaxis]
  rows = np.array([0,3,2])[np.newaxis,:]
  mat[rows, cols]
  ```
- mat [ 

]
  ```

# Index-broadcasted slicing

- say you want a A = n x n x n x n array
- $A_{iiii} = U$; $A_{ijij} = V$; $A_{iijj} = J$ (i != j)

```
def get_A(n, U=0., V=0., J=0.):
    A = np.zeros((n, n, n, n))
    i = np.arange(n)[:, np.newaxis]
    j = np.arange(n)[np.newaxis, :]
    A[i, j, i, j] = V
    A[i, i, j, j] = J
    A[i, i, i, i] = U  # override V
    return A
```

# Reduction

- reductions by default over the whole array
- many support axis parameter
```
A.sum([axis = ....])
A.prod([axis = .... ])
A.cumsum([axis = ... ])
```
- combined tensor/inner product
```
np.tensordot(A, B, axes)
```
- Einstein index summation
```
np.einsum(subscripts, A, B, ... )
```

# Data types

- Each array has a data type (default np.double)

```
np.array([...], dtype=np.double)
```

- Convert, e.g., to integer or logical for a mask

```
np.asarray(A, dtype=np.int)
np.asarray(A, dtype=np.bool)
```

- Careful when modifying inline!

```
A = np.zeros(3, int)
A += 0.1
```

# Record types

- Allow you to store heterogenuous arrays (like MATLAB „record arrays“)
```
X = np.zeros(5,
    dtype=[("id", np.int),
           ("marks", np.double, 3)])
```
- dtype = list (!) of tuples (name, type, shape)
- Use:
```
X["id"]
X["marks"][:, 1]
```

# Other cool numpy methods

- `.sort(...)`
  `.argsort(...)`
- `.amin(...)`
  `.argmin(...)`
- `.all(...)`
  `.any(...)`
- `np.unique(...)`
- `np.where(C, T, F)`
- Documentation! Google your problem!

- x = np.arange(9)
- x.shape
- x.dtype
- y = x.reshape(2, 5)
- y = x.reshape(3, 3)
- y.shape
- x.shape
- x * y
- y + np.ones(3)
- x[...,None] + np.ones(4)
- z = 3*np.eye(3)
- y * z
- y.dot(z)
- r = np.zeros(3, [("a", int),
  ("b", double, 3)])
- r["b"]

play around a bit and read the docs

permute rows of a
4 x 4 –matrix filled with 1 ... 16

scale the *rows* of the previous
matrix by the factors 1, 2, 9, 7

write a matrix-matrix multiplication
using broadcasting (without *dot*)

permute rows and columns of a
4 x 4 –matrix filled with 1 ... 16

create the Levi-Civita symbol in four
dimensions; use it to calculate a
volume spanned by four 4-vectors

create a random 10x10 matrix A
with 0s and 1s. Find the row/column
permutation which
block-diagonalises A.

# Plotting

# MatPlotLib

- MatLab-inspired plotting library
- Importing

```
import matplotlib.pyplot as pl
```

# Interactive mode

- Scripted mode:

```
pl. [...]
pl.show()    # clears the plot
```

- Interactive mode:

```
pl.interactive(True)
pl. [...]
```

- IPython notebook inline mode

```
%matplotlib inline
```

# Line plots

- `x = np.linspace(0, np.pi, 100)`
- `pl.plot(x, np.sin(x))`
- `pl.plot(x, np.cos(x))`
- `[pl.show()]`

# Line plots

- `x = np.linspace(0, np.pi, 100)`
- First dimension is
```
pl.plot(x, np.transpose(np.sin(x),
                        np.cos(x)))
```
- Errorbars: `pl.errorbar(...)`
- Plot style:
```
pl.plot(x, y, '+-')
```

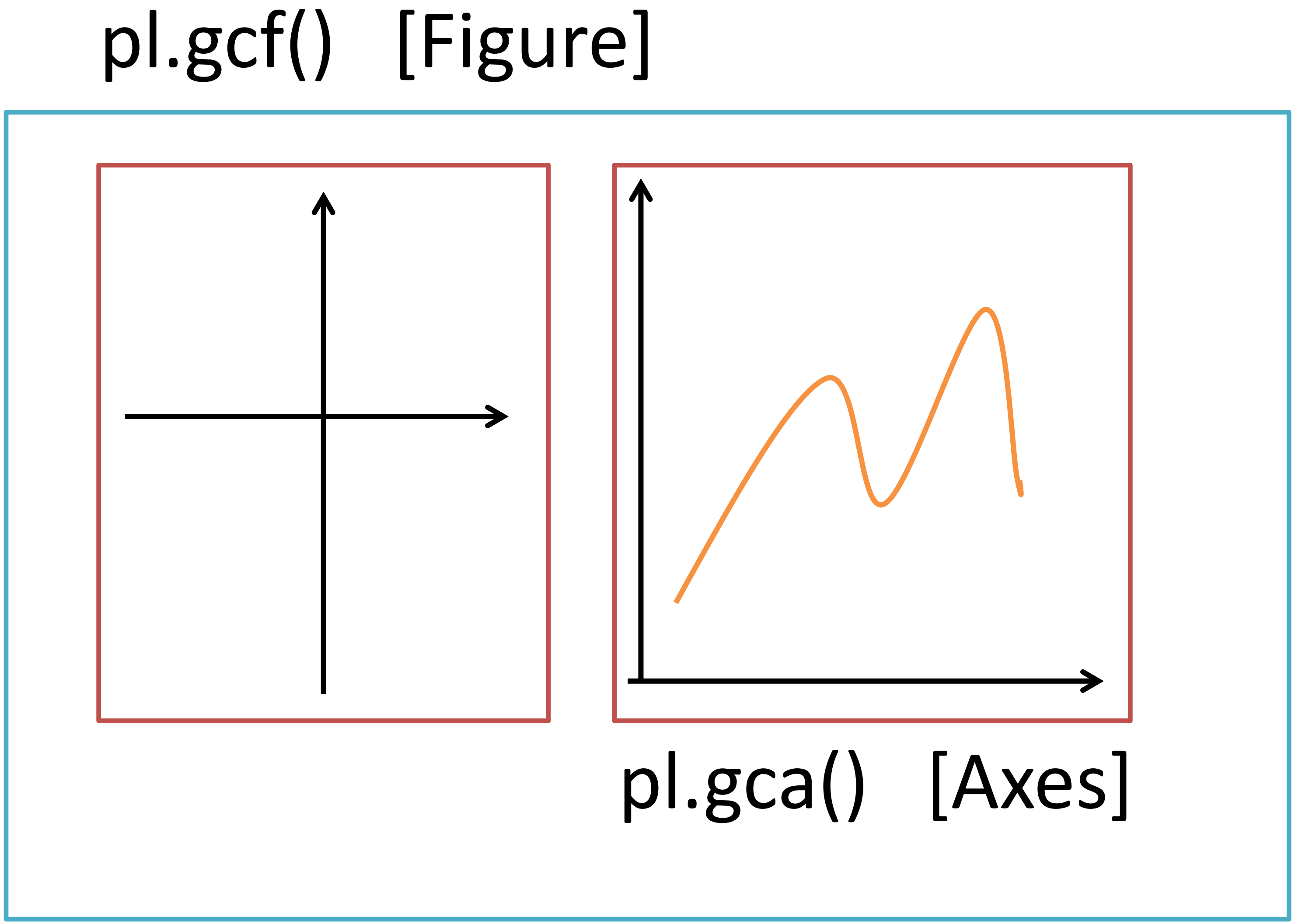
# Annotating

- `pl.plot(..., label="...")`
- ```
  pl.xlabel()
  pl.ylabel()
  pl.title()
  ```
- Latex powered:
  ```
  pl.xlabel(r"$i\omega_n$")
  ```
  ```

# Setting parts of the plot

- `pl.grid()`
- `pl.xlim()`
  `pl.xscale()`
  `pl.xticks()`
  `pl.ylim()`
- `pl.semilogy()`
- After plotting:
  `pl.gca().set_yscale('log')`

# Figures/Axes/Objects

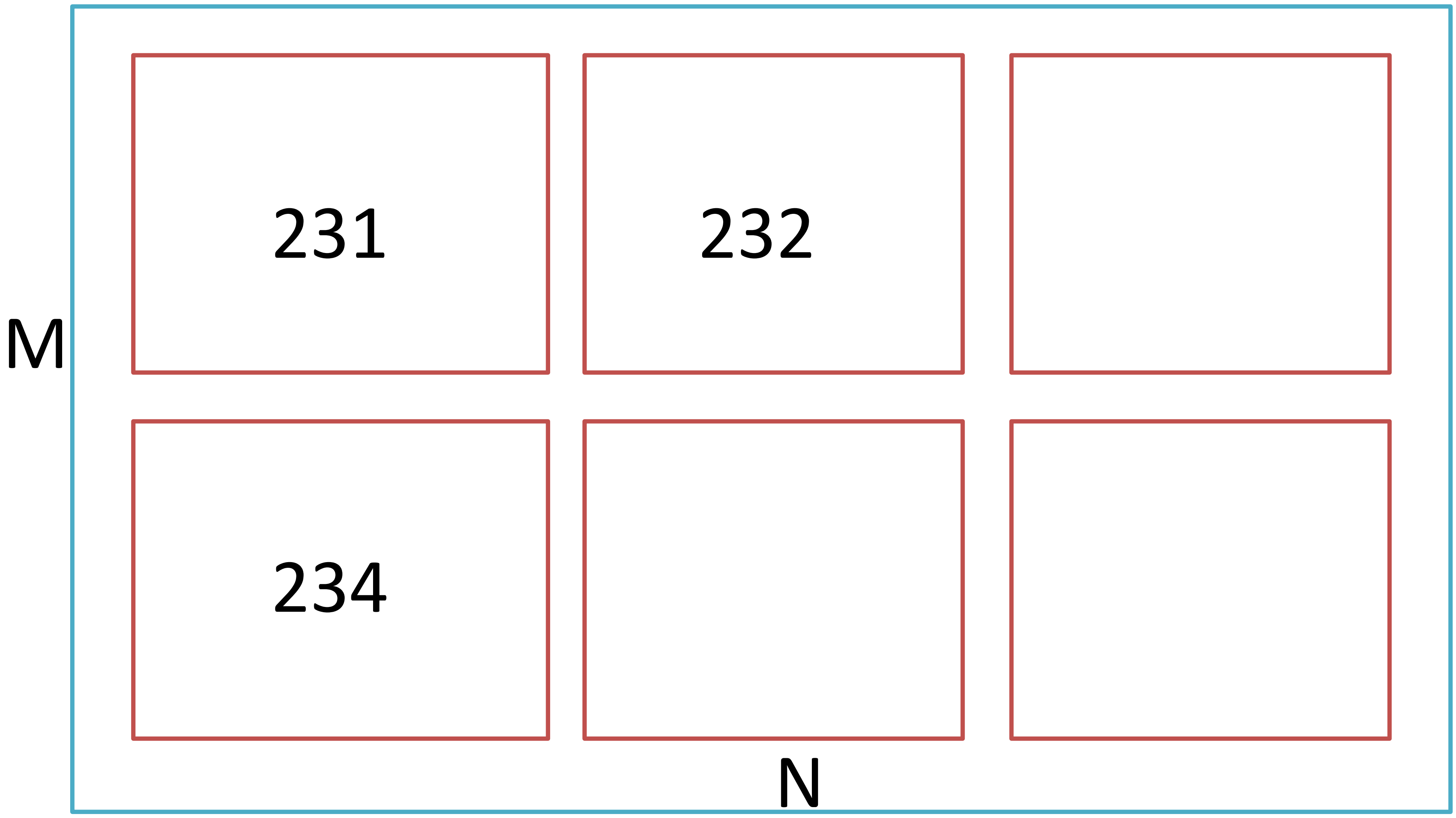


# Figures/Axes/Objects

- You can create, store and manipulate Figures and Axes objects (object-oriented)
- Or use "current" objects: gcf(), gca()
- Or use "direct" functions: pl....()
- Interactively, do it quick'n'dirty
- Think about it some more in scripts and programs

# Subplots

pl.subplot(MNk) *or* pl.subplot(M,N,k)

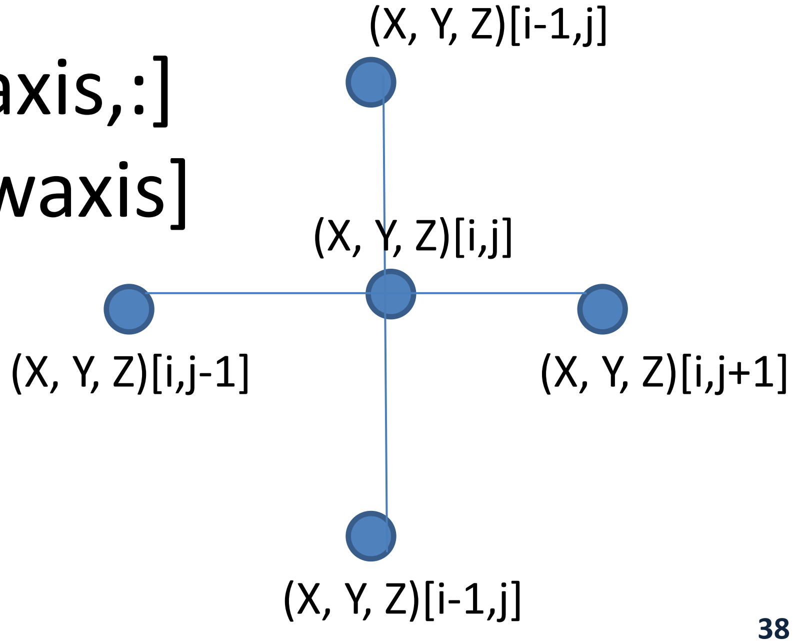


# 3D-plots

- Pseudocolor 2D: `pl.pcolormesh(...)`
- Requires special import (even if "unused")
  `from mpl_toolkits.mplot3d import Axes3D`
- Set Axes object to be 3D
  `ax = pl.gca(projection='3d')`
- `ax.plot_wireframe( ...)`
- `ax.plot_surface(...)`
- `ax.plot_trisurf(...)`

# 3D-plots

- `ax.plot_wireframe(X, Y, Z)`
  `ax.plot_surface(X, Y, Z)`
- X, Y, Z are all 2D arrays of same shape
- X = np.linspace(...)[np.newaxis,:]
  Y = np.linspace(...)[:, np.newaxis]
  Z = f(X, Y)



# Loading NumPy data from file

- Text files

```
x = np.loadtxt( ... )
```

- From HDF5 file

```
import h5py
f = h5py.File( ... )
x = f["..."].value
```

- Matlab files

```
x = scipy.io.loadmat( ... )
```

# Saving NumPy data to file

- Text files
```
np.savetxt( ... )
```
- Numpy Format
```
np.savez( ... )
```
- HDF5 file
```
import h5py
f = h5py.File( ..., "w" )
```
- Matlab files
```
scipy.io.savemat( ... )
```

- **import** matplotlib.pyplot **as** pl
- x = np.linspace(0, 1, 50)
- x
- pl.plot(x, np.sin(x))
- pl.xlabel(r"$x$")
- pl.show()
- pl.show()
- pl.interactive(True)
- pl.plot(x, np.sin(x))
- pl.xlabel(r"$y$")
- xx = x[np.newaxis,:]
  yy = x[:, np.newaxis]
- xx.shape
- pl.pcolormesh(xx, yy, np.sin(xx*yy))

load some of your research data (or something else) into a numpy array and make some cool plots

plot "1/sin(x)" and "1/cos(x)" in a single plot reaaally nice (with grid, labels, etc. – a paper grade plot)

plot sin, cos, tan and their inverses in a two-by-three subplot (also paper-grade ☺)

plot a half-sphere of radius 1

plot the shape of the Earth as approximated by the WGS84 datum surface (oblate spheroid)