Introduction To Python Programming

III: Numpy, SciPy and Matplotlib

Jan Moren, SCDA

OIST

OKINAWA INSTITUTE OF SCIENCE AND TECHNOLOGY GRADUATE UNIVERSITY

沖縄科学技術大学院大学

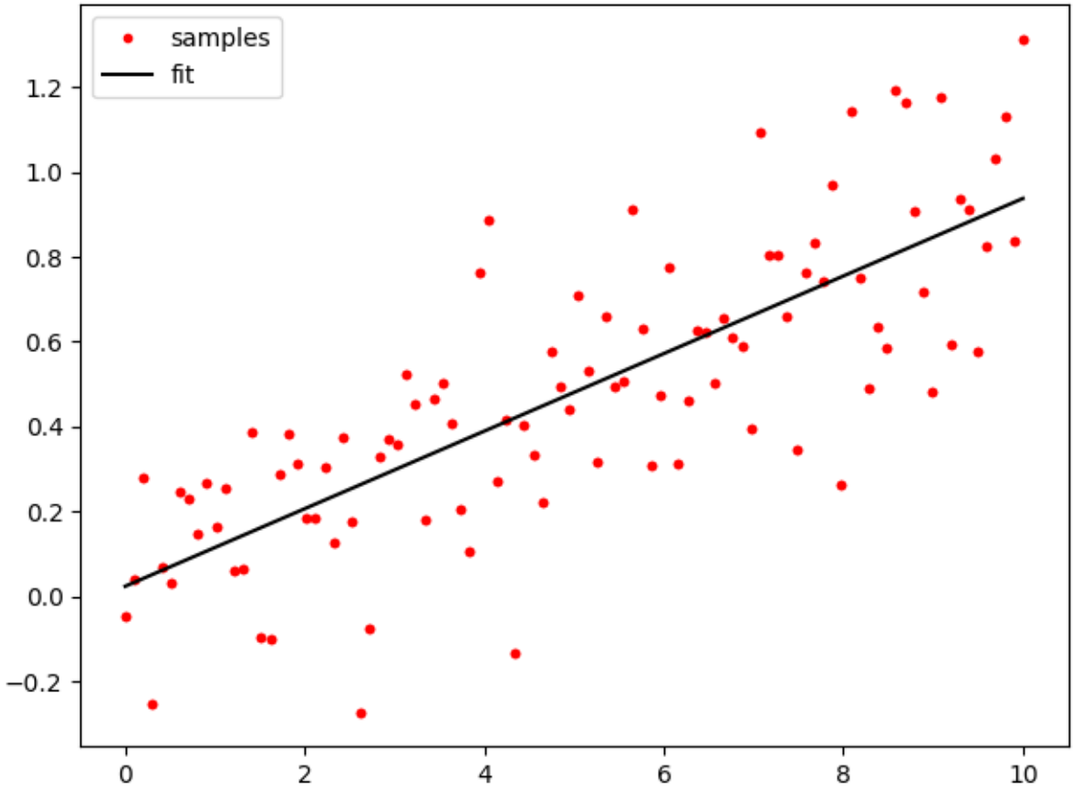
The Libraries

- NumPy: Base library. Fast arrays and matrices, FFT, linear algebra, dates, random numbers, etc
- SciPy: Uses NumPy. Large collection of high-quality implementations of numerical algorithms: stats, ODE solvers, integration, optimization, signal processing…
- Matplotlib: Data visualization for Nympy and Scipy. 2-D and 3-D plots, animations, and interactive visualizations.

SymPy:	Symbolic computation
Pandas:	Time series analysis
scikit-learn:	Machine learning

A quick example

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

x = np.linspace(0,10,100)
y = np.random.normal(x/10, 0.2)

p = stats.linregress(x,y)

plt.figure()
plt.plot(x,y, 'r.', label='samples')
plt.plot(x, x*p.slope+p.intercept,
    "k", label='fit')

plt.legend()
```



A quick example

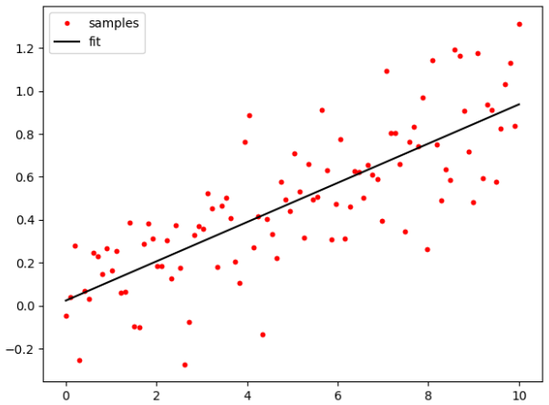
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

x = np.linspace(0,10,100)
y = np.random.normal(x/10, 0.2)

p = stats.linregress(x,y)

plt.figure()
plt.plot(x,y, 'r.', label='samples')
plt.plot(x, x*p.slope+p.intercept,
    "k", label='fit')

plt.legend()
```

Import numpy, name it “np”

Import matplotlib.pyplot, name it “plt”

- We often rename modules for ease of use; “np” and “plt” are standard names.

Import the “stats” module from scipy



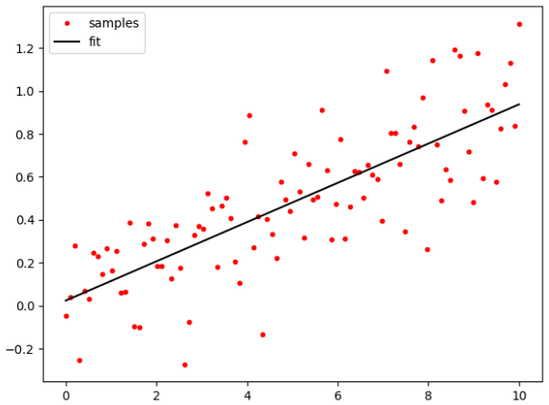
A quick example

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

x = np.linspace(0,10,100)
y = np.random.normal(x/10, 0.2)

p = stats.linregress(x,y)

plt.figure()
plt.plot(x,y, 'r.', label='samples')
plt.plot(x, x*p.slope+p.intercept,
    "k", label='fit')

plt.legend()
```

Get an *array* of 100 numbers from 0 to 10

Get an array of normal random numbers, with the mean at the “x/10” values, and variance 0.2

Linear regression to x and y



OIST

A quick example

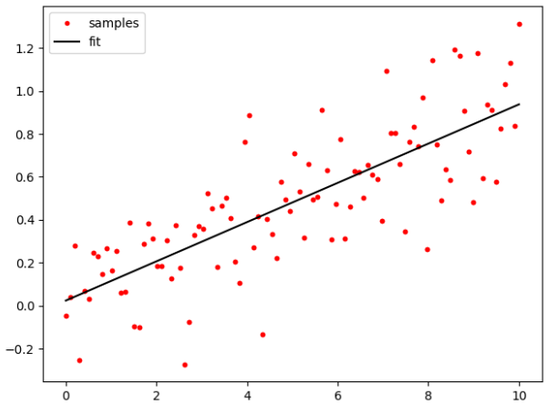
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

x = np.linspace(0,10,100)
y = np.random.normal(x/10, 0.2)

p = stats.linregress(x,y)

plt.figure()
plt.plot(x,y, 'r.', label='samples')
plt.plot(x, x*p.slope+p.intercept,
    "k", label='fit')

plt.legend()
```

Create a figure to plot into

Plot the (x,y) points in red (“r”) dots (“.”) with label “samples”

Plot a line with slope p.slope and and zero at p.intercept, name it “fit”

Add the legend to the plot



Arrays

The base Numpy data structure is the *array*:

```
  # 1-d vector from a list
  a = np.array( [1,2,3] )

  # 2-d array from a list of lists
  b = np.array([[1,2], [3,4]])
  b
array([[1, 2],
       [3, 4]])

  a = np.outer([1,2,3], [1,2,3])
  b = np.zeros_like(a)+2

  a*b                    # elementwise

array([[ 2,  4,  6],
       [ 4,  8, 12],
       [ 6, 12, 18]])
```

Arrays

- Stored as a single memory area
- All elements are the same type
  (there's an exception...)
- very efficient, compatible with C and Fortran arrays
- array is an *iterable*:
  - can convert to/from lists
  - can loop over the elements

Arrays

The base Numpy data structure is the *array*:

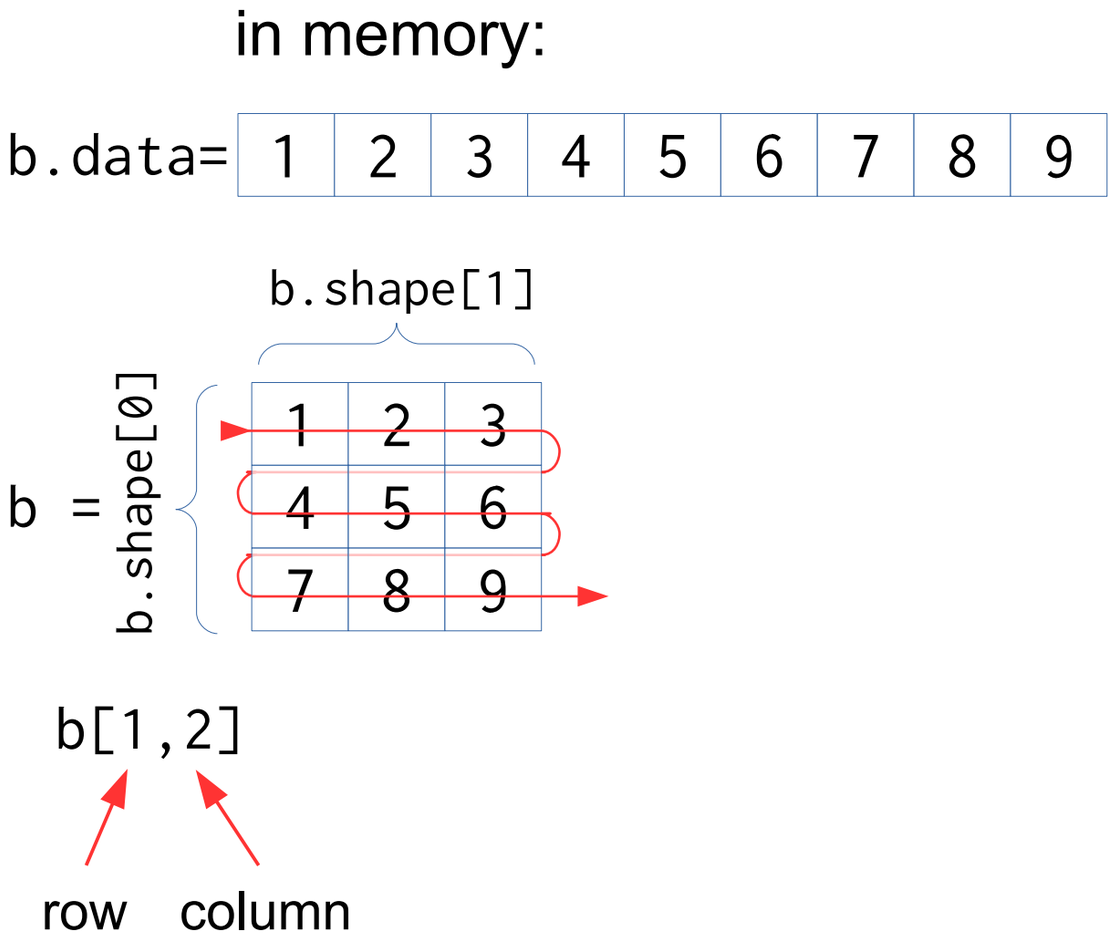
```
  # 1-d vector from a list
  a = np.array( [1,2,3] )

  # 2-d array from a list of lists
  b = np.array([[1,2], [3,4]])
  b
array([[1, 2],
       [3, 4]])

  a = np.outer([1,2,3], [1,2,3])
  b = np.zeros_like(a)+2

  a*b                    # elementwise

array([[ 2,  4,  6],
       [ 4,  8, 12],
       [ 6, 12, 18]])
```

Why Arrays?

- The storage is very efficient
- They are fast
- You can quickly operate on all elements in an array
- You can treat them as matrices
  - same as Matlab matrices
  - many high-performance operations available

Array layout and shapes

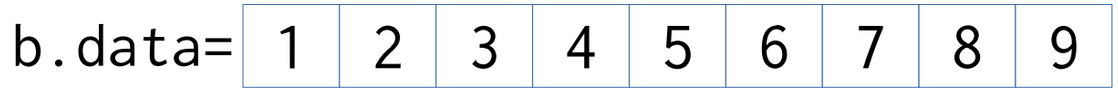
```
  b = np.arange(1,10).reshape(3,3)

  b.size          # nr elements
9
  b.ndim          # dimensions
2
  b.shape         # size in each dimension
(3, 3)

  b.reshape(2,4)# ERROR  2×4 != 9

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



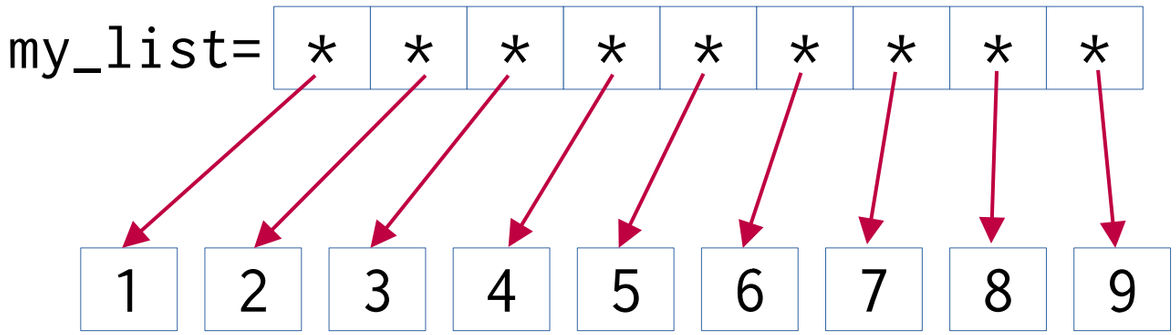
Array layout and shapes

```
  b = np.arange(1,10).reshape(3,3)

  b.size          # nr elements
9
  b.ndim          # dimensions
2
  b.shape         # size in each dimension
(3, 3)

  b.reshape(2,4)# ERROR  2×4 != 9

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

in memory:



compare with list:



Arrays and matrices

What about ‘matrix’?

- Really the same as an array:
  - Same representation
  - mostly same methods
  - very cheap to convert with "asmatrix"
- But, multiplication is a matrix multiplication, not elementwise
- Also, can use MATLAB-like init

Most code uses array in practice

```
  # MATLAB-like initialization
  m  = np.mat('1, 2; 3, 4')

  m1 = np.asmatrix(a)
  m2 = np.asmatrix(b)

  m1*m2               # dot product

matrix([[12., 12., 12.],
        [24., 24., 24.],
        [36., 36., 36.]])

  a.dot(b)            # also dot product

matrix([[12., 12., 12.],
        [24., 24., 24.],
        [36., 36., 36.]])
```

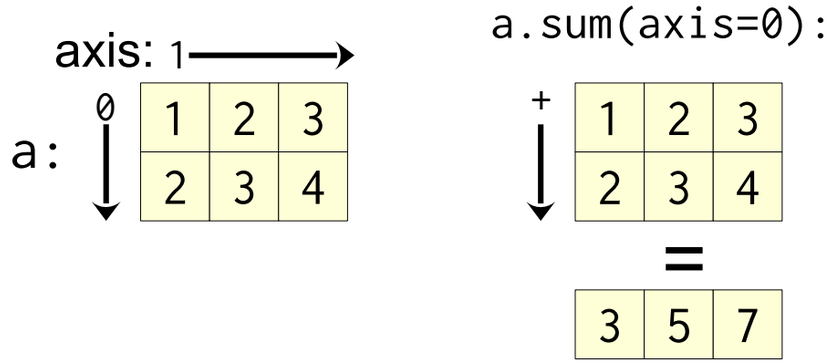
Create Arrays

```
  # Create arrays:
  a = np.array([[1,2,3], [2,3,4]])
  b = np.ones_like(a) # same shape with '1'
  c = np.zeros((2,3,4)) # 3d array of '0'

  I = np.eye(3)         # identity
  I
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

  # you can specify the numerical type
  ia = np.ones((2,2), dtype=np.complex64)
  ia
array([[1.+0.j, 1.+0.j],
       [1.+0.j, 1.+0.j]], dtype=complex64)
```

- give a sequence of elements
- create pre-filled array of specific size
- Special array creation:
  - `np.eye, np.diag, np.random ...`
  - many python modules can convert to/from numpy arrays
- You can specify the data type

Array operations

- Many functions operate on the elements of an array
  - `max()`, `min()`, `+`, `cumsum()`, `sin()`, `conjugate()`, `round()`, `prod()` …
- You can set an axis for operations:



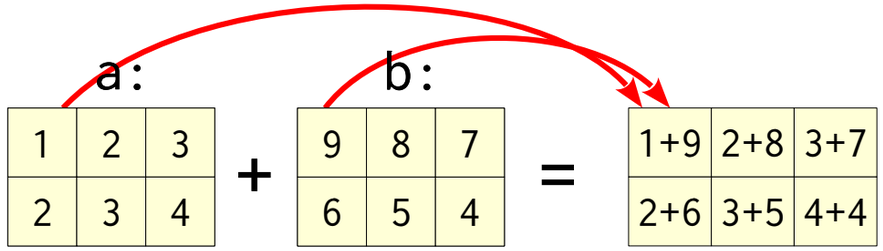
- operations give you a new array

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

  a.sum()
15
  a.sum(axis=0) # operate along one axis
array([3, 5, 7])

  np.log(a)     # all can be called like this
array([[0.     , 0.69314, 1.09861],
       [0.69314, 1.09861, 1.38629]])

  b = a.copy()
  b.T           # Transposition
array([[1, 2],
       [2, 3],
       [3, 4]])

  b.dot(a.T)    # dot product
array([[14, 20],
       [20, 29]])
```

Arrays: Broadcasting

Operations are elementwise:

```
  a = np.array([[1,2,3], [2,3,4]])
  b = np.array([[9,8,7], [6,5,4]])
  a+b
array([[10, 10, 10],
       [ 8,  8,  8]])
```



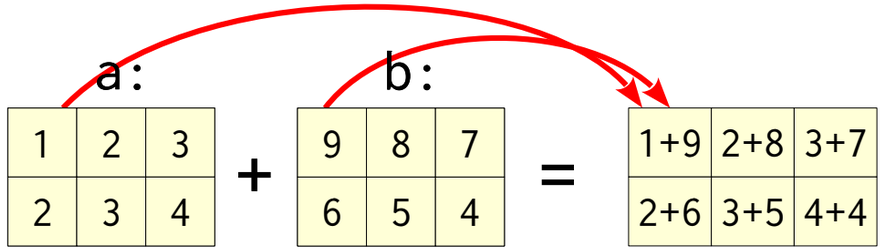
The operation happens on each pair of elements in the two arrays

→ the arrays have to be the same size

Arrays: Broadcasting

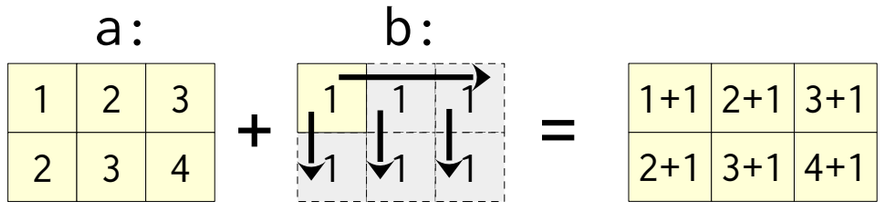
Operations are elementwise:

```
  a = np.array([[1,2,3], [2,3,4]])
  b = np.array([[9,8,7], [6,5,4]])
  a+b
array([[10, 10, 10],
       [ 8,  8,  8]])
```



What about this?

```
  a = np.array([[1,2,3], [2,3,4]])
  b = 1
  a+b
array([[2, 3, 4],
       [3, 4, 5]])
```

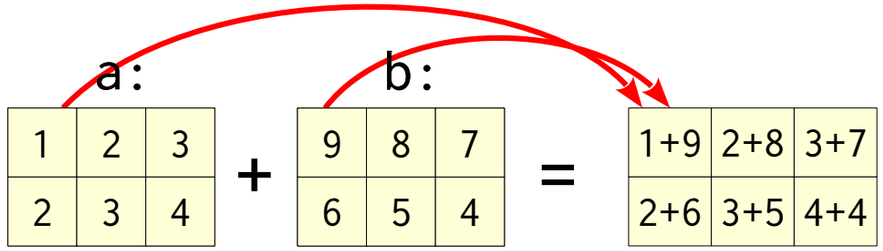


Broadcasting fills in the single value into an array of the right size, so that the elementwise operation can proceed.

Arrays: Broadcasting

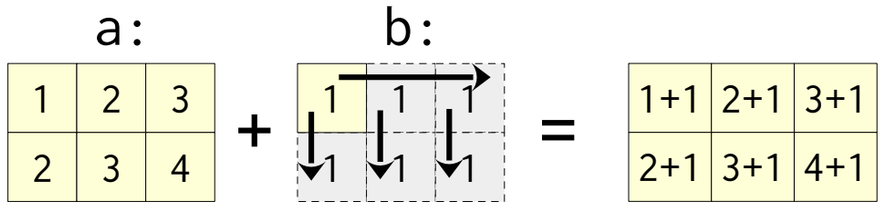
Operations are elementwise:

```
  a = np.array([[1,2,3], [2,3,4]])
  b = np.array([[9,8,7], [6,5,4]])
  a+b
array([[10, 10, 10],
       [ 8,  8,  8]])
```



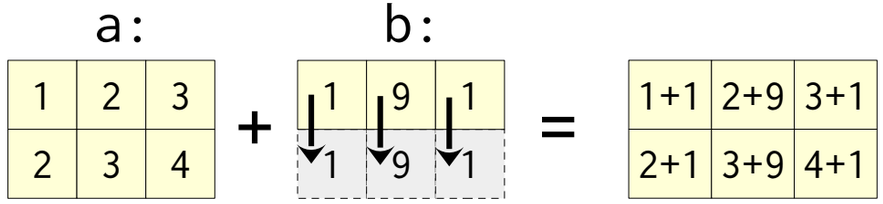
What about this?

```
  a = np.array([[1,2,3], [2,3,4]])
  b = 1
  a+b
array([[2, 3, 4],
       [3, 4, 5]])
```



And what about this?

```
  a = np.array([[1,2,3], [2,3,4]])
  b = np.array([1,9,1])
  a+b
array([[2, 11, 4],
       [3, 12, 5]])
```



Find information

How to find out more

- `lookfor("text")` looks for the text anywhere in numpy
- `function?` gives you documentation and examples for a function in ipython
- `np.info(function)` is another way to get the same documentation
- Documentation:

  https://numpy.org/doc/1.18/

```
  np.lookfor("median")
Search results for 'median'
---------------------------
numpy.median
    Compute the median along the specified axis.
numpy.nanmedian
    Compute the median along the specified axis, ...
...
numpy.quantile
    Compute the q-th quantile of the data along ...

  np.median?
Signature: np.median(a, axis=None, out=None, overw ...
Docstring:
Compute the median along the specified axis.

Returns the median of the array elements.

Parameters
----------
a : array_like
    Input array or object that can be converted to ...
```

Visualisation Example: sin()

calculate and plot sin(x) and sin(x/2)

- `np.linspace(s,e,n)` generates n points between start and end
- `plt.ion()` for interactive mode
- `figure()`: use `figsize` (8,6 inches) and dpi (128) to set the plot size.
  - 8,6 inches × 128 dpi = 1024×768 pixels
- `plt.plot()`:
  - arrays with x and y coordinates
  - set line width, color, markers etc.

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

x = np.linspace(0, 2*np.pi, 50) # 50 pts in [0..2*pi)
s1 = np.sin(x)
s2 = np.sin(x/2)

plt.ion()                       # interactive mode
plt.figure(figsize=(8,6), dpi=128)

plt.plot(x,s1,"b-", lw=2)   # lw=linewidth
plt.plot(x,s2, ".-", c="red") # red dot-dash line
```

Visualisation Example: sin()

calculate and plot sin(x) and sin(x/2)

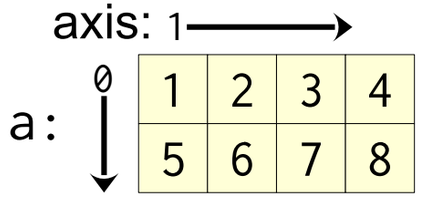
- `plt.ion()` for interactive mode
- `figure()`: use `figsize` (8,6 inches) and dpi (128) to set the plot size.
- `plt.plot()`:
  - arrays with x and y coordinates
  - set line width, color, markers etc.
- `plt.xlabel()` sets label for x axis
- `plt.tight_layout()` removes extra space
- `plt.savefig()` saves figure to file
- `plt.show()` shows plot if not interactive

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

x = np.linspace(0, 2*np.pi, 50) # 50 pts in [0..2*pi)
s1 = np.sin(x)
s2 = np.sin(x/2)

plt.ion()                       # interactive mode
plt.figure(figsize=(8,6), dpi=128)

plt.plot(x,s1,"b-", lw=2)   # lw=linewidth
plt.plot(x,s2, ".-", c="red") # red dot-dash line

plt.xlabel("angle")
plt.ylabel("awesomness")

plt.tight_layout()          # remove extra space
plt.savefig("awesome.png")  # svg, pdf, other formats
plt.show()                  # show on screen
```

Array indexing

arrays are indexed:

`a[row, col, depth ...]`



- begins with 0
- negative counts from the end
- can also use the “list of lists” way to index:

  `a[row][col][depth]`

- a list as index:
  - list of elements to pick in order
  - creates a copy of the data

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

  a[0, 0]        # row, column from 0,0
1
  a[0][0]        # same thing
1
  a[1]           # single index=entire row
array([ 5, 6, 7, 8])

  a[0, -1]       # -1 = last element
4

  a[0, [0,3,2]] # select elements
array([1, 4, 3])
```

Array slices

Slice: a “cut-out” piece of an array

- Slices are *not* copies
- specify slices with ‘:’

  `[first elem:last elem+1]`

a:

1	2	3	4	5	6	7	8

b:

1	2	3	4
5	6	7	8

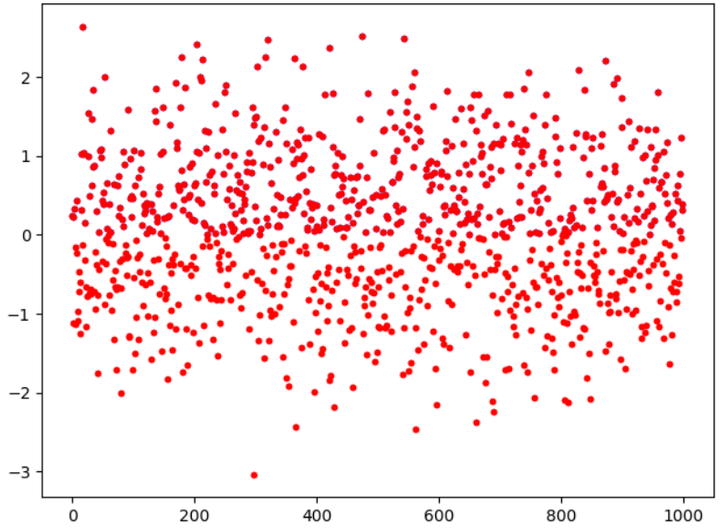
```
a[:3] = [0:3]   # start of array:3
a[2:] = [2:8]   # 2:end of array
a[:]  = [0:8]   # all elements
```

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

  a[2:4]          # start:end+1
array([3, 4])

  a[2]            # 1 value
3
  a[2:3]          # 1-element array
array([3])        # 2:3 is DIFFERENT from 2!

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

  b[:, 1]
array([2, 6])     # 1d array

  b[:, 1:2]
array([[2],       # 2d array
       [6]])
```

Array indexing

Using boolean indexes:

```
a = np.random.normal(size=1000)
x = np.arange(a.size)
plt.plot(x, a, "r.")
```



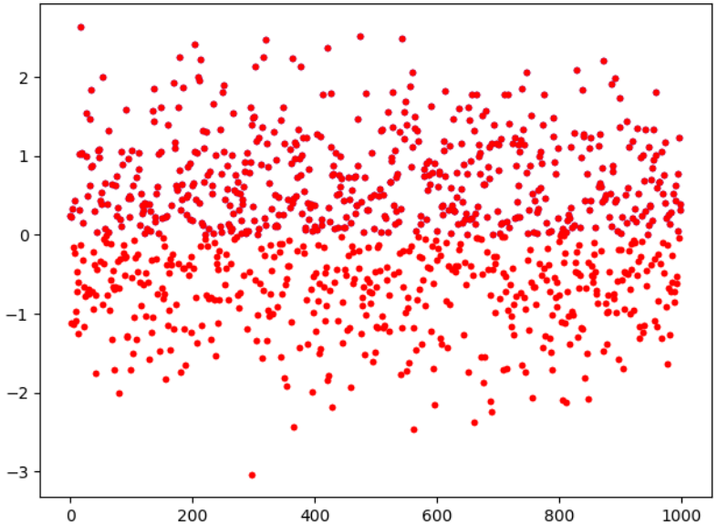
Use a test to pick what values to use

For example: pick and plot values of ‘a’ that are greater than or equal to 0

Array indexing

Using boolean indexes:

```
  a = np.random.normal(size=1000)
  x = np.arange(len(a))
  plt.plot(x, a, "r.")
  apos = (a>=0)
array([ True, False,  True,  True,  True …
```

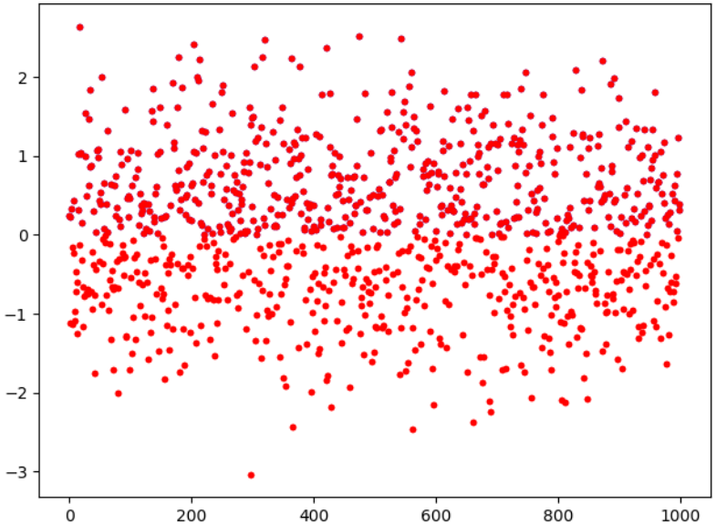


- Compare each element in a.
- Return a new array with "True" or "False" in each position.

Array indexing

Using boolean indexes:

```
  a = np.random.normal(size=1000)
  x = np.arange(len(a))
  plt.plot(x, a, "r.")
  apos = (a>=0)
array([ True, False,  True,  True,  True …

## Small example ##

  b = np.random.randint(-5,5,8)
array([-2,  4, -3,  1, -1,  2,  1,  4])

  bpos = (b>=0)
  b[bpos]
array([4, 1, 2, 1, 4])
```

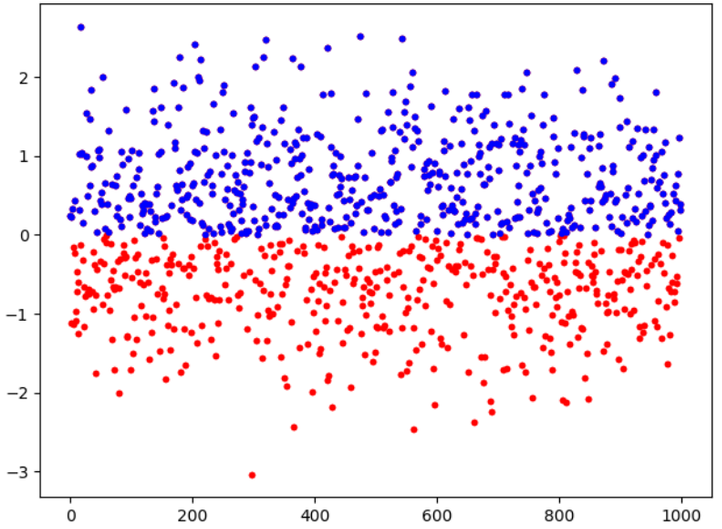


- Compare each element in a.
- Return a new array with "True" or "False" in each position.
- A boolean index returns the elements that are "True"

Array indexing

Using boolean indexes:

```
  a = np.random.normal(size=1000)
  x = np.arange(len(a))
  plt.plot(x, a, "r.")
  apos = (a>=0)
array([ True, False,  True,  True,  True …

  plt.plot(x[apos], a[apos], "b.")
```

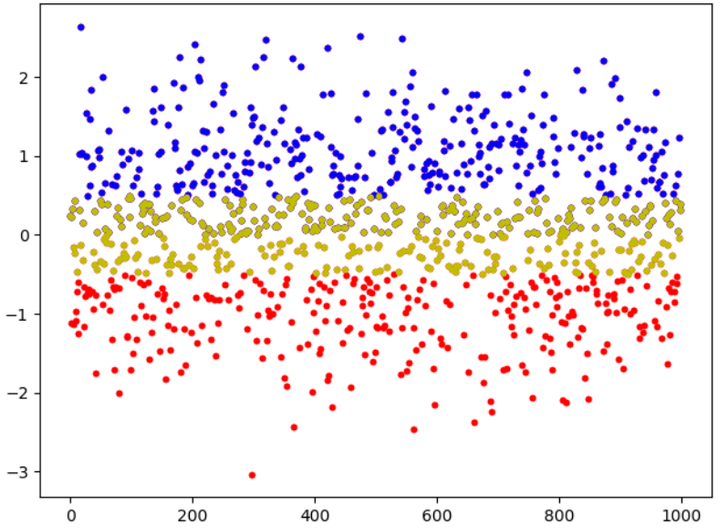


- Compare each element in a.
- Return a new array with "True" or "False" in each position.
- A boolean index returns the elements that are "True"

Array indexing

Using boolean indexes:

```
  a = np.random.normal(size=1000)
  x = np.arange(len(a))
  plt.plot(x, a, "r.")
  apos = (a>=0)
array([ True, False,  True,  True,  True …

  plt.plot(x[apos], a[apos], "b.")

  # combine boolean arrays with '&' and '|'
  amid = (a<0.5) & (a>-0.5)

  plt.plot(x[amid], a[amid], "y.")
```



- Compare each element in a.
- Return a new array with "True" or "False" in each position.
- A boolean index returns the elements that are "True"

Speed

in ipython we can use "timeit" to measure execution time

Elementwise multiplication using Numpy:

```
a = np.ones(10000)*10
b = np.ones_like(a)*2
c = np.zeros_like(a)

%timeit c=a*b
```

```
3.35 µs ± 3.42 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Elementwise multiplication using our own loop:

```
%%timeit
for i in range(len(a)):
    c[i] = a[i]*b[i]
```

```
4.76 ms ± 22.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Speed

in Ipython we can use "timeit" to measure execution time

Elementwise multiplication using Numpy:

```
a = np.ones(10000)*10
b = np.ones_like(a)*2
c = np.zeros_like(a)

%timeit c=a*b
```

```
3.35 µs ± 3.42 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Elementwise multiplication using our own loop:

```
%%timeit
for i in range(len(a)):
    c[i] = a[i]*b[i]
```

```
4.76 ms ± 22.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Our loop is *1400×* slower than Numpy!

Speed

in ipython we can use "timeit" to measure execution time

Elementwise multiplication using Numpy:

```
a = np.ones
b = np.ones
c = np.zero

%timeit c=a

3.35 µs ± 3.4
dev. of 7 run
```

Elementwise multiplication using our own loop:

```
a)):
]

loop (mean ±
00 loops each)
```

Always spend as much time as possible inside Numpy and Scipy functions!

→ If there is a function to do something, use it. Do not write it yourself.

loading, saving and displaying

reading, writing images

- plt.imread() reads an image
  - 3-d array (height, width, color)
- plt.imshow() shows array as image
- plt.pcolormesh() shows as mesh
- plt.imsave() saves a plot as image

```
  # one way to read an image: use matplotlib
  img = plt.imread("shisa.jpg")

  # height, width, color channels
  img.shape
(200, 250, 3)

  plt.imshow(i)    # show as image

  # coordinates of the corners
  x = np.arange(0, 251, 1)
  y = np.arange(201, 0, -1)

  # show mesh, green channel, grayscale
  plt.pcolormesh(x,y,img[:,:,1], cmap="gray")

  plt.imsave("shisa_green.jpg")
```

Reading data

`genfromtxt()`

- Reads a text-format data file
- skips comments, empty lines

  you can set what counts as comment etc.
- set the data type with "`dtype=`"
- returns a Numpy array

```
# example data file. genfromtxt() skips '#'
# comments and empty lines automatically

1   10 11     # whitespace is default separator
2   9  12
```

```
  np.genfromtxt("data1.txt")
array([[ 1., 10., 11.],
       [ 2.,  9., 12.]])  # float type

  np.genfromtxt("data1.txt", dtype="int")
array([[ 1, 10, 11],
       [ 2,  9, 12]])     # integer
```

Reading data

`genfromtxt()`

- Reads a text-format data file
- skips comments, empty lines

  you can set what counts as comment etc.
- set the data type with "`dtype=`"
- returns a Numpy array
- "`delimiter=`" sets separator char
- "`skip_header=`" and "`skip_footer=`" skips lines at the beginning and end

But what if you have columns with different data types?

```
count, val1, val2      # header line
1,     10,   11        # comma-separated data
2,     9,    12
```

```
  np.genfromtxt("data2.txt")
array([[nan, nan, nan],
       [nan, nan, 11.],
       [nan, nan, 12.]])  # that didn't go well

  np.genfromtxt("data2.txt", skip_header=1,
                delimiter=",", dtype="int")
array([[ 1, 10, 11],
       [ 2,  9, 12]])     # much better!
```

Reading data

`genfromtxt()` can only do so much

- Matlab .mat: "scipy.io" module
- CSV: Python "CSV" module
  Pandas read_csv()
- HDF5: Python "h5py" module
- NetCDF: "scipy.io" module
- IDL .sav: "scipy.io" module
- DICOM: Python "pydicom" module
  Python "dicom-numpy" module

...

...

Scipy

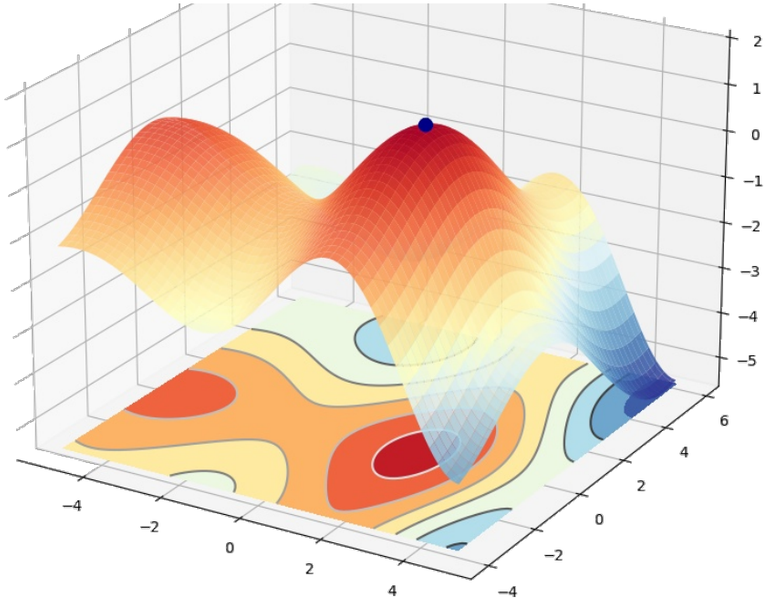
• special	Defines special functions
• integrate	Integration and ODE solvers
• **optimize**	**find minima & roots, fit curves, LP**
• interpolate	data interpolation
• fftpack	FFT, discrete cosine, sine
• signal	convolution, filters, spectral analysis
• linalg	superset of Numpy linear algebra
• sparse	sparse matrices, graphs, algebra
• spatial	Voronoi, KDtree, convex hulls
• stats	Statistics
• ndimage	image analysis
• IO	Matlab, IDL, Netcdf, other file formats

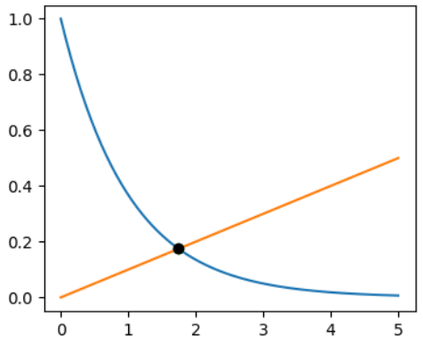
Optimization

Find the maximum/minimum/zero value of a function

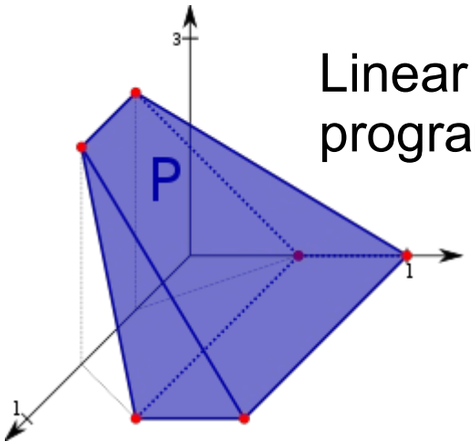
- You can optimize (almost) anything you can describe with a function:
  - simulation parameters
  - fit your model to data
  - optimize a process
  - assignment problems
  - numerical methods (finding roots, extrema, etc.)

Parameter optimization





numerical methods



Linear programming

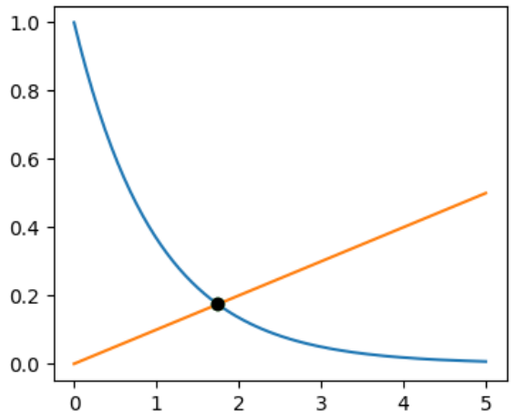
Scipy Exercise:
Find the root (zero) of a function

We want the intersection of two functions `exp(-x)` and `x/10`

1: define a python function `f(x)`

```
exp(-x) - x/10
```

2: find where `f(x)==0`




```python
from scipy import optimize as opt

# define our function f(x)



# use a function in opt to find root of f(x)
```

see: https://docs.scipy.org/doc/scipy/reference

Also:

```
help(opt)
help(opt.<method>)
```

Scipy Exercise:
Find the root (zero) of a function

We want the intersection of two functions `exp(-x)` and `x/10`

1: define a python function `f(x)`

```
exp(-x) - x/10
```

2: find where `f(x)==0`

Look for root finding methods:

ex: brentq, newton, fsolve

```
from scipy import optimize as opt

# define our function f(x)



# use a function in opt to find root of f(x)
```

see: https://docs.scipy.org/doc/scipy/reference

Also:

```
help(opt)
help(opt.<method>)
```

Scipy Exercise:
Find the root (zero) of a function

We want the intersection of two functions `exp(-x)` and `x/10`

1: define a python function `f(x)`

```
exp(-x) - x/10
```

2: find where `f(x)==0`

Look for root finding methods:

ex: brentq, newton, fsolve

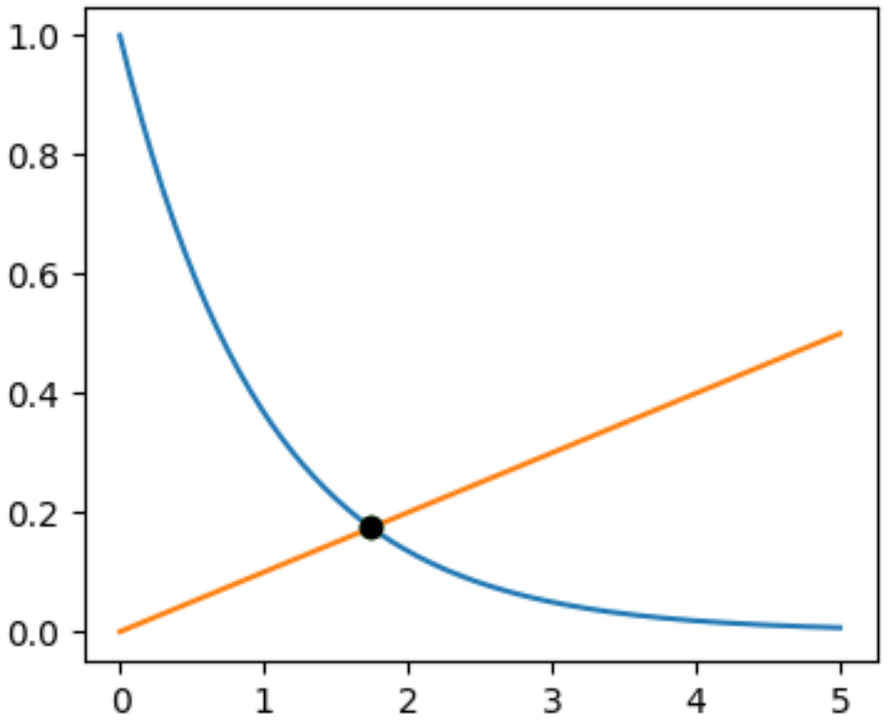
```
from scipy import optimize as opt

# define our function f(x)
def f(x):
    return np.exp(-x) - x/10

# use a function in opt to find root of f(x)
opt.brentq(f, 0, 10)
1.7455280027406994
```

see: https://docs.scipy.org/doc/scipy/reference

Also:

```
help(opt)
help(opt.<method>)
```

Scipy Exercise:
Find the root (zero) of a function




```python
from scipy import optimize as opt

# define our function f(x)
def f(x):
    return np.exp(-x) - x/10

# use a function in opt to find root of f(x)
p = opt.brentq(f, 0, 10)

x=np.linspace(0,5,200)
plt.plot(x, np.exp(-x))
plt.plot(x, x/10)
plt.plot(p, p/10, "ko")   # k=black, o=circle
```

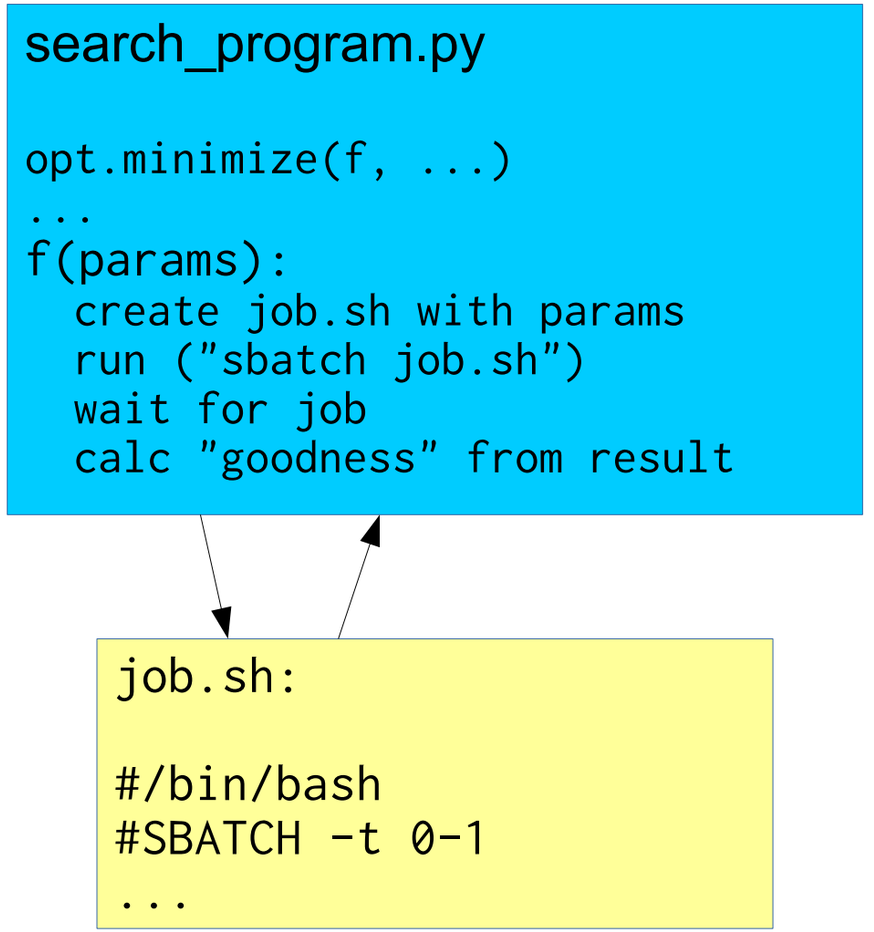
Scipy Optimize: A few Thoughts

- opt.minimize is a general interface for finding function minima:

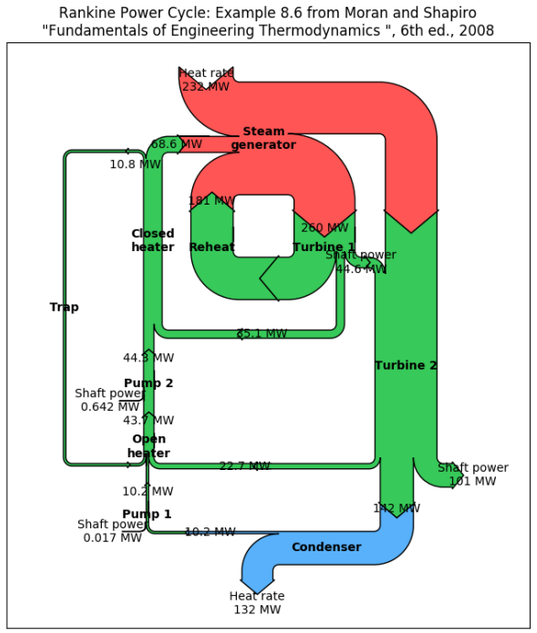
```
opt.minimize(f, [x0, y0...])
```

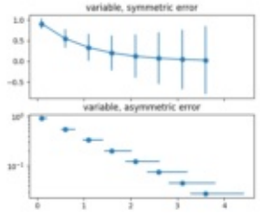
- *Many* HPC applications need parameter tuning. Tuning by hand is error prone, grid search is inefficient.
- The objective function f can be anything — including a separate cluster job

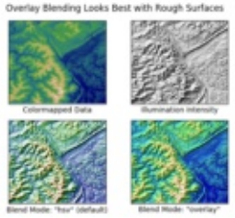
  → We can optimize our parameters automatically with python and a cluster

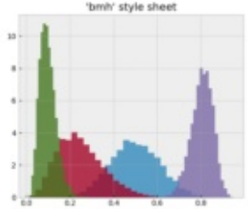


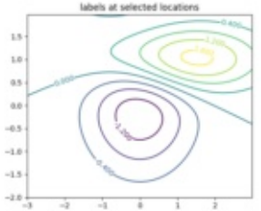
Plots

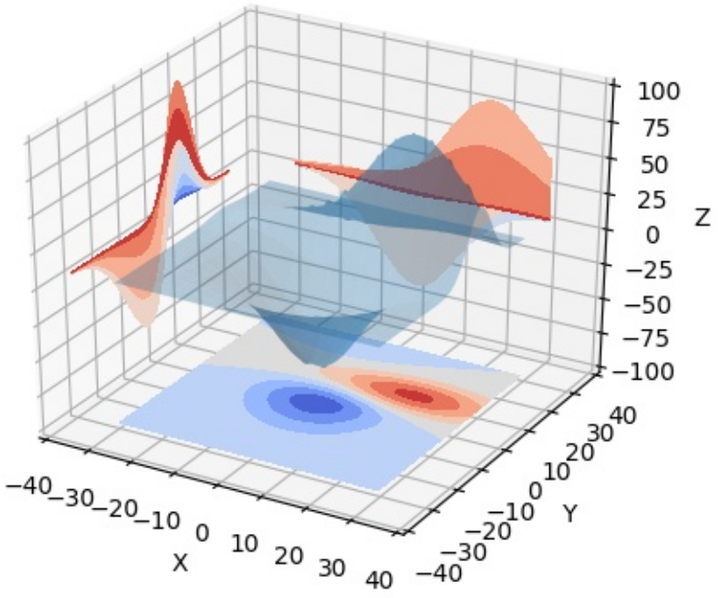

Rankine Power Cycle: Example 8.6 from Moran and Shapiro
"Fundamentals of Engineering Thermodynamics ", 6th ed., 2008
Heat rate
232 MW
Steam generator
68.6 MW
10.8 MW
181 MW
260 MW
Closed heater
Reheat
Turbine 1
Shaft power
44.6 MW
Trap
65.1 MW
44.5 MW
Pump 2
Shaft power
0.642 MW
43.7 MW
Open heater
22.7 MW
Turbine 2
Shaft power
191 MW
10.2 MW
Pump 1
Shaft power
0.017 MW
10.2 MW
132 MW
Condenser
Heat rate
132 MW

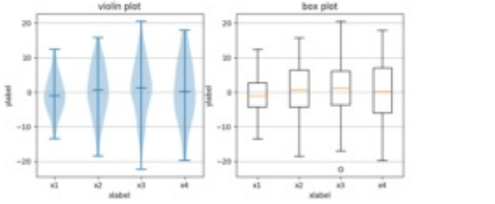

variable, symmetric error
variable, asymmetric error

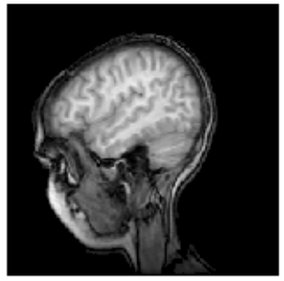

Overlay Blending Looks Best with Rough Surfaces
Colormapped Data
Illumination Intensity
Blend Mode: "hsv" (default)
Blend Mode: "overlay"

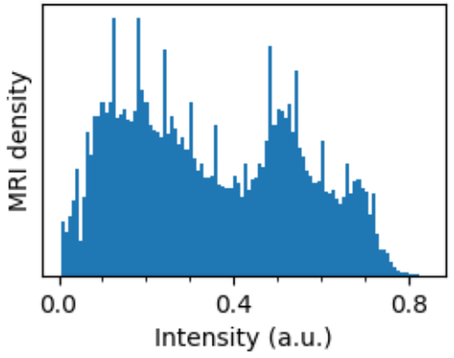

'bmh' style sheet

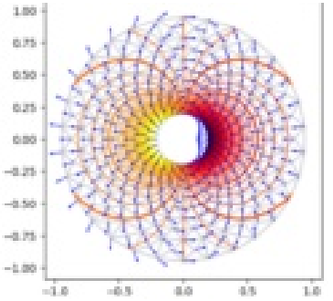

labels at selected locations

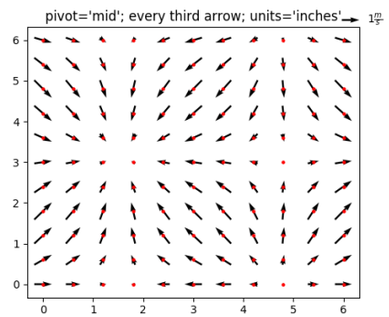

100
75
50
25
0
−25
−50
−75
−100
Z
−40 −30 −20 −10 0 10 20 30 40
X
40 30 20 10 0 −10 −20 −30 −40
Y

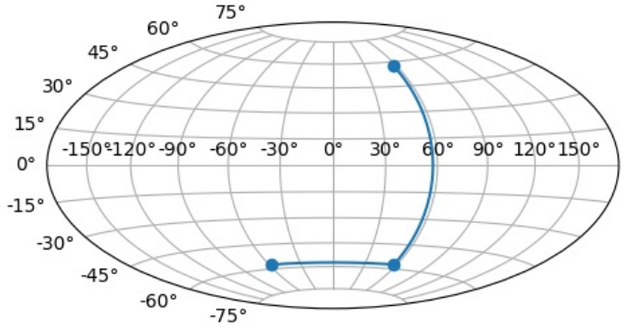

violin plot
box plot

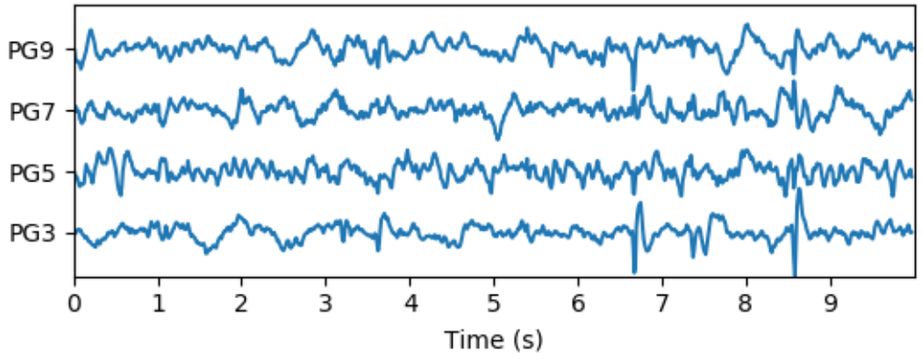


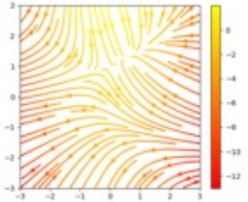

MRI density
0.0
0.4
0.8
Intensity (a.u.)

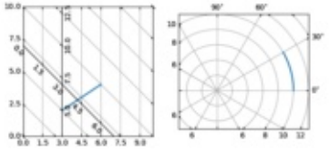



pivot='mid'; every third arrow; units='inches'


75°
60°
45°
30°
15°
0°
-15°
-30°
-45°
-60°
-75°
-150° -120° -90° -60° -30° 0° 30° 60° 90° 120° 150°


PG9
PG7
PG5
PG3
0 1 2 3 4 5 6 7 8 9
Time (s)





OIST

Plots

```
plt.ion()          # interactive mode
#plt.ioff()        # stop interactive mode

plt.figure()       # create a new figure

plt.plot(x,y,…)    # plot 1-d data


# save figure to file
plt.savefig("myfig.png")

plt.show()     # if we didn't use ion()
```

- The Pyplot interface ("plt.") keeps track of figures and axes for you
- In "interactive mode" any change immediately updates the plot

  → good for interactive use
- non-interactive mode only draws the graph when you save or show the figure

  → great for programs, or when you plot lots of data

Plots

Creating the plot

```
plt.plot( … )     # creates figure

# figure parameters:

figure(nr, figsize, dpi, facecolor)

figsize=(x, y)        # size in inches
dpi = n               # pixels per inch
facecolor = color     # background color

# colors
'r'                 # red
'red'               # red
'#ff0000'           # red
[1.0, 0.0, 0.0]     # red
```

- colors can be given in many ways:

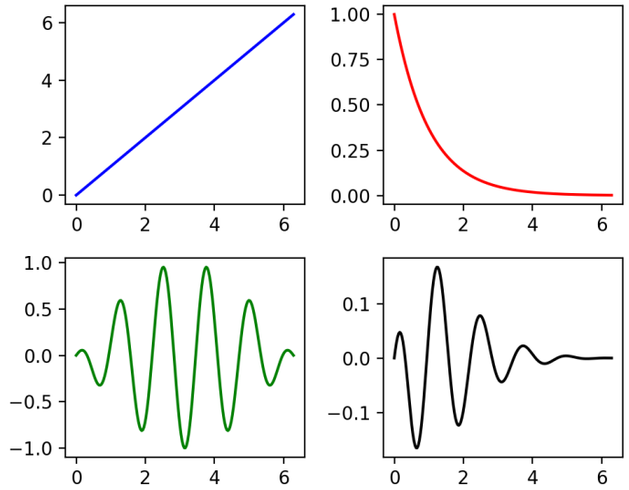
'r'	single color letter
'red'	HTML name
'#ff0000'	Red-Green-Blue triplet in hex
[1,0,0]	RGB as [0...1] triplet

Plots

```
f,ax = plt.subplots(2,2)

ax[0,0].plot(x,y1,'b')  # plot each
ax[0,1].plot(x,y2,'r')  # quadrant
ax[1,0].plot(x,y3,'g')
ax[1,1].plot(x,y4,'k')
```

Subplots

- Make multiple plots in grids
- There are a number of ways to create subplots (including doing it "by hand")
- `plt.subplots()` is pretty flexible and not hard to use.



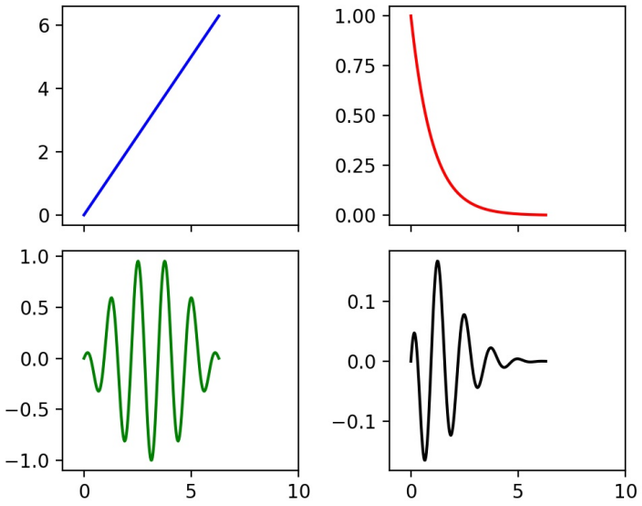
Plots

```
f,ax = plt.subplots(2,2, sharex=True)

ax[0,0].plot(x,y1,'b')  # plot each
ax[0,1].plot(x,y2,'r')  # quadrant
ax[1,0].plot(x,y3,'g')
ax[1,1].plot(x,y4,'k')

ax[1,1].clear()
plt.sca(ax[1,1])     # set current axes
plt.plot(x,y4,'k')   # can plot with
                     # pyplot

ax[1,1].set_xlim([-1,10])  # x axes are
                           # connected
```

Subplots

- sharex() and sharey() makes the x or y-axis common across rows/cols or across all plots.

  → axis changes in one plot affects all connected plots



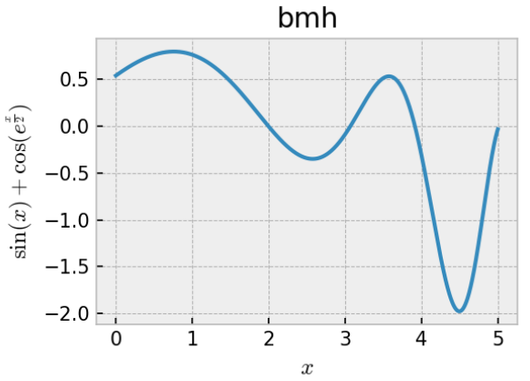
OIST

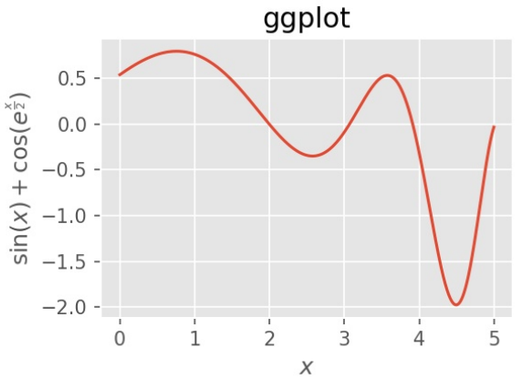
Plot Styles

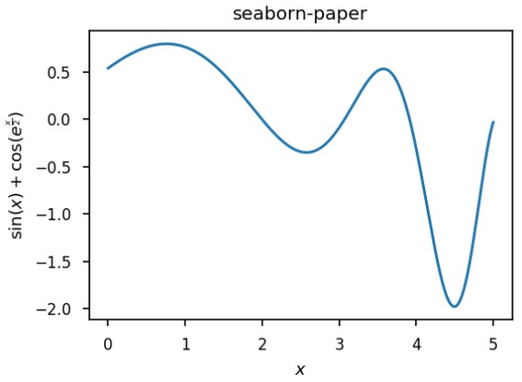
```
# styles

plt.style.use("ggplot")

print(plt.style.available)
['seaborn-paper', 'classic', …
```

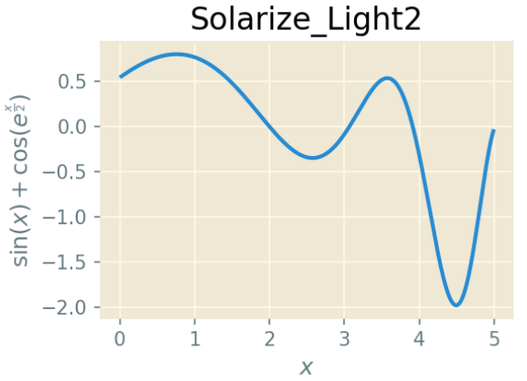
Styles set visual defaults

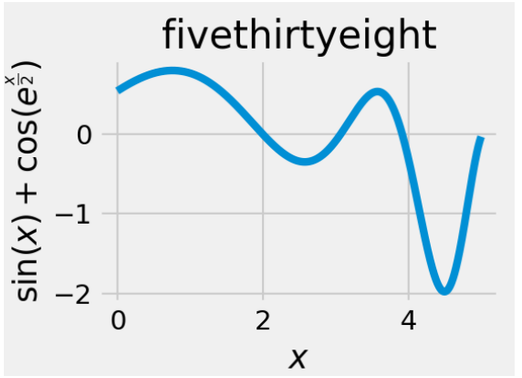
- Not all defaults are actively set, so settings from previous styles may linger.

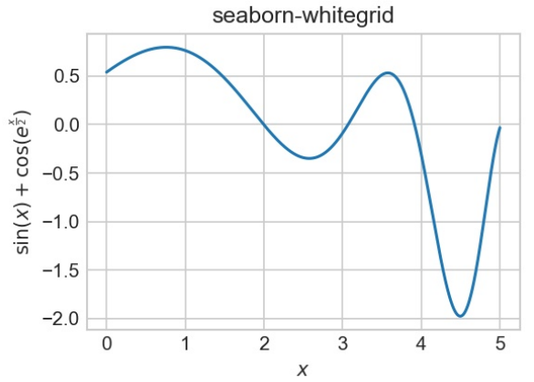








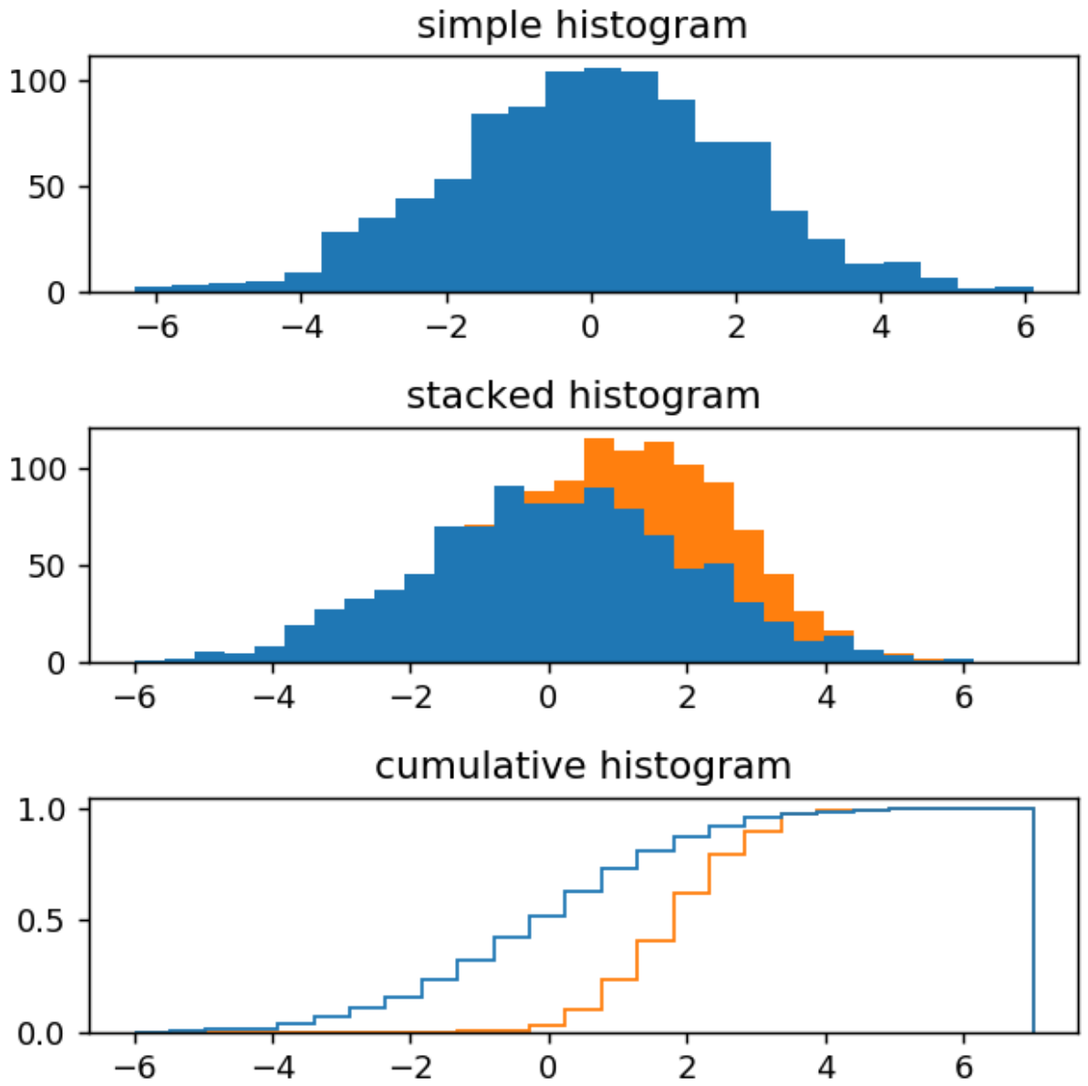




OIST

Histogram

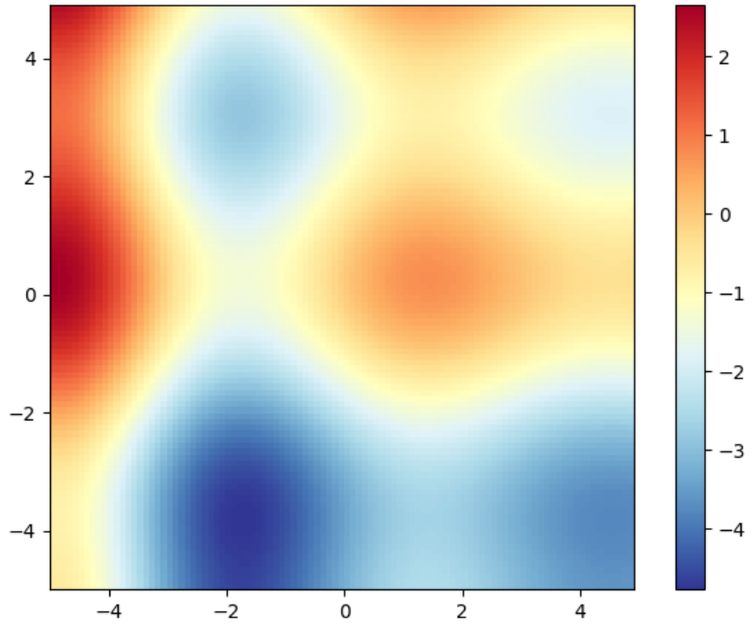
```
  p = np.random.normal(scale=2, size=1000)
  plt.hist(p, bins='auto')

  p2=np.random.normal(loc=2, size=300)
  plt.hist((p,p2),
    histtype='barstacked', # stack bars
    bins=30,               # set bin number
    range=(-6,7))          # min and max

# can also give explicit bin positions
  bins=[0.0, 1.0, …
  cumulative=True        # cumultive hist
  density=True           # normalized scale
  weights=[…]            # weigh points
```



Mesh and Contour Plots

```
# create coordinates
xc = np.arange(-5,5,0.1)
yc = np.arange(-5,5,0.1)
x,y = np.meshgrid(xc,yc)

# create some 2-d data
z = np.sin(x)*np.exp(-x/5) +
    np.cos(y)-np.exp(-y/4)

f,ax = plt.subplots()
ax.set_aspect('equal')

p = ax.pcolormesh(x,y,z,    # coord, vals
  cmap=plt.cm.RdYlBu_r)     # color map

f.colorbar(p)
```

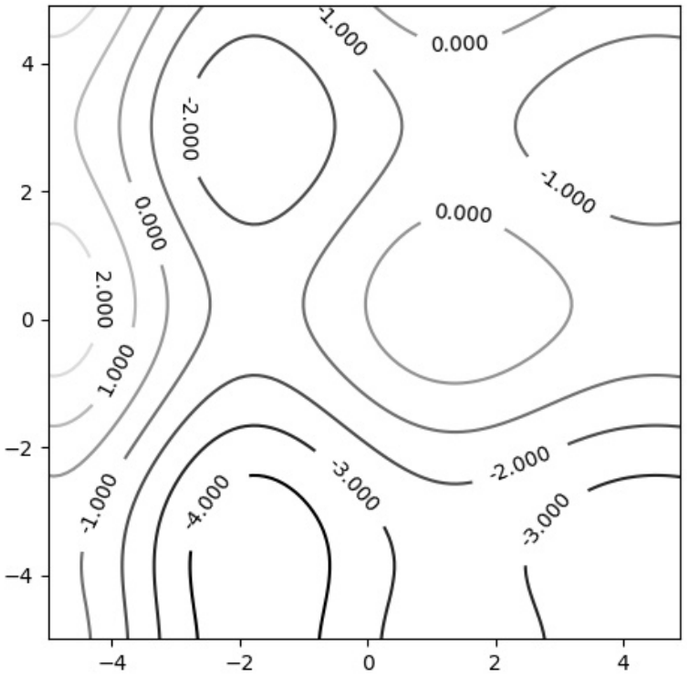
- 2d data is an array of z-values.
  - optionally give X and Y coordinates for each Z data points ("meshgrid")
  - pcolormesh plots z values as colors:



Mesh and Contour Plots

```
# create coordinates
xc = np.arange(-5,5,0.1)
yc = np.arange(-5,5,0.1)
x,y = np.meshgrid(xc,yc)

# create some 2-d data
z = np.sin(x)*np.exp(-x/5) +
    np.cos(y)-np.exp(-y/4)

f,ax = plt.subplots()
ax.set_aspect('equal')

c = ax.contour(x,y,z,    # coord, vals
  8,                     # number of levels
  vmax=3,                # upper limit
  cmap=plt.cm.gray)      # color map

c.clabel(colors="k")     # numerical labels
```

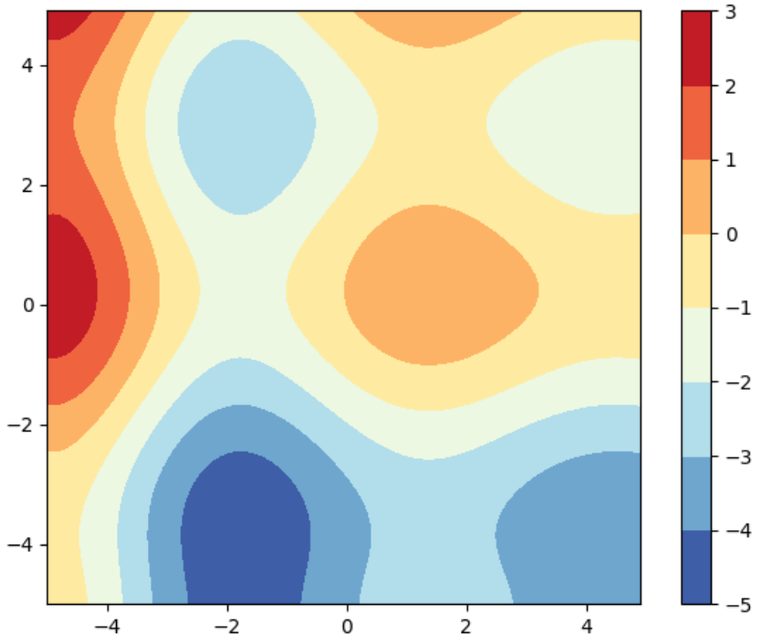
- 2d data is an array of z-values.
  - optionally give X and Y coordinates for each Z data points ("meshgrid")
  - contour plots



Mesh and Contour Plots

```
# create coordinates
xc = np.arange(-5,5,0.1)
yc = np.arange(-5,5,0.1)
x,y = np.meshgrid(xc,yc)

# create some 2-d data
z = np.sin(x)*np.exp(-x/5) +
    np.cos(y)-np.exp(-y/4)

f,ax = plt.subplots()
ax.set_aspect('equal')

s = ax.contourf(x,y,z,  # coord, vals
  8,                    # number of levels
  cmap=plt.cm.RdYlBu_r) # color map

f.colorbar(s)
```

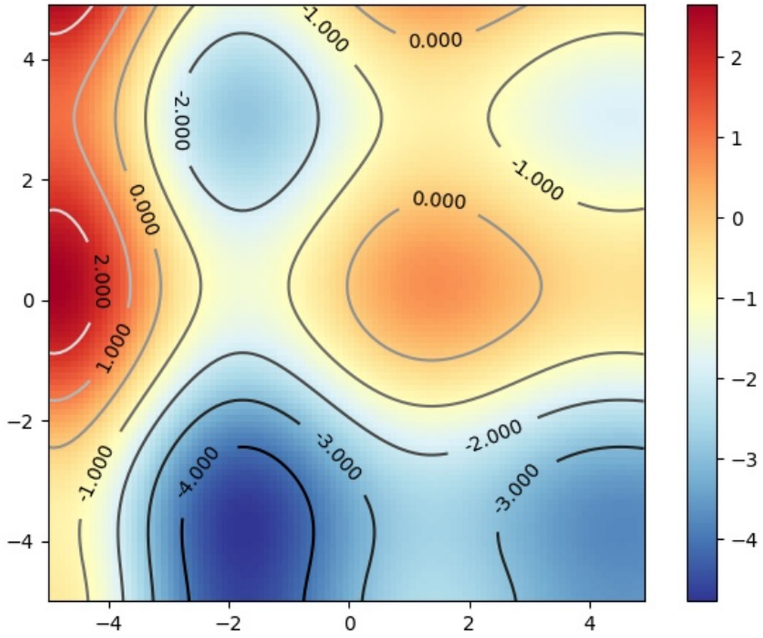
- 2d data is an array of z-values.
  - optionally give X and Y coordinates for each Z data points ("meshgrid")
  - filled contour plots:



Mesh and Contour Plots

```
f,ax = plt.subplots()
ax.set_aspect('equal')

p = ax.pcolormesh(x, y, z, 8
        cmap=plt.cm.RdYlBu_r)

c = ax.contour(x, y, z, 8,
        vmax=3,
        cmap=plt.cm.gray)

f.colorbar(p)
c.clabel(colors="k")
```

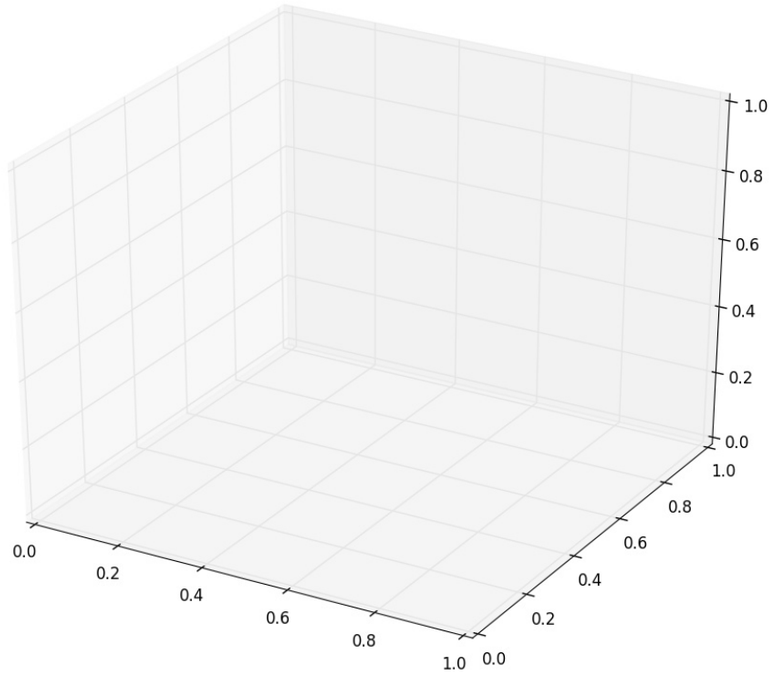
- 2d data is an array of z-values.
  - optionally give X and Y coordinates for each Z data points ("meshgrid")
  - combine plots for best results:



3D Plots

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

f = plt.figure(figsize=(10,8))
f.add_subplot(111, projection='3d')
ax = f.gca()
```

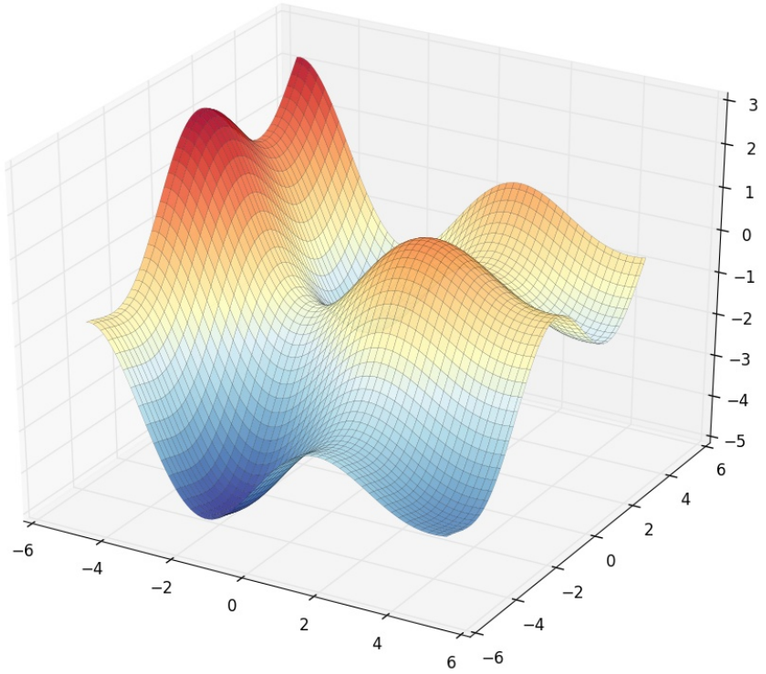
- import Axes3d from mpl_toolkits
  - adds 3d-related methods to matplotlib
  - still need to use older "add_subplot" instead of "subplots()"



3D Plots

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

f = plt.figure(figsize=(10,8))
f.add_subplot(111, projection='3d')
ax = f.gca()

s=ax.plot_surface(x,y,z,
      rstride=2, cstride=2,
      lw=0.1,
      alpha=0.9,
      cmap=plt.cm.RdYlBu_r)
```

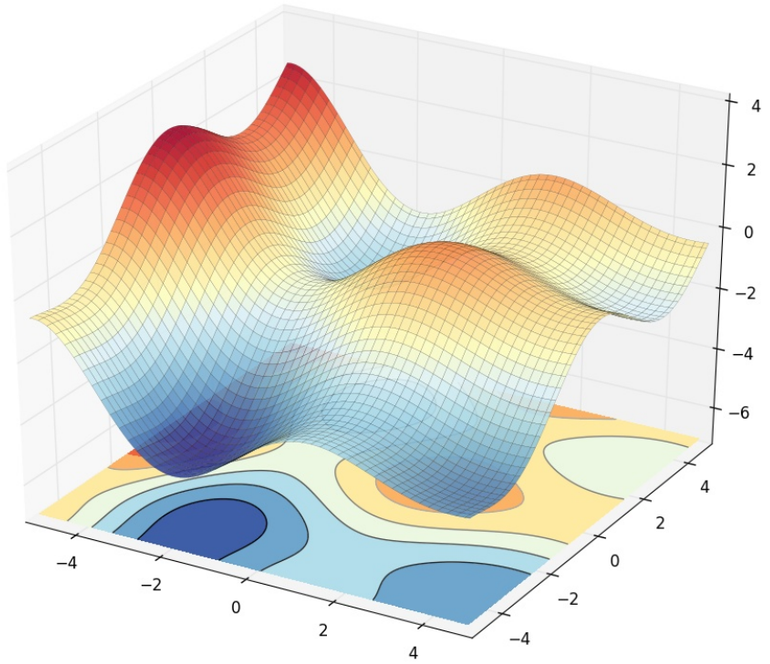
- plot_surface() plots the 3d surface
  - rstride 2 - one gridline every 2 data rows
  - lw - grid line width
  - alpha - surface transparency



3D Plots

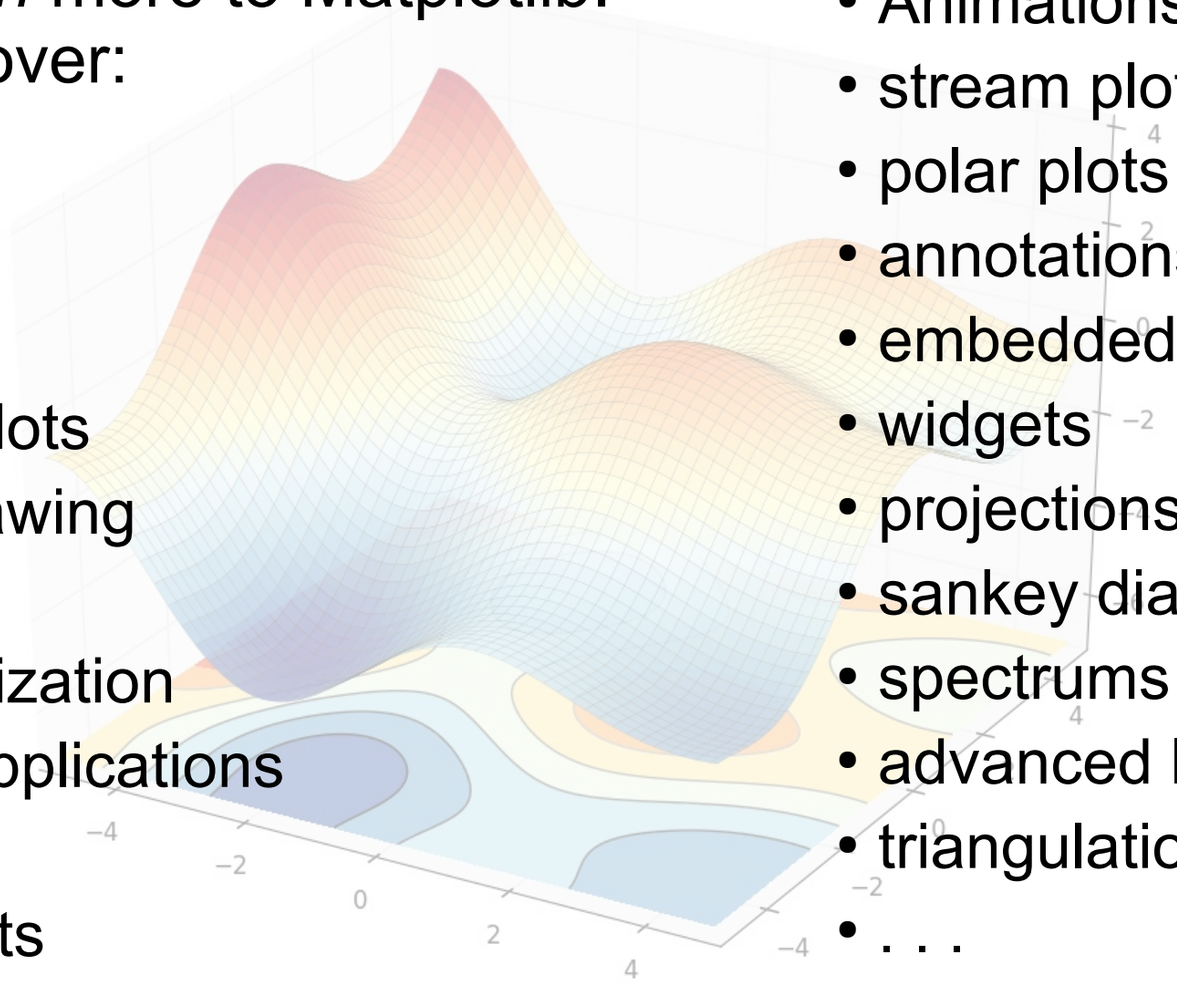
```
f = plt.figure(figsize=(10,8))
f.add_subplot(111, projection='3d')
ax = f.gca()

s=ax.plot_surface(x,y,z,rstride=2,
      cstride=2,lw=0.1,alpha=0.9,
      cmap=plt.cm.RdYlBu_r)

ax.contourf(x,y,z,
      zdir='z',
      offset=-7,
cmap=plt.cm.RdYlBu_r)

ax.contour(x,y,z, zdir='z', offset=-7,
lw=0.5, cmap=plt.cm.gray)

ax.set_zlim((-7,4))
```

- use contour() to plot the 2d contour
  - zdir - which data is "down"
  - offset - where to put the surface
  - set_zlim() set the floor of the graph



Other

- There is *much* more to Matplotlib. We did not cover:
  - Bar charts
  - Pie charts
  - scatterplots
  - parametric plots
  - free-form drawing
  - boxplots
  - image visualization
  - Interactive applications
  - Hinton plots
  - output formats
  - Animations
  - stream plots
  - polar plots
  - annotations
  - embedded plots
  - widgets
  - projections
  - sankey diagrams
  - spectrums and spectograms
  - advanced layouts
  - triangulations
  - . . .



Finally!

please fill in our feedback form!

`https://groups.oist.jp/scs/introduction-python-feedback`

Or go to SCDA → Documentation →
Introduction to Python → Feedback page