ASTR496: FOUNDATIONS OF DATA SCIENCE FOR ASTRO

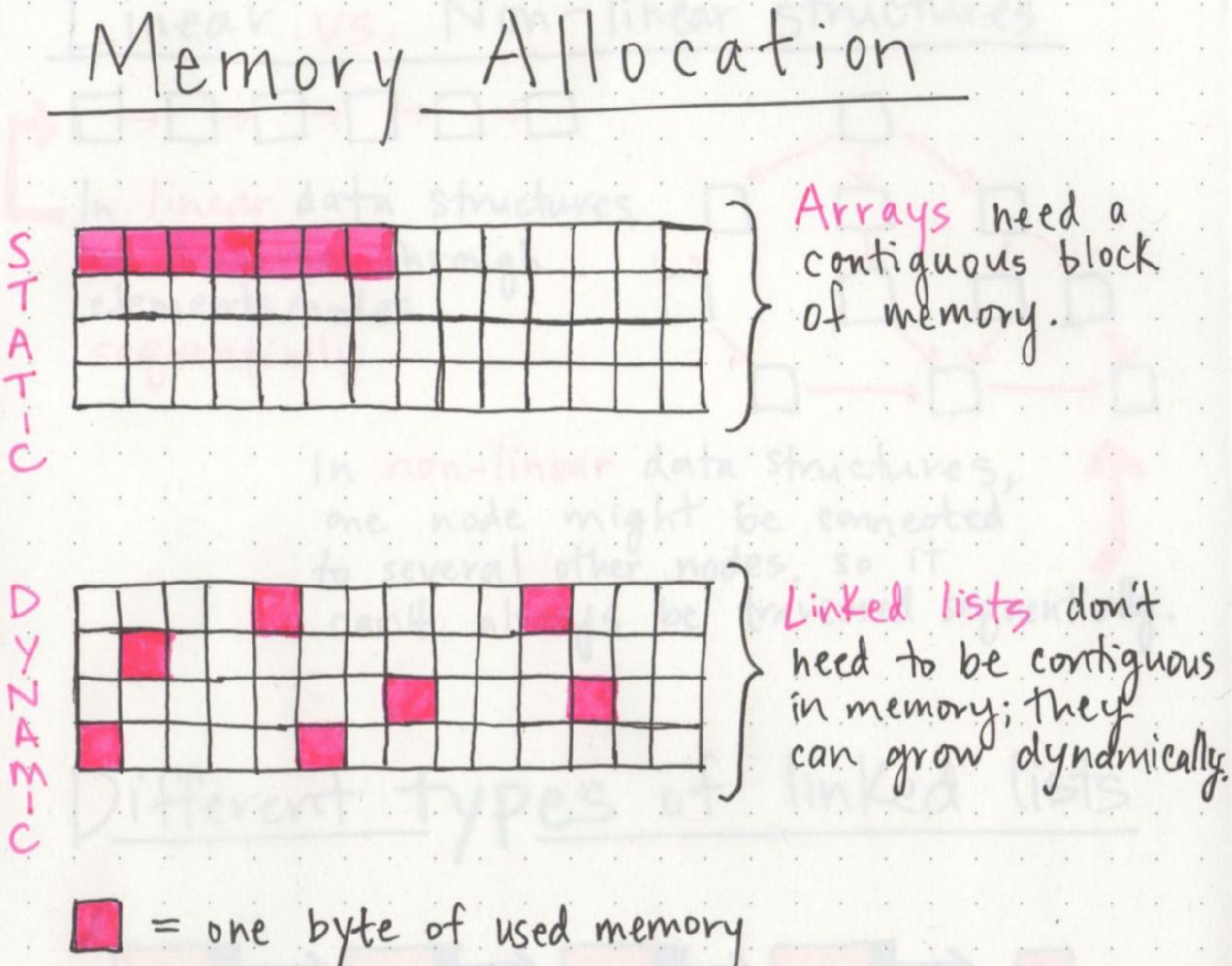
PROF. GAUTHAM NARAYAN

WEEK 1: PYTHON CRASH COURSE

NUMPY

- NumPy is a module designed to make numerical and scientific operations in Python easier and more efficient
- Customarily, you import the NumPy module with renaming as
  `import numpy as np`
- Numpy supports (some of these are functions, some are classes, some are submodules)
  - **Efficient N-dimensional array objects** (`np.ndarray`)
  - Linear algebra (`np.linalg`)
  - Interpolation (`np.interp`)
  - Numerical integration (`np.trapz`)
  - Statistics (`np.mean, np.std, np.percentile etc`)
  - Fourier transforms and signal processing (`np.fft`)
  - Random numbers (`np.random`)

ARRAYS VS LISTS

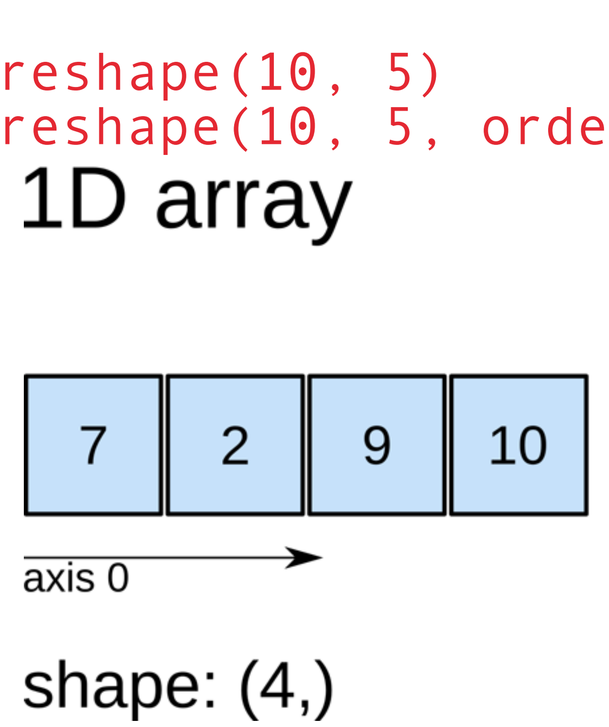


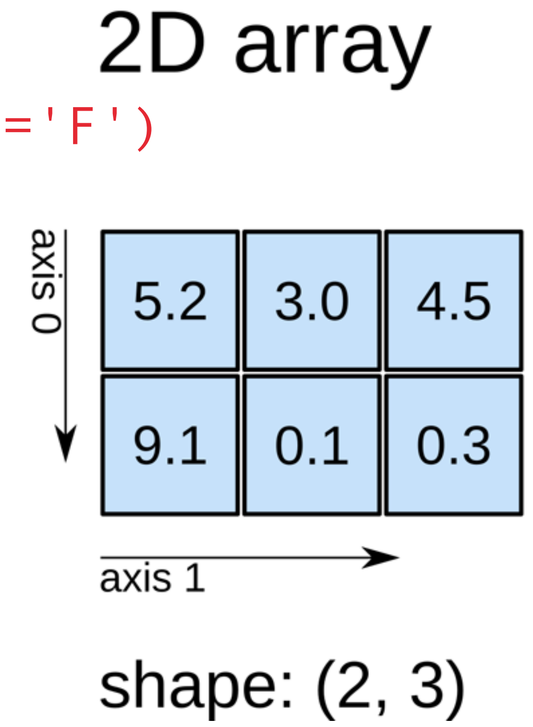
- The most basic unit for operations with numpy is the `ndarray`
- An **array** is a *contiguous* block of memory containing elements of the **same type and size**
- A **list** can contain elements of **different types and sizes**, and elements can be inserted into it or removed from it
- Arrays are more efficient, but less flexible.

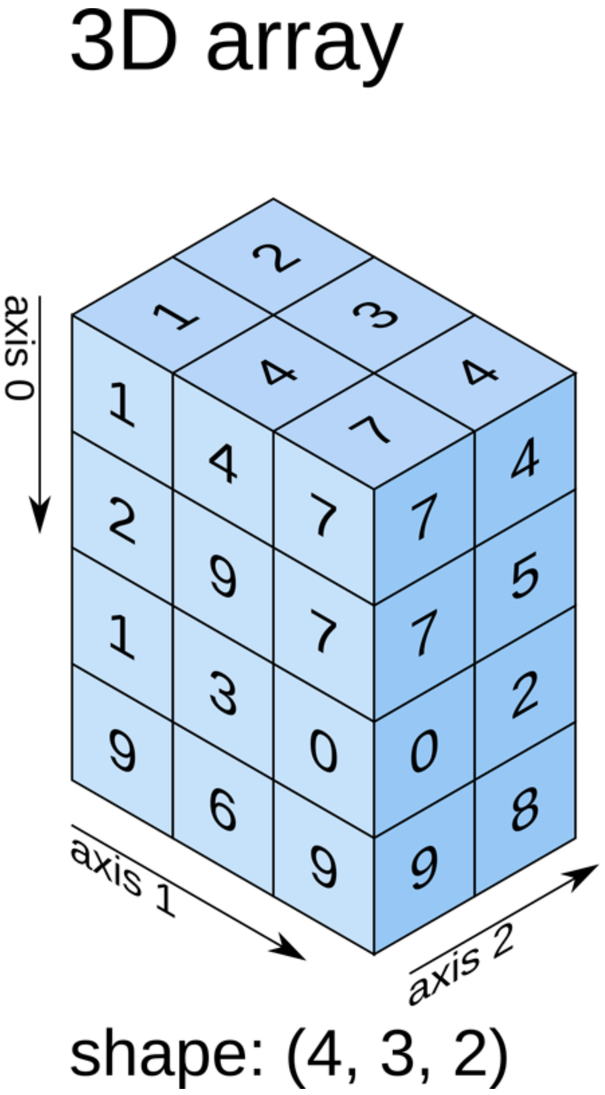
ARRAY PROPERTIES

- **shape:** number and extent of dimensions (`array.size, array.shape`)
- **dtype:** Type of data stored (int, float, complex, etc. `array.dtype`)
- **order:** Order of elements in memory (set when created or reshaped)
  default: “C” (row-major: last element cycles first)
  alternative: “F” (column-major: first element cycles first)
  Compare

```
np.arange(50).reshape(10, 5)
np.arange(50).reshape(10, 5, order='F')
```







CREATING ARRAYS

- Creating an array of a desired shape

```
np.zeros((4, 5))        # array shape (4,5) with zeros
np.ones((4, 5))         # or with ones
np.empty((4, 5))        # or without initalization
```

- Creating an array from a list of values

```
np.array([[20.3, -41.2, 17.8], [31.3, 14.9, 21.0]])
```

- Creating an array spanning a range

```
np.arange(2, 3, 0.1)    # values from 2<=x<3 in 0.1 steps
np.linspace(2, 3, 9)    # 9 equally spaced from 2<=x<3
np.logspace(-2, 7, 20) # now with logarithmic spacing!
```

INDEXING ARRAYS

- Referring to an element

```
a = np.arange(100).reshape(10, 10)
a[9,7]
```

multi-D indexing!!! Can't do that with a list

- Slicing – returns a "**view**" of an array, not a copy

```
a[1:5:2,::3]  # same semantics as lists start:stop:stride
```

- Indexing with index arrays/tuples – returns a copy

```
B, C = np.array([7, 3, 2]), np.array([0, 4, 3])
a[B,C]          # a 1D array containing [a[7,0], a[3,4], a[2,3]]
a[B]            # a 2D array containing [a[7,:], a[3,:], a[2,:]]
D = ((7, 3, 2), (0, 4, 3))
a[D]            # a 1D array containing [a[7,0], a[3,4], a[2,3]]
```

- indexing with a Boolean mask – returns a copy

```
E = (a > 32) & (a < 78)
a[E]            # 1D array with elements of a where E is True
```

BROADCASTING

- Converting scalars or arrays so that they are the same shape when an element-by- element operation is to be done is called **broadcasting**.

```
In [1]: a = np.arange(10).reshape(2,5)

In [2]: b = np.arange(5)

In [3]: c = 6

In [4]: a*b
Out[4]:
array([[ 0,  1,  4,  9, 16],
       [ 0,  6, 14, 24, 36]])

In [5]: a*c
Out[5]:
array([[ 0,  6, 12, 18, 24],
       [30, 36, 42, 48, 54]])
```

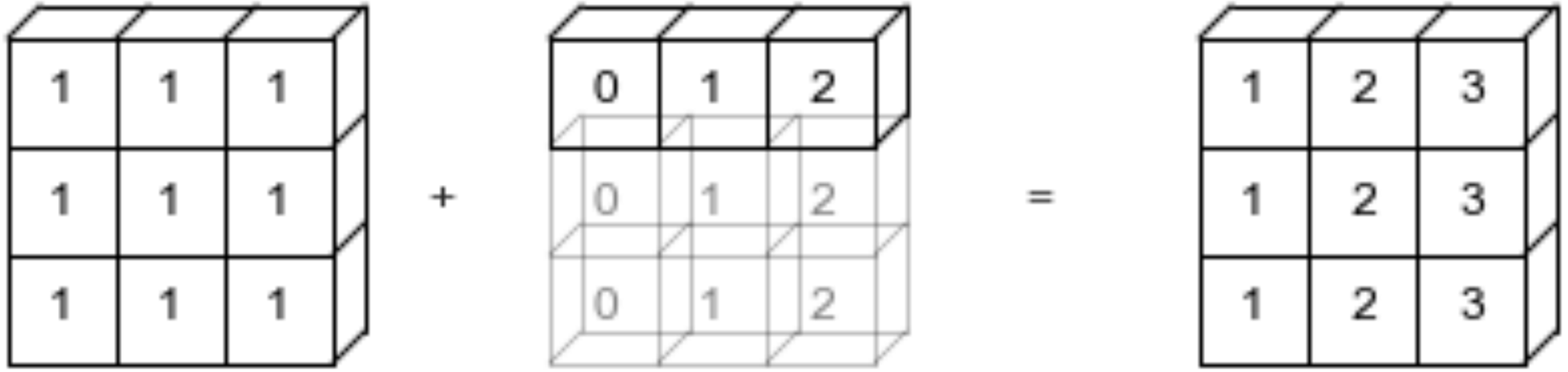
b is treated as (1,5)

c is treated as (1,1)

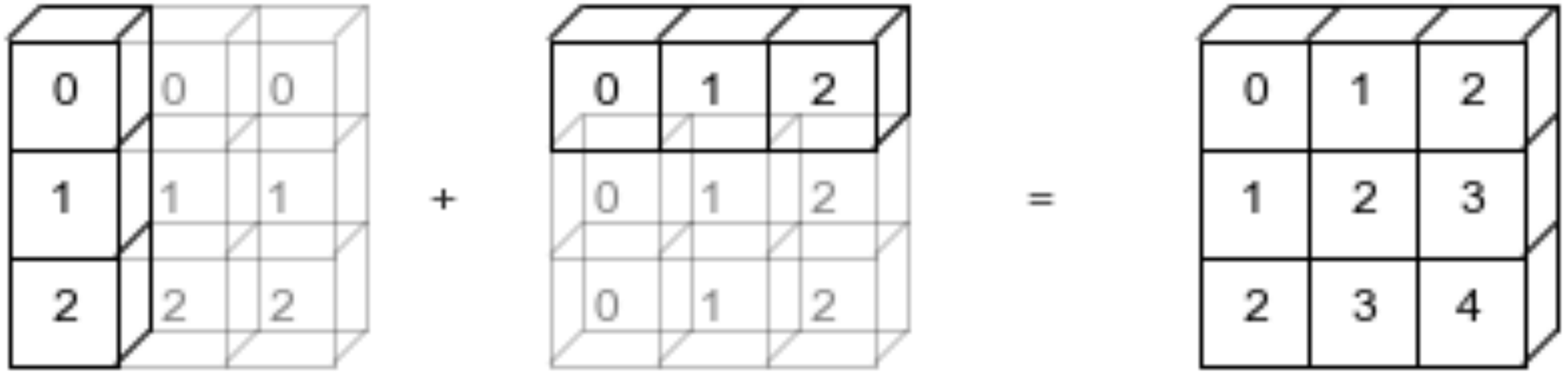
BROADCASTING

$\texttt{np.arange}(3) + 5$

$\texttt{np.ones}((3,3)) + \texttt{np.arange}(3)$



$\texttt{np.arange}(3).\texttt{reshape}((3,1)) + \texttt{np.arange}(3)$



IN CLASS EXERCISE 1: MANIPULATING NUMPY ARRAYS

- Create a **10x5x4** array **A** containing 200 values logarithmically spaced between **10^-3 and 10^+7**. Print **A[3,0,2]**, **A[5,4,1]**, and the **last element of A**.
- Create an array **B** containing

$$B = \begin{bmatrix} -2 & 8 & 10 & 1 \\ 17 & 9 & 2 & 0 \\ 1 & 6 & -4 & 10 \\ 3 & 8 & -9 & 4 \end{bmatrix}$$

  and set all the elements of **B larger than 4 equal to 4**. Print **B**.
- Create a 10x10 array **C** that contains **zeros in all elements for which either index is odd** and ones in all other elements. Print **C**.

```python
import numpy as np

A = np.logspace(-3., 7., 200).reshape(10, 5, 4)
print(A[3,0,2], A[5,4,1], A[-1,-1,-1])


B = np.array([[-2, 8, 10, 1], [17, 9, 2, 0], \
              [1, 6, -4, 10], [3, 8, -9, 4]])
B[B > 4] = 4
print(B)


C = np.zeros((10, 10))
C[::2,::2] = 1.
print(C)
```

WHOLE ARRAY OPERATIONS

element-by-element operation

list methods still work

multiply array by scalar

Numpy has versions of math functions that work on arrays

```
In [1]: x = np.array([1, 2, 3, 4, 5])

In [2]: y = np.array([6, 7, 8, 9, 10])

In [3]: print(x*y)
[ 6 14 24 36 50]

In [4]: print(sum(x*y))
130

In [5]: z = 14

In [6]: print(x*z)
[14 28 42 56 70]

In [7]: print(np.exp(-x))
[0.36787944 0.13533528 0.04978707 0.01831564 0.00673795]
```

DATATYPES IN NUMPY

- NumPy defines a class called `dtype` that provides different data types. In addition to the normal Python classes (which have NumPy equivalents, e.g. np.int):

```
np.int8      8-bit integer        np.int16     16-bit integer
np.int32     32-bit integer       np.int64     64-bit integer

np.float16   16-bit float         np.float32   32-bit float
np.float64   64-bit float         np.float128  128-bit float

np.complex64 two 32-bit floats    np.complex128 two 64-bit floats
np.complex256   two 128-bit floats
```

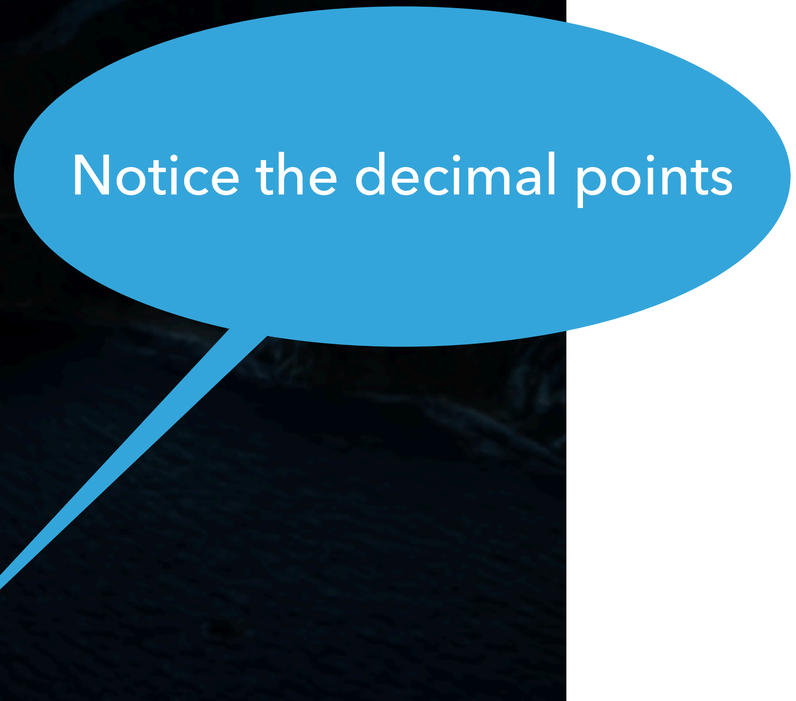
- Some NumPy array-creation routines take an optional argument dtype that determines the data type:

```
np.array([3, 6, 9], dtype=np.complex)
                    # produces array([ 3.+0.j,  6.+0.j,  9.+0.j])
np.arange(2, 10, 2, dtype=np.float)
                    # produces array([ 2.,  4.,  6.,  8.])
```

DATATYPES IN NUMPY

- Since types are objects in NumPy, you can manipulate them inside your program

```
In [1]: def convert(x):
   ...:             newkind = input('Convert to float (f) or double (d)? ')
   ...:             newkind = { 'f': 'float32', 'd': 'float64' }[newkind]
   ...:             return np.cast[newkind](x)
   ...:

In [2]: a = np.arange(10)

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

In [4]: a.dtype
Out[4]: dtype('int64')

In [5]: convert(a)
Convert to float (f) or double (d)? d
Out[5]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```



THE KEY RULE OF NUMPY ARRAYS

```
In [1]: A = np.arange(100000)

In [2]: B = np.random.rand(100000)

In [3]: %timeit for i in np.arange(0, len(A), 2): A[i] = A[i] + B[i]
116 ms ± 1.96 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [4]: %timeit A[::2] = A[::2] + B[::2]
86.2 µs ± 3.3 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

notice the prefix!!

Do not iterate with loops if you can help it.

ND-ARRAYS, MESHGRIDS

```
In [1]: x = np.linspace(0., 2.*np.pi, 10001)

In [2]: y = np.zeros(x.shape)

In [3]: %timeit for i in range(10001): y[i] = np.sin(x[i])
12.2 ms ± 2 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [4]: %timeit y = np.sin(x)
85.2 µs ± 1.3 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

- What if you want to iterate over a **multidimensional** grid?

Three 3D arrays from 0-3, 0-4, 0-5
w/ 10 samples each

```
In [1]: z, y, x = np.mgrid[0:3:10j, 0:4:10j, 0:5:10j]

In [2]: r = np.sqrt(x**2 + y**2 + z**2)
```

ND-ARRAYS, MESHGRIDS

- You can also make meshes using `meshgrid`:

```
In [1]:  x = np.linspace(0., 10., 5)
   ...:  y = np.linspace(0., 20., 10)
   ...:  xx, yy = np.meshgrid(x, y)
```

This is better than `mgrid` when your mesh spacings are irregular

- Can also rely on broadcasting if you have a function of (x,y) to evaluate at the mesh points:

```
In [2]: z = x * y.reshape(10, 1)

In [3]: z2 = x * y[:,np.newaxis]
```

`newaxis` creates a dummy axis of length 1 wherever you want.

RANDOM NUMBERS – NUMPY.RANDOM

```
In [1]: import numpy.random as npr

In [2]: npr.rand(2,3,4)
Out[2]:
array([[[0.49788492, 0.2762824 , 0.03427953, 0.24141776],
        [0.094409  , 0.9169308 , 0.15206474, 0.9088985 ],
        [0.28938606, 0.73952203, 0.60718676, 0.71310171]],

       [[0.79350954, 0.64027908, 0.39693911, 0.13895758],
        [0.16018318, 0.26148142, 0.67456118, 0.85441903],
        [0.62835152, 0.88403737, 0.20142244, 0.16059226]]])

In [3]: npr.randint(5, 18, 10)
Out[3]: array([ 9,  7, 12, 12, 17, 15,  6, 11,  5, 13])

In [4]: npr.normal(0, 1, 12)
Out[4]:
array([ 0.30886622,  0.44317973,  0.19026378, -0.21285009,  0.81316372,
       -0.52248537, -0.48294371, -2.17994937,  0.9157902 ,  0.15418031,
       -1.50098894, -0.33141593])
```

2x3x4 array of floats in [0,1)

10 random integers between 5 and 18

12 random numbers from a normal distribution w/ mean=0 std=1

SORTING AND SEARCHING

Sorting:

```
a = np.array([[3, 5, -1], [8, 12, 20]])
np.sort(a, axis=1)          # return copy of a sorted along axis 1
a.sort(axis=0)              # in-place sort of a along axis 0
np.argsort(a)               # return indices that would sort a along axis 0
```

Searching:

```
np.argmax(a, axis=1)        # index of max value along axis 1
np.nanargmax(a, axis=1).    # same ignoring NaNs
np.nonzero(a)               # indices of non-zero elements of a
np.where(a > 0)             # indices where a > 0
np.where(a > 0, np.log(a), 0.)    # if a > 0 returns np.log(a) else 0
```

In where, the three arguments are expected to have the same rank; if they don't, they are broadcast as necessary (in this case 0. is broadcast to a 2x3 array) Note: and/or/etc. test the whole array, not element-by-element; use, e.g., `np.where(((a>0) & (b<0))|(c>0))` instead.

IN CLASS EXERCISE 2: RANDOM NUMBERS, MESHGRIDS AND WHOLE ARRAY OPS

- Construct two **random**, **uniformly** sampled 1D coordinate arrays **x** and **y**, each with 100 samples ranging from 0 to 1, and **sort** them.

- Construct 2D **meshgrids** from **x** and **y**. Use the meshgrids to construct the 2D uniformly sampled function

$$z(x, y) = \exp\left[-\frac{x^2 + y^2}{0.05}\right]$$

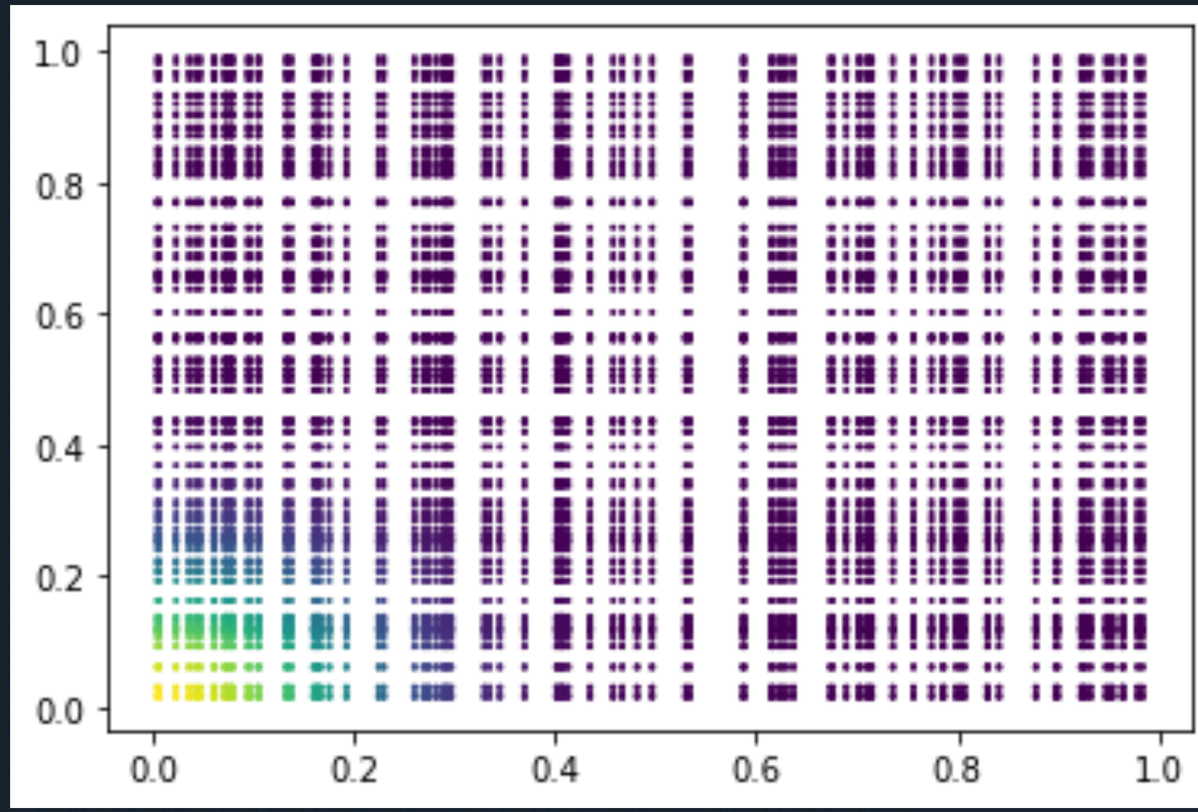
- NumPy provides a `histogram` function that we can use to find the frequency of occurrence of different values of z. The most common usage is

```
hist, bin_edges = np.histogram(z, bins=10)
```

where the bins argument supplies the number of bins (or a list of bin right-edge values), **hist** contains the histogram counts, and **bin_edges** contains the bin edges (its length is len(hist) + 1).

- Use the histogram function to construct a histogram of z values with bin width 0.1, and print the counts in a table like **[bin left edge] [bin right edge] [counts in bin]**

```python
import numpy as np
import numpy.random as npr

x = np.sort(npr.rand(100))
y = np.sort(npr.rand(100))

xx, yy = np.meshgrid(x, y)
z = np.exp(-(xx**2 + yy**2)/0.05)


nb = int((z.max() - z.min())/0.1) + 1
hist, edges = np.histogram(z, bins=nb)

for i in range(len(hist)):
    print(f"{edges[i]:4.1f}  {edges[i+1]:4.1f}    {hist[i]:4d}")
# visualize the meshgrid

import matplotlib.pyplot as plt
plt.scatter(xx, yy, c=z, s=1)
plt.show()
```



SORTING AND SEARCHING

Sorting:

```
a = np.array([[3, 5, -1], [8, 12, 20]])
np.sort(a, axis=1)          # return copy of a sorted along axis 1
a.sort(axis=0)              # in-place sort of a along axis 0
np.argsort(a)               # return indices that would sort a along axis 0
```

Searching:

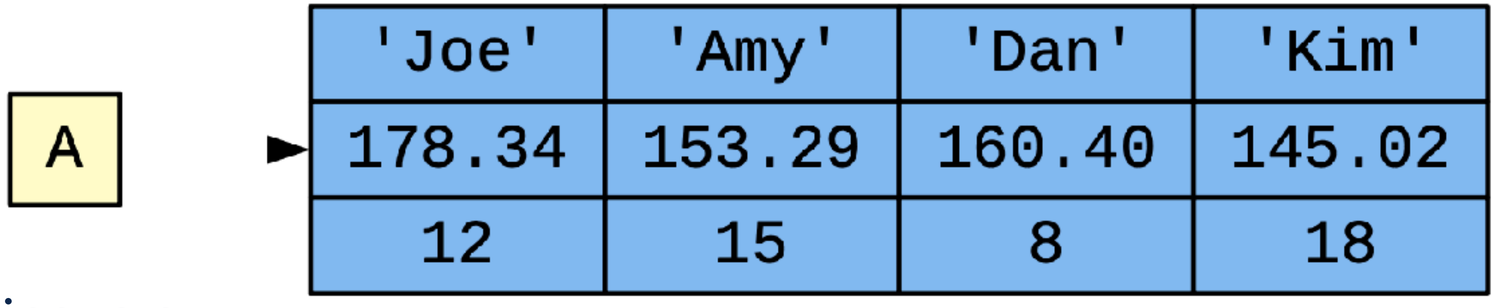
```
np.argmax(a, axis=1)        # index of max value along axis 1
np.nanargmax(a, axis=1).    # same ignoring NaNs
np.nonzero(a)               # indices of non-zero elements of a
np.where(a > 0)             # indices where a > 0
np.where(a > 0, np.log(a), 0.)    # if a > 0 returns np.log(a) else 0
```

In where, the three arguments are expected to have the same rank; if they don't, they are broadcast as necessary (in this case 0. is broadcast to a 2x3 array) Note: and/or/etc. test the whole array, not element-by-element; use, e.g., `np.where(((a>0) & (b<0))|(c>0))` instead.

RECORD DATA TYPES

- Group data together in an array using structures. Example:
  `A=np.empty((10,), dtype=[('name','S3'), ('height','f8'), ('score','i4')])`

  produces a 1D array of length 10 elements. Each contains a 3-character string name, a double-precision float height, and an integer score.



- Can reference using, e.g.,
  `A[2]['name'] = 'Dan'`
  `A[:4]['score'] = [12, 15, 8, 18]`

- Example type descriptors: '**b**' (Boolean), '**S**' (string), '**i**' (integer), '**f**' (float), '**c**' (complex) ... can also use string name of a NumPy data type or a dtype object (e.g. **np.float32**). Can also specify a shape for each field!

RECORD DATA TYPES

- Example:

```
A = np.empty((10,), dtype=[('id', np.int), ('obs', 'float32',
(2,2))])
```

Each element contains:

id: an integer

obs: a 2x2 array of 32-bit floats

- You can manipulate the structure dtype directly using

```
d = A.dtype    # which is equivalent to
d = np.dtype([('id', np.int), ('obs', 'float32', (2,2))])
```

- and get information about it:

```
d.names  # tuple of field names ('id', 'obs')
d.fields # dictionary-like object with keys given by field
         # names and values given by tuple of type and byte offset
```

READING RECORD DATA FROM A FILE

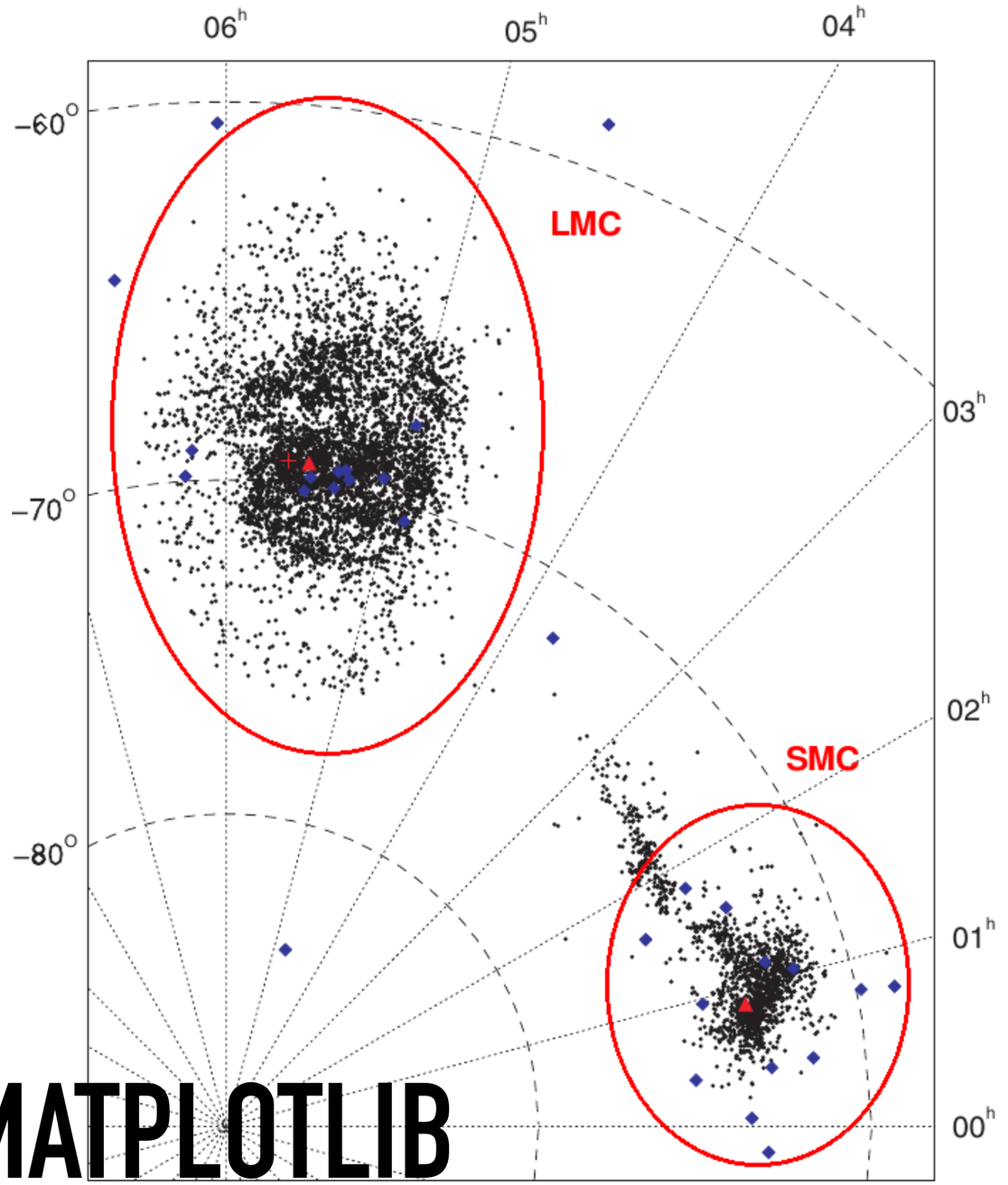
- `genfromtxt` reads tabulated data from a text file and correctly handles missing values, skips comments, assigns column names, etc. e.g:

```
data=np.genfromtxt('mydata.dat',\
skip_header=3, names=True,\
usecols=(0, 2))
print(data['t'], data['energy'])
```

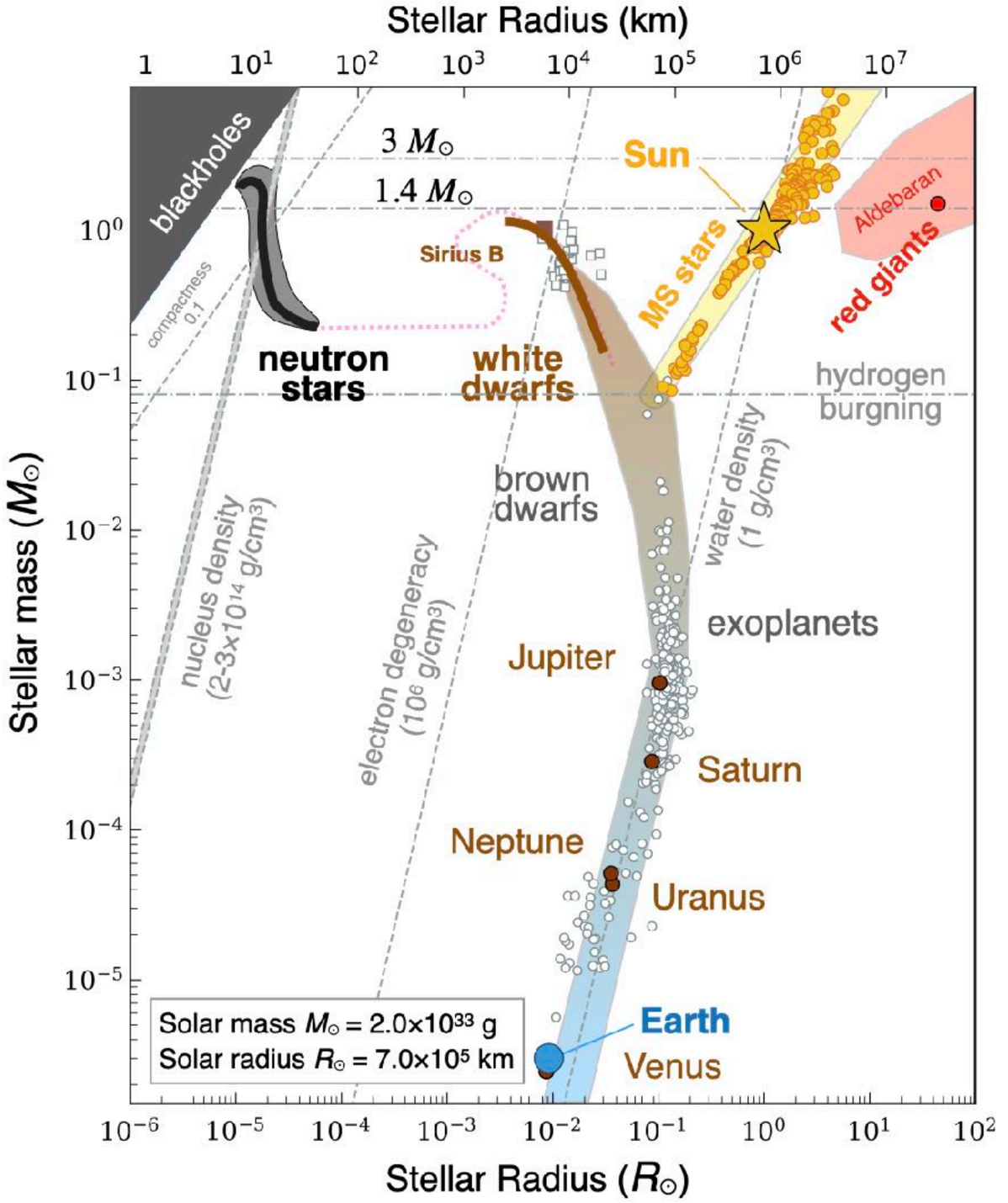
- Can supply a value to use when columns are missing data, change delimiter character, comment character, have columns return different data types, etc. If your file has no missing data, you can use `loadtxt` instead

```
# Tabulated data from run 23
# 2/21/19

# t          mass          energy
0.0          12.312        4.163
1.5          11.973        3.982
2.0          10.691        3.811
2.5          10.101        3.552
# end of data
```

01-Numpy.key
02-Matplotlib.key


06h
05h
04h
−60°
−70°
−80°
03h
02h
01h
00h
LMC
SMC

MATPLOTLIB

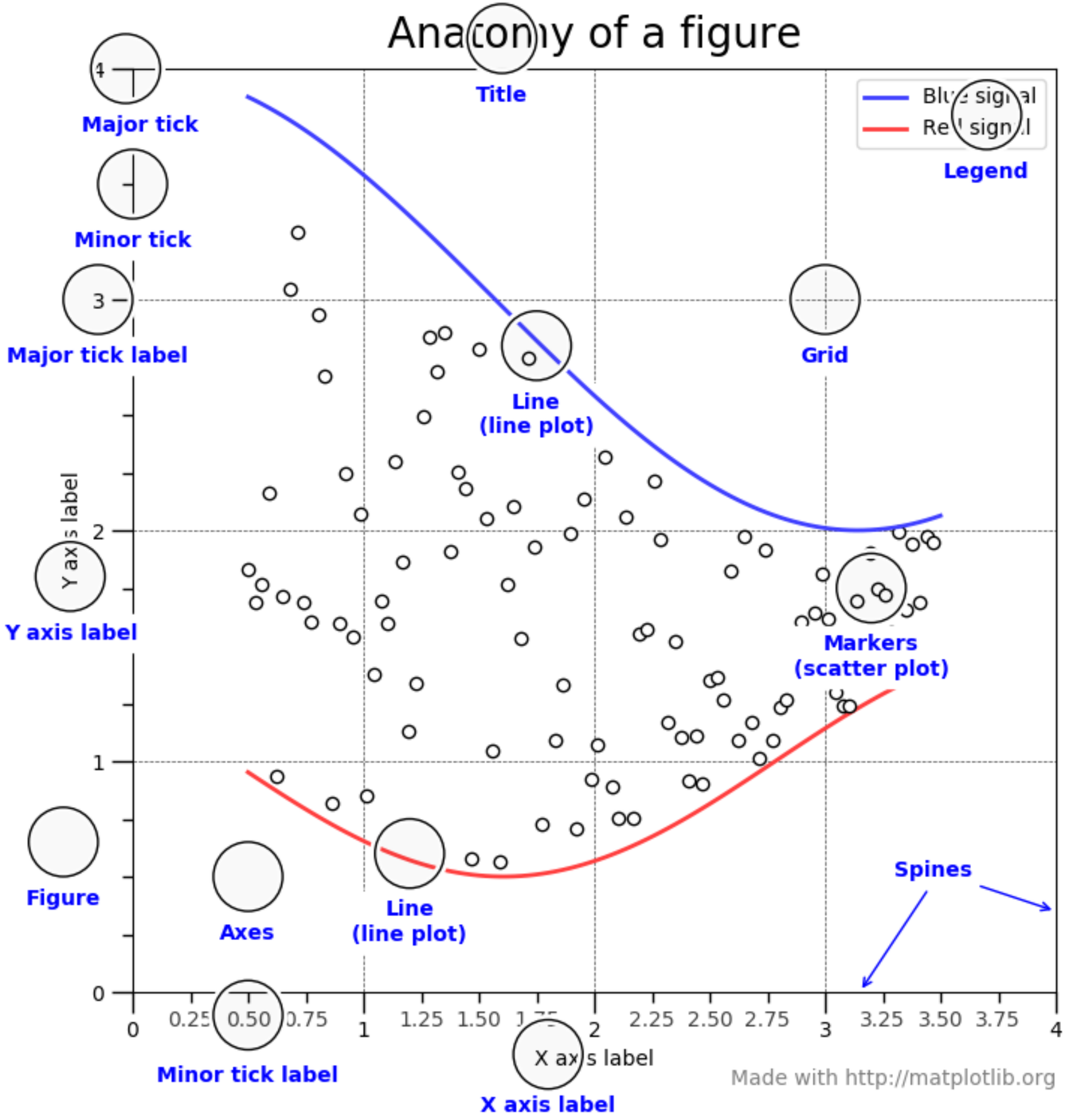

Stellar Radius (km)
1
10¹
10²
10³
10⁴
10⁵
10⁶
10⁷
blackholes
3 M⊙
1.4 M⊙
Sun
Sirius B
MS stars
Aldebaran
red giants
compactness 0.1
neutron stars
white dwarfs
hydrogen burgning
brown dwarfs
nucleus density (2-3x10¹⁴ g/cm³)
electron degeneracy (10⁶ g/cm³)
water density (1 g/cm³)
exoplanets
Jupiter
Saturn
Neptune
Uranus
Earth
Venus
Solar mass M⊙ = 2.0x10³³ g
Solar radius R⊙ = 7.0x10⁵ km
Stellar mass (M⊙)
Stellar Radius (R⊙)

MATPLOTLIB HELP

- Official Matplotlib tutorial https://matplotlib.org/stable/tutorials/index.html
- Matplotlib examples – often the best way to find out how to do what you want https://matplotlib.org/stable/gallery/index.html
- Pyplot interface tutorial https://matplotlib.org/stable/tutorials/introductory/pyplot.html
- Nicolas Rougier's tutorial https://www.labri.fr/perso/nrougier/teaching/matplotlib/
- Python4astronomers Matplotlib tutorial https://python4astronomers.github.io/plotting/matplotlib.html
- Astro Plot of the Week: https://twitter.com/plotastro
- Guidance for choosing plot colors and styles to accommodate color blindness https://towardsdatascience.com/two-simple-steps-to-create-colorblind-friendly-data-visualizations-2ed781a167ec

WHAT IS MATPLOTLIB

- A plotting library for Python that makes it easy to generate and manipulate 2D and 3D plots
- Two main ways to use Matplotlib:
  - **Object-oriented**: use Matplotlib routines to create objects representing figures, axes, plots, annotations, etc., whose properties we can modify using the objects' associated methods
  - **Interactive:** Using the `matplotlib.pyplot` convenience functions: provides a “stateful” interface to Matplotlib (commands change the current state) - use within Jupyter notebooks and at the ipython shell
- Often we will need to switch between these modes.
  - For example, you could use the pyplot interface to create a plot, then use the `gca()` (“get current axes”) function to get a pointer to the Axes object for this plot, then use that object’s methods to modify the positioning or style of tick marks.


Anatomy of a figure
Title
Major tick
Minor tick
Major tick label
Legend
Grid
Line
(line plot)
Markers
(scatter plot)
Y axis label
Figure
Axes
Line
(line plot)
Spines
Minor tick label
X axis label
Made with http://matplotlib.org

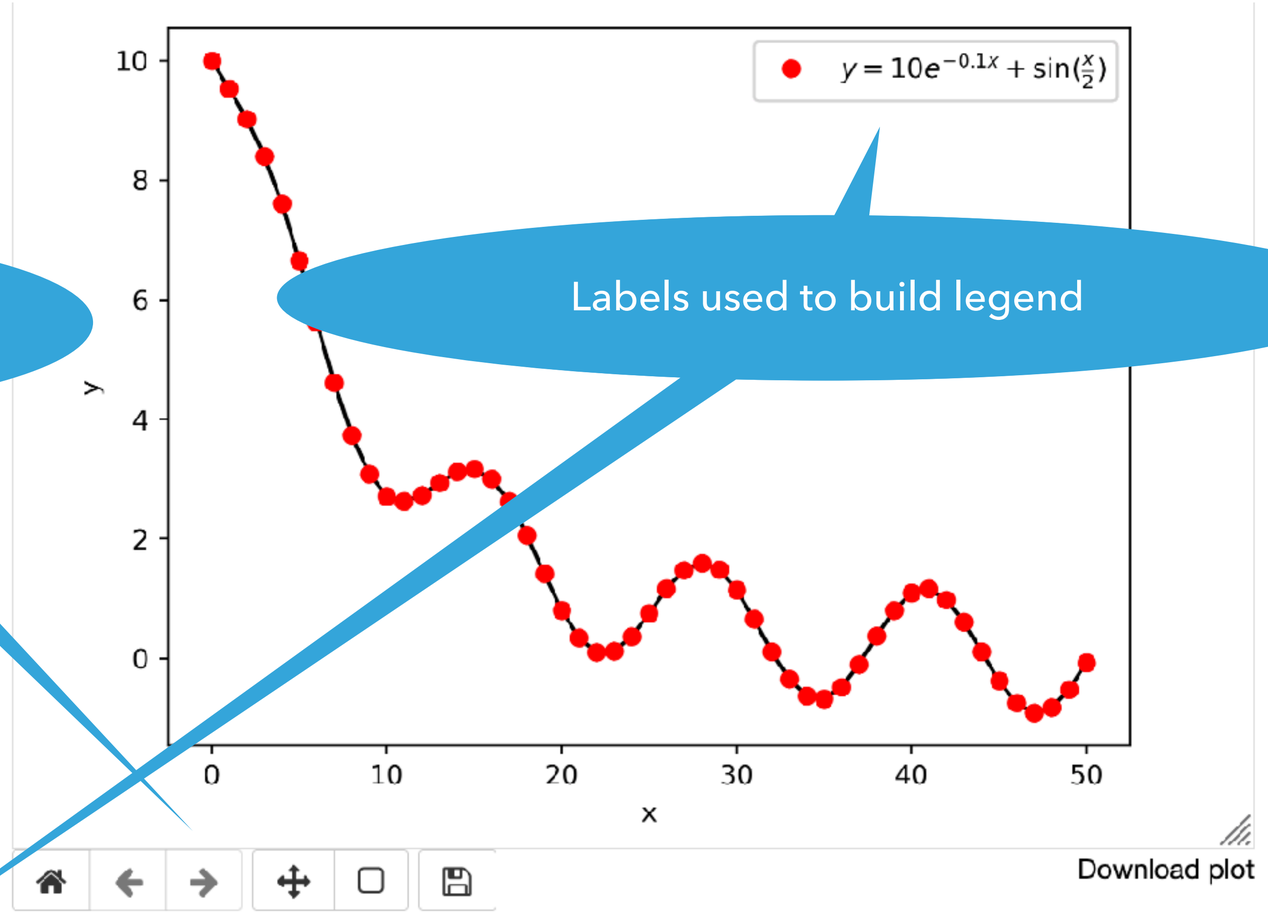
AN EXAMPLE OF CREATING A PLOT FROM SCRATCH

Only needed in a jupyter notebook

Widget controls - reset, prev, next, pan, zoom, save

Labels used to build legend

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


x = np.arange(0, 51, 1)
y = 10*np.exp(-0.1*x) + np.sin(0.5*x)

plt.plot(x, y, 'k-')
plt.plot(x, y, 'ro', label=r'$y = 10e^{-0.1x} + \sin(\frac{x}{2})$')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Format specification

INTERACTIVE WORK WITH MATPLOTLIB

- If you run that code at the python/ipython prompt you'll find that `plt.show()` creates a window but then prevents further code from executing while the window is open: **blocking**

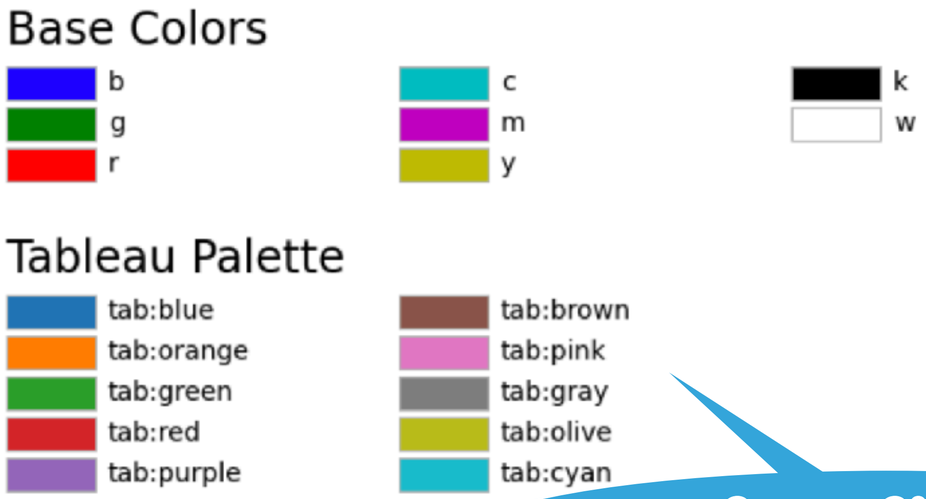
- It's possible to create plot windows that don't block the terminal

Implicit x values

```
plt.plot([1, 2, 3, 4, 5, 6])
plt.show(block=False)          # show plot and return immediately
plt.ylabel('new label')        # oops! forgot to add y-label
plt.draw()                     # updates current plot window
plt.plot([2, 4, 5, 3, 1, 0])   # add a new plot
plt.draw()                     # update plot window
```

- Inside a Jupyter session, use the `%matplotlib notebook` (or `%matplotlib inline`) magic command to cause Matplotlib plots to be displayed inside the Jupyter session. These will not block input.

SPECIFYING LINE/POINT STYLES



Can use C0..C9 to refer to these colors

Multiple sets of x-, y-values & format codes can be specified on one line:

```
plt.plot(x, np.sin(x), ‘ro',\
         x, np.cos(x), 'gx')
```

(sin(x) using red dots and cos(x) using green x's)

Code	Style	Code	Style	Code	Style
'-'	solid line	'- -'	dashed line	'- .'	dash-dot
':'	dotted line	'.'	point	','	pixel
'o'	circle	'v'	triangle down	'^'	triangle up
'<'	triangle left	'>'	triangle right	's'	square
'p'	pentagon	'*'	star	'h'	hexagon
'+'	plus	'x'	x	'D'	diamond

MORE LINE PROPERTIES

- The plot command actually returns a list of `Line2D` objects, one per curve. Using methods supplied by the `Line2D` class, we can modify the properties of these curves.

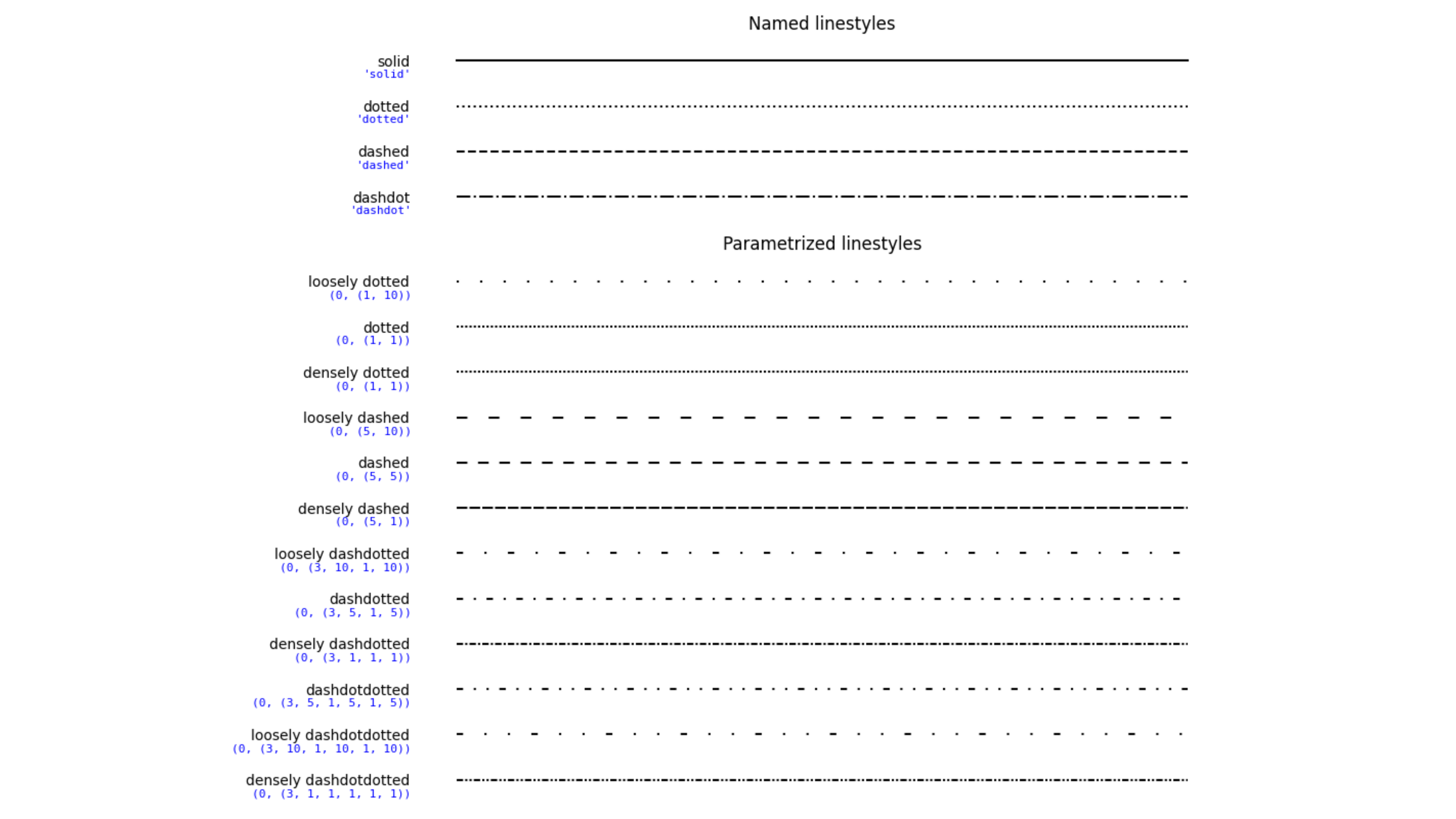
```
line1, line2 = plt.plot(x, np.sin(x), x, np.cos(x), linewidth=2.0)
line1.set_linewidth(2.0)
plt.setp(line2, color='r', linewidth=2.0)
```

`setp()` can also operate on a list of `Line2D` objects

- Some available properties:

```
color   # color specification - HTML code, RGB triplet, any CSS name
xdata
ydata
label
linestyle          # can also specify line styles in a parametric form
linewidth
marker
markeredgewidth
markeredgecolor
markerfacecolor
markersize
visible            # true or false
```

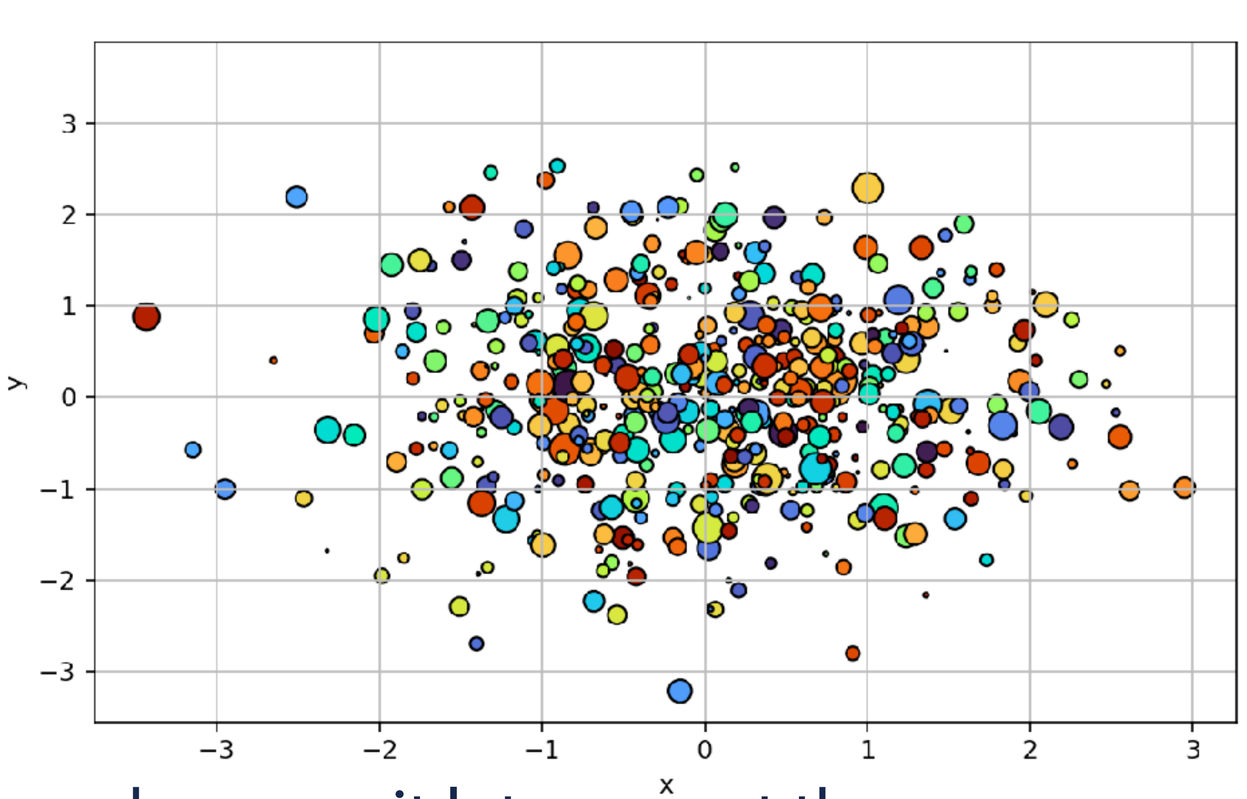
Use plt.get() to list the current properties of any Matplotlib object


Named linestyles
solid
'solid'
dotted
'dotted'
dashed
'dashed'
dashdot
'dashdot'
Parametrized linestyles
loosely dotted
(0, (1, 10))
dotted
(0, (1, 1))
densely dotted
(0, (1, 1))
loosely dashed
(0, (5, 10))
dashed
(0, (5, 5))
densely dashed
(0, (5, 1))
loosely dashdotted
(0, (3, 10, 1, 10))
dashdotted
(0, (3, 5, 1, 5))
densely dashdotted
(0, (3, 1, 1, 1))
dashdotdotted
(0, (3, 5, 1, 5, 1, 5))
loosely dashdotdotted
(0, (3, 10, 1, 10, 1, 10))
densely dashdotdotted
(0, (3, 1, 1, 1, 1, 1))

SCATTER PLOTS

- You can generate scatter plots using `plot()` or using `scatter()`:

```
x = np.random.randn(1000)
y = np.random.randn(1000)
colors = np.random.rand(1000)
sizes = 50*np.random.randn(1000)
plt.scatter(x, y, s=sizes, c=colors,\
 cmap='turbo', edgecolor='black')
plt.grid()
ax = plt.gca()
ax.set_xlabel('x')
ax.set_ylabel('y')
fig = plt.gcf()
title = fig.suptitle('blah', fontsize=20)
title.set_fontname('Arial')
title.set_fontstyle('oblique')
title.set_color('C1')
```



- `scatter()` is more flexible for this case because it lets you set the characteristics of each point separately.

- You can also specify the dot style using the `marker` argument. This takes a string code with the same meanings as used for `plot()`

IN CLASS EXERCISE 1: READING FILES

- **Download** the file "**L14_halos.dat**" from Github if you've not git pulled yet. This is a text file containing x, y, z positions and masses for dark matter halos in a cosmological simulation. Positions are in Mpc and masses are in M⊙.

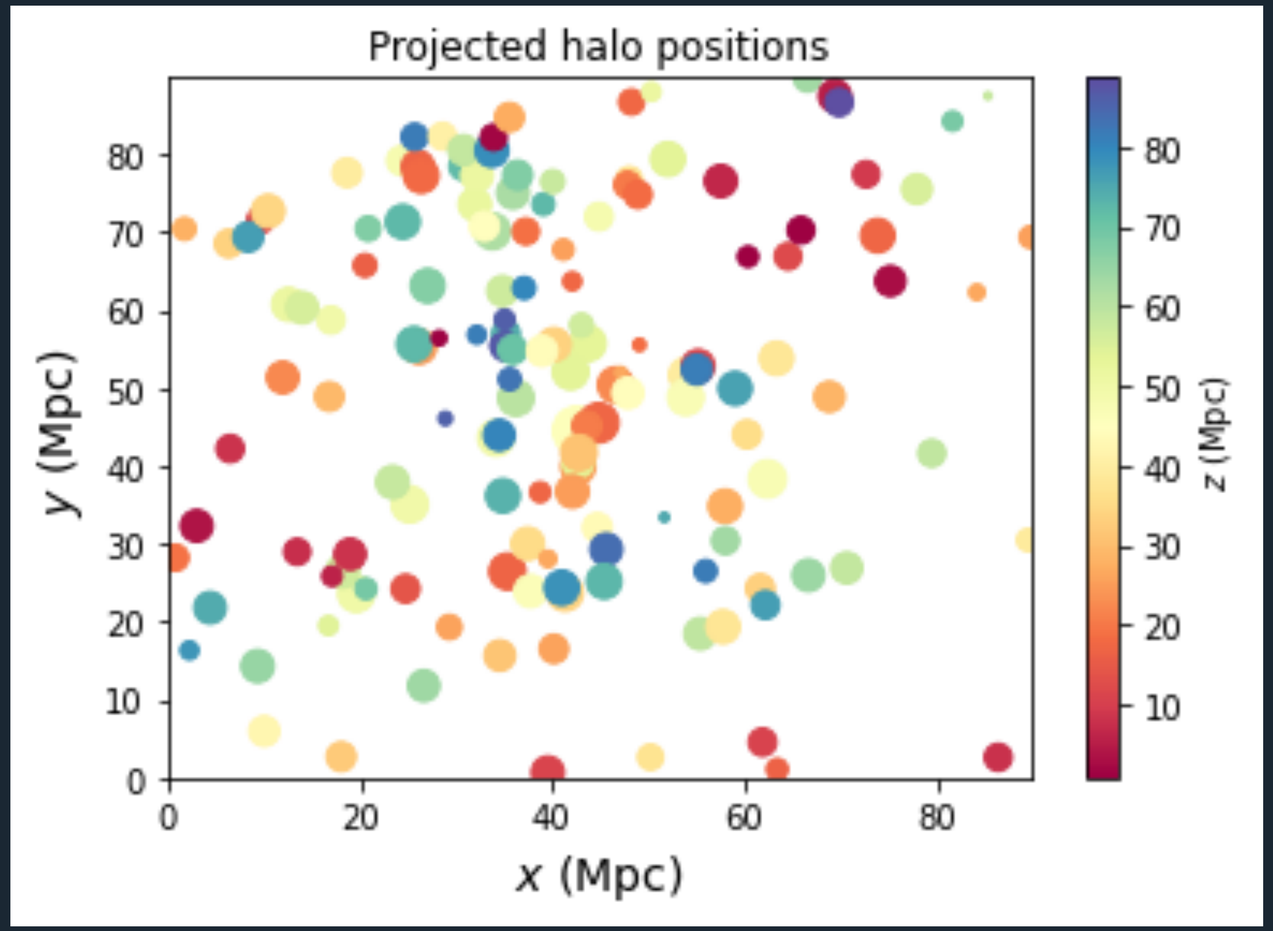
- **Create a scatter plot** of the (x,y) coordinates of the halos. Represent each halo by a circle whose **size is proportional to the logarithm of the halo's mass**:

$$\text{size} = 70[\log(M/M_\odot) - 11.5]$$

- Using the "Spectral" Matplotlib color map, **make the color of each circle proportional to the z-coordinate** of the corresponding halo. (You can change the color map using `plt.set_cmap('Spectral')` or as an argument to `scatter()`.)

- Include an appropriate title and axis labels.

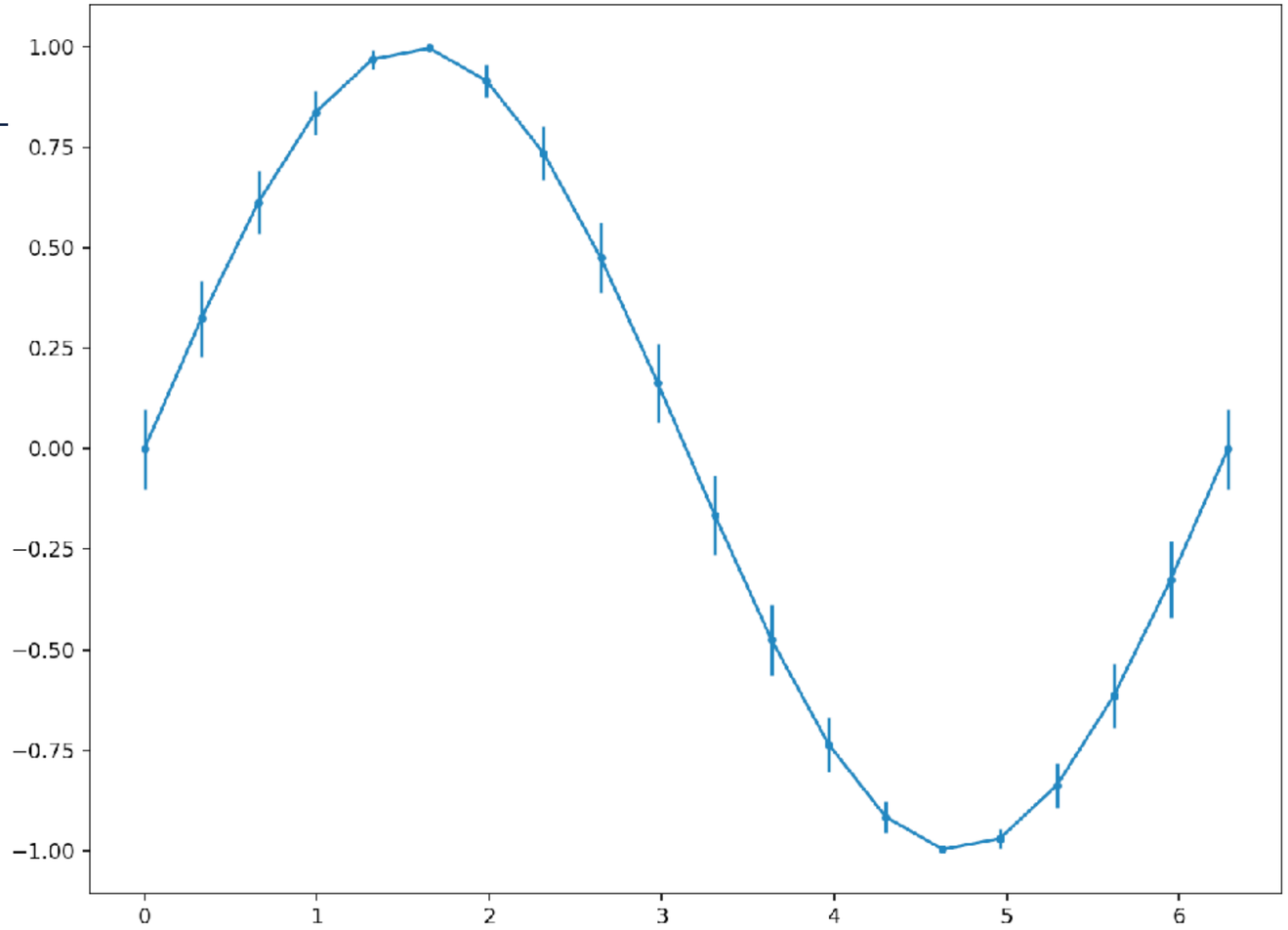
```python
import numpy as np
import matplotlib.pyplot as plt

x, y, z, m = np.loadtxt('L14_halos.dat', unpack=True, skiprows=3)

logm  = 70*(np.log10(m) - 11.5)

plt.scatter(x, y, s=logm, c=z)
plt.set_cmap("Spectral")

# we'll discuss colorbars next time
cbar = plt.colorbar()
cbar.set_label(r'$z$ (Mpc)')

plt.title('Projected halo positions')
plt.xlabel(r'$x$ (Mpc)', fontsize=14)
plt.ylabel(r'$y$ (Mpc)', fontsize=14)

plt.xlim(0., 89.8)
plt.ylim(0., 89.8)
plt.show()
```



ERROR BARS

- Use `errorbar()` to add error bars to the current plot:

```
x = np.linspace(0, 2*np.pi, 20)
y = np.sin(x)
y_errors = np.abs(0.1*np.cos(x))
plt.plot(x, y)
plt.errorbar(x, y,\
 yerr=y_errors, fmt='.-)
```



- An optional `xerr` argument can be used to specify error bars in x.
- You can specify `ecolor` and `elinewidth` (defaults are the same as the values used for the plot itself). The `capsize` argument gives the size of the error bar caps in pixels.
- For asymmetrical error bars, pass a list of error arrays:

```
plt.errorbar(x, y, yerr=[ neg_y_errors, pos_y_errors ], fmt='.-')
```

AXIS LIMITS AND TICK MARKS

- Use `axis()` to set the axis limits for the current plot:

```
plt.axis([xmin, xmax, ymin, ymax])
```

- Alternately

```
plt.xlim([xmin, xmax])
plt.ylim([ymin, ymax])
```

- Also,

```
plt.axis('off')      # turn off axis lines and labels
plt.axis('equal').   # make axis scaling square - same scale on x & y
plt.axis('tight').   # adjust limits to include all data
```

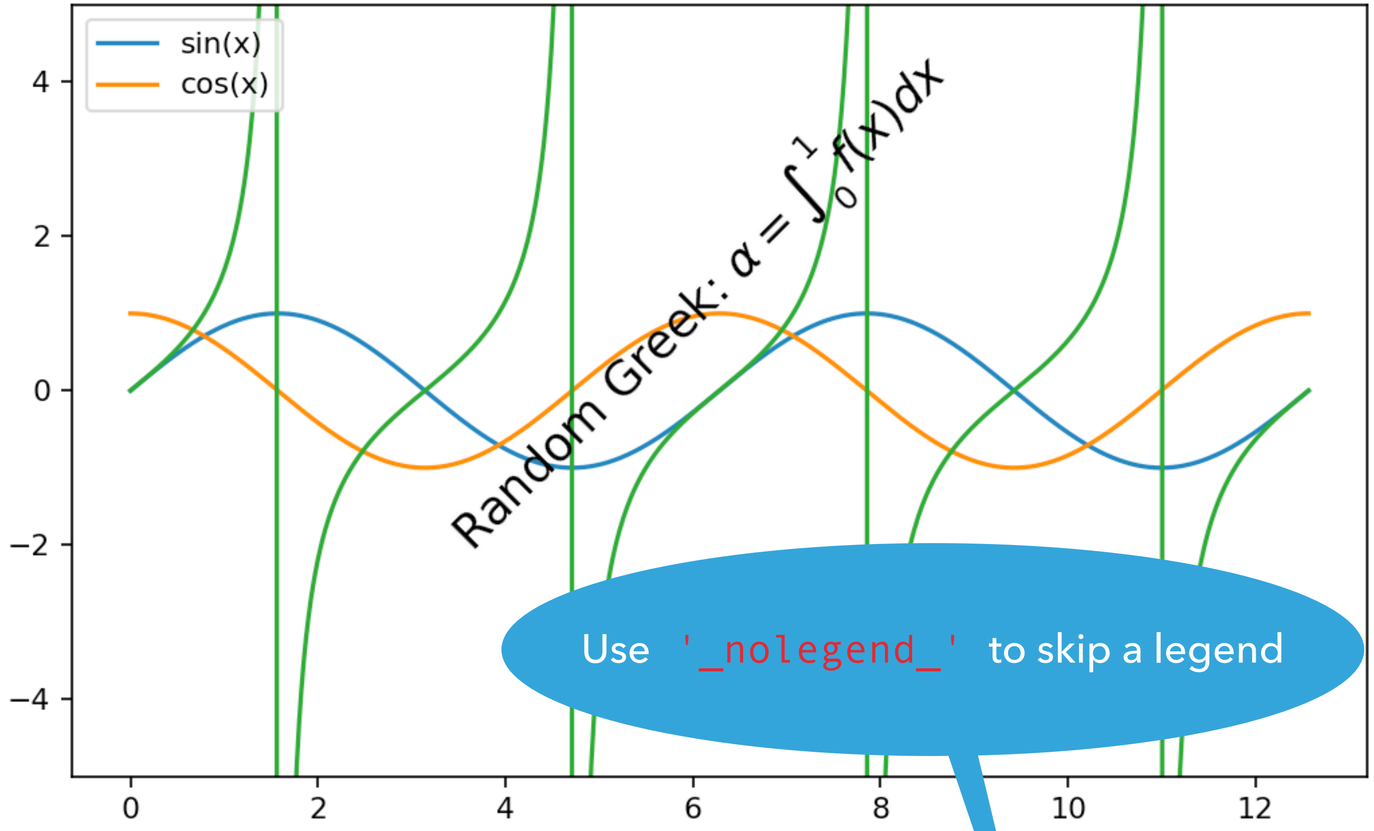
- Control the placement of tick marks using `xticks()` and `yticks()`

```
plt.xticks([0, 1, 2, 3, 4], ['a', 'b', 'c', 'd', 'e'])
```

These take two lists: the values at which to place ticks, and the labels to use. Other text attributes (e.g. `rotation`, `font`) can be given as keyword arguments.

TEXT ON FIGURES

- Remember Python 'raw string' – prefixed with a ‘r’ - escape expressions (e.g., '\n') are not interpreted
- The string can contain LaTeX !! (have to use raw strings)
- Some useful arguments: `rotation`, `color`, `fontsize`, `fontname`, `fontstyle`, `fontweight`, `horizontalalignment`, `verticalalignment`
- Create a legend for your plot using `legend()`. With no arguments, it will construct a legend using the `label` attributes of your plots.
- Create a legend for your plot using `legend()`. With no arguments, it will construct a legend using the label attributes of your plots. Can also pass a list of labels or a list of Line2D objects and labels.

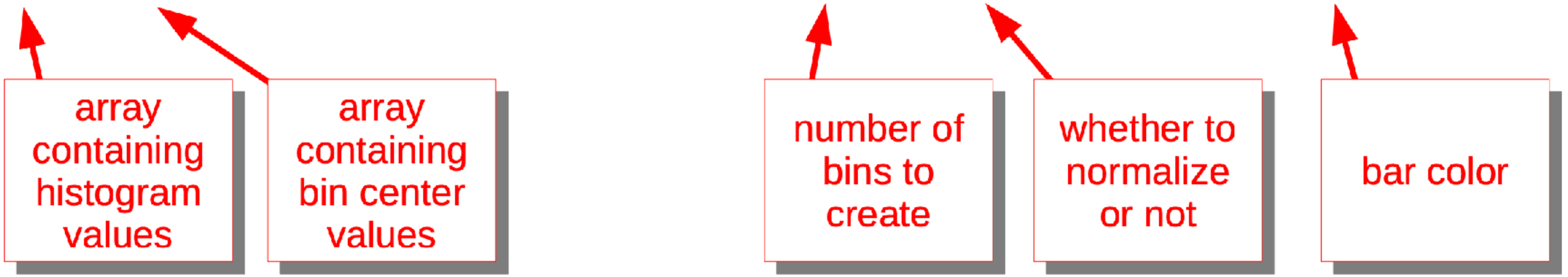



```
x = np.linspace(0, 4*np.pi, 1000)
plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.plot(x, np.tan(x), label='_nolegend_')
plt.legend(loc='upper left')
plt.ylim(-5, 5)
plt.text(np.pi, -2,\
 r'Random Greek: $\alpha = \int_0^1 f(x) dx$',\
 fontsize=16, rotation=45)
```

HISTOGRAMS

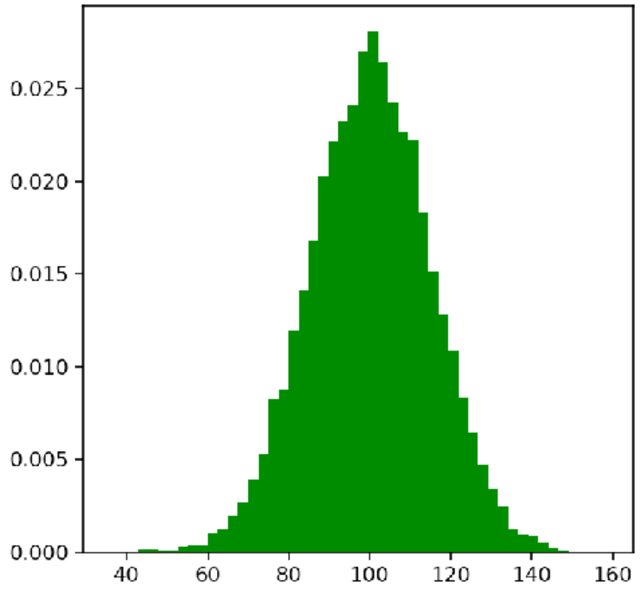
- Histograms can be drawn using `hist()`:

```
x = 100. + 15.*np.random.randn(10000)
n, bins, patches = plt.hist(x, 50, density=True, facecolor='g')
```



- Other arguments accepted by `hist()`:

```
range        # (lower, upper) range for bins
weights      # array of weights, same shape as data
cumulative   # if True, do cumulative histogram
histtype     # type of histogram (e.g. 'bar', 'step')
log          # if True, make bin axis logarithmic
             # and only return nonempty bins
```



OUTPUT FROM A PROGRAM TO AN IMAGE FILE

- Use `imsave()` to save a 2D array to an image file:

```
plt.imsave('myplot.png', z, format='png')
```

- `imsave()` also takes cmap and origin arguments.

- What if you want to capture the current figure to a file?

```
plt.savefig('myplot.png', format='png')
```

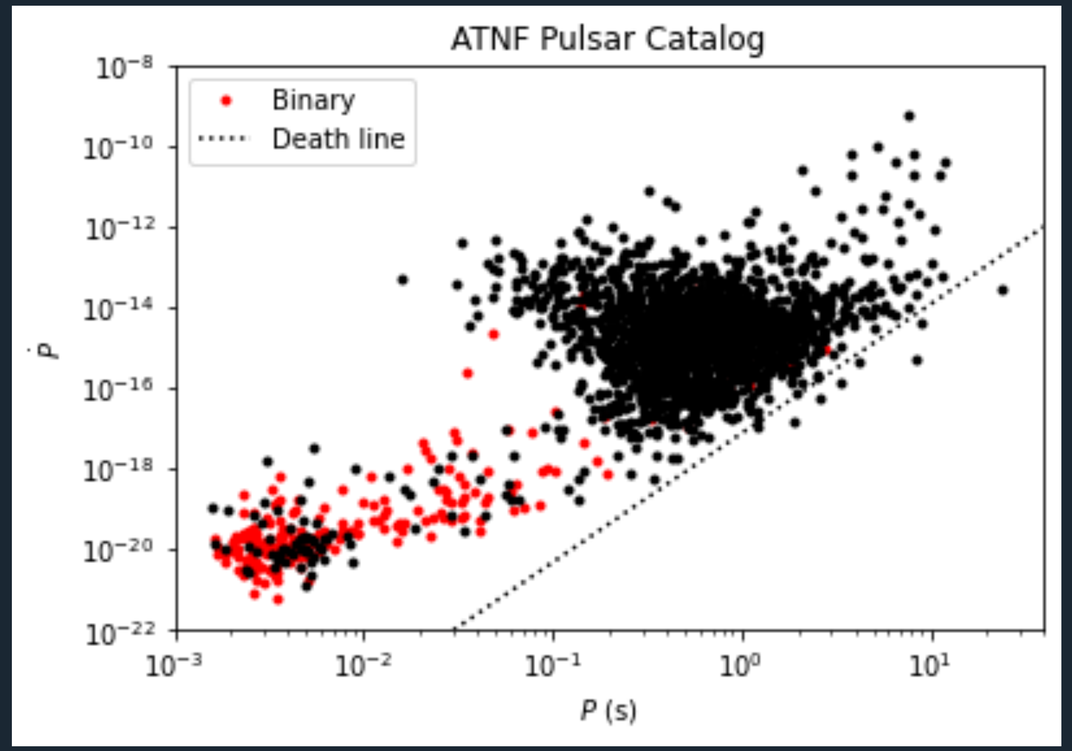
- This also takes orientation, `papertype`, `bbox_inches`, `pad_inches`, and `dpi` keyword arguments (for instance, use `dpi=300` for a publication-quality plot).

IN CLASS EXERCISE 2: OVERPLOTTING, LINE STYLES

- **Download** the file "**L14_atnf_pulsars.dat**" from the Github. This is a **fixed-width** ASCII table containing data from the Australia Telescope National Facility (ATNF) catalog of pulsars in our Galaxy known as of 1/12/20.
- Use astropy.io.ascii to read the table. You might need to give the reader the hint that the format is '`fixed_width`' and the **header starts on line 2**. We've not covered using this, but that's the point!
- Extract the pulsar periods (**column 'period'**), rate of change of periods (**column 'period_dot'**), and companion types (**column 'companion_type'**).
- The companion type will be **masked out if the pulsar is an isolated pulsar**, or else **one of 'MS', 'NS', 'CO', 'He'**, or '**UL**' depending on the type of binary companion. **Make two boolean mask arrays** to select the pulsars that are **in binaries vs. not in binaries**. Hint: use the mask attribute of the 'companion_type' column object.
- Make a **log-log scatter plot of P_dot vs. P**. Make binary pulsars and isolated pulsars **two separate colors**, and produce an appropriate legend. Make appropriate axis labels (P is in units of seconds, while P_dot is dimensionless).
- Finally, **overplot the "pulsar death line"** as a dotted line. This is (roughly) the function

and corresponds to the line beyond which pulsa $\dot{P}_{\rm death} = 7.14 \times 10^{-18} (P/{\rm sec})^{3.2}$

```python
import matplotlib.pyplot as plt
import numpy as np
from astropy.io import ascii

data = ascii.read('L14_atnf_pulsars_20200112.dat',\
                  format='fixed_width', header_start=2)

P    = data['period']
Pdot = data['period_dot']
ctype= data['companion_type']

notbin = ctype.mask
binary = ~ctype.mask

plt.loglog(P[binary], Pdot[binary], 'r.', label='Binary')
plt.loglog(P[notbin], Pdot[notbin], 'k.', label='')
ldeath = np.array([0.03, 40.])
plt.plot(ldeath, 7.14e-18*ldeath**3.2, 'k:', \
         label='Death line')

plt.xlabel(r'$P\ ({\rm s})$')
plt.ylabel(r'${\dot P}$')
plt.title('ATNF Pulsar Catalog')

plt.legend(loc='best')
plt.xlim([0.001, 40.])
plt.ylim([1.e-22, 1.e-8])
plt.show()
```



MULTI-PANEL PLOTS

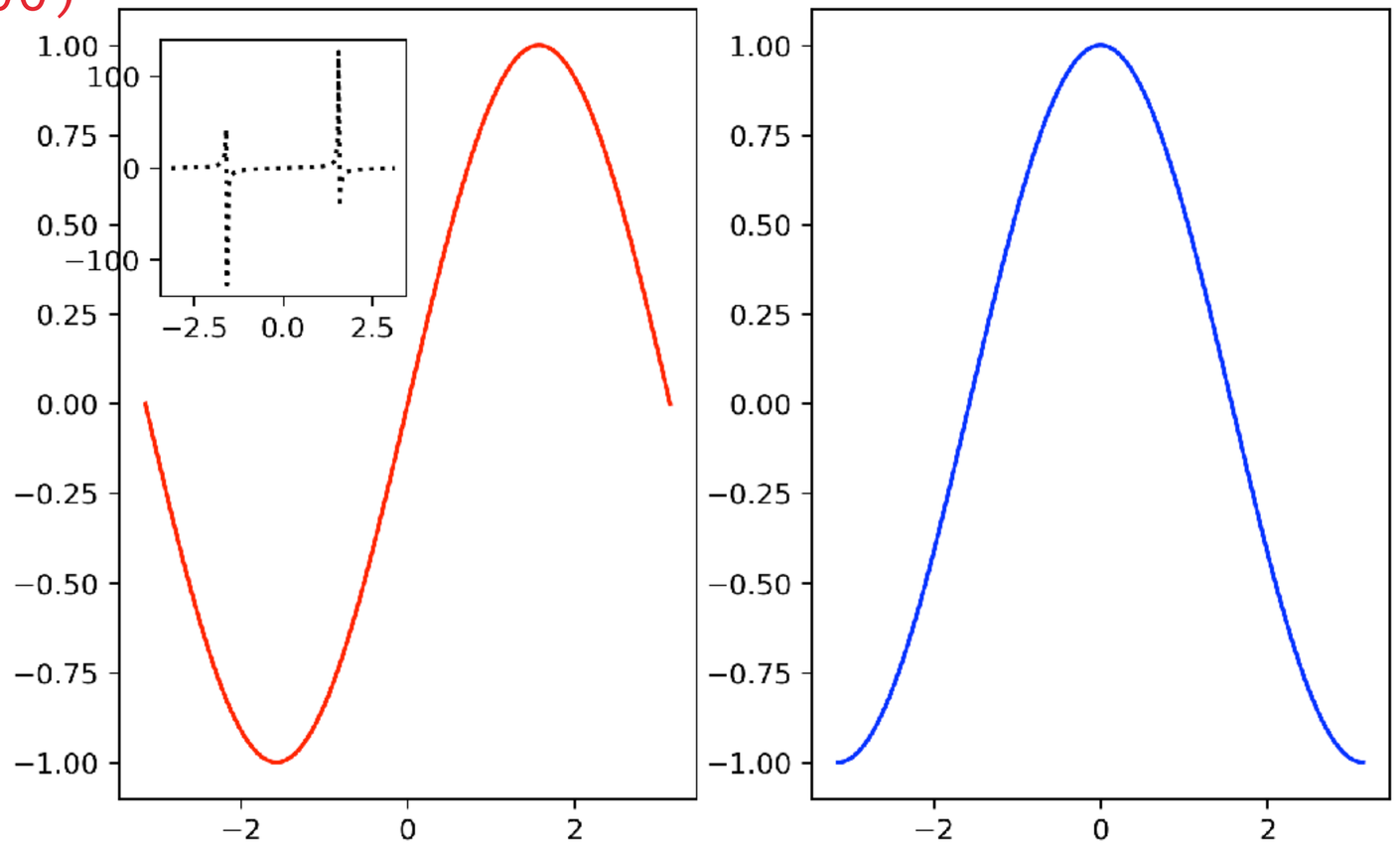
- You already know you can create multiple `Figure` objects. Each figure can have multiple `AxesSubplot` instances

```
fig = plt.figure(figsize=(8, 5))
ax1 = fig.add_subplot(1,2,1)  # make 1 row, 2 cols, select subplot 1
ax2 = fig.add_subplot(122)    # this selects subplot 2
x = np.linspace(-np.pi, np.pi, 200)
ax1.plot(x, np.sin(x), 'r-')
ax2.plot(x, np.cos(x), 'b-')
```

If < 10 subplots, can drop commas

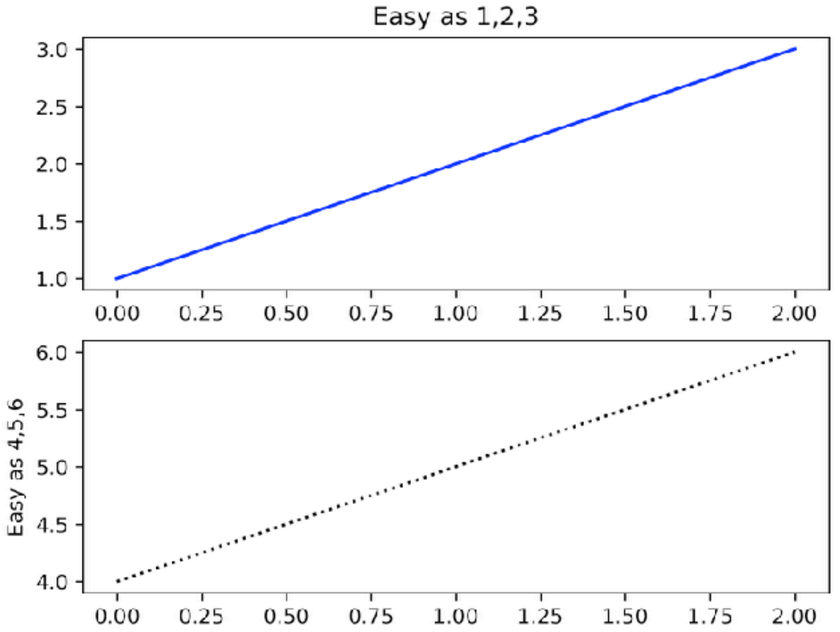
- You can also add an axis anywhere

```
ax3 = fig.add_axes([0.15, 0.6,\
          0.15, 0.25])
ax3.plot(x, np.tan(x), ‘k:')
```

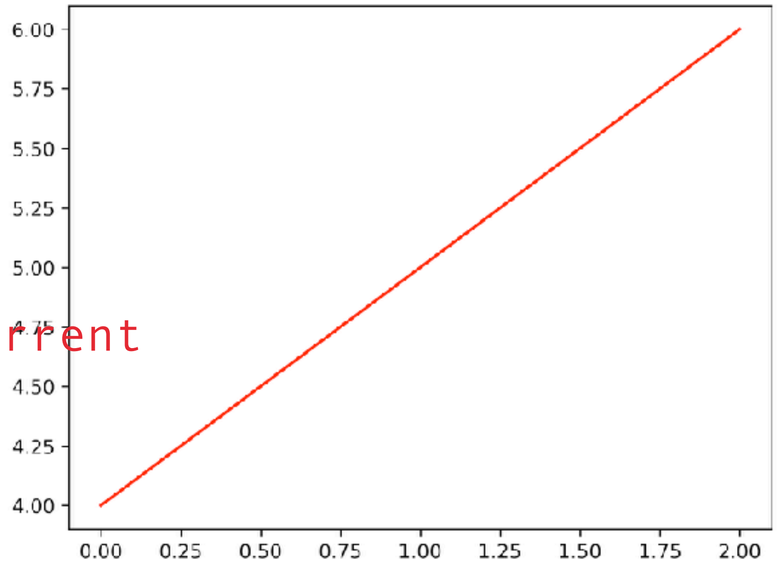


SWITCHING SUBPLOTS/FIGURES AND CLEARING

- Once you create subplots, you can switch between them easily. This is obvious if using object-oriented mode, where each subplot gets it's own variable, but also works in interactive mode
- Use `cla()` to clear an axes, or `clf()` to clear figure `close()` frees figure memory.



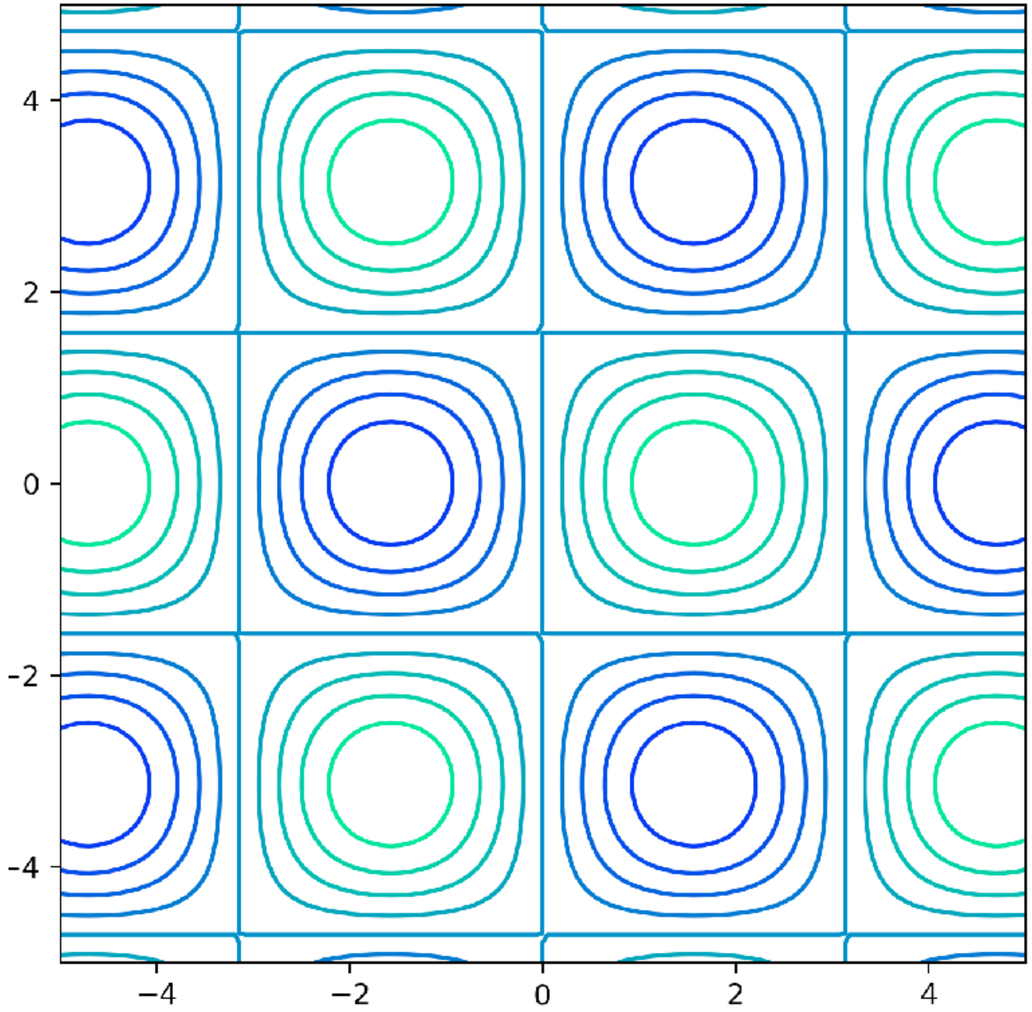

```
plt.figure(2)       # the second figure
plt.subplot(211)    # the first subplot in the second figure
plt.plot([1,2,3], 'b-')
plt.subplot(212)    # the second subplot in the second figure
plt.plot([4,5,6], 'k:')
plt.figure(3)       # a third figure
plt.plot([4,5,6], 'r-')  # creates a subplot(111) by default
plt.figure(2)       # figure 2 current; subplot(212) still current
plt.subplot(211)    # make subplot(211) in figure 2 current
plt.title('Easy as 1,2,3')  # subplot 211 tit
```



2D CONTOUR PLOTS

- Contour plots can be drawn using `contour()`, which operates on a 2D array/list:

```
xx = yy = np.linspace(-5, 5, 100)
x, y = np.meshgrid(xx, yy)   # create 2D arrays with (x, y) values
z = np.sin(x)*np.cos(y)
fig4 = plt.figure(figsize=(6, 6))
ax4 = fig4.add_subplot(1,1,1)
# 10 is number of contour levels (optional)
ax4.contour(x, y, z, 10, cmap='winter')
```



- Here x and y are optional
  (just as x- values are optional for `plot()`).
- Filled contours can be created with `contourf()`
- The palette used for the colors is controlled by `cmap`. We can add a plot key using `colorbar()`.
- Contour labels can be added using `clabel()`.

CHOOSING COLORMAPS

- Choose different colormaps for quantities that are always positive/negative or that have both positive and negative values.

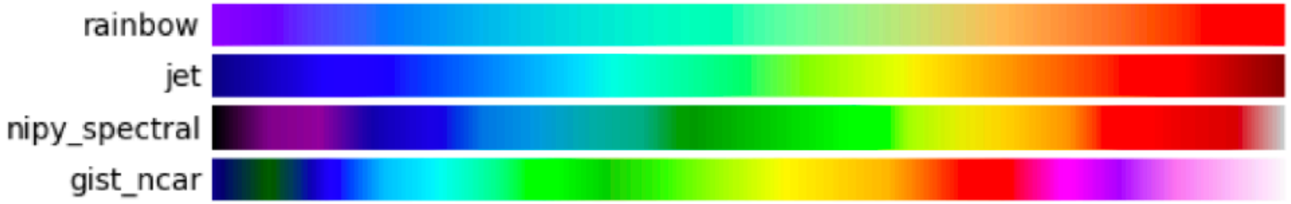
steadily increasing values

values whose magnitude diverges from a central value

- Try to choose “perceptually uniform” color maps to avoid artifacts and make plots easier to read for colorblind users or when rendered as grayscale.

better

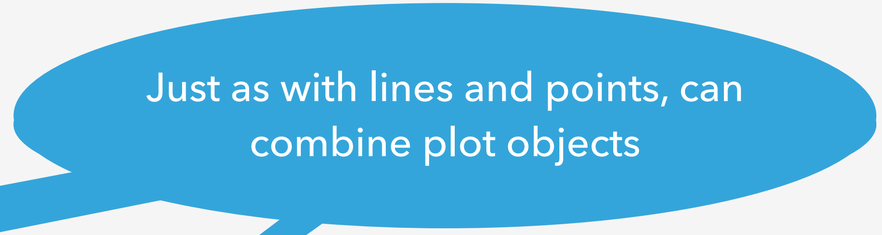
not preferred

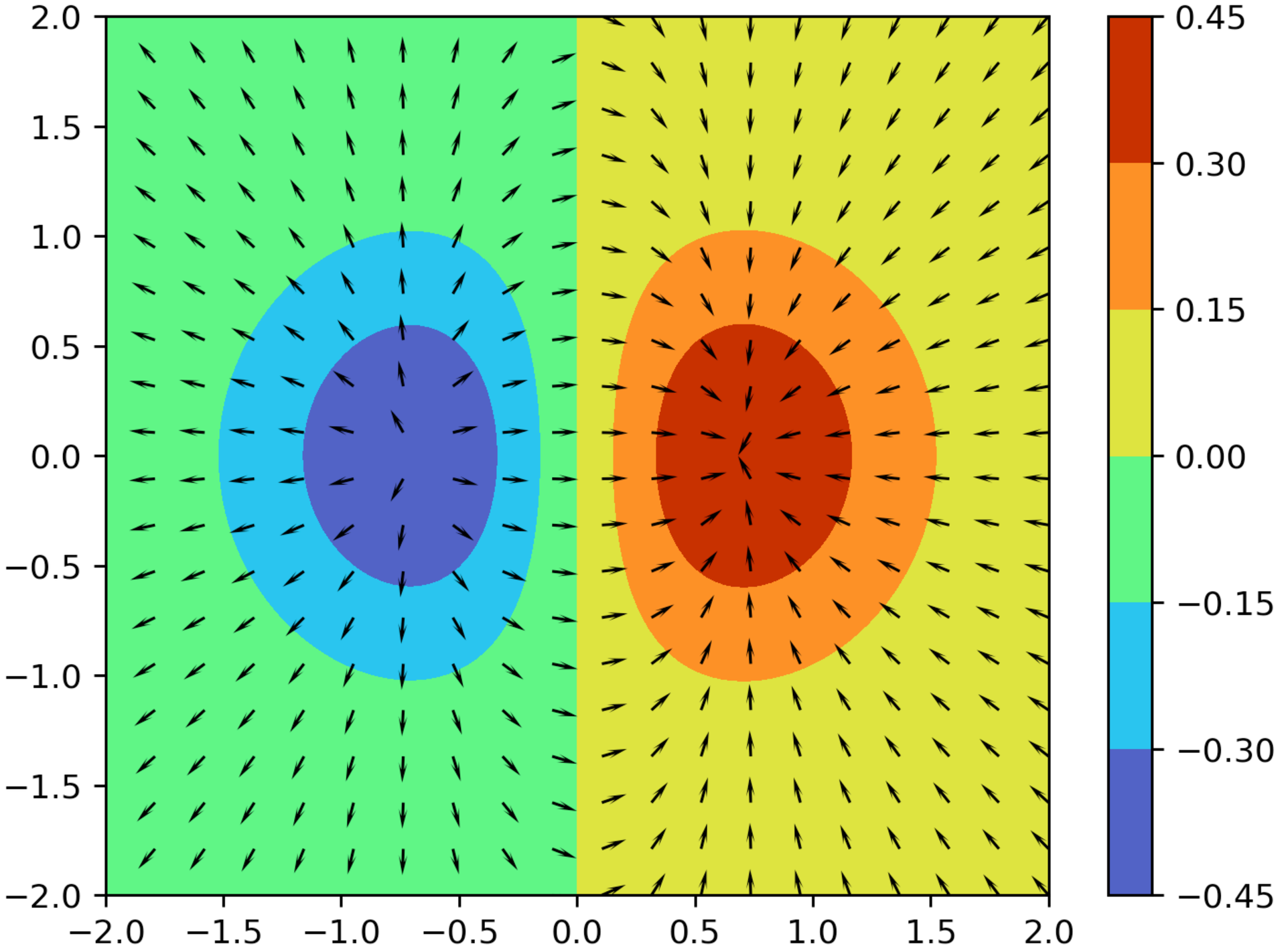


QUIVER PLOTS AND COMBINING MULTIPLE OBJECTS ON THE SAME FIGURE

- Use `quiver()` to create a vector field from two 2D arrays representing x- and y-components of the field, respectively:

```
fig5 = plt.figure()
ax5 = fig5.add_subplot(1,1,1)

X, Y = np.mgrid[-2:2:100j, -2:2:100j]
Z = X*np.exp(-(X**2 + Y**2))

# Contour Plot but Filled this time
cp = ax5.contourf(X, Y, Z, cmap='turbo')   # like contour but filled
cb = fig5.colorbar(cp)                     # add a colorbar automatically populated using the filled contour

# Vector Field
YT, XT = np.mgrid[-2:2:20j, -2:2:20j]          # notice we're using the same ranges, but fewer samples
U =(1 - 2*(XT**2))*np.exp(-((XT**2)+(YT**2)))
V = -2*XT*YT*np.exp(-((XT**2)+(YT**2)))
speed = np.sqrt(U**2 + V**2)
UN = U/speed
VN = V/speed

# Quiver plot overlaid on contours
quiv = ax5.quiver(XT, YT, UN, VN,  color='k', headlength=7)

# save the figure
fig5.savefig('quiver_filled_contour.png')
```



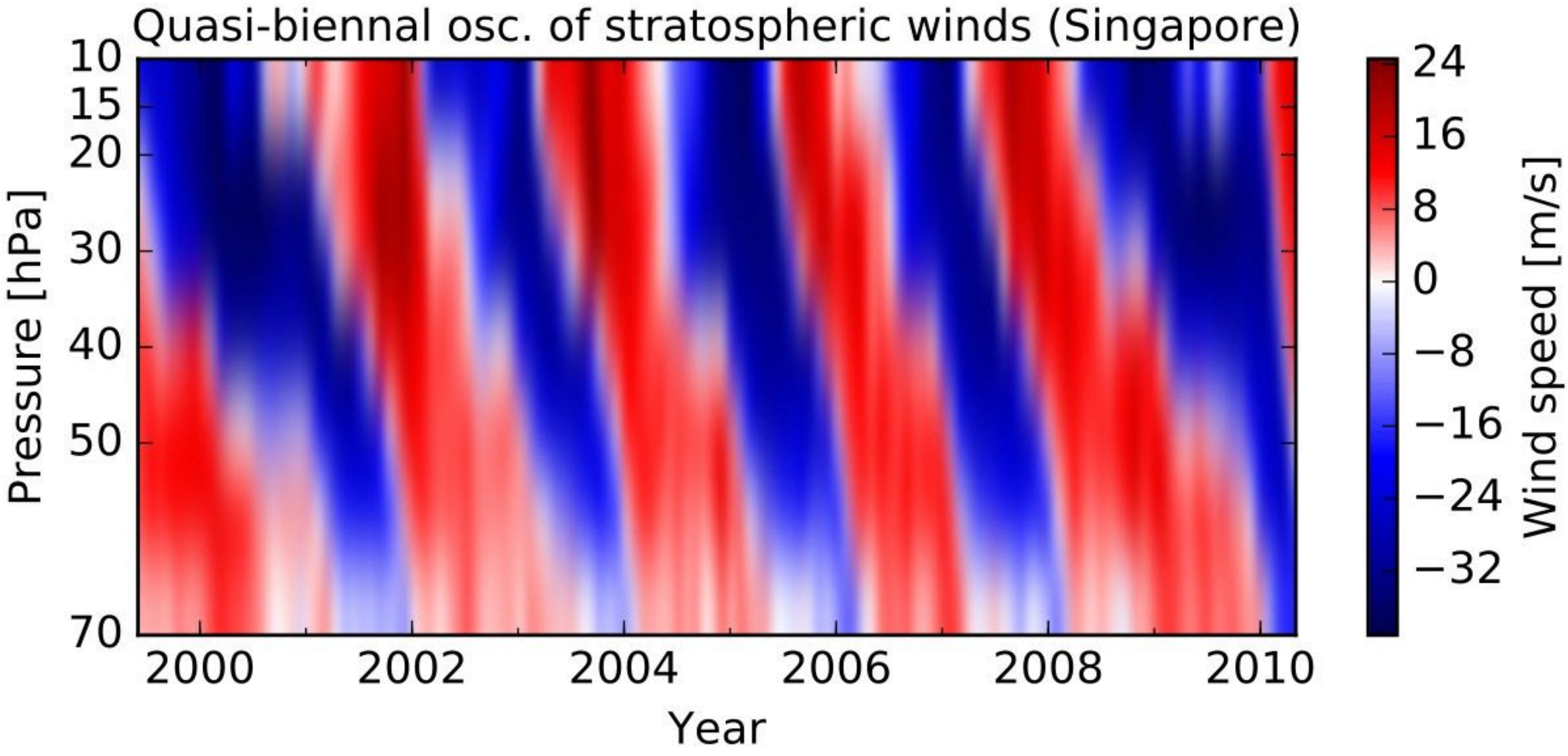

2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
0.45
0.30
0.15
0.00
−0.15
−0.30
−0.45

CONTROLLING THE COLOR MAP AND PLOT KEY

- The colorbar() routine can take several additional keyword arguments:

```
orientation   'vertical' or 'horizontal'
fraction      fraction of original axes to use for colorbar (default 0.15)
pad           fraction of original axes between colorbar and new axes
shrink        fraction by which to shrink the colorbar
aspect        aspect ratio of the colorbar
```

- Color maps create a mapping between color numbers and actual colors (ie. the **palette**). These are handled using the `matplotlib.cm` module, which allows you to create and use new color maps. Using pyplot routines you can set the "current" colormap to any of the built-in color maps: https://matplotlib.org/stable/tutorials/colors/colormaps.html
- Check your colormap for how accesible it is to those with color blindness


Quasi-biennal osc. of stratospheric winds (Singapore)
Pressure [hPa]
10
15
20
30
40
50
70
2000
2002
2004
2006
2008
2010
Year
24
16
8
0
−8
−16
−24
−32
Wind speed [m/s]

SAVING AND RESTORING PLOTS

- Matplotlib can read in **PNG** images natively (and other formats too if the **Python Imaging Library (PIL)** is installed)
- Once you have the image as a 2D array – or if you've generated a 2D array yourself – you can plot it as an image using `imshow()`
- Default origin is upper-left corner, but you can change this using `origin` keyword: `'upper'` or `'lower'`
- You can specify axis ranges using extent keyword `(extent=[xmin,xmax,ymin,ymax])`.

```
fig6 = plt.figure()
ax6 = fig6.add_subplot()
img = plt.imread('quiver_filled_contour.png') # load the previously saved
plot
ax6.imshow(img);
```