Numpy Essentials: Parts 1 and 2

Part 2, Section 1 with Terezija Semenski (LinkedIn Learning)

Matplotlib basics

Data visualization is extremely important in every field of science, especially when it comes to data science. It is easier for the human brain to understand and remember pictures than it is to remember numbers and words. Visualization also makes it effortless to detect trends, patterns, and relationships in groups of data.

Matplotlib is the most popular visualization library that focuses on generating static publicly quality 2D and 3D graphs, as well as animated and interactive visualizations.

Some of the many advantages of Matplot library include, it's easy to get started. Matplotlib is extremely powerful because it allows users to create numerous and diverse plot types. It can be used in variety of user interfaces such as IPhython shells, Python scripts, Jupyter notebooks, as well as web applications and GUI toolkits. It has support for LaTeX-formatted labels and texts and offers control of every aspect of a figure or a plot. It supports high quality output in various formats including PNG, SVG and PDF.

One of the key features of Matplotlib that I find valuable is the possibility to use a programmatic approach in which graphs are created by writing code. You control every aspect of their appearance instead of manually creating graphs using a graphical user interface. This is is extremely important because programmatically created graphics can be made reproducible or easily adjusted when data is updated and are time-saving, as there is no need to redo lengthy and tedious procedures in a GUI. Finally, Matplotlib is open source and therefore data scientists and developers can use it for free.

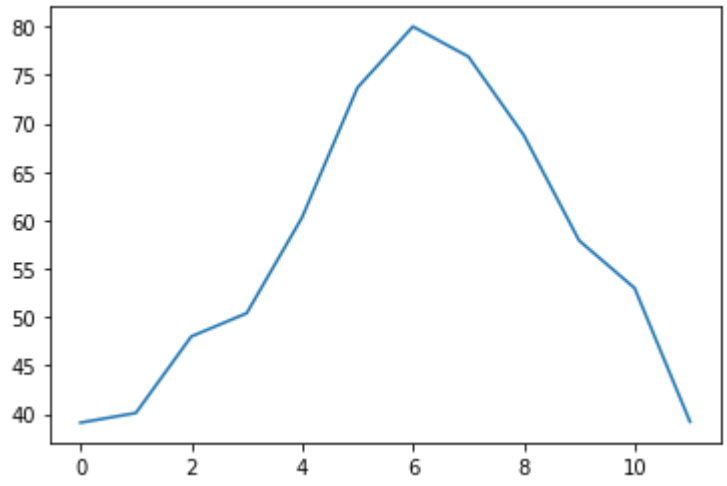
```
%matplotlib notebook
%matplotlib inline
```

%matplotlib notebook -> interactive features
%matplotlib inline -> prints in the notebook directly

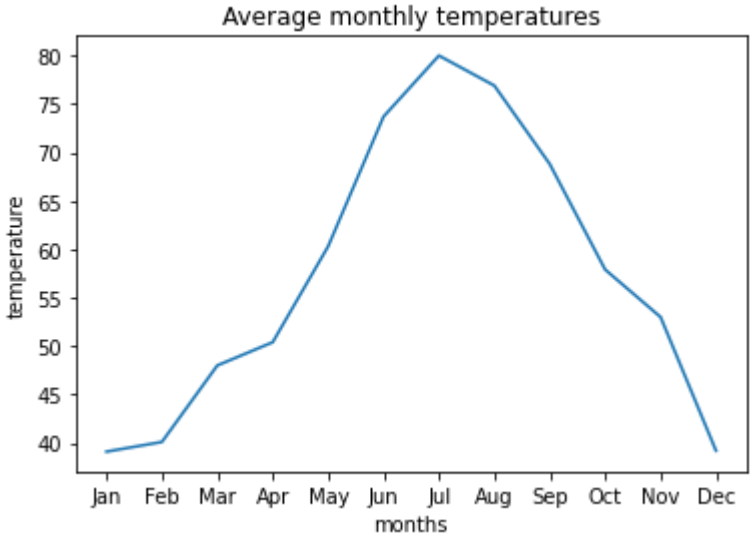
```
import numpy as np
import matplotlib.pyplot as plt
```

When a single parameter is passed, the data values are on the Y axis and the indices are on the X axis.

```
average_monthly_temperatures = [39.1, 40.1, 48.0, 50.4, 60.3, 73.7, 80.0, 76.9, 68.8, 5
fig = plt.figure()
plt.plot(average_monthly_temperatures)
plt.show()
```




```
average_monthly_temperatures = [39.1, 40.1, 48.0, 50.4, 60.3, 73.7, 80.0, 76.9, 68.8, 5
months=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
fig = plt.figure()
plt.plot(months,average_monthly_temperatures)
plt.title("Average monthly temperatures")
plt.xlabel("months")
plt.ylabel("temperature")
plt.show()
```



Saving the figure:

```
fig.savefig('average_monthly_temperatures.png')
```

```
fig.savefig('average_monthly_temperatures.pdf')
```

```
!ls -lh average_monthly_temperatures.png
```

```
-rw-r--r-- 1 root root 19K Oct 22 13:04 average_monthly_temperatures.png
```

```
!ls -lh average_monthly_temperatures.pdf
```

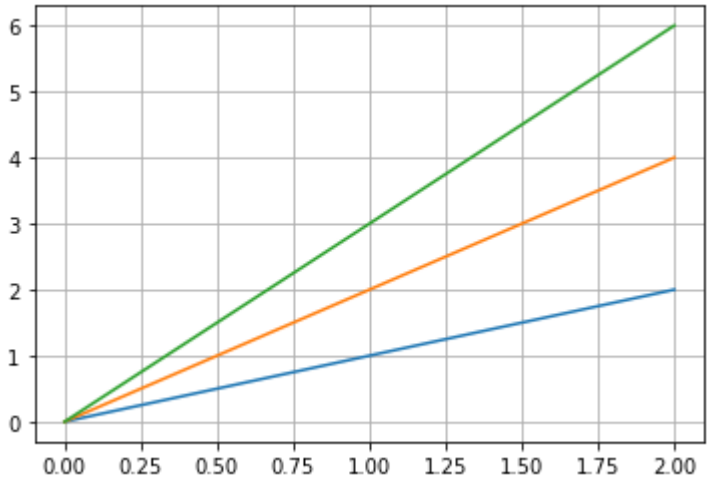
```
-rw-r--r-- 1 root root 13K Oct 22 13:04 average_monthly_temperatures.pdf
```

Understanding figures

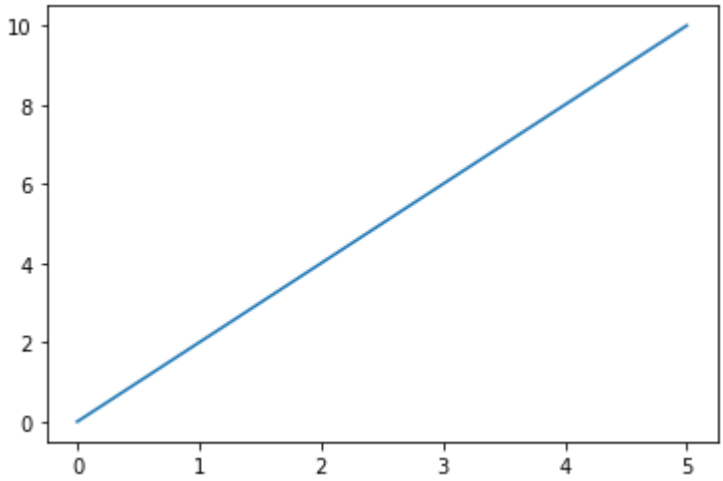
The **figure** is by definition, a high level Matplotlib object that contains all the elements of the output graph. We can arrange multiple graphs in a different ways, so to form a figure. Every element of a figure is customizable. As we can see from the picture, axes is a subsection of a figure where our graph is plotted.

Axes contains a title, x-label, and y-label. Our figure contains only one axis. But a figure can have multiple axis. Each represented one or more graphs. The important thing to notice is the difference between axis and axes. Axis are the number lines that show the scale of the plotted graphs. As we have previously seen in our two dimensional graph, we had two axis, x-axis and y-axis. For three dimensional graphs, we will have three axis. A great thing to add to our graph is a grid that is drawn along major ticks of x and y-axis. So we can easily read the coordinates of various points and understand the function that is drawn.

```
x = np.arange(3)
plt.plot(x,x)
plt.plot(x,2*x)
plt.plot(x,3*x)
plt.grid(True)
plt.show()
```



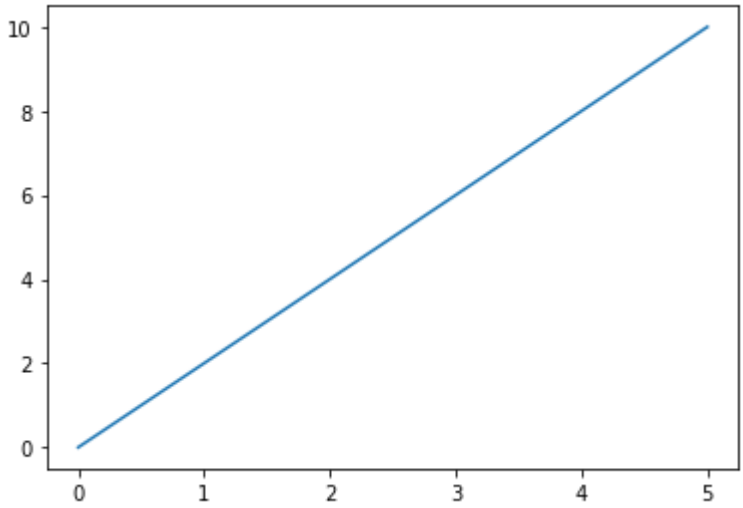

```
x = np.linspace(0,5,5)
y=2*x
plt.plot(x,y)
plt.show()
```



Object-Oriented Method:

The central principle is to create figure objects and then call methods over this. We are going to type, **`fig = plt.figure`** . Now, when we run the code, you see that the figure object is created. Now, we can add axes to figure by typing, **`axes = fig.add_axes()`** . As arguments for add axes method, we'll pass in a list that contains floating points that represent the **left** of the axis, **bottom** of the axis, **width** and height. So we'll know exactly where our axes will be placed. Next we want to plot on that set of axes. We'll do this by typing, **`axes.plot(x, y)`** , and **`plt.show()`** .

```
fig = plt.figure()
axes = fig.add_axes([0.1,0.1,0.8,0.8])
axes.plot(x,y)
plt.show()
```



Matplotlib subplots functionality

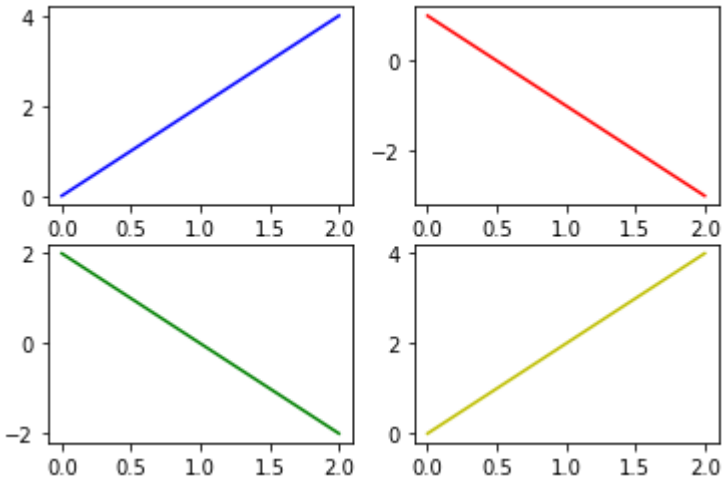
Subplots are group of small axis that can stand together between a single figure. They can be in sets, grids or plots or some other complicated layout.

There are two ways to do this: **`plt.subplot()`** function that creates only a single subplot between a grid and **`plt.subplots()`** function that creates a full grid of subplots at once.

The **`plt.subplot()`** function takes these arguments, **number of rows**, **number of columns** and the third argument is the **index of the plot** we're referring to.

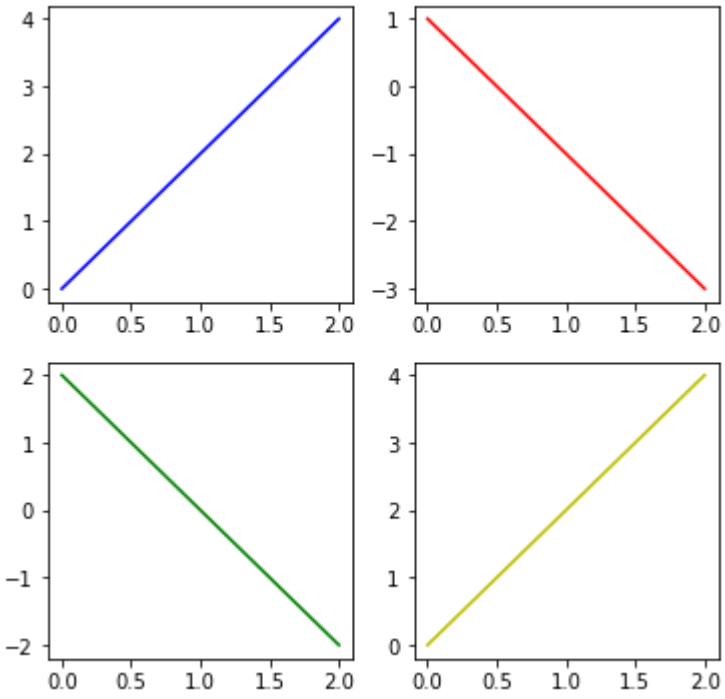
```
fig=plt.figure()
x=np.arange(3)
y=2*x
plt.subplot(2,2,1)
plt.plot(x,y,'b')

plt.subplot(2,2,2)
plt.plot(x,1-y,'r')

plt.subplot(2,2,3)
plt.plot(x,2-y,'g')
```

```
plt.subplot(2,2,4)
plt.plot(x,y,'y')

plt.show()
```



This way of creating subplots can become tedious. In cases where we are creating a large grid of subplots, the way to do this elegantly is with the **`plt.subplots()`** function. We can create all four subplots with just one line of code by typing **`fig, axes = plt.subplots(2, 2, figsize=(6,6)`** Now notice here we still have to copy the plot function four times in order to plot our four graphs.

```
fig, axs = plt.subplots(2, 2, figsize=(6,6))
axs[0, 0].plot(x, y, 'b')
axs[0, 1].plot(x, 1-y, 'r')
axs[1, 0].plot(x, 2-y, 'g')
axs[1, 1].plot(x, y, 'y')
plt.show()
```

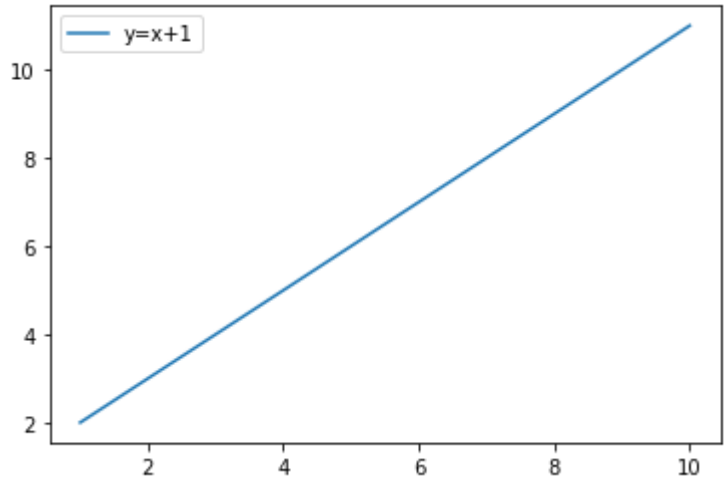


Understanding Legends

The legend is the description of each of the graphs on given axis. Matplotlib has a built-in function to create a legend called **`plt.legend()`** .

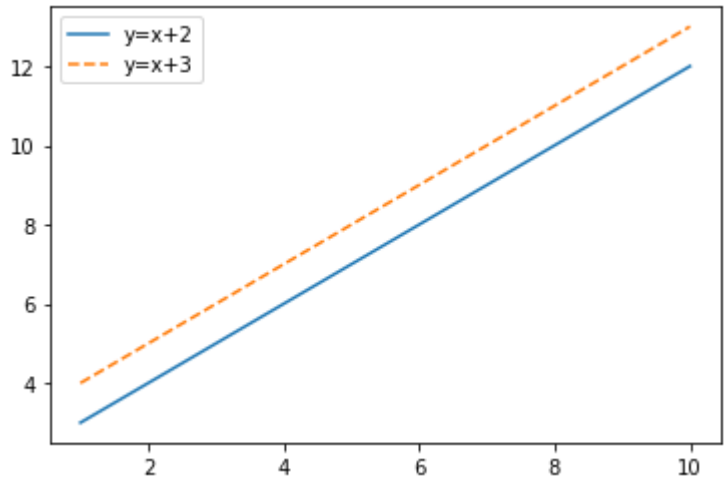
```
x = np.linspace(1,10)
first_line = plt.plot(x, x+1, label= 'y=x+1')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fda72a1f510>
```




```
second_line, = plt.plot(x,x+2,linestyle='solid')
second_line.set_label('y=x+2')
third_line, = plt.plot(x,x+3,linestyle='dashed')
third_line.set_label('y=x+3')
plt.legend()
```

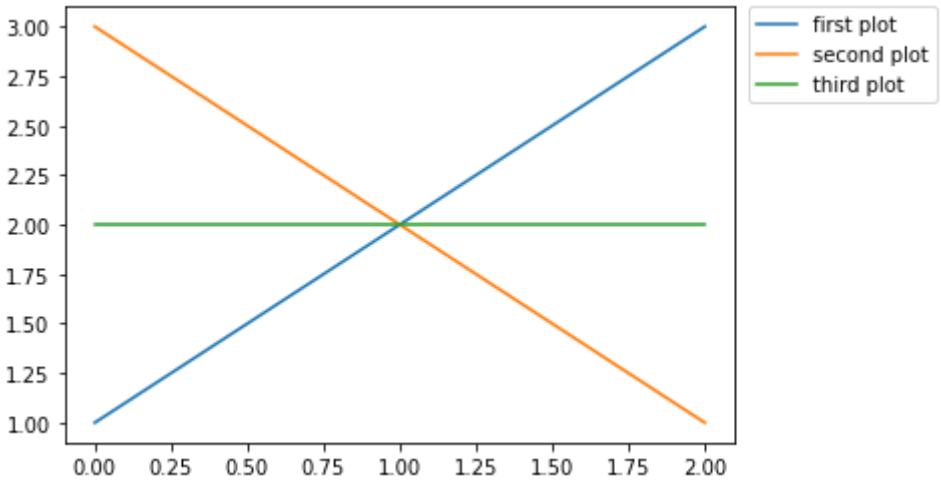
```
<matplotlib.legend.Legend at 0x7fda729c8c10>
```



The **`bbox_to_anchor`** keyword gives a great degree of control for manual legend placement. For example, if you want your axes legend located at the figure's top right-hand corner instead of the axes' corner, simply specify the corner's location and the coordinate system of that location:

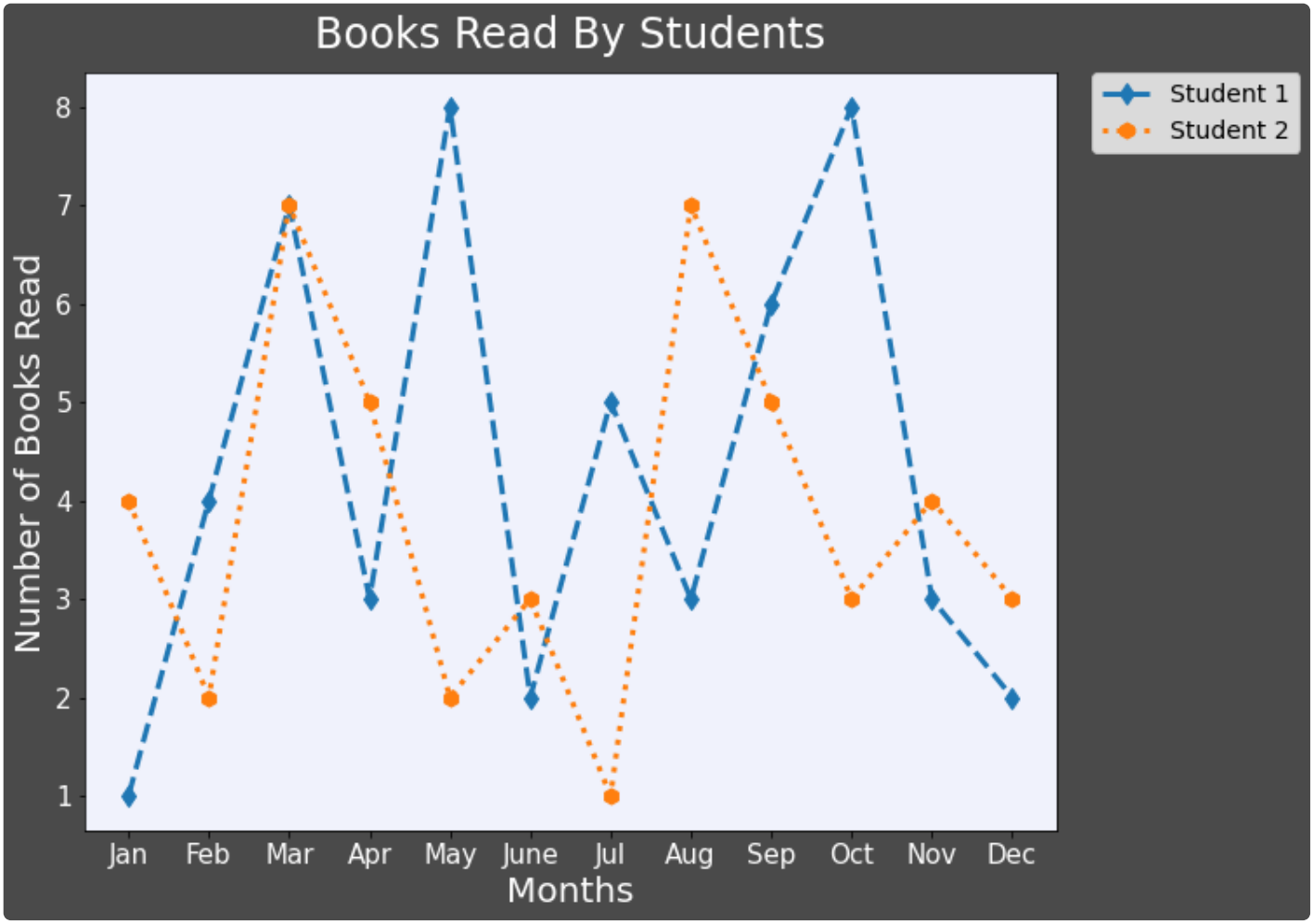
```
ax.legend(bbox_to_anchor=(1, 1), bbox_transform=fig.transFigure)
```

```
first_plot,=plt.plot([1,2,3],label='first plot')
second_plot,=plt.plot([3,2,1],label='second plot')
third_plot,=plt.plot([2,2,2],label='third plot')
plt.legend(bbox_to_anchor=(1.02, 1.0), borderaxespad=0);
```



SECTION CHALLENGE: plot a graph according to directions given in video

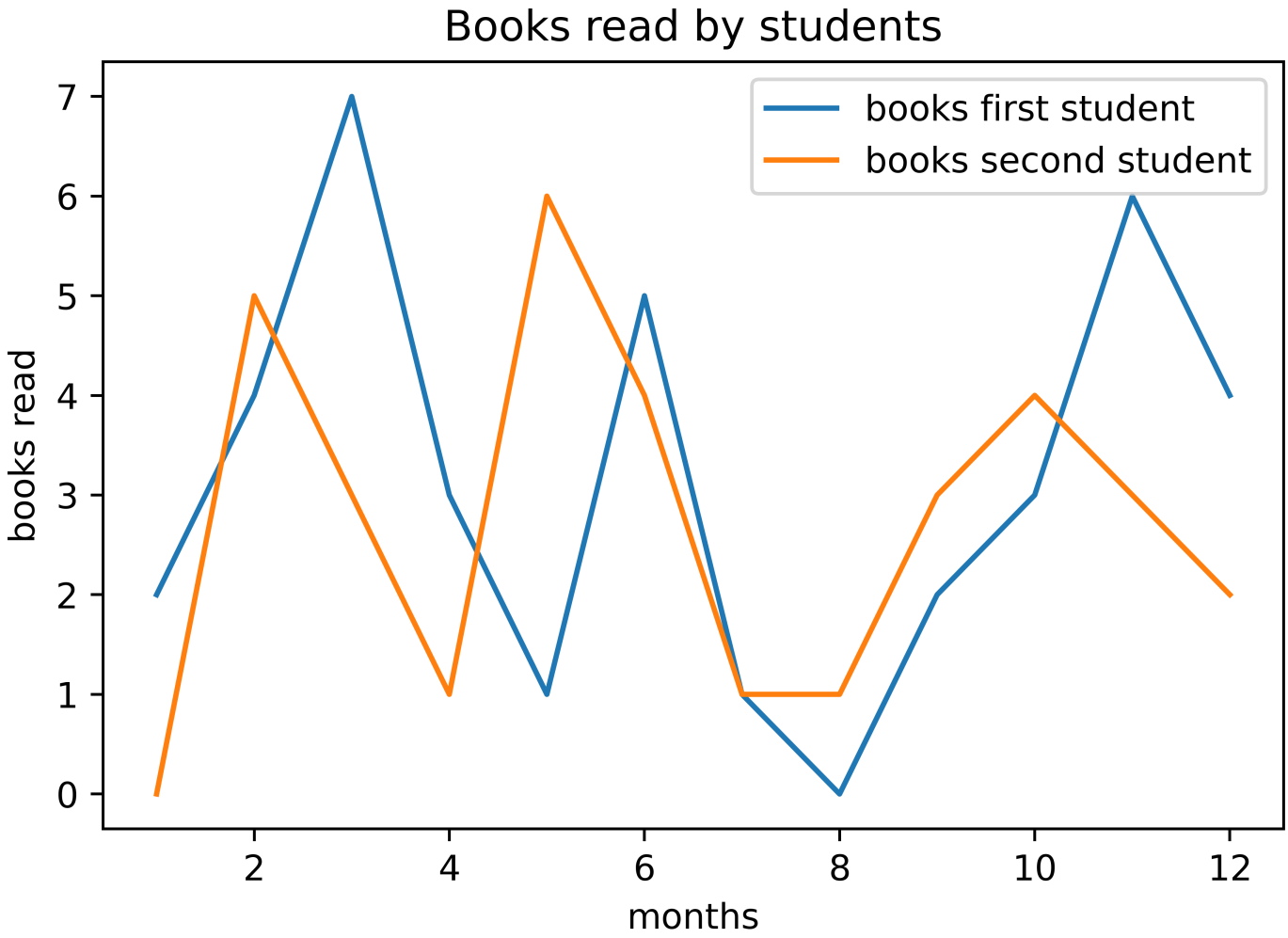
```
first_student = [1, 4, 7, 3, 8, 2, 5, 3, 6, 8, 3, 2]
second_student = [4, 2, 7, 5, 2, 3, 1, 7, 5, 3, 4, 3]
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June',
          'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

fig = plt.figure(figsize=(10,8), facecolor='#4a4a4a')
ax = plt.axes()
ax.set_facecolor('#f0f2fc')
plt.plot(months, first_student, label="Student 1", linestyle='dashed', linewidth=3, mar
plt.plot(months, second_student, label="Student 2", linestyle='dotted', linewidth=3, ma
ax.set_ylabel("Number of Books Read", fontsize=20, color="white")
ax.set_xlabel('Months', fontsize=20, color="white")
plt.title("Books Read By Students", fontsize=24, color="white", pad=14)
plt.xticks(fontsize = 15, color="white")
plt.yticks(fontsize = 15, color="white")
plt.legend(bbox_to_anchor=(1.25, 1.0), borderaxespad=0, fontsize=14);
plt.show()
```



Implementing a figure

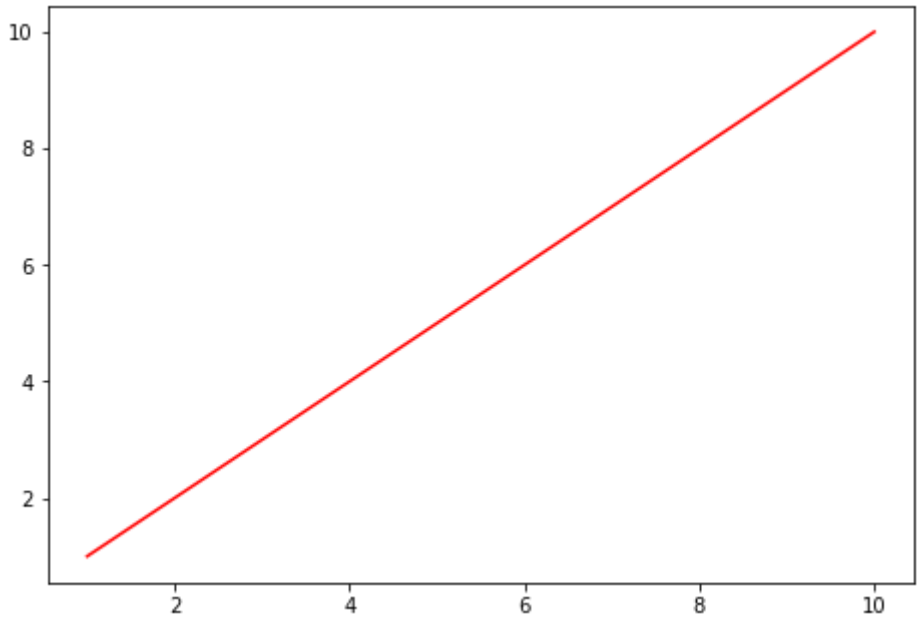
```
import matplotlib.pyplot as plt
%matplotlib inline
```

```
plt.figure(dpi=720)
first_student_books=[2,4,7,3,1,5,1,0,2,3,6,4]
second_student_books=[0,5,3,1,6,4,1,1,3,4,3,2]
first_line=plt.plot(range(1,13),first_student_books)
second_line=plt.plot(range(1,13),second_student_books)
plt.xlabel('months')
plt.ylabel('books read')
plt.legend(['books first student','books second student'],loc=1)
plt.title('Books read by students')
plt.show()
```

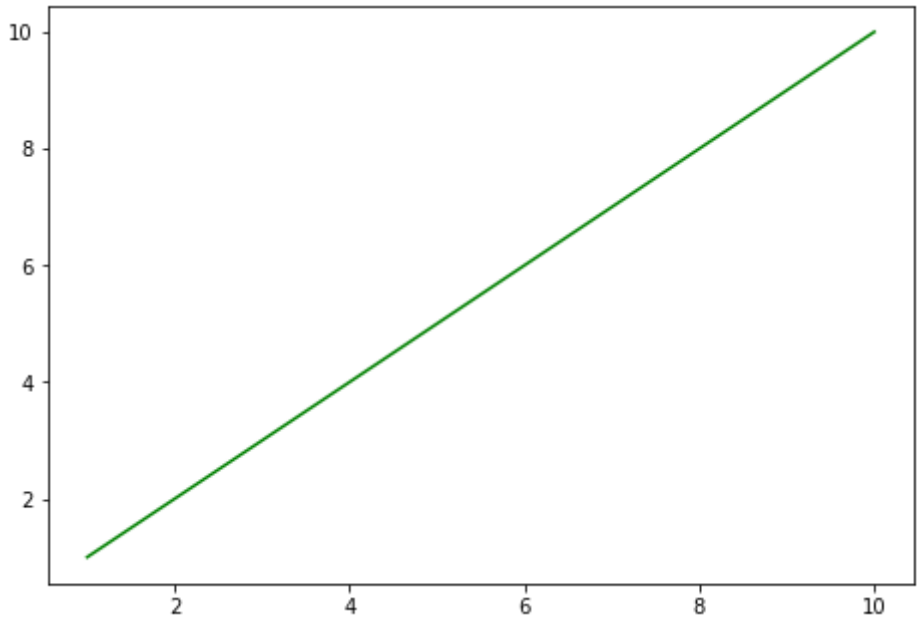


Colors and Styles

```
first_figure = plt.figure()
x = np.linspace(1, 10)
y = np.linspace(1, 10)
ax=first_figure.add_axes([0,0,1,1])
ax.plot(x,y, color='red');
```

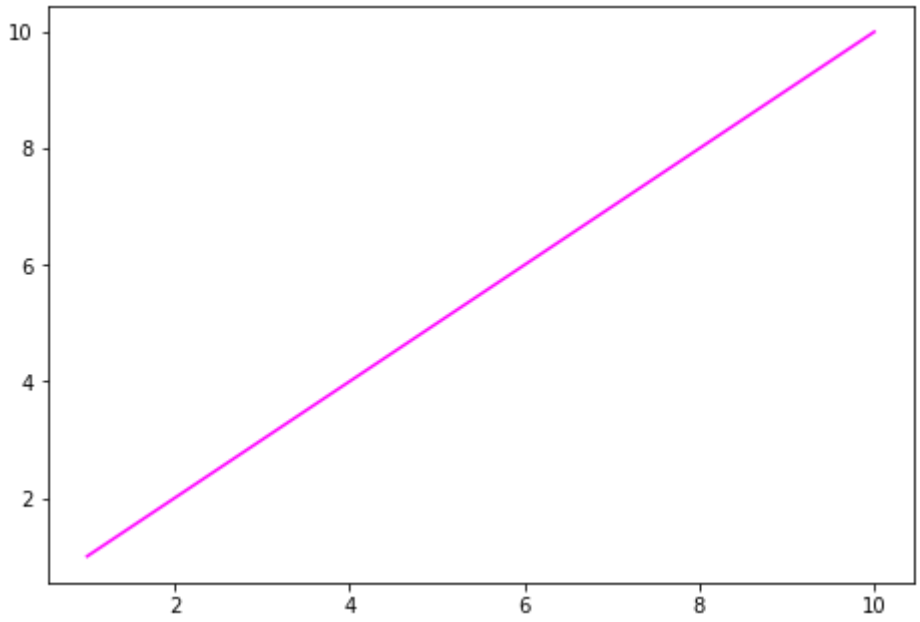


```
second_figure = plt.figure()
ax=second_figure.add_axes([0,0,1,1])
ax.plot(x,y, color='g');
```



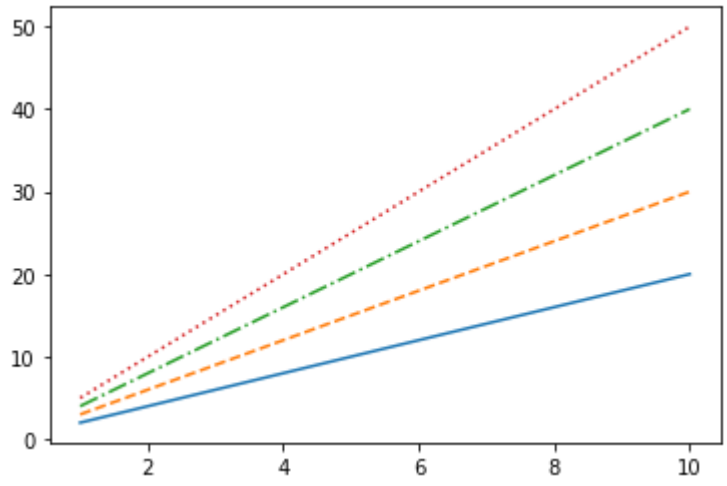

```
third_figure = plt.figure()
ax=third_figure.add_axes([0,0,1,1])
ax.plot(x,y, color='#FF00FF')
```

```
[<matplotlib.lines.Line2D at 0x7fda715ef190>]
```



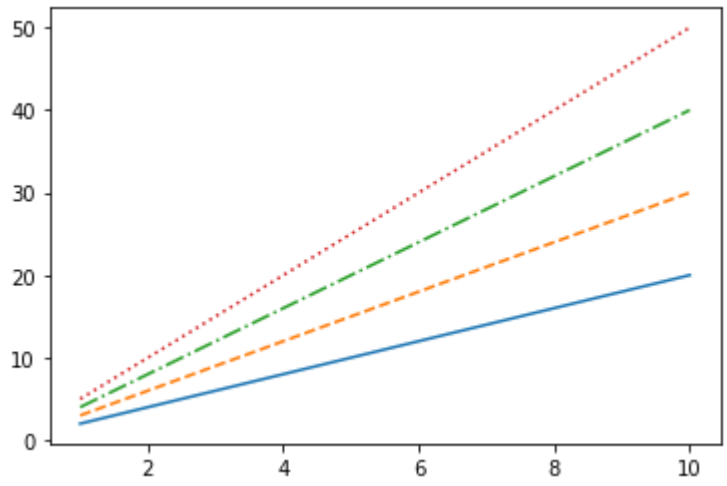

```
plt.plot(x,2*x,linestyle='solid')
plt.plot(x,3*x,linestyle='dashed')
plt.plot(x,4*x,linestyle='dashdot')
plt.plot(x,5*x,linestyle='dotted')
```

```
[<matplotlib.lines.Line2D at 0x7fda72cbd250>]
```

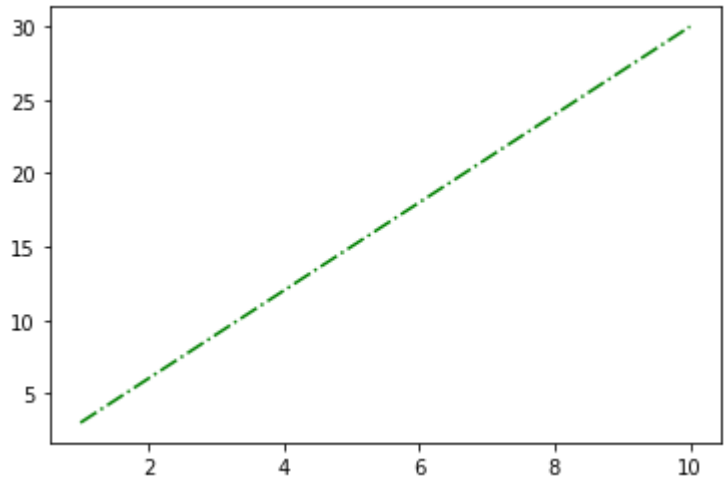



```
plt.plot(x,2*x,linestyle='-')
plt.plot(x,3*x,linestyle='--')
plt.plot(x,4*x,linestyle='-.')
plt.plot(x,5*x,linestyle=':')
```

```
[<matplotlib.lines.Line2D at 0x7fda70818710>]
```




```
plt.plot(x, 3*x ,'-.g');
```



Advanced Matplotlib commands

When we use Matplotlib for plotting, it will automatically create a linear scale. Sometimes, creating plots on a linear scale won't give us clear and valuable results. The solution for our struggle is to use one of the three types of non-

linear scales: **`logarithmic`** , **`symmetrical logarithmic`** , or **`logit scale`** .

We will use **`logarithmic scale`** when we have a series of values where each value equals the previous value multiplied with a constant. In that case, values can be represented by equidistant ticks on the logarithmic scale.

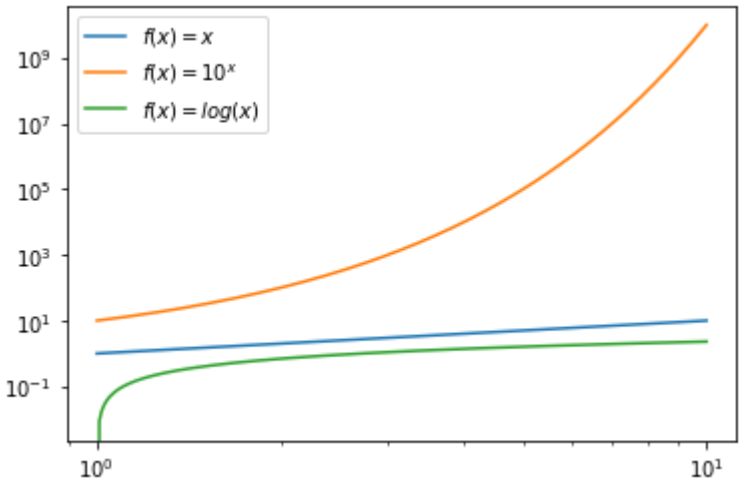
`Symmetrical logarithmic` scale is used when we want to represent non-positive numbers. The logarithmic scale is one of the most used nonlinear scales. Usually we'll use powers of 10. We could also use some other bases that would narrow or widen the spacing of plotted elements.

Use:

- **`plt.xscale()`**
- **`plt.yscale()`**

Change the base of the scale by passing **`basex=`** or **`basey=`** .

```
# $ makes label italics

x = np.linspace(1, 10, 1024)
plt.xscale('log')
plt.yscale('log')

plt.plot(x, x, label ='$f(x)=x$')
plt.plot(x, 10**x, label ='$f(x)=10^x$')
plt.plot(x, np.log(x),label ='$f(x)=log(x)$')

plt.legend()
plt.show()
```



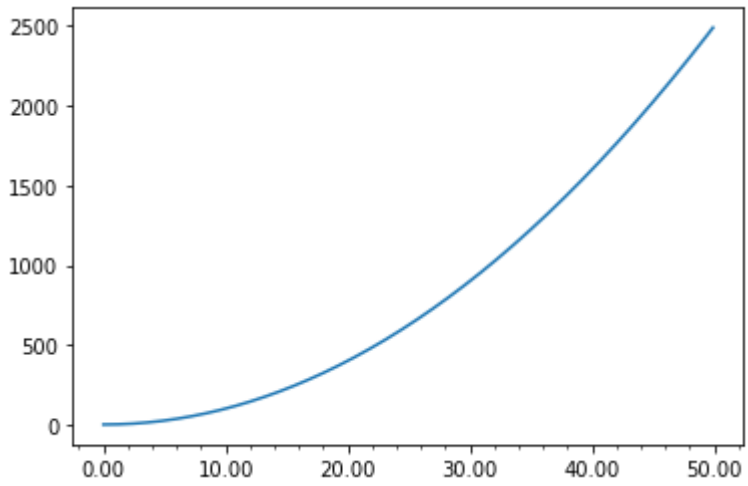
`Matplotlib.ticker` is a Matplotlib module that provides a general tick management system so we can have full control of tick placement using different classes.

For info on setting a formatter and how to use these aspects of Matplotlib.

```
from matplotlib import ticker
```

```
from matplotlib.ticker import (MultipleLocator, AutoMinorLocator)
```

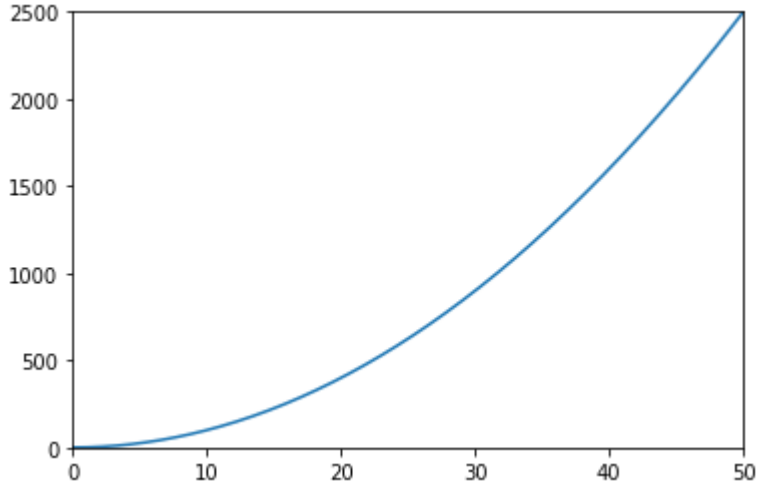
```
x = np.arange(0.0, 50.0, 0.1)
y = x**2

fig, ax = plt.subplots()
ax.plot(x,y)

formatter = ticker.FormatStrFormatter('%1.2f')
ax.xaxis.set_major_locator(MultipleLocator(10))
ax.xaxis.set_major_formatter(formatter)

ax.xaxis.set_minor_locator(MultipleLocator(2))

plt.show()
```



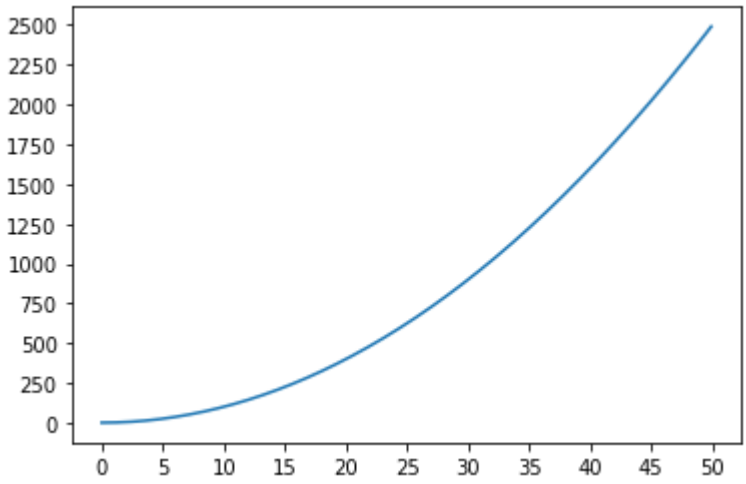
Setting axis limits:

- **ax.set_xlim([low, high])**
- **ax.set_ylim([low, high]**

```
x = np.arange(0.0, 50.0, 0.1)
y = x**2
fig, ax = plt.subplots()
ax.plot(x,y)

ax.set_xlim([0, 50])
ax.set_ylim([0, 2500])

plt.show()
```




```
x = np.arange(0.0, 50.0, 0.1)
y = x**2

fig, ax = plt.subplots()
ax.plot(x,y)

ax.set_xticks([0,5,10,15,20,25,30,35,40,45,50])
ax.set_yticks([0,250,500,750,1000,1250,1500,1750,2000,2250,2500])

plt.show()
```



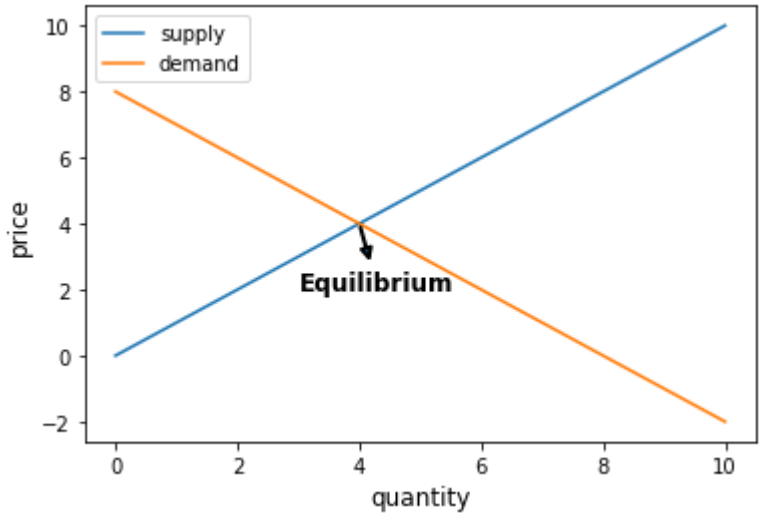
Adding annotations

Annotations are used to describe specific details on the plot so we can draw attention to points of interest on the graph, call out surprising features, or explain significance of a wiggle. Matplotlib provides a few modules to add text, arrows, and shapes on our plot. So we can add text annotations, arrows, graphical annotations, and even image annotations.

Use:

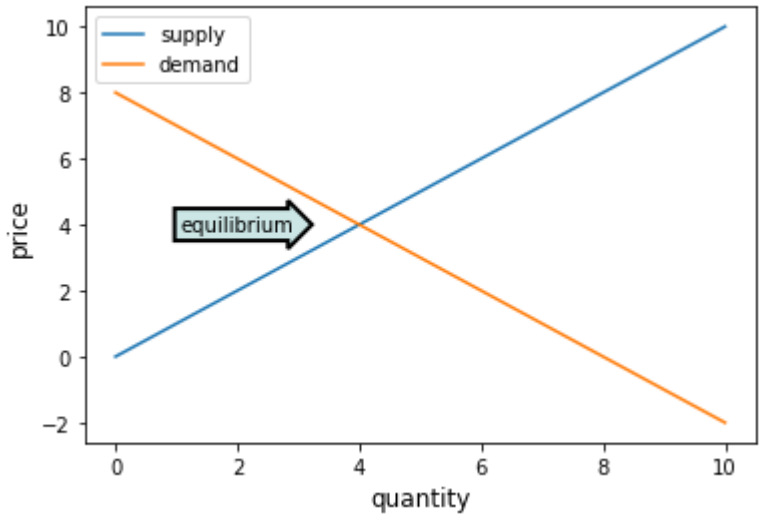
- **`ax.annotate()`**
- **`arrowprops=dict(linewidth=, arrowstyle='')`**

```
x = np.linspace(0, 10)

y1 = x
y2 = 8-x

fig, ax = plt.subplots()
plt.plot(x,y1,label='supply')
plt.plot(x,y2,label='demand')

ax.annotate("Equilibrium", xy=(4,4), xytext=(3,2), \
             fontsize=12, fontweight='semibold',\
             arrowprops=dict(linewidth=2, arrowstyle="<|-"))


plt.xlabel('quantity',fontsize=12)
plt.ylabel('price',fontsize=12)


plt.legend()
plt.show()
```




```
x = np.linspace(0, 10)
y1 = x
y2 = 8-x

# Plot the data
fig, ax = plt.subplots()
plt.plot(x,y1,label='supply')
plt.plot(x,y2,label='demand')

# Annotate the equilibrium point with arrow and text
bbox_props = dict(boxstyle="rarrow", fc=(0.8, 0.9, 0.9), lw=2)
t = ax.text(2,4, "equilibrium", ha="center", va="center", rotation=0,
            size=10,bbox=bbox_props)

# Label the axes
plt.xlabel('quantity',fontsize=12)
```

```
plt.ylabel('price',fontsize=12)


plt.legend()
plt.show()
```



Matpolitlib Patches Class (polygons, etc):

We can also add different graphical annotations using the Matplotlib class called **`patches`** . The most used shapes are circle, ellipse, wedge and polygon.

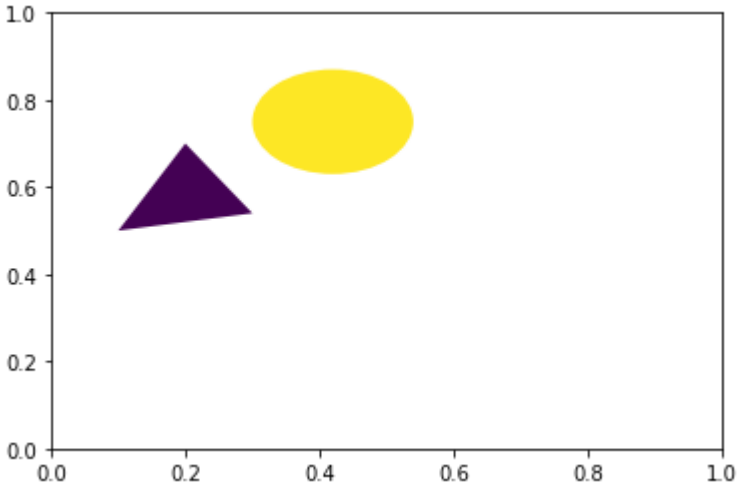
First, you have to import circle, polygon, and ellipse from the **`patches`** class and **`PatchCollection`** from **`collections`** .

We'll define fig and ax by calling **`plt.subplots()`** function and **`patches`** . Draw a circle by passing as parameters coordinates for the center and radius. And for the triangle, pass coordinates of three points to polygon patch.

Lastly, we just need to draw the patches and show our figure.

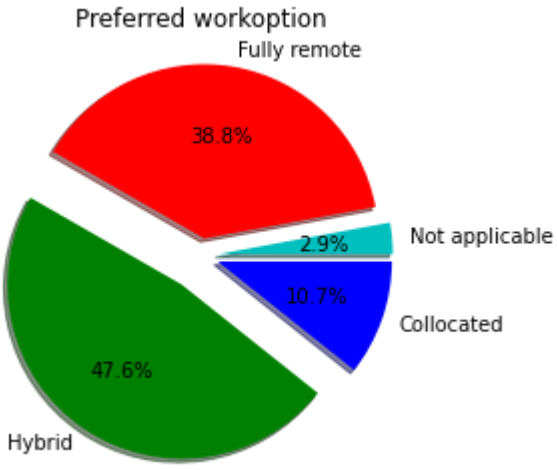
```
from matplotlib.patches import Circle, Polygon
from matplotlib.collections import PatchCollection


fig, ax = plt.subplots()
patches = []

# draw circle and triangle
circle = Circle((.42,.75),0.12)
triangle = Polygon([[.1,.5],[.2,.7],[.3,.54]], True)

patches += [circle,triangle]

# Draw the patches
colors = 100*np.random.rand(len(patches)) # set random colors
p = PatchCollection(patches)
p.set_array(np.array(colors))
ax.add_collection(p)
```

```
# Show the figure
plt.show()
```



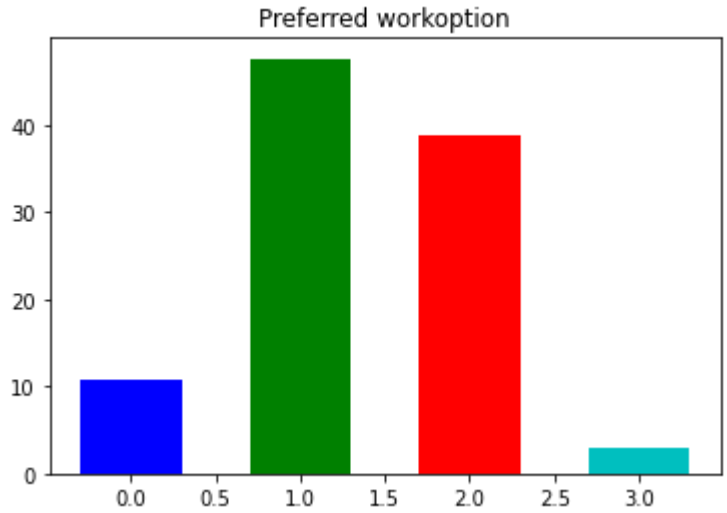
Creating pie charts and bar charts

```
preferred_workoption = [10.7, 47.6, 38.8, 2.9]
colors = ['b', 'g', 'r', 'c']
labels = ['Collocated', 'Hybrid', 'Fully remote', 'Not applicable']
explode = (0.1, 0.2, 0.1, 0.1)  # 0 leaves wedge in place, 0+ pushes it out that much

plt.pie(preferred_workoption, colors=colors, labels=labels,
        explode=explode, autopct='%1.1f%%',
        counterclock=False, shadow=True)

plt.title('Preferred workoption')
plt.show()
```




```
preferred_workoption = [10.7, 47.6, 38.8, 2.9]
colors = ['b', 'g', 'r', 'c']
labels = ['Collocated', 'Hybrid', 'Fully remote', 'Not applicable']
widths= [0.6, 0.6, 0.6, 0.6]
plt.bar(range(0, 4), preferred_workoption, width=widths, color=colors, align='center')
plt.title('Preferred workoption')
```

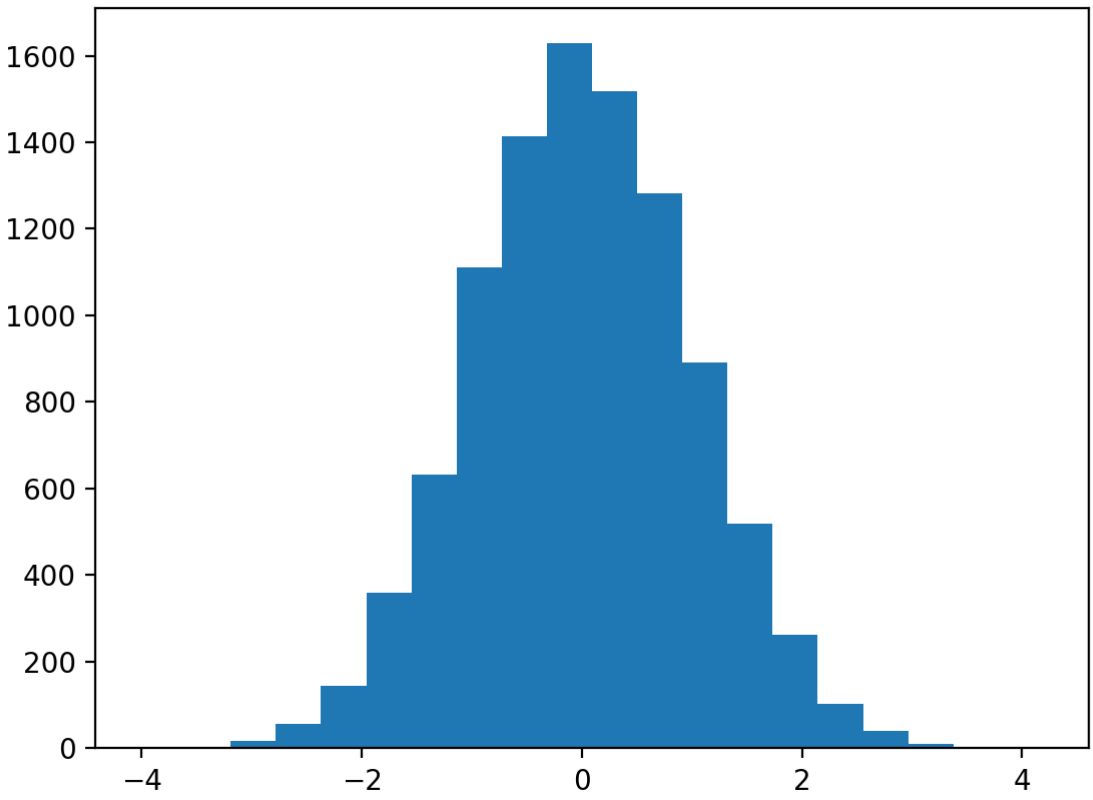
```
plt.show()
```



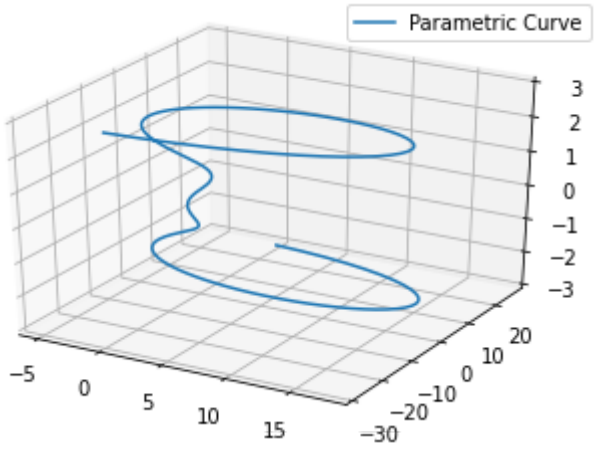
Advanced plots: Histograms and 3D

```
from mpl_toolkits.mplot3d import Axes3D # For 3D plots
```

```
X = np.random.randn(10000)
plt.hist(X, bins = 20)
plt.show()
```



```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
theta = np.linspace(-3 * np.pi, 3 * np.pi, 200)
z = np.linspace(-3, 3, 200)
r = z**3 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)

ax.plot(x, y, z, label='Parametric Curve')
ax.legend()
plt.show()
```



Numpy Essentials: Parts 1 and 2

Part 2, Section 2 with Terezija Semenski (LinkedIn Learning)

Universal functions

NumPy has a rich collection of universal functions, or ufuncs, that you can use to eliminate loops and optimize your code. Universal functions are basically Python objects that belong to NumPy ufunc class and encapsulate behavior of a function. You have already experienced some of that in NumPy part 1 where we learned arithmetic functions and statistical functions. Many other examples of universal functions can be found in trigonometry, summary statistics, and comparison operations

```
from __future__ import print_function
import numpy as np
```

```
numbers=np.arange(1,11)
numbers
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

`np.sin()` and **`np.log`** => Applied to an array in an element-by-element fashion

```
np.sin(numbers)
```

```
array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 , -0.95892427,
       -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849, -0.54402111])
```

```
np.log(numbers)
```

```
array([0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791,
       1.79175947, 1.94591015, 2.07944154, 2.19722458, 2.30258509])
```

Custom Functions with `np.frompyfunc()`

You can create your own universal function using three simple steps.

- First, use a def keyword to define a function.
- Second, add created function to NumPy ufunc library using the **`np.frompyfunc()`** method.
- Third, call this function over a NumPy array.

Info about **`np.frompyfunc()`** : Parameters ---------- func : Python function object An arbitrary Python function. nin : int The number of input arguments. nout : int The number of objects returned by `func` . identity : object, optional The value to use for the `~numpy.ufunc.identity` attribute of the resulting object. If specified, this is equivalent to setting the underlying C `identity` field to `PyUFunc_IdentityValue` . If omitted, the identity is set to `PyUFunc_None` . Note that this is *not* equivalent to setting the identity to `None` , which implies the operation is reorderable.

```
    Returns
    -------
    out : ufunc
        Returns a NumPy universal function (``ufunc``) object.
```

```
# creating numpy array
integers = np.arange(1, 101)
print("integers     :", *integers)

# creating own function
def modulo(val):
  return (val % 10)

# adding into numpy
mod_10=np.frompyfunc(modulo, 1, 1)

# using function over numpy array
mod_integers=mod_10(integers)
print("mod_integers :", *mod_integers)
```

```
integers     : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
mod_integers : 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
```

Introducing strides

Strides are the indexing scheme in ndarrays and specify the number of bytes to jump to find the next element and give insight into the memory usage of the data.

```
numbers = np.arange(10, dtype = np.int8)
numbers
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int8)
```

```
numbers.strides   # This tells us each element is 1 byte apart from the next
```

```
(1,)
```

```
numbers.shape = 2,5
numbers
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]], dtype=int8)
```

```
numbers.strides  # first integer is bytes to next row, second is bytes to next column
```

```
(5, 1)
```

```
first_array = np.zeros((100000,))
first_array
```

```
array([0., 0., 0., ..., 0., 0., 0.])
```

```
second_array = np.zeros((100000 * 100, ))[::100]
second_array
```

```
array([0., 0., 0., ..., 0., 0., 0.])
```

```
first_array.shape
```

```
(100000,)
```

```
second_array.shape
```

```
(100000,)
```

```
first_array.strides
```

```
(8,)
```

```
second_array.strides
```

```
(800,)
```

```
%timeit first_array.sum()
```

```
29.8 µs ± 378 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%timeit second_array.sum()
```

```
146 µs ± 864 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Structured arrays

Numpy has a special type of arrays called **structured** or **record** arrays. They're effective in cases when you're performing computations and want to keep closely related data together. We can use them for grouping data of different types and sizes. The way to achieve that is with data containers called fields. Each data field contains data with the same or different type or size.

```
student_records = np.array([('Lazaro','Oneal', '0526993', 2009, 2.33), ('Dorie','Salina
       dtype=[('name', (np.str_, 10)),('surname', (np.str_, 10)), ('id', (np.str_,7)),(
student_records
```

```
array([('Lazaro', 'Oneal', '0526993', 2009, 2.33),
       ('Dorie', 'Salinas', '0710325', 2006, 2.26),
       ('Mathilde', 'Hooper', '0496813', 2000, 2.56),
       ('Nell', 'Gomez', '0740631', 2003, 2.22),
       ('Lachelle', 'Jordan', '0490888', 2003, 2.13),
       ('Claud', 'Waller', '0922492', 2004, 3.6 ),
       ('Bob', 'Steele', '0264843', 2002, 2.79),
       ('Zelma', 'Welch', '0885463', 2007, 3.69)],
      dtype=[('name', '<U10'), ('surname', '<U10'), ('id', '<U7'), ('graduation_year',
'<i4'), ('gpa', '<f8')])
```

```
student_records[['id','graduation_year']]
```

```
array([('0526993', 2009), ('0710325', 2006), ('0496813', 2000),
       ('0740631', 2003), ('0490888', 2003), ('0922492', 2004),
       ('0264843', 2002), ('0885463', 2007)],
      dtype={'names':['id','graduation_year'], 'formats':['<U7','<i4'], 'offsets':
[80,108], 'itemsize':120})
```

np.sort()

Another interesting feature of structured arrays is that you can use a sort function and passed order as a parameter. We need to pass the value of the field according to which we want to sort our array.

```
students_sorted_by_surname = np.sort(student_records, order='surname')
print('Students sorted according to the surname :\n', students_sorted_by_surname)
```

```
Students sorted according to the surname :
 [('Nell', 'Gomez', '0740631', 2003, 2.22)
 ('Mathilde', 'Hooper', '0496813', 2000, 2.56)
 ('Lachelle', 'Jordan', '0490888', 2003, 2.13)
 ('Lazaro', 'Oneal', '0526993', 2009, 2.33)
 ('Dorie', 'Salinas', '0710325', 2006, 2.26)
 ('Bob', 'Steele', '0264843', 2002, 2.79)
 ('Claud', 'Waller', '0922492', 2004, 3.6 )
 ('Zelma', 'Welch', '0885463', 2007, 3.69)]
```

```
students_sorted_by_grad_year = np.sort(student_records, order='graduation_year')
print('Students sorted according to the graduation year :\n', students_sorted_by_grad_y
```

```
Students sorted according to the graduation year :
 [('Mathilde', 'Hooper', '0496813', 2000, 2.56)
 ('Bob', 'Steele', '0264843', 2002, 2.79)
 ('Lachelle', 'Jordan', '0490888', 2003, 2.13)
 ('Nell', 'Gomez', '0740631', 2003, 2.22)
 ('Claud', 'Waller', '0922492', 2004, 3.6 )
 ('Dorie', 'Salinas', '0710325', 2006, 2.26)
```

```
('Zelma', 'Welch', '0885463', 2007, 3.69)
('Lazaro', 'Oneal', '0526993', 2009, 2.33)]
```

Dates and time in NumPy

Dates and time are critical especially in the time series analytics. It's a specific way of analyzing a sequence of data points collected over internal of time. Time series analytics is used in many different industries like finance, retail, and economics. Some examples of time series analytics used in many different areas and industries include weather data like rainfall measurements and temperature readings, heart rate monitoring, financial data, quarter sales, stock prices and interest rates, and industry forecasts.

If you're familiar with Python, you know that the daytime option is used for date time types. NumPy has a similar data time object called datetime 64. Datetime 64 object is constructed from the ISO 8601 string universal date format. The default date unit supported are years, months, weeks and days. While the time units are hours, minutes, seconds and milliseconds. It also accepts the string NAT which stands for Not a Time Value.

```
np.datetime64('2022-03-01')
```

```
numpy.datetime64('2022-03-01')
```

```
np.datetime64('2022-03')
```

```
numpy.datetime64('2022-03')
```

np.busday_count()

NumPy has many useful functions that deal with dates called "bus days functions", short for business days functions. For instance, if we want to find out the number of business days in 2022, we would call the **busday_count()** function and pass arguments of the current year, 2022, and the following year, 2023.

If we want to find out the number of weekdays in June, 2022, we would pass two arguments 2022-06 and 2022-07.

```
    Parameters

    begindates : array_like of datetime64[D]
        The array of the first dates for counting.
    enddates : array_like of datetime64[D]
        The array of the end dates for counting, which are excluded
        from the count themselves.
```

```
print('Number of weekdays in 2022:')
print(np.busday_count('2022','2023'))
```

```
Number of weekdays in 2022:
260
```

```
print('Number of weekdays in June 2022:')
np.busday_count('2022-06', '2022-07')
```

```
Number of weekdays in June 2022:

22
```

```
print("Number of weekdays in October 2022: ", np.busday_count('2022-10', '2022-11'))
```

```
Number of weekdays in October 2022:  21
```

np.is_busday()

Another useful function **np.is_busday()** . As its name suggests, it checks if the date passed as an argument is a valid business day.

```
    Parameters
    ----------
    dates : array_like of datetime64[D]
        The array of dates to process.
```

```
np.is_busday(np.datetime64('2022-06-05'))
```

```
False
```

Linear algebra capabilities in NumPy

The amount of data needed for machine learning and deep learning models increased tremendously and created the need for vectorized or matrix operations.

There is a field of mathematics called linear algebra that deals with linear equations and their representations in vector spaces and through matrix's. NumPy provides different types of objects to solve mathematical problems and **np.linalg** package contains linear algebra functions. In this lesson, we'll cover only a matrix object, leaving scalars, vectors, and tensors behind. Matrix objects inherit all the attributes and functions from ndarray with only one difference: it's two dimensional while ndarray can be any dimension.

```
import numpy as np
```

```
first_array = np.arange(16).reshape(4,4)
first_array
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

np.matrix()

```
first_matrix = np.matrix(first_array)
first_matrix
```

```
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
```

```
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])
```

np.identity()

Identity matrix is a matrix where every diagonal element is one and all other elements are zero. It is denoted with a capital letter I and a number in subscript that represents its size. We'll achieve that by typing second_matrix = np.matrix(np.identity 4)

```
second_matrix = np.matrix(np.identity(4))
second_matrix
```

```
matrix([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

np.matmul()

This function returns the matrix product of two arrays. While it returns a normal product for 2-D arrays, if dimensions of either argument is >2, it is treated as a stack of matrices residing in the last two indexes and is broadcast accordingly.

On the other hand, if either argument is 1-D array, it is promoted to a matrix by appending a 1 to its dimension, which is removed after multiplication.

```
matrix_a=np.random.randint(5,size=(2,3))
matrix_a
```

```
array([[2, 0, 0],
       [2, 3, 4]])
```

```
matrix_b=np.random.randint(5,size=(3,2))
matrix_b
```

```
array([[3, 3],
       [4, 2],
       [1, 1]])
```

```
np.matmul(matrix_a,matrix_b)
```

```
array([[ 6,  6],
       [22, 16]])
```

```
np.matmul(matrix_b, matrix_a)
```

```
array([[12,  9, 12],
       [12,  6,  8],
       [ 4,  3,  4]])
```

```
ones = np.ones(5).reshape(5, 1)
```

```
ones
```

```
array([[1.],
       [1.],
       [1.],
       [1.],
       [1.]])
```

```
another = np.array([[3, 6, 13, 4, 4]])
```

```
anotherone = np.array([3, 6, 13, 4, 4])
```

```
anotherone.shape
```

```
(5,)
```

```
another.shape
```

```
(1, 5)
```

```
answer = np.matmul(ones, another)
```

```
answer
```

```
array([[ 3.,  6., 13.,  4.,  4.],
       [ 3.,  6., 13.,  4.,  4.],
       [ 3.,  6., 13.,  4.,  4.],
       [ 3.,  6., 13.,  4.,  4.],
       [ 3.,  6., 13.,  4.,  4.]])
```

np.linalg.inv()

We use numpy.linalg.inv() function to calculate the inverse of a matrix. The inverse of a matrix is such that if it is multiplied by the original matrix, it results in identity matrix.

The matrix must be a square matrix, having the same number of rows and columns.

```
matrix_c=np.matrix("0 1 2;1 0 3;4 -3 8")
matrix_c
```

```
matrix([[ 0,  1,  2],
        [ 1,  0,  3],
        [ 4, -3,  8]])
```

```
inverse = np.linalg.inv(matrix_c)
inverse
```

```
matrix([[-4.5,  7. , -1.5],
        [-2. ,  4. , -1. ],
        [ 1.5, -2. ,  0.5]])
```

```
print(matrix_c*inverse)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

np.mat() interprets a given input as a matrix.

np.linalg.solve() gives the solution of linear equations in the matrix form.

```
A =np.mat("1 -2 1;0 2 -8;-4 5 9")
A
```

```
matrix([[ 1, -2,  1],
        [ 0,  2, -8],
        [-4,  5,  9]])
```

```
b = np.array([0, 16, -18])
b
```

```
array([  0,  16, -18])
```

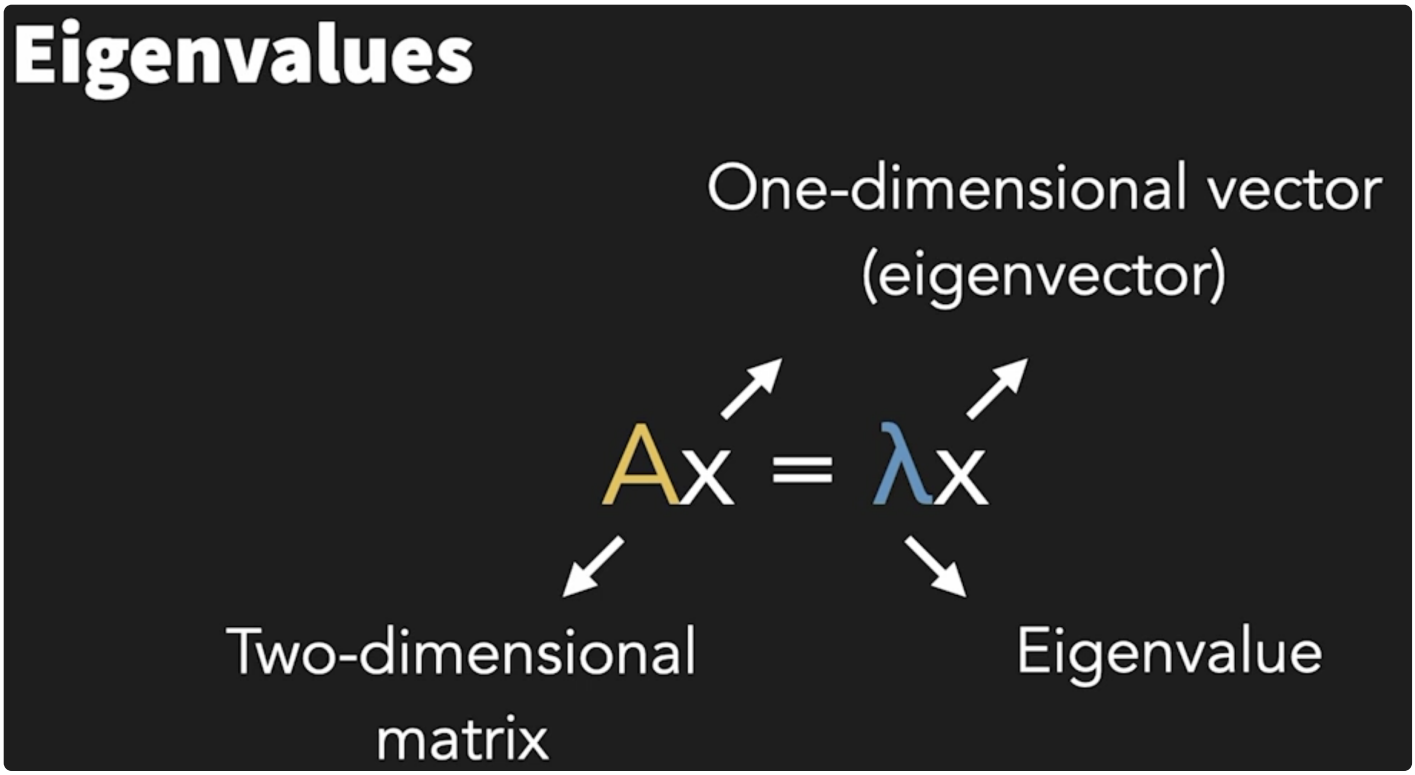
```
x = np.linalg.solve(A, b)
print("Solution", x)
```

```
Solution [58. 32.  6.]
```

Decomposition

Matrix decomposition or matrix factorization is a process of splitting a matrix into parts. You probably recall prime factorization for math where you were finding which prime numbers multiplied together to make the original number.

Well, this is quite similar. The most famous metric decomposition techniques are: lower-upper decomposition, singular value decomposition, QR decomposition, and Cholesky factorization. We'll start by understanding the basics of Eigenvalues and Eigenvectors. And then explore the most commonly used decompositions, singular value decomposition and QR decomposition.



Eigenvalues are scalar solutions to the equation Ax equals lambda x, where A is a two dimensional matrix, x is a one dimensional vector called a eigenvector, and lambda is eigenvalues

NumPy is equipped with the **`np.linalg`** sub package that has two functions, **`np.linalg.eig()`** , which returns a couple of eigenvalues and eigenvectors and **`np.linalg.eigvals()`** , which returns the eigenvalues.

```
first_matrix=np.matrix([[4,8],[10,14]])
print("Matrix:\n",first_matrix)
```

```
Matrix:
 [[ 4  8]
 [10 14]]
```

```
eigenvalues, eigenvectors = np.linalg.eig(first_matrix)
print("Eigenvalues:\n", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

```
Eigenvalues:
 [-1.24695077 19.24695077]
Eigenvectors:
 [[-0.83619408 -0.46462222]
 [ 0.54843365 -0.885509  ]]
```

```
eigenvalues= np.linalg.eigvals(first_matrix)
print("Eigenvalues:", eigenvalues)
```

```
Eigenvalues: [-1.24695077 19.24695077]
```

`np.linalg.svd()`

Singular Value Decomposition, or SVD, decomposes the matrix into singular vectors and singular values. It's used in computer vision, signal processing, natural language processing, and machine learning.

Singular Value Decomposition means when arr is a 2D array, it is factorized as u and vh, where u and vh are 2D unitary arrays and s is a 1D array of a's singular values. numpy.linalg.svd() function is used to compute the factor of an array by Singular Value Decomposition.

This returns a tuple that contains matrices U and V and diagonal values for the matrix sigma.

```
Syntax : numpy.linalg.svd(a, full_matrices=True, compute_uv=True, hermitian=False)

Parameters :
a (…, M, N) array : A real or complex array with a.ndim >= 2.

full_matrices(bool, optional) : If True (default), u and vh have the shapes (…, M, M)
and (…, N, N), respectively. Otherwise, the shapes are (…, M, K) and (…, K, N),
 respectively, where K = min(M, N).

compute_uv(bool, optional) : Whether or not to compute u and vh in addition to s.
Its default value is True.

hermitian(bool, optional) : If True, a is assumed to be Hermitian (symmetric if real-
enabling a more efficient method for finding singular values. Its default value is Fa
```

```
A = np.mat("3 1 4;1 5 9;2 6 5")
print("A\n", A)

U, Sigma, V = np.linalg.svd(A, full_matrices=False)

print("U: ",U)
print("Sigma : ",Sigma)
print("V : ", V)
```

```
A
 [[3 1 4]
 [1 5 9]
 [2 6 5]]
U:  [[-0.32463251  0.79898436  0.50619929]
 [-0.75307473  0.1054674  -0.64942672]
 [-0.57226932 -0.59203093  0.56745679]]
Sigma :  [13.58235799  2.84547726  2.32869289]
V :  [[-0.21141476 -0.55392606 -0.80527617]
 [ 0.46331722 -0.78224635  0.41644663]
 [ 0.86060499  0.28505536 -0.42202191]]
```

```
print("Product\n", U * np.diag(Sigma) * V)
```

```
Product
```

```
[[3. 1. 4.]
[1. 5. 9.]
[2. 6. 5.]]
```

M=Q*R

QR decomposition is used to decompose square or rectangular matrix M, internal or outgoing or matrix Q, an upper triangle matrix R.

np.linalg.qr() QR factorization of a matrix is the decomposition of a matrix say 'A' into 'A=QR' where Q is orthogonal and R is an upper-triangular matrix. We factorize the matrix using numpy.linalg.qr() function.

```
Syntax : numpy.linalg.qr(a, mode='reduced')
Parameters :
a : matrix(M,N) which needs to be factored.
```

```
A
```

```
matrix([[3, 1, 4],
        [1, 5, 9],
        [2, 6, 5]])
```

```
b = np.array([1,2,3]).reshape(3,1)
q, r = np.linalg.qr(A)
x = np.dot(np.linalg.inv(r), np.dot(q.T, b))
x
```

```
matrix([[ 0.26666667],
        [ 0.46666667],
        [-0.06666667]])
```

```
np.linalg.solve(A,b)
```

```
array([[ 0.26666667],
       [ 0.46666667],
       [-0.06666667]])
```

Polynomial mathematics, **np.polynomial()**

The word polynomial comes from the Greek word poly, meaning many, and the Latin word nomial, meaning term. So it's name means many terms. Just a quick reminder from math, polynomial is an expression involving the sum of powers in one or more variables multiplied by coefficients.

Examples of polynomial functions are linear, quadratic, cubic, and quartic functions. NumPy contains the sub module polynomial which provides functions and classes for working with polynomials.

```
from numpy.polynomial import polynomial
```

np.polynomial.Polynomial()

```
Parameters:
coef : array_like

Polynomial coefficients, in increasing order.
For example, (1, 2, 3)
implies P_0 + 2P_1 + 3P_2
where the P_i are a graded polynomial basis.
```

```
first_polynomial = np.polynomial.Polynomial([2, -3, 1])
first_polynomial
```

$x \mapsto 2.0 - 3.0\,x + 1.0\,x^2$

np.polynomial.Polynomial.fromroots()

Generate a monic polynomial with given roots.

Return the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * \ldots * (x - r_n),$$

where the r_n are the roots specified in roots. If a zero has multiplicity n, then it must appear in roots n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then roots looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are c, then

$$p(x) = c_0 + c_1 * x + \ldots + x^n$$

The coefficient of the last term is 1 for monic polynomials in this form.

Parameters rootsarray_like Sequence containing the roots.

```
second_polynomial = np.polynomial.Polynomial.fromroots([1, 2])
second_polynomial
```

$x \mapsto 2.0 - 3.0\,x + 1.0\,x^2$

np.roots()

return the roots of a polynomial with coefficients given in p. The values in the rank-1 array p are coefficients of a polynomial. If the length of p is n+1 then the polynomial is described by:
p[0] * x**n + p[1] * x**(n-1) + … + p[n-1]*x + p[n]

```
Syntax : numpy.roots(p)
```

```
Parameters :
p : [array_like] Rank-1 array of polynomial coefficients.

Return : [ndarray] An array containing the roots of the polynomial.
```

```
first_polynomial.roots()
```

```
array([1., 2.])
```

```
second_polynomial.roots()
```

```
array([1., 2.])
```

$y=x^4+2x^3+3x^2+4x+5$, x=1

y=?

np.polyval(poly, x)

evaluates a polynomial at specific values.

If 'N' is the length of polynomial 'p', then this function returns the value

p[0]*x(N-1) + p[1]*x**(N-2) + ... + p[N-2]*x + p[N-1]**

```
Parameters :
p : [array_like or poly1D] polynomial coefficients are given in decreasing order
of powers. If the second parameter (root) is set to True then array values are the
roots of the polynomial equation.

For example : poly1d(3, 2, 6) = 3x2 + 2x + 6

x : [array_like or poly1D] A number, an array of numbers, for evaluating 'p'.
```

```
np.polyval([5,4,3,2,1], 1)
```

```
15
```

```
third_polynomial = np.polynomial.Polynomial([1,2,3,4,5])
third_polynomial
```

$$x \mapsto 1.0 + 2.0\,x + 3.0\,x^2 + 4.0\,x^3 + 5.0\,x^4$$

np.polynomial.polynomial.Polynomial.integ(m=1, k=[], lbnd=None)

```
Return a series instance that is the definite integral of the current series.
```

```
Parameters
mnon-negative int
The number of integrations to perform.

karray_like
Integration constants. The first constant is applied to the first integration,
the second to the second, and so on. The list of values must less than or equal to m
 in length and any missing values are set to zero.

lbndScalar
The lower bound of the definite integral.

Returns
new_seriesseries
A new series representing the integral. The domain is the same as the domain of the
integrated series.
```

```
integral=third_polynomial.integ()
integral
```

$$x \mapsto 0.0 + 1.0\,x + 1.0\,x^{2} + 1.0\,x^{3} + 1.0\,x^{4} + 1.0\,x^{5}$$

np.polynomial.polynomial.Polynomial.deriv(m=1)

```
Return a series instance of that is the derivative of the current series.

Parameters
mnon-negative int
Find the derivative of order m.

Returns
new_series
A new series representing the derivative. The domain is the same as the domain of the
differentiated series.
```

```
integral.deriv()
```

$$x \mapsto 1.0 + 2.0\,x + 3.0\,x^{2} + 4.0\,x^{3} + 5.0\,x^{4}$$

```
derivative=third_polynomial.deriv()
derivative
```

$$x \mapsto 2.0 + 6.0\,x + 12.0\,x^{2} + 20.0\,x^{3}$$

Application: linear regression

Linear regression is a simple, one of the most important and widely used model for machine learning algorithms. Machine learning algorithms are divided into two categories, supervised machine learning algorithms and

unsupervised machine learning algorithms.

For unsupervised learning algorithms, the system will only use input data without any labels. For supervised learning algorithms, the input data set and the corresponding output or true prediction are available. And these algorithms try to find the relationship between inputs and outputs.

Linear regression is a supervised machine learning method most commonly used for finding out the relationship between variables and forecasting. By definition, linear regression is estimating an unknown variable in a linear fashion, based on some other known variables.

Visually we fit a line or a hip plane in higher dimensions through our data points. It can be applied to various kinds of business and scientific problems. For example, stock prices, property price, (indistinct) prices, sales and GDP growth rate predictions.

Let's explore one of the most famous problems, prediction of property prices on a simplification. We can get statistical data for the house price indices and pick the average house price from 2012 to 2021. We want to find out the average house price for the year 2022. We can assume their relation is squared, so we want to find the polynomial y equals ax squared plus bx plus B to represent the relations. Y will represent price at year x.

House market

- Input: Price data from 2012 - 2021
- Output: Avarage house market 2022?
- Assume Relationship - squared
- $y=ax^2+bx+c$

```
year = np.arange(1,11)
price = np.array([129000, 133000, 138000, 144000, 142000, 141000, 150000, 135000, 13400
year
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

np.polyfit(x, y, deg)

```
Fit a polynomial p(x) = p[0] * x**deg + ... + p[deg] of degree deg to points (x, y).
Returns a vector of coefficients p that minimises the squared error in the order
deg, deg-1, … 0.

The Polynomial.fit class method is recommended for new code as it is more
stable numerically. See the documentation of the method for more information.

Parameters
xarray_like, shape (M,)
x-coordinates of the M sample points (x[i], y[i]).

yarray_like, shape (M,) or (M, K)
y-coordinates of the sample points. Several data sets of sample points sharing the sa
x-coordinates can be fitted at once by passing in a 2D-array that contains one
dataset per column.
```

degint
Degree of the fitting polynomial

```
a, b, c = np.polyfit(year, price, 2)
print ("a:",a)
print ("b:",b)
print ("c:",c)
```

a: -594.6969696969664

b: 7032.575757575716

c: 122516.66666666669

```
print("Estimated price for 2022:",a*11**2 + b*11 + c )
```

Estimated price for 2022: 127916.66666666663

```
import matplotlib.pyplot as plt

plt.plot(year,price, color = 'blue')
plt.scatter(year,price, color = 'blue')
plt.scatter(11, a*11**2 + b*11 + c ,color='red')
plt.title('Linear regression')
plt.xlabel('year')
plt.ylabel('average house price')
```

Text(0, 0.5, 'average house price')