# STATS 507
# Data Analysis in Python

Week9-2: Matplotlib and Pandas

Dr. Xian Zhang

Adapted from slides by Professor Jeffrey Regier

# Part 1: Matplotlib

# What is `matplotlib`

Matplotlib is a comprehensive **library** for creating static, animated, and interactive **visualizations** 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`

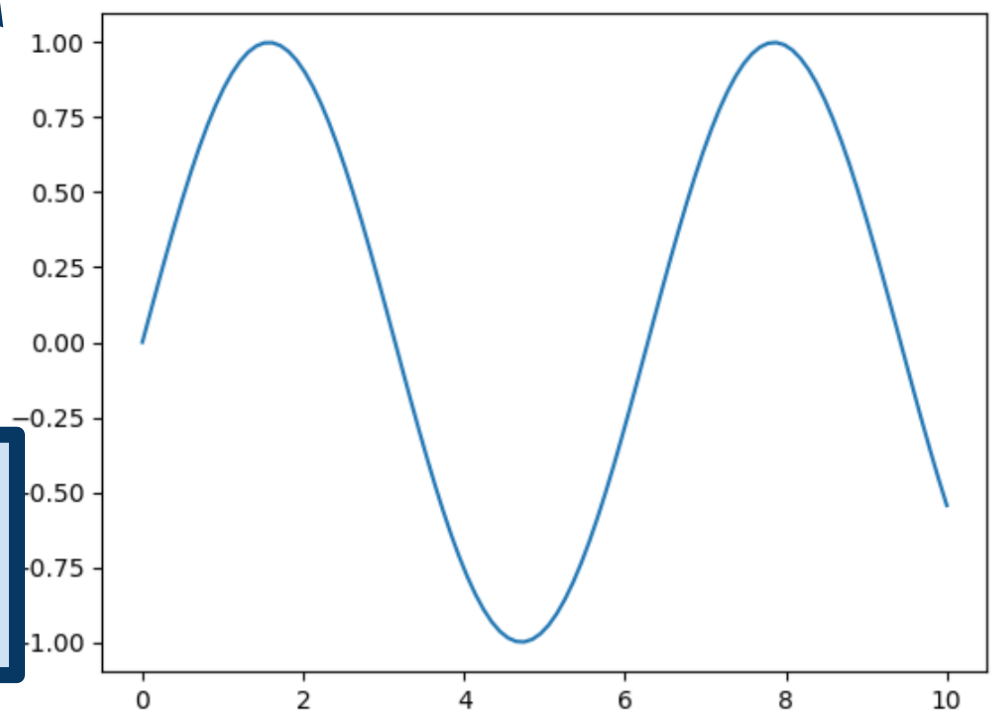
`matplotlib.pyplot.plot(x,y)`

plots `y` as a function of `x`.

`matplotlib.pyplot.plot(y)`

default x-axis to `np.arrange(len(x))`

```python
import numpy as np
import matplotlib.pyplot as plt
# Create some example data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Create the plot
plt.plot(x,y)
# Display the plot
plt.show()
```



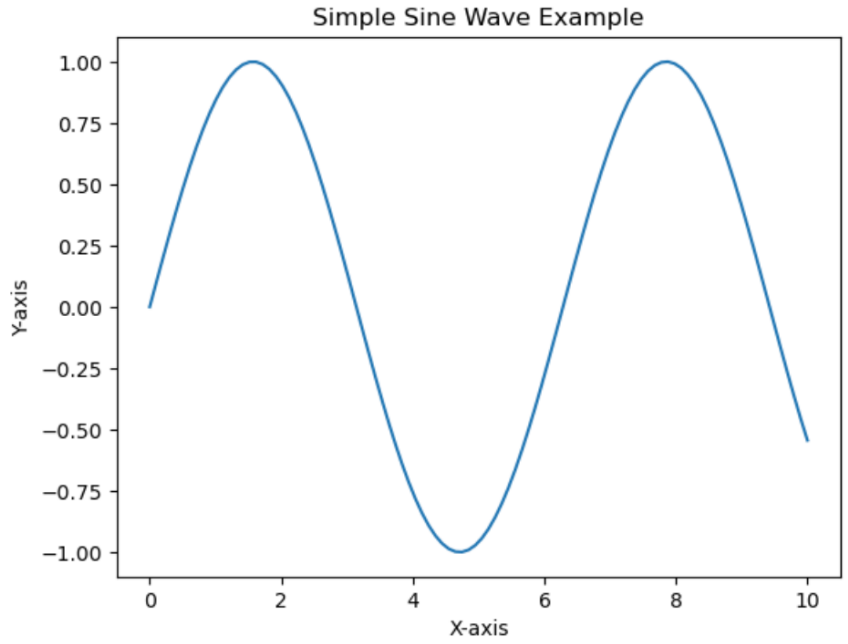

`matplotlib.pyplot` is the main plotting interface in Matplotlib, It provides an implicit, MATLAB-like way of plotting, often imported as `plt`

# Basic plotting: `matplotlib.pyplot.plot`

`matplotlib.pyplot.plot(x,y)`

plots `y` as a function of `x`.

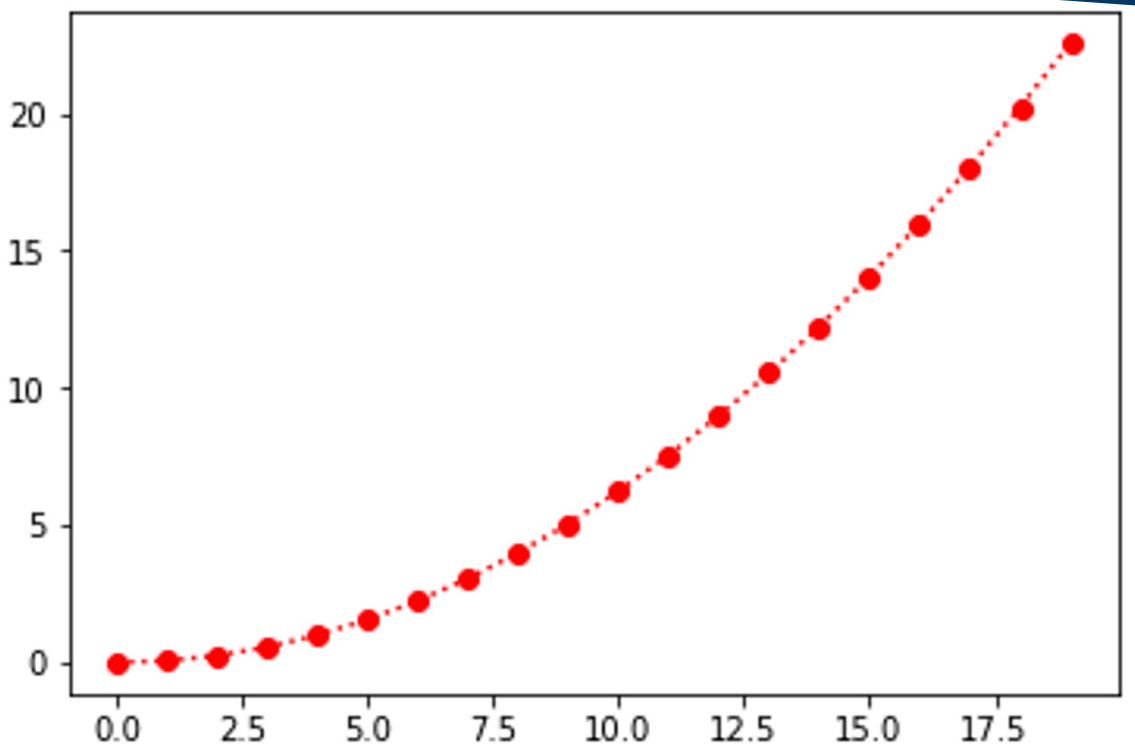
```python
import numpy as np
import matplotlib.pyplot as plt
# Create some example data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Create the plot
plt.plot(x,y)
# Add a title
plt.title("Simple Sine Wave Example")
# Add an axis label
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
# Display the plot
plt.show()
```




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

# Customizing plots

```
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.

**matplotlib.pyplot.plot**

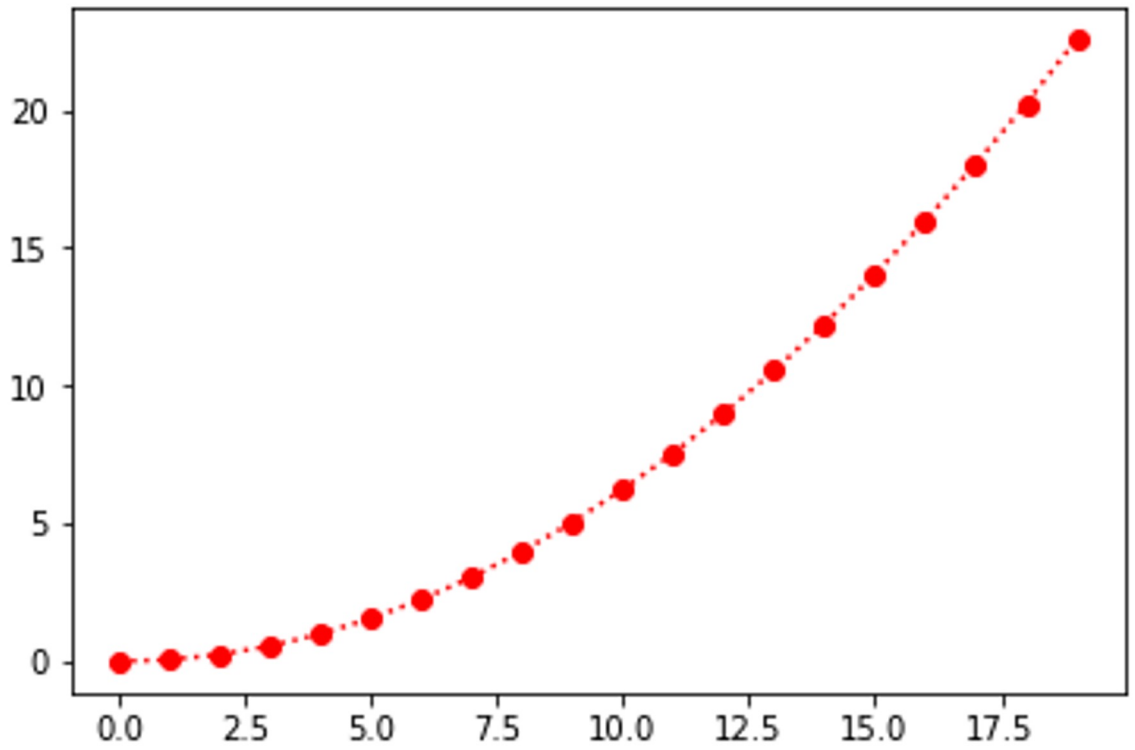
`matplotlib.pyplot.plot(*args, **kwargs)`

positional argument

Optional keyword arguments for **customizing** the plot

# Customizing plots

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



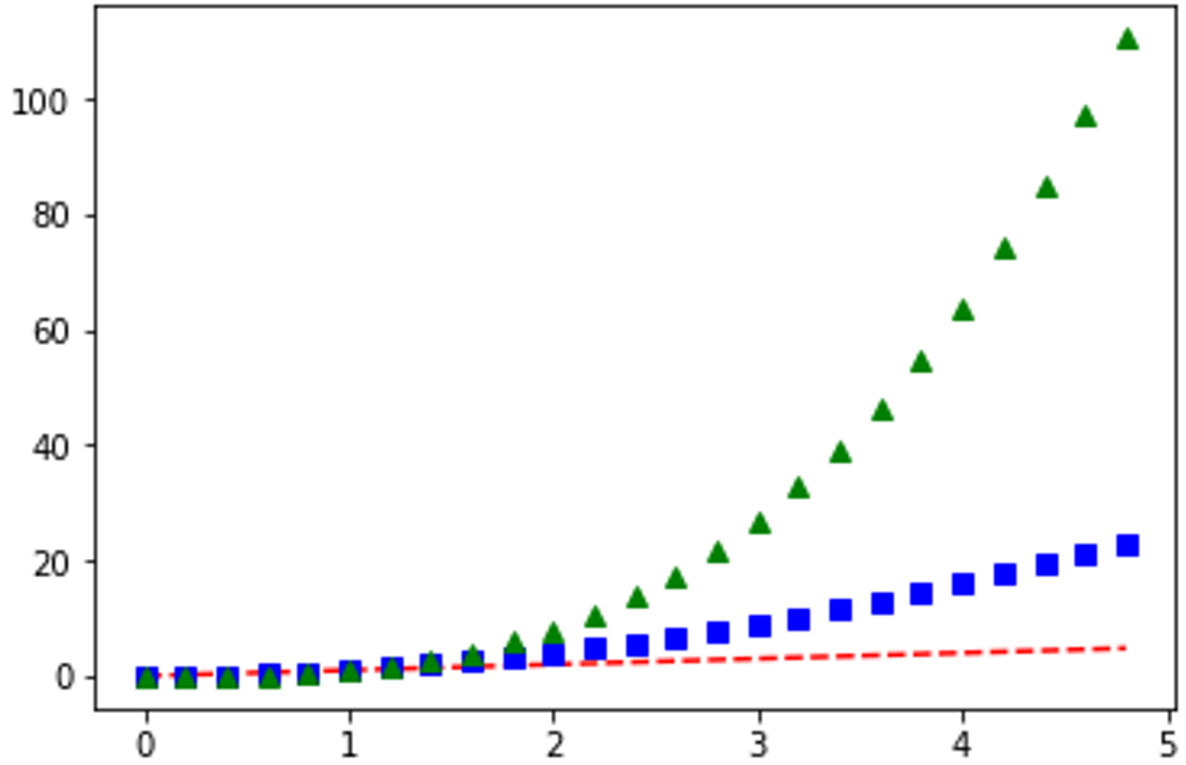

Long form of the command on the previous slide. Same plot!

format **string** characters are accepted to control the color, line style or marker:

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

# Multiple lines in a single plot

```
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^')
```

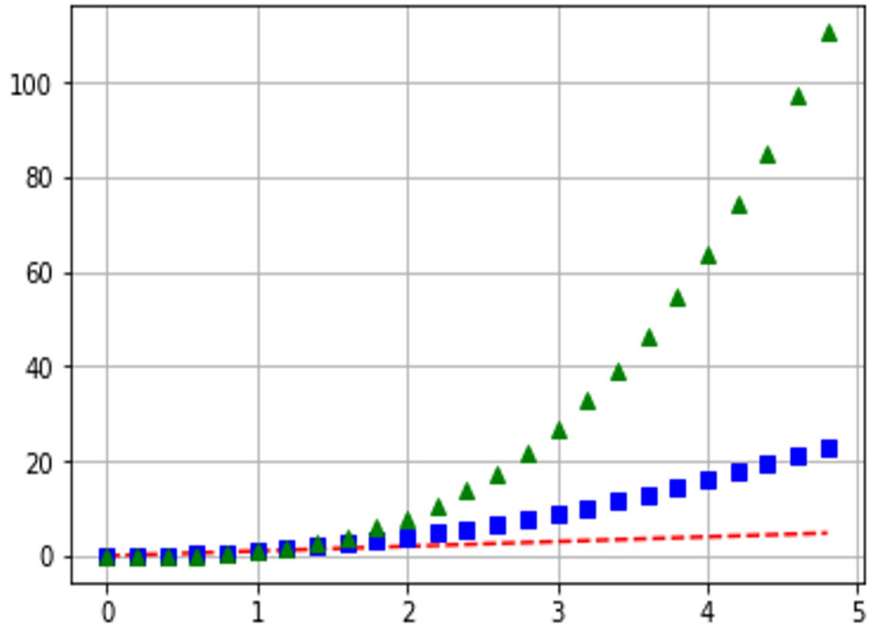



**Note:** more complicated specification of individual lines can be achieved by adding them to the plot one at a time.

# Multiple lines in a single plot

```
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()
```

`plt.grid` to apply grid to the figure

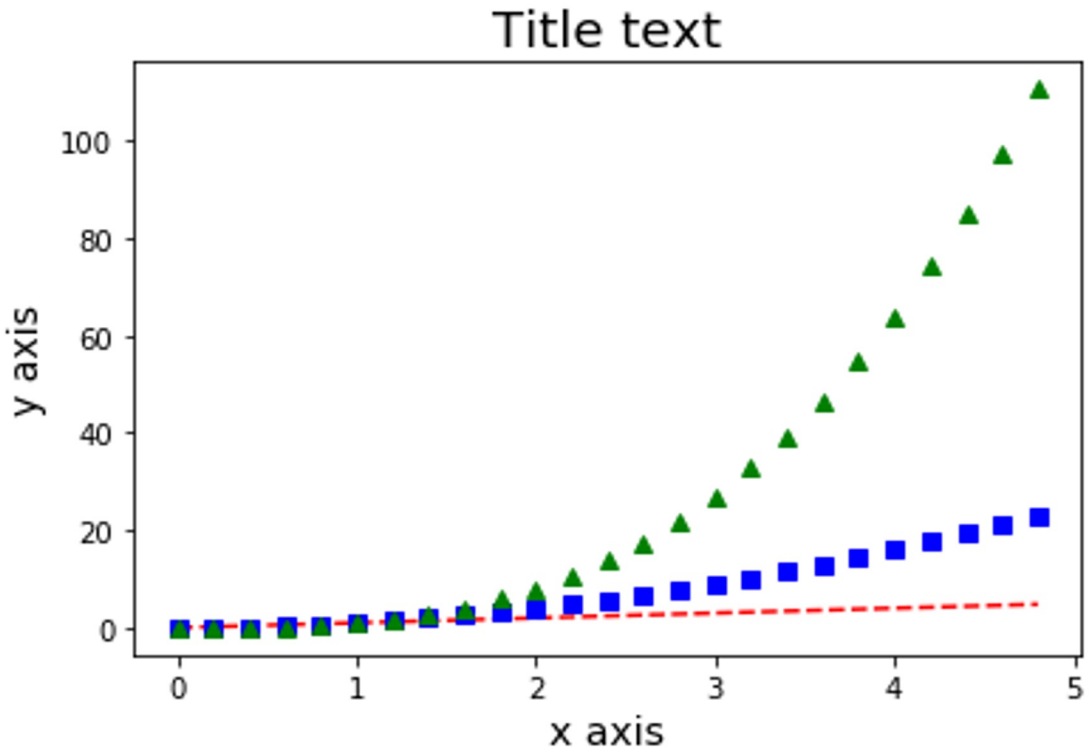



**Note:** same plot as previous slide, but specifying one line at a time.

# Titles and axis labels

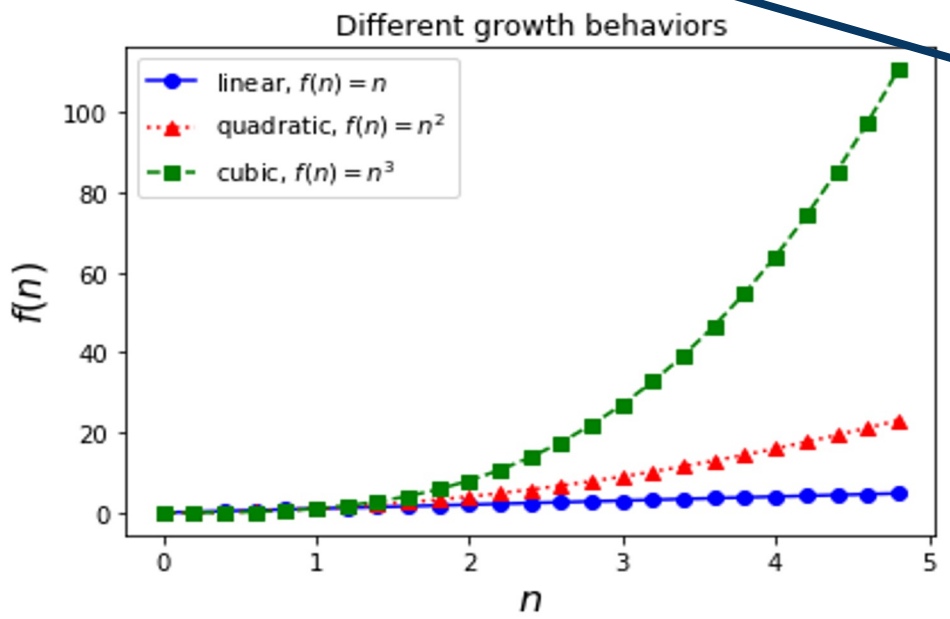
```
1 t = np.arange(0., 5., 0.2)
2 plt.title('Title text', fontsize=18)
3 plt.xlabel('x axis', fontsize=14)
4 plt.ylabel('y axis', fontsize=14)
5 _ = plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

Change font sizes



# Legends

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

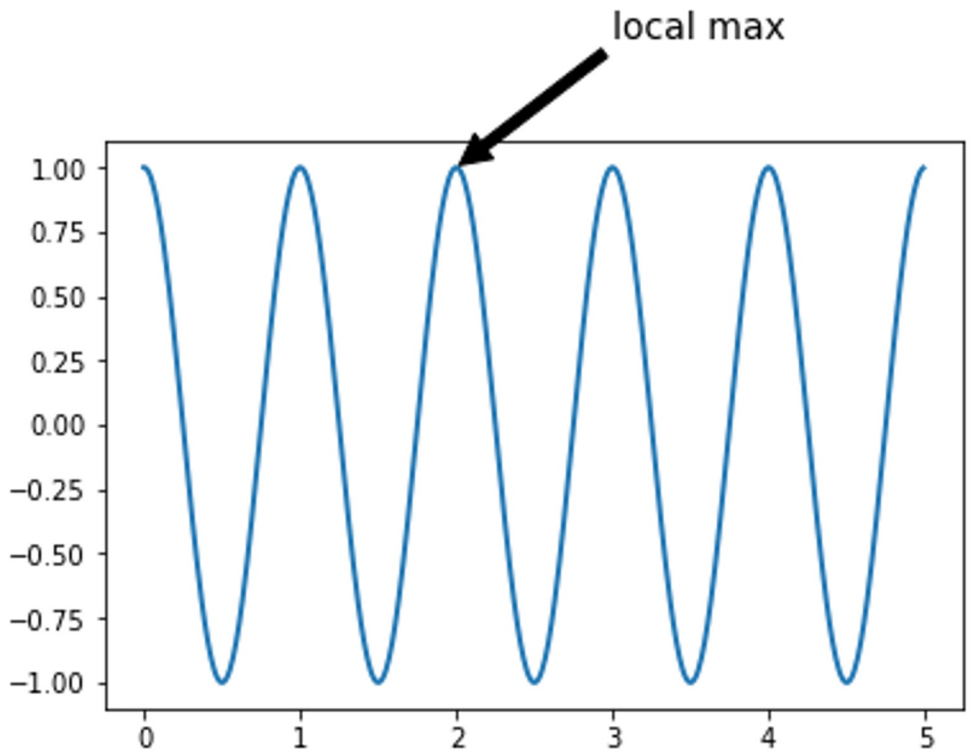



Can use LaTeX in labels, titles, etc.

`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) )
```



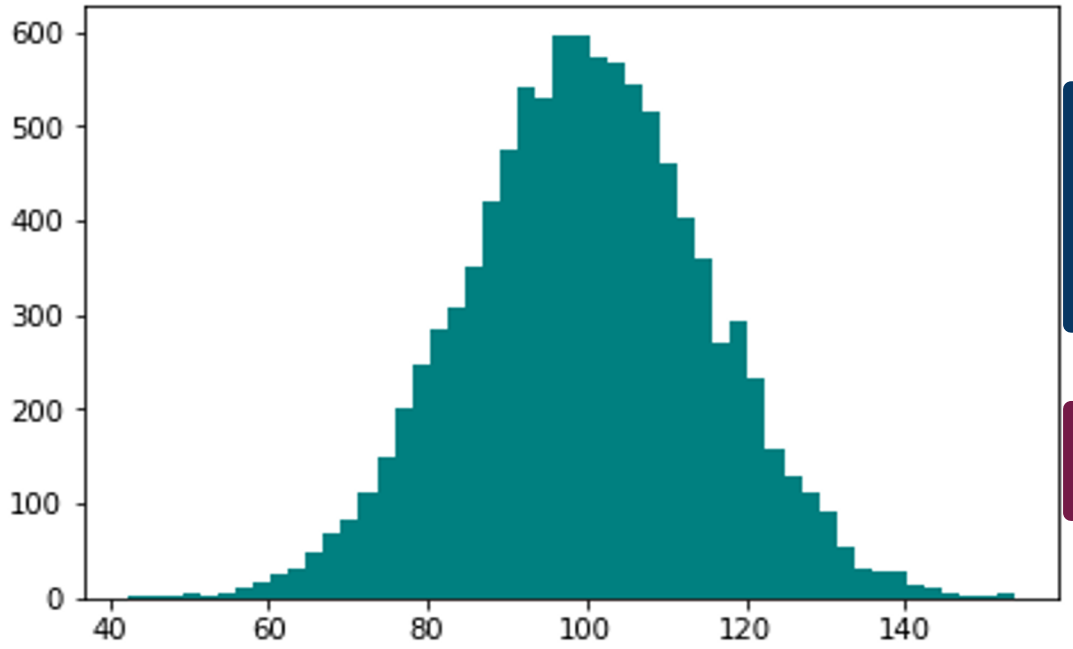

Specify text coordinates and coordinates of the arrowhead using the *coordinates of the plot itself*. This is pleasantly different from many other plotting packages, which require specifying coordinates in pixels or inches/cms.

# Plotting histograms: `pyplot.hist()`

```
1  mu, sigma = (100, 15)
2  x = np.random.normal(mu,sigma,10000)
3  # hist( data, nbins, ... )
4  (n, bins, patches) = plt.hist(x, 50, density=False, facecolor='teal')
5  n
```

```
array([  1.,   1.,   2.,   4.,   3.,   5.,  11.,  18.,  26.,  30.,  47.,
        68.,  82., 113., 150., 201., 246., 285., 309., 352., 420., 475.,
       541., 529., 597., 595., 572., 566., 543., 515., 462., 404., 360.,
       270., 294., 233., 159., 128., 111.,  92.,  54.,  32.,  28.,  28.,
        15.,  11.,   5.,   2.,   1.,   4.])
```

Bin counts: defines the number of equal-width bins in the histogram or the bin edges.

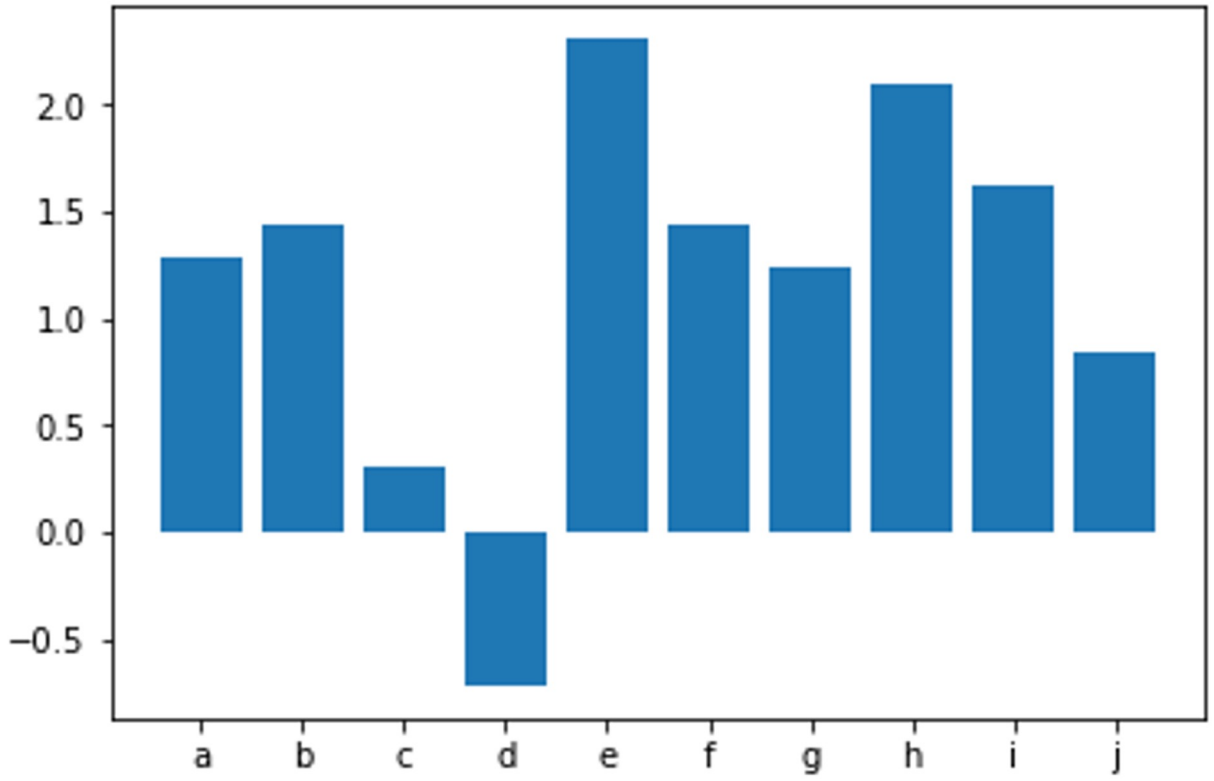



Note that if `density=True`, then these will be chosen so that the histogram "integrates" to 1. It is normalized so the total area equals 1.

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

# Bar plots

```
1 import string
2 t = np.arange(10)
3 s = np.random.normal(1,1,10)
4 mylabels = list(string.ascii_lowercase[0:len(t)])
5 _ = plt.bar(t, s, tick_label=mylabels, 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

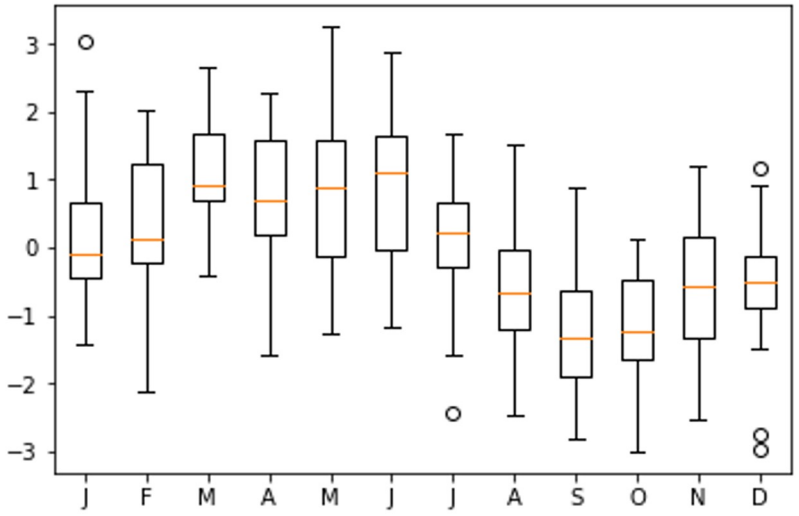
Horizontal analogue given by `barh`
http://matplotlib.org/api/_as_gen/matplotlib.pyplot.barh.html#matplotlib.pyplot.barh

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

# Box & whisker plots

Draw a box and whisker plot. The box extends from the first quartile (Q1: a statistical measure that represents the 25th percentile of a dataset.) to the third quartile (Q3) of the data, with a line at the median.

```
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…

`pyplot.pie(x, … )`

http://matplotlib.org/api/_as_gen/matplotlib.pyplot.pie.html#matplotlib.pyplot.pie

# Subplots

`subplot(nrows, ncols, plot_number)`

Shorthand: `subplot(XYZ)`

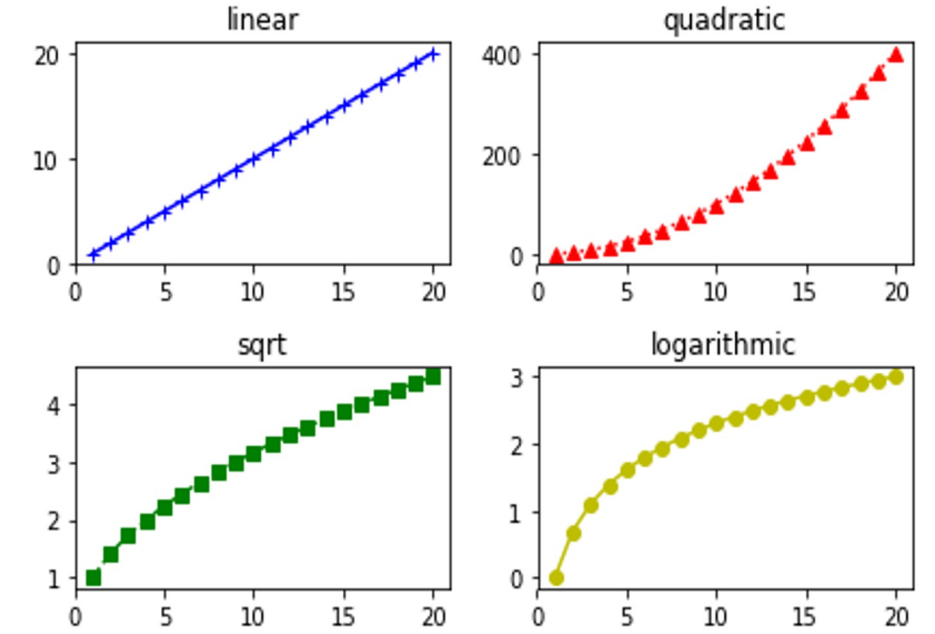
Makes an X-by-Y plot

Picks out the Z-th plot

Counting in row-major order

`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.

```
1  t=np.arange(20)+1
2  plt.subplot(221)
3  plt.plot(t,t,'-+b')
4  plt.title('linear')
5  plt.subplot(222)
6  plt.title('quadratic')
7  plt.plot(t, t**2, ':^r')
8  plt.subplot(223)
9  plt.title('sqrt')
10 plt.plot(t,np.sqrt(t), '--sg')
11 plt.subplot(224)
12 plt.title('logarithmic')
13 plt.plot(t,np.log(t), '-oy')
14 _ = plt.tight_layout()
```



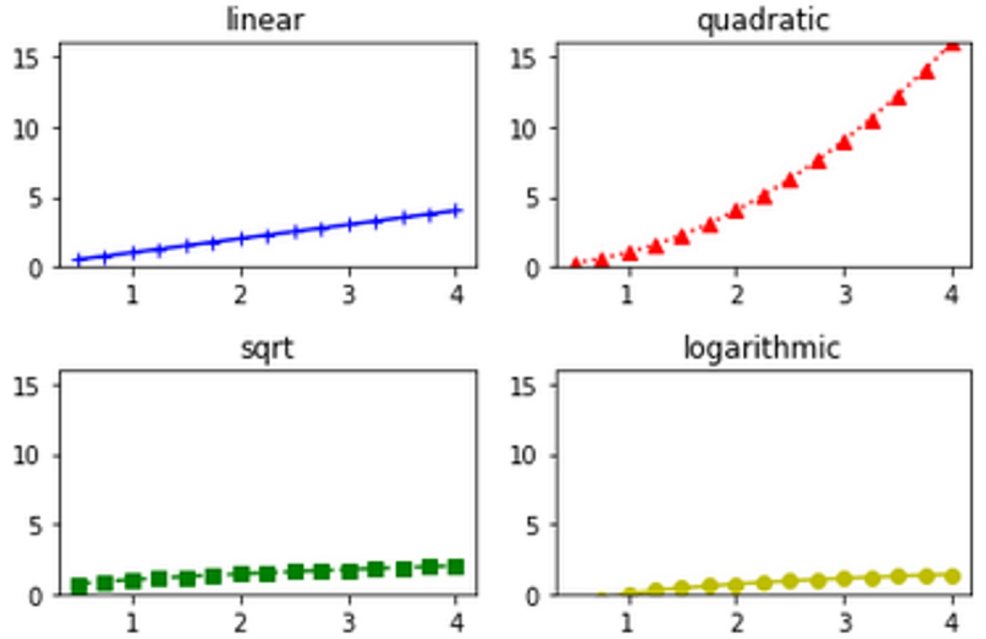
# Specifying axis ranges

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

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

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

```
t = np.arange(0.5,4.25,0.25)
ymax = np.max(t**2)
plt.subplot(221)
plt.plot(t,t,'-+b')
plt.title('linear')
plt.subplot(222)
plt.title('quadratic')
plt.plot(t, t**2, ':^r')
plt.subplot(223)
plt.title('sqrt')
plt.plot(t,np.sqrt(t), '--sg')
plt.subplot(224)
plt.title('logarithmic')
plt.plot(t,np.log(t), '-oy')
for subplt in range(221,225):
    plt.subplot(subplt)
    plt.ylim([0,ymax])
_ = plt.tight_layout()
```



# Nonlinear axis

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

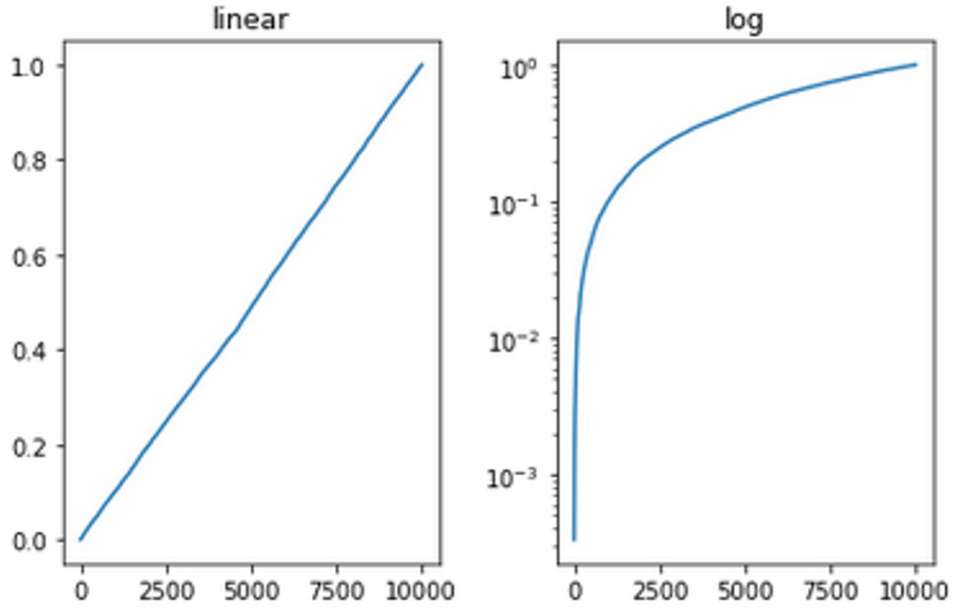
Built-in scales:

- 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

```
y = np.random.uniform(0,1,10000); y.sort()
x = np.arange(len(y))
plt.subplot(121)
plt.plot(x,y)
plt.yscale('linear'); plt.title('linear')
plt.subplot(122)
plt.plot(x, y)
plt.yscale('log'); plt.title('log')
_ = plt.tight_layout()
```



# Saving images

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

Or 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

```
1 random_signs = np.sign(np.random.rand(1000)-0.5)
2 plt.grid(True)
3 plt.title('Random walk of 1000 steps')
4 # cumsum() returns cumulative sums
5 _ = plt.plot(np.cumsum(random_signs))
6 plt.savefig('random_walk.svg')
```

# Animations

`matplotlib.animate` package generates animations

We 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:

https://matplotlib.org/stable/api/animation_api.html

# In-class practice

# Part 2: Intro to Pandas

# Recall Scope of this class

**Part 1: Introduction to Python**

Data types, functions, classes, objects, functional programming

**Part 2: Numerical Computing and Data Visualization**

numpy, scipy, matplotlib, scikit-learn

**Part 3: Dealing with structured data**

**Pandas**, **SQL**, real datasets

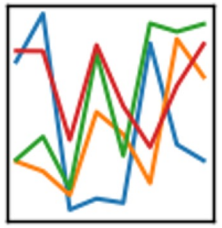
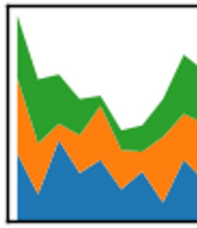
**Part4: Intro to Deep Learning**

PyTorch, Perceptron, Multi-layer perceptron, SGD, regularization, ConvNets

# Pandas

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

Yet another **open-sourced**, **practical**, modern data science tool in Python…

- Database-like structures, largely similar to those available in R
- Well integrated with `NumPy/SciPy`
- Low-level ops implemented in Cython (C+Python=Cython)
- Optimized for most common operations
- E.g., vectorized operations, operations on rows of a table

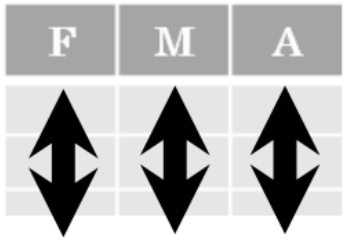
**From the documentation:** pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing **practical**, **real world** data analysis in Python.

https://pandas.pydata.org/ https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

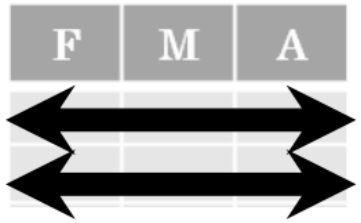
# Tidy Data

A foundation for wrangling in pandas.

In a tidy data set:




&




Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

# Basic Data Structures

**Series**: represents a one-dimensional **labeled** array

Labeled just means that there is an **index** into the array

Support vectorized operations

**DataFrame**: table of rows, with labeled columns

Like a spreadsheet or an R data frame

Support `NumPy ufuncs` (provided data are numeric)

1. Pandas Series Basics (creations and properties)

2. Pandas Series Operations

3. Pandas DataFrames

# pandas Series Basics

Pandas series are Pandas data structure built on top of NumPy arrays.

- Series also contain a **custom index** and an **optional name**, in addition to the array of data.
- They can be created from other data types, but are usually imported form external sources (csv, Excel or SQL database).
- Two or more Series grouped together form a Pandas DataFrame.

```
1 import pandas as pd
2 import numpy as np
3 numbers = np.random.randn(5)
4 s = pd.Series(numbers)
5 s
```

```
0   -0.318743
1    0.807948
2   -0.216362
3   -0.356014
4    1.542122
dtype: float64
```

- pd is the standard alias for Pandas library
- Pd.series function converts NumPy Arrays into Pandas Series.

# `pandas` Series Creation

Can create a `Pandas` series from **array-like** structure (e.g., Python list, `NumPy` array, dict).

```
1 import pandas as pd
2 import numpy as np
3 numbers = np.random.randn(5)
4 s = pd.Series(numbers)
5 s
```

```
0   -0.318743
1    0.807948
2   -0.216362
3   -0.356014
4    1.542122
dtype: float64
```

By default, indices are integers, starting from 0, just like you're used to.

`pandas` tries to infer this data type automatically.

```
1 idx = ['a','b','c','d','e']
2 s = pd.Series(numbers, index=idx)
3 s
```

```
a   -0.318743
b    0.807948
c   -0.216362
d   -0.356014
e    1.542122
dtype: float64
```

But we can specify a different set of indices if we so choose.

**Warning:** providing too few or too many indices is a `ValueError`.

# `pandas` Series Creation

Can create a `Pandas` series from a **dictionary.**

```
1 d = {'dog':3.1415,'cat':42,'bird':0,'goat':1.618}
2 s = pd.Series(d)
3 s
```

```
bird      0.0000
cat      42.0000
dog       3.1415
goat      1.6180
dtype: float64
```

Can create a series from a dictionary. Keys become indices.

```
1 inds = ['dog','cat','bird','goat','cthulu']
2 s = pd.Series(d, index=inds)
3 s
```

```
dog         3.1415
cat        42.0000
bird        0.0000
goat        1.6180
cthulu         NaN
dtype: float64
```

Index `'cthulu'` doesn't appear in the dictionary, so `pandas` assigns it `NaN`, the standard "missing data" symbol.

# `pandas` Series properties

Pandas Series have these key properties:

- **values** – the data array in the Series
- **index** – the index array in the Series
- **name** – the optional name for the Series *(useful for accessing columns in a DataFrame)*
- **dtype** – the data type of the elements in the values array

```
numbers = np.array([2, 50.0, 113.0, 4, 9])
s = pd.Series(numbers, name = "sale")
```

```
# Print out some common and useful attributes
print("\nCommon attributes:")
print(f"Name: {s.name}")
print(f"dtype: {s.dtype}")
print(f"Index: {s.index}")
print(f"Size: {s.size}")
print(f"Shape: {s.shape}")
print(f"Value: {s.values}")
```

```
Common attributes:
Name: sale
dtype: float64
Index: RangeIndex(start=0, stop=5, step=1)
Size: 5
Shape: (5,)
Value: [  2.  50. 113.   4.   9.]
```

1. Pandas Series Basics (creations and properties)

2. Pandas Series Operations

3. Pandas DataFrames

# Indexing

Indexing works like you're used to and supports slices, but **not** negative indexing.

```
1 s = pd.Series([2,3,5,7,11])
2 s[0]
```

```
2
```

This object has type `np.int64`

```
1 s[1:3]
```

```
1    3
2    5
dtype: int64
```

This object is another `pandas` Series.

```
1 s[-1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-22-0e2107f91cbd> in <module>()
----> 1 s[-1]
```

# Indexing

```
1 s = pd.Series([2,3,5,7,11], index=['a','a','a','a','a'])
2 s
```

```
a     2
a     3
a     5
a     7
a    11
dtype: int64
```

**Caution:** indices need not be unique in `pandas` Series. This will only cause an error if/when you try to perform an operation that requires unique indices.

```
1 s['a']
```

```
a     2
a     3
a     5
a     7
a    11
dtype: int64
```

# Slicing and func like Numpy

```
1 s
```

```
dog          3.1415
cat         42.0000
bird         0.0000
goat         1.6180
cthulu          NaN
dtype: float64
```

Series objects are like `np.ndarray` objects, so they support all the same kinds of slice operations, but note that the indices come along with the slices.

```
1 s[s>0]
```

```
dog       3.1415
cat      42.0000
goat      1.6180
dtype: float64
```

Series objects even support most `numpy` functions that act on arrays.

```
1 s**2
```

```
dog           9.869022
cat        1764.000000
bird          0.000000
goat          2.617924
cthulu             NaN
dtype: float64
```

# Modifying `Series` like a dictionary

Series objects are `dict`-like, in that we can **access** and **update** entries via their keys.

```
1 s
```

```
dog         3.1415
cat        42.0000
bird        0.0000
goat        1.6180
cthulu         NaN
dtype: float64
```

```
1 s['goat']
```

```
1.6180000000000001
```

```
1 s['cthulu']=-1
2 s
```

```
dog         3.1415
cat        42.0000
bird        0.0000
goat        1.6180
cthulu     -1.0000
dtype: float64
```

Like a dictionary, accessing a non-existent key is a `KeyError`.

```
1 s['penguin']
```

```
---------------------------------
KeyError
<ipython-input-48-a7df9b66ea8a> i
----> 1 s['penguin']
```

**Note:** I cropped out a bunch of the error message, but you get the idea.

**Not shown:** Series also support the `in` operator: `x in s` checks if `x` appears as an index of Series `s`. Series also supports the dictionary `get` method.

```
1 s
```

```
dog         3.1415
cat        42.0000
bird        0.0000
goat        1.6180
cthulu     -1.0000
dtype: float64
```

```
1 s['cthulu'] = (1,1)
```

```
---------------------------------
ValueError
<ipython-input-50-47579d9278ca>
----> 1 s['cthulu'] = (1,1)
```

Entries of a Series can be of (almost) any type, and they may be mixed (e.g., some floats, some ints, some strings, etc), but they **CAN NOT be lists/tupes**…

```
/Users/keith/anaconda/lib/python2.7/site-packages/pandas
    744                 # GH 6043
    745                 elif _is_scalar_indexer(indexer):
--> 746                     values[indexer] = value
    747
    748                 # if we are an exact match (ex-broad

ValueError: setting an array element with a sequence.
```

More information on indexing: https://pandas.pydata.org/pandas-docs/stable/indexing.html

# Universal functions on `Series`

```
1 s
```

```
dog        3.1415
cat            42
bird            0
goat        1.618
cthulu      abcde
dtype: object
```

Series support universal functions, so long as all their entries support operations.

```
1 s + 2*s
```

```
dog                    9.4245
cat                       126
bird                        0
goat                    4.854
cthulu     abcdeabcdeabcde
dtype: object
```

Series operations require that keys be shared. Missing values become `NaN` by default.

```
1 d = {'dog':2,'cat':1.23456}
2 t = pd.Series(d)
3 t
```

```
cat    1.23456
dog    2.00000
dtype: float64
```

```
1 s+t
```

```
bird          NaN
cat       43.2346
cthulu        NaN
dog        5.1415
goat          NaN
dtype: object
```

To reiterate, Series objects support most `numpy` ufuncs. For example, `np.sqrt(s)` is valid, so long as all entries are positive.

# Methods

Series have an optional `name` attribute.

After it is set, `name` attribute can be changed with `rename` method.

**Note:** this returns a new Series. It **does not** change `s.name`.

```
1 s
```

```
bird      0.0000
cat      42.0000
dog       3.1415
goat      1.6180
dtype: float64
```

```
1 s.name = 'aminals'
2 s
```

```
bird      0.0000
cat      42.0000
dog       3.1415
goat      1.6180
Name: aminals, dtype: float64
```

```
1 s.rename('animals')
```

```
bird      0.0000
cat      42.0000
dog       3.1415
goat      1.6180
Name: animals, dtype: float64
```

This will become especially useful when we start talking about DataFrames, because these name attributes will be column names.

# Mapping and linking `Series` values

Series `map` method works analogously to Python's `map` function. Takes a function and applies it to every entry.

```
1 s = pd.Series(['dog','goat','skunk'])
2 s
```

```
0       dog
1      goat
2     skunk
dtype: object
```

```
1 s.map(lambda s:len(s))
```

```
0    3
1    4
2    5
dtype: int64
```

# Mapping `Series` values

```
1 s = pd.Series(['fruit', 'animal', 'animal', 'fruit', 'fruit'],
2               index=['apple','cat', 'goat','banana','kiwi'])
3 s
```

```
apple      fruit
cat       animal
goat      animal
banana     fruit
kiwi       fruit
dtype: object
```

```
1 t = pd.Series({'fruit':0,'animal':1})
2 s.map(t)
```

```
apple     0
cat       1
goat      1
banana    0
kiwi      0
dtype: int64
```

Series `map` also allows us to change values based on another Series. Here, we're changing the fruit/animal category labels to binary labels.

# Other things

HW5 due

HW6 out

Coming next:

- More on Pandas
- Intro to SQL