STAT606
Computing for Data Science and Statistics

Lecture 12: matplotlib
Plotting with matplotlib

matplotlib is a plotting library for use in Python

Similar to R’s ggplot2 and MATLAB’s plotting functions

For MATLAB fans, matplotlib.pyplot implements MATLAB-like plotting:
  http://matplotlib.org/users/pyplot_tutorial.html

Sample plots with code:
  http://matplotlib.org/tutorials/introductory/sample_plots.html
Basic plotting: `matplotlib.pyplot.plot`  

`matplotlib.pyplot.plot(x, y)`  
plots `y` as a function of `x`.  

`matplotlib.pyplot(t)`  
sets x-axis to `np.arange(len(t))`
Basic plotting: `matplotlib.pyplot.plot`

Jupyter “magic” command to make images appear in-line.

Reminder: Python `'_'_` is a placeholder, similar to MATLAB `‘~’`. Tells Python to treat this like variable assignment, but don’t store result anywhere.
Customizing plots

```python
1 x = np.arange(0, 5, 0.25, dtype='float')
2 _ = plt.plot(x**2, ':ro')
```

Second argument to `pyplot.plot` specifies line type, line color, and marker type.
Customizing plots

```python
x = np.arange(0, 5, 0.25, dtype='float')
_ = plt.plot(x**2, color='red', linestyle=':', marker='o')
```

Long form of the command on the previous slide. Same plot!

A full list of the long-form arguments available to `pyplot.plot` are available in the table titled “Here are the available Line2D properties.”: [http://matplotlib.org/users/pyplot_tutorial.html](http://matplotlib.org/users/pyplot_tutorial.html)
Multiple lines in a single plot

```python
1 t = np.arange(0., 5., 0.2)
2 # plt.plot(xvals, y1vals, traits1, y2vals, traits2, ...)
3 _ = plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

**Note:** more complicated specification of individual lines can be achieved by adding them to the plot one at a time.
Multiple lines in a single plot: long form

```python
1 t = np.arange(0., 5., 0.2)
2 plt.grid()
3 plt.plot(t, t, 'r--')
4 plt.plot(t, t**2, 'bs')
5 plt.plot(t, t**3, 'g^')
6 _ = plt.show()
```

**Note:** same plot as previous slide, but specifying one line at a time so we could, if we wanted, use more complicated line attributes.

*plt.grid* turns grid lines on/off.
Titles and axis labels

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

t = np.arange(0., 5., 0.2)
plt.grid()
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.title('Profits as a function of goats')
plt.xlabel('Goats')
plt.ylabel('Profits')
plt.show()
```

Specifying titles and axis labels couldn’t be more straight-forward.
Titles and axis labels

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

# Create a range of numbers
r = np.arange(0., 5., 0.2)

# Change the title and axis labels
plt.title('Title text', fontsize=18)
plt.xlabel('x axis', fontsize=14)
plt.ylabel('y axis', fontsize=14)

# Plot a graph
plt.plot(r, r, 'r--', label='r--')
plt.plot(r, r**2, 'bs', label='r^2')
plt.plot(r, r**3, 'g^', label='r^3')
plt.legend()

plt.show()
```
Legends

```python
plt.xlabel("$n$", fontsize=16)  # set the axes labels
plt.ylabel("$f(n)$", fontsize=16)
plt.title("Different growth behaviors")  # set the plot title
plt.plot(t, t, '-ob', label='linear, $f(n)=n$')
plt.plot(t, t**2, ':^r', label='quadratic, $f(n)=n^2$')
plt.plot(t, t**3, '--sg', label='cubic, $f(n)=n^3$')
_=plt.legend(loc='best')  # places legend at best location
```

- Can use LaTeX in labels, titles, etc.
- `pyplot.legend` generates legend based on label arguments passed to `pyplot.plot`. `loc='best'` tells `pyplot` to place the legend where it thinks is best.
Annotating figures

```
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t) # np.pi == 3.14159...
plt.plot(t, s, lw=2) # plot the cosine.
# Annotate the figure with an arrow and text.
_ = plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
    fontsize=14,
    arrowprops=dict(facecolor='black', shrink=0.02))
```

Specify text coordinates and coordinates of the arrowhead using the coordinates of the plot itself. This is pleasantly different from many other plotting packages, which require specifying coordinates in pixels or inches/cms.
Plotting histograms: `pyplot.hist()`

```python
mu, sigma = (100, 15)
x = np.random.normal(mu, sigma, 10000)
# hist( data, nbins, ... )
(n, bins, patches) = plt.hist(x, 50, density=False, facecolor='teal')
n
array([[ 1.,  1.,  2.,  4.,  3.,  5., 11., 18., 26., 30., 47.,
      541., 529., 597., 595., 572., 566., 543., 515., 462., 404., 360.,
      15.,  11.,  5.,  2.,  1.,  4.])
```

Bin counts. Note that if `density=True`, then these will be chosen so that the histogram “integrates” to 1.

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.hist.html
Bar plots

```python
bar(x, height, *, align='center', **kwargs)
```

```
t = np.arange(10)
s = np.random.normal(1, 1, 10)
plt.bar(t, s, align='center')
```

Full set of available arguments to `bar(...)` can be found at
http://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar

Horizontal analogue given by `barh`
http://matplotlib.org/api/_as_gen/matplotlib.pyplot.barh.html#matplotlib.pyplot.barh
Tick labels

Can specify what the x-axis tick labels should be by using the `tick_label` argument to plot functions.
Box & whisker plots

```python
K=12; n=25
draws = np.zeros((n,K))
for k in range(K):
    mu = np.sin(2*np.pi*k/K)
draws[::,k] = np.random.normal(mu, 1, n)
_ = plt.boxplot(draws, labels=list('JFMAMJJASOND'))
```

`plt.boxplot(x,...) : x is the data. Many more optional arguments are available, most to do with how to compute medians, confidence intervals, whiskers, etc. See http://matplotlib.org/api/_as_gen/matplotlib.pyplot.boxplot.html#matplotlib.pyplot.boxplot`
Pie Charts

Don’t use pie charts!

A table is nearly always better than a dumb pie chart; the only worse design than a pie chart is several of them, for then the viewer is asked to compare quantities located in spatial disarray both within and between charts [...]

Given their low [information] density and failure to order numbers along a visual dimension, pie charts should never be used.

Edward Tufte
*The Visual Display of Quantitative Information*

But if you must...

```python
pyplot.pie(x, ...)  
```

[hyperlink](http://matplotlib.org/api/ as_gen/matplotlib.pyplot.pie.html#matplotlib.pyplot.pie)
Heatmaps and tiling

```python
n=20
x = np.arange(1,n+1)
M = x*np.reshape(x,(n,1))
_ = plt.imshow(M)
```

*`imshow` is `matplotlib` analogue of MATLAB’s `imagesc`, R’s `image`. Lots of optional extra arguments for changing scale, color scheme, etc. See documentation: https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow*
These three lines create an object, `mvn1`, representing a multivariate normal distribution.
Drawing contours

```python
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:,:,0] = x; pos[:,:,1] = y
_ = plt.contour(x, y, mvn1.pdf(pos))
```

*mgrid* is short for “mesh grid”. Note the syntax: square brackets instead of parentheses. *mgrid* is an object, not a function!
Drawing contours

Here, `mgrid` stores a grid of (x,y) pairs, so this line actually generates a 100-by-100 grid of (x,y) coordinates, hence the tuple assignment.

```python
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvn1 = scipy.stats.multivariate_normal(mu,Sigma)
x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:,:,0] = x; pos[:,:,1] = y
_= plt.contour(x, y, mvn1.pdf(pos))
```
pos is a 3-dimensional array. Like a box of numbers. We're going to plot a surface, but at each (x,y) coordinate, the surface value depends on both x and y.
The reason for building `pos` the way we did is apparent if we read the documentation for `scipy.stats.(dist).pdf`. 

```python
mu=np.array([0.5,0.5])
Sigma=np.array([[0.1,0.05],[0.05,0.1]])
mvnl = scipy.stats.multivariate_normal(mu,Sigma)
x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:,:,0] = x; pos[:,:,1] = y
_ = plt.contour(x, y, mvnl.pdf(pos))
```
Drawing contours

```python
mu = np.array([0.5, 0.5])
Sigma = np.array([[0.1, 0.05], [0.05, 0.1]])

mvnl = scipy.stats.multivariate_normal(mu, Sigma)

x, y = np.mgrid[0:1:.01, 0:1:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

_ = plt.contour(x, y, mvnl.pdf(pos))
```

**matplotlib.contour** takes a set of x coordinates, a set of y coordinates, and an array of their corresponding values.

**matplotlib.contour** offers plenty of optional arguments for changing color schemes, spacing of contour lines, etc.

[https://matplotlib.org/api/contour_api.html](https://matplotlib.org/api/contour_api.html)
Subplots

```
plt.subplot(221)
plt.plot(t, t, '-ob')
plt.title('linear')
plt.subplot(222)
plt.title('quadratic')
plt.plot(t, t**2, '-or')
plt.subplot(223)
plt.title('sqrt')
plt.plot(t, np.sqrt(t), '--og')
plt.subplot(224)
plt.title('logarithmic')
plt.plot(t, np.log(t), '-oy')
```

= plt.tight_layout()

---

**Shorthand:** `subplot(XYZ)`

- Makes an $X$-by-$Y$ plot
- Picks out the $Z$-th plot
- Counting in row-major order

---

`t` = np.arange(20) + 1

**tight_layout()** automatically tries to clean things up so that subplots don’t overlap. Without this command in this example, the labels “sqrt” and “logarithmic” overlap with the x-axis tick labels in the first row.
Specifying axis ranges

plt.xlim([lower, upper]) for x-axis

plt.ylim([lower, upper]) sets y-axis limits

For-loop goes through all of the subplots and sets their y-axis limits
Nonlinear axes

Scale the axes with `plt.xscale` and `plt.yscale`

Built-in scales:
- Linear (`'linear'`)
- Log (`'log'`)
- Symmetric log (`'symlog'`)
- Logit (`'logit'`)

Can also specify customized scales: [https://matplotlib.org/devel/add_new_projection.html#adding-new-scales](https://matplotlib.org/devel/add_new_projection.html#adding-new-scales)
Saving images

`plt.savefig(filename)` will try to automatically figure out what file type you want based on the file extension.

Can make it explicit using `plt.savefig('filename', format='fmt')`

Options for specifying resolution, padding, etc: [https://matplotlib.org/api/_as_gen/matplotlib.pyplot.savefig.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.savefig.html)
Animations

matplotlib.animate package generates animations

I won’t require you to make any, but they’re fun to play around with (and they can be a great visualization tool)

The details are a bit tricky, so I recommend starting by looking at some of the example animations here: http://matplotlib.org/api(animation_api).html#examples