AGENDA

1. Introduction to Numpy, Mathematics with Numpy
2. Initializing and Reorganizing of Arrays
3. Visualization using Seaborn and Matplotlib

Data Science Academy

I

Introduction to Numpy
Mathematics with Numpy

Data Science Academy

Numpy

- NumPy (Numerical Python) is an open source Python library that's used in almost every field of science and engineering.
- It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems.
- NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development.
- The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

Installing NumPy

- To install NumPy, we strongly recommend using a scientific Python distribution. If you're looking for the full instructions for installing NumPy on your operating system, you can find all of the details here.

- If you already have Python, you can install NumPy with:

```
conda install numpy
```

or

```
pip install numpy
```

- If you don't have Python yet, you might want to consider using Anaconda. It's the easiest way to get started.

- The good thing about getting this distribution is the fact that you don't need to worry too much about separately installing NumPy or any of the major packages that you'll be using for your data analyses, like pandas, Scikit-Learn, etc.

How to import Python

- Any time you want to use a package or library in your code, you first need to make it accessible.
- In order to start using NumPy and all of the functions available in NumPy, you'll need to import it. This can be easily done with this import statement:

```
import numpy as np
```

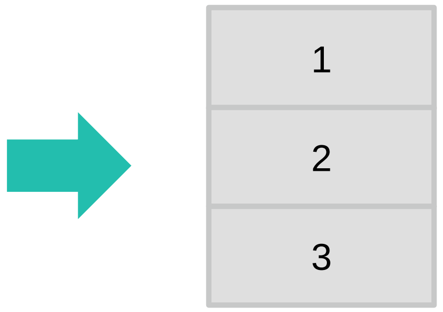
- We shorten numpy to np in order to save time and also to keep code standardized so that anyone working with your code can easily understand and run it.

The Basics

- NumPy’s main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers.
- In NumPy dimensions are called *axes*.
- To create a NumPy array, you can use the function np.array().
- All you need to do to create a simple array is pass a list to it. If you choose to, you can also specify the type of data in your list. You can find more information about data types here.

Numpy array:

```
In  []:  import numpy as np

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



The Basics

- NumPy’s array class is called **ndarray**. It is also known by the alias array. Note that `numpy.array` is not the same as the Standard Python Library class `array.array`, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an ndarray object are:

  - `ndarray.ndim`

    the number of axes (dimensions) of the array.

  - `ndarray.shape`

    the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim

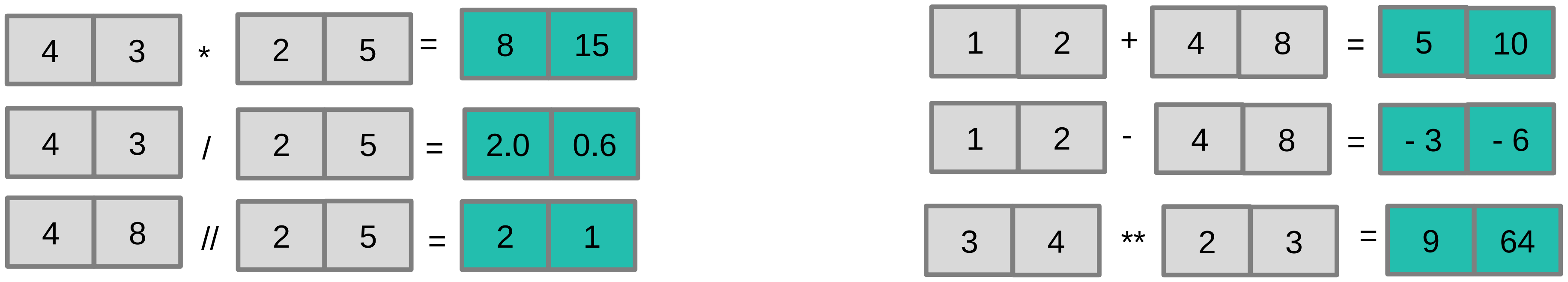
  - `ndarray.dtype`

    an object describing the type of the elements in the array. One can create or specify dtype’s using standard Python types. Additionally, NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

Vector operations

- NumPy contains a large number of various mathematical operations. NumPy provides standard trigonometric functions, functions for arithmetic operations, handling complex numbers, etc.

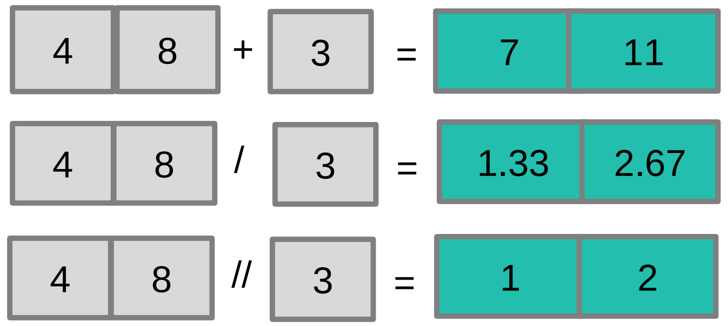
- Arithmetic is one of the places where NumPy speed shines most. NumPy allows the manipulation of whole arrays just like ordinary numbers:

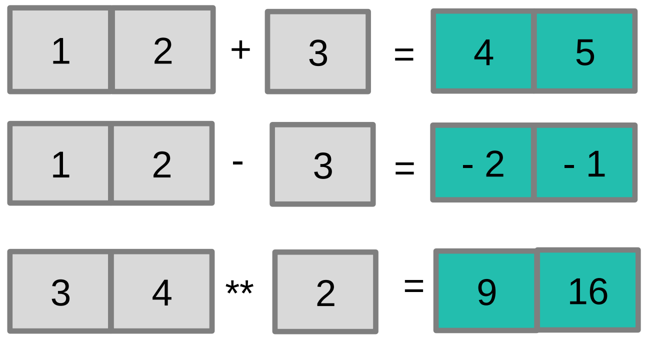


```
In  [ ]:  np.float(64)

In  [ ]:  np.int(64)
```

Vector operations

- The same way ints are promoted to floats when adding or subtracting, scalars are promoted (aka *broadcasted*) to arrays:





```
In  [ ]:  np.float(64)
```

```
In  [ ]:  np.int(64)
```

Vector operations

- Most of the math functions have NumPy counterparts that can handle vectors:

$$a^2 = \boxed{2}\boxed{3} ** \boxed{2} = \boxed{4}\boxed{9}$$

$$\sqrt{a} = \left(\boxed{4}\boxed{9}\right) = \boxed{2.}\boxed{3.}$$

$$e^a = \left(\boxed{1}\boxed{2}\right) = \boxed{2.718}\boxed{7.389}$$

$$\ln a = \left(\boxed{\texttt{np.e}}\boxed{\texttt{np.e**2}}\right) = \boxed{1.}\boxed{2.}$$

Vector operations

- Scalar product has an operator of its own:

$$\vec{a} \cdot \vec{b} = \text{np.dot}\left(\boxed{1}\boxed{2}\,,\ \boxed{3}\boxed{4}\right)$$

$$= \boxed{1}\boxed{2}\ @\ \boxed{3}\boxed{4} = \boxed{11}$$

$$\vec{a} \times \vec{b} = \text{np.cross}\left(\boxed{2}\boxed{0}\boxed{0}\,,\ \boxed{0}\boxed{3}\boxed{0}\right) = \boxed{0}\boxed{0}\boxed{6}$$

Vector operations

- You don’t need loops for trigonometry either:

np.sin (| Np.pi | Np.pi / 2 |) = | 0 | 1 |

np.arcsin (| 0 | 1 |) = | 0 | 1.57 |

sin	arcsin	sinh	arcsinh
cos	arccos	cosh	arccosh
tan	arctan	tanh	arctanh

np.hypot (| 3. | 5. | , | 4.. | 12.. |) = | 5.. | 13.. |

Vector operations

- Arrays can be rounded as a whole:

np.floor	(	1.1	1.5	1.9	2.5	)	=	1.	1.	1.	2.
np.ceil	(	1.1	1.5	1.9	2.5	)	=	2.	2.	2.	3.
np.round	(	1.1	1.5	1.9	2.5	)	=	1.	2.	2.	3.

II

Initializing and Reorganizing of Arrays

Data Science Academy

Initializing

In [ ]:

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

1.	2.	3.

```
.dtype == dtype('float64')
.shape == (3,)
```

- Besides creating an array from a sequence of elements, you can easily create an array filled with 0's:

In [ ]:

```
b = np.zeros(3, int)
```

0	0	0

```
.dtype == dtype('int64')
```

- It is often necessary to create an empty array which matches the existing one by shape and elements type:

In [ ]:

```
c = np.zeros_like(a)
```

0.	0.	0.

```
.dtype == dtype('float64')
.shape == (3,)
```

Initializing

- Actually, all the functions that create an array filled with a constant value have a _like counterpart:

```
In [ ]: np.zeros(3)
```

0	0	0

```
In [ ]: np.ones(3)
```

1	1	1

```
In [ ]: np.empty(3)
```

5e-296	7e-297	1e-296

```
In [ ]: np.full(3,7.)
```

7.	7.	7.

```
In [ ]: np.array([1,2,3])
```

1	2	3

```
In [ ]: np.zeros_like(a)
```

0.	0.	0.

```
In [ ]: np.ones_like (a)
```

1.	1.	1.

```
In [ ]: np.empty_like (a)
```

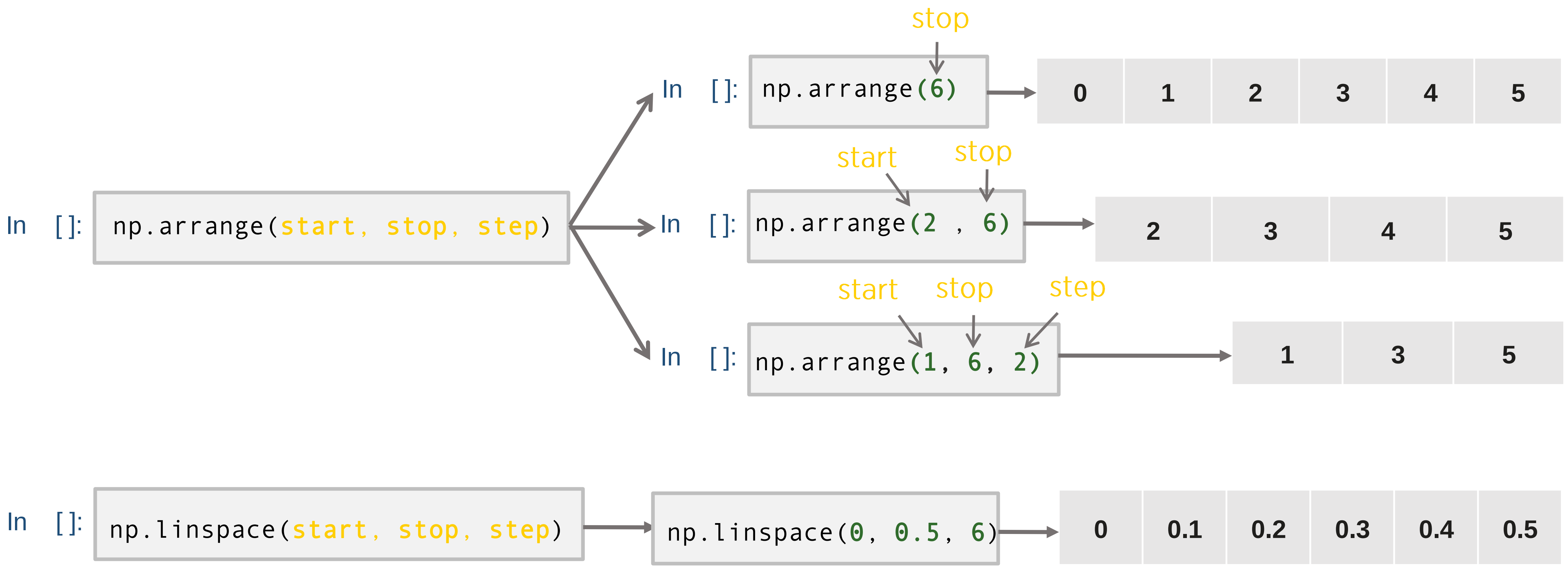
540876987	1630433390	2036429426

```
In [ ]: np.full_like(a,7)
```

7.	7.	7.

Initializing

- There are as many as two functions for array initialization with a monotonic sequence in NumPy:



In [ ]: np.arrange(start, stop, step)
stop
In [ ]: np.arrange(6)
0 1 2 3 4 5
start stop
In [ ]: np.arrange(2 , 6)
2 3 4 5
start stop step
In [ ]: np.arrange(1, 6, 2)
1 3 5
In [ ]: np.linspace(start, stop, step)
np.linspace(0, 0.5, 6)
0 0.1 0.2 0.3 0.4 0.5

Initializing

- For testing purposes it is often necessary to generate random arrays:

```
In  [ ]: np.random.randint(0,10,3)
```

→

4	3	7

uniform, $x \epsilon [0,10)$

Careful!

- np.random.randint(0,10) is [0,10), but random.randint(0,10) is [0,10]

```
In  [ ]: np.random.rand(3)
```

→

0.7	0.3	0.8

uniform, $x \epsilon [0,1)$

```
In  [ ]: np.random.rand(3)
```

→

0.4	-1.1	0.8

normal $\mu = 0,\ \sigma = 1$

```
In  [ ]: np.random.uniform(1,10,3)
```

→

5.1	2.7	7.2

uniform, $x \epsilon [0,10)$

```
In  [ ]: np.random.normal(5,2,3)
```

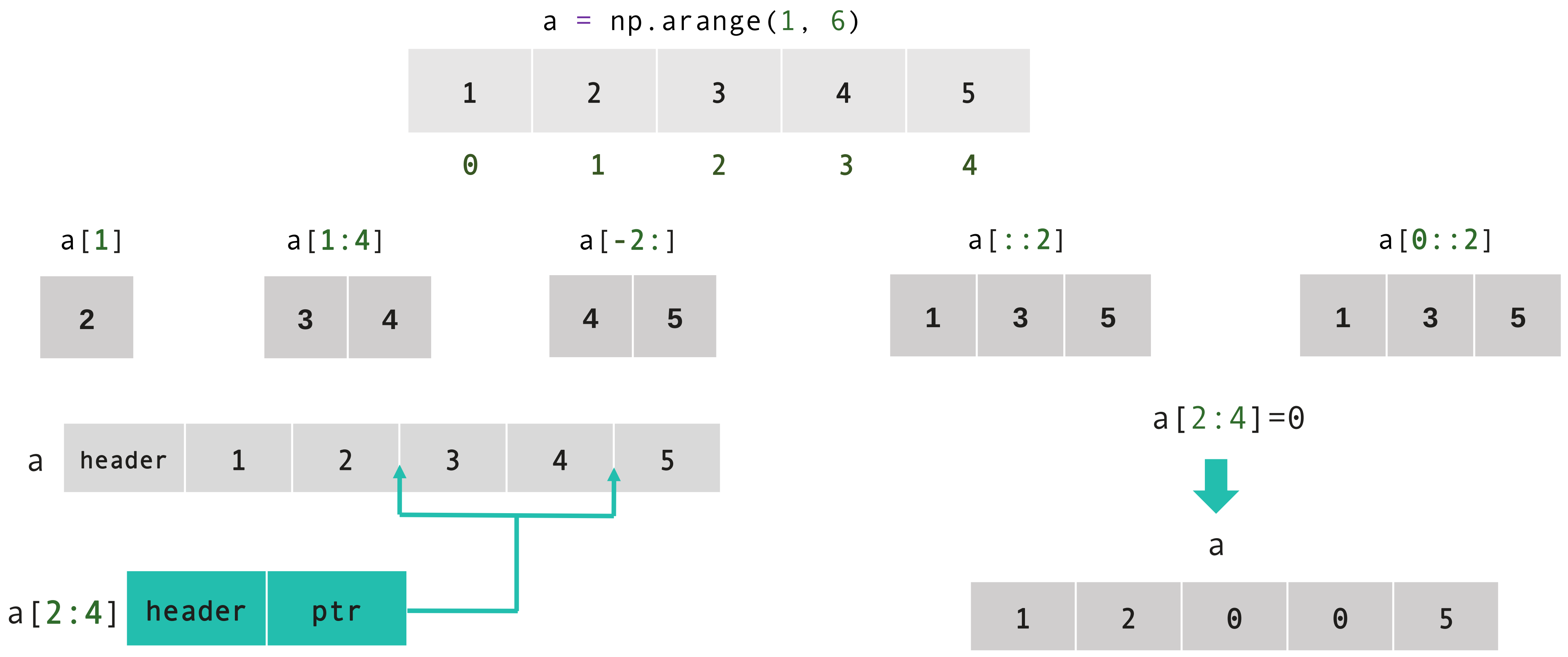
→

4.5	3.2	6.7

normal $\mu = 0,\ \sigma = 1$

Vector indexing

- Once you have your data in the array, NumPy is brilliant at providing easy ways of giving it back:



Reshaping and Transposing

- It’s common to need to transpose your matrices. NumPy arrays have the property T that allows you to transpose a matrix.

- Transposing is switching its rows with its columns.

data

1	2
3	4
5	6

data.T

1	3	5
2	4	6

Data type objects

- A data type object (an instance of numpy.dtype class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data:

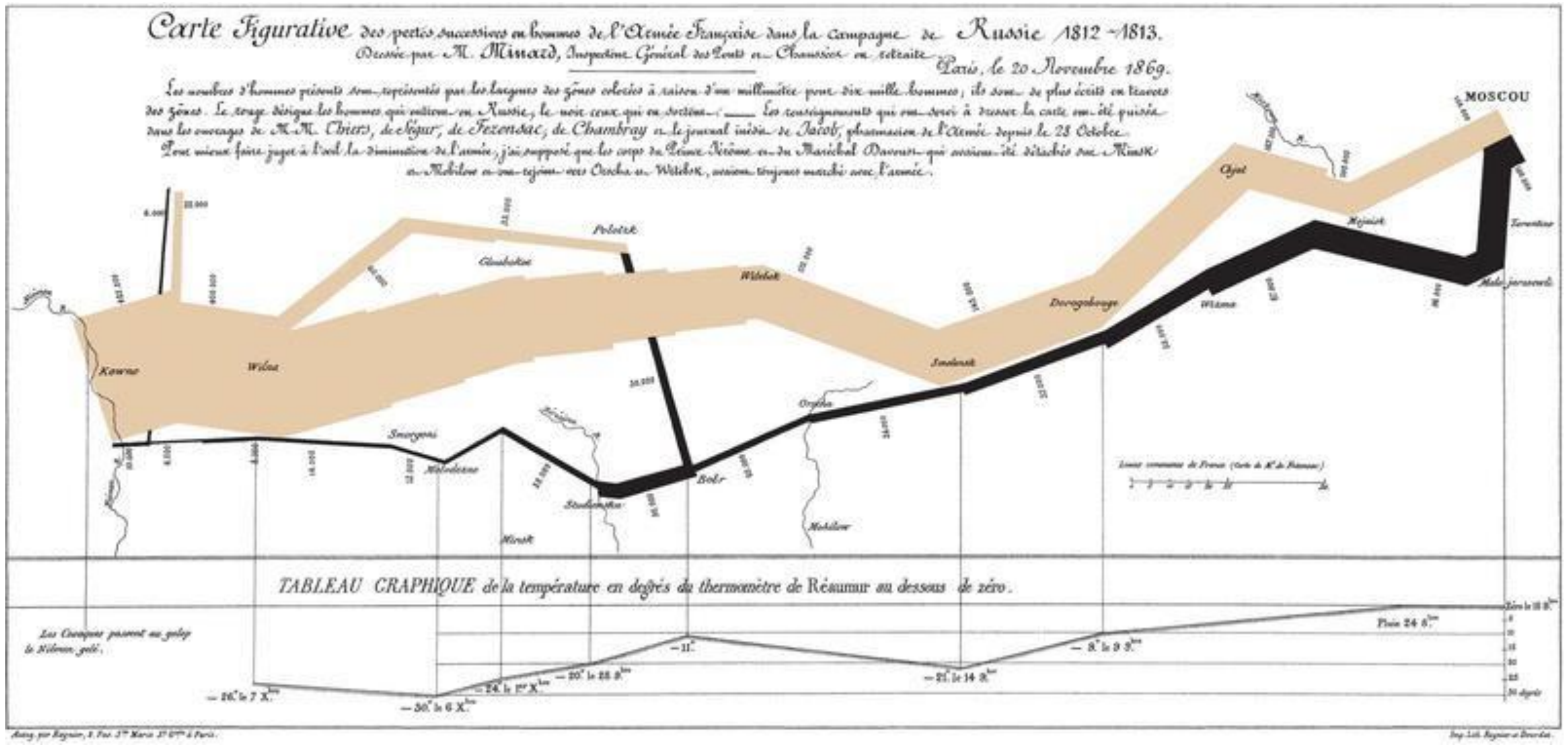
1. Type of the data (integer, float, Python object, etc.)
2. Size of the data (how many bytes is in *e.g.* the integer)
3. Byte order of the data (little-endian or big-endian)
4. If the data type is structured data type, an aggregate of other data types, (*e.g.*, describing an array item consisting of an integer and a float),
   A. what are the names of the "fields" of the structure, by which they can be accessed,
   B. what is the data-type of each field, and
   C. which part of the memory block each field takes.
5. If the data type is a sub-array, what is its shape and data type.

III
Visualization using Seaborn and Matplotlib

Data Science Academy

The map of Napoleon’s Russian campaign

- Minard is best known for his cartographic depiction of numerical data on a map of Napoleon's disastrous losses suffered during the Russian campaign of 1812.


Carte Figurative des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812-1813.
Dressée par M. Minard, Inspecteur Général des Ponts et Chaussées en retraite.
Paris, le 20 Novembre 1869.
MOSCOU
Kowno
Wilna
Smolensk
Polotzk
Witebsk
Dorogobouge
Mojaisk
Bobr
Smorgoni
Molodeczno
Minsk
Orscha
Mohilow
TABLEAU GRAPHIQUE de la température en degrés du thermomètre de Réaumur au dessous de zéro.

Data visualization

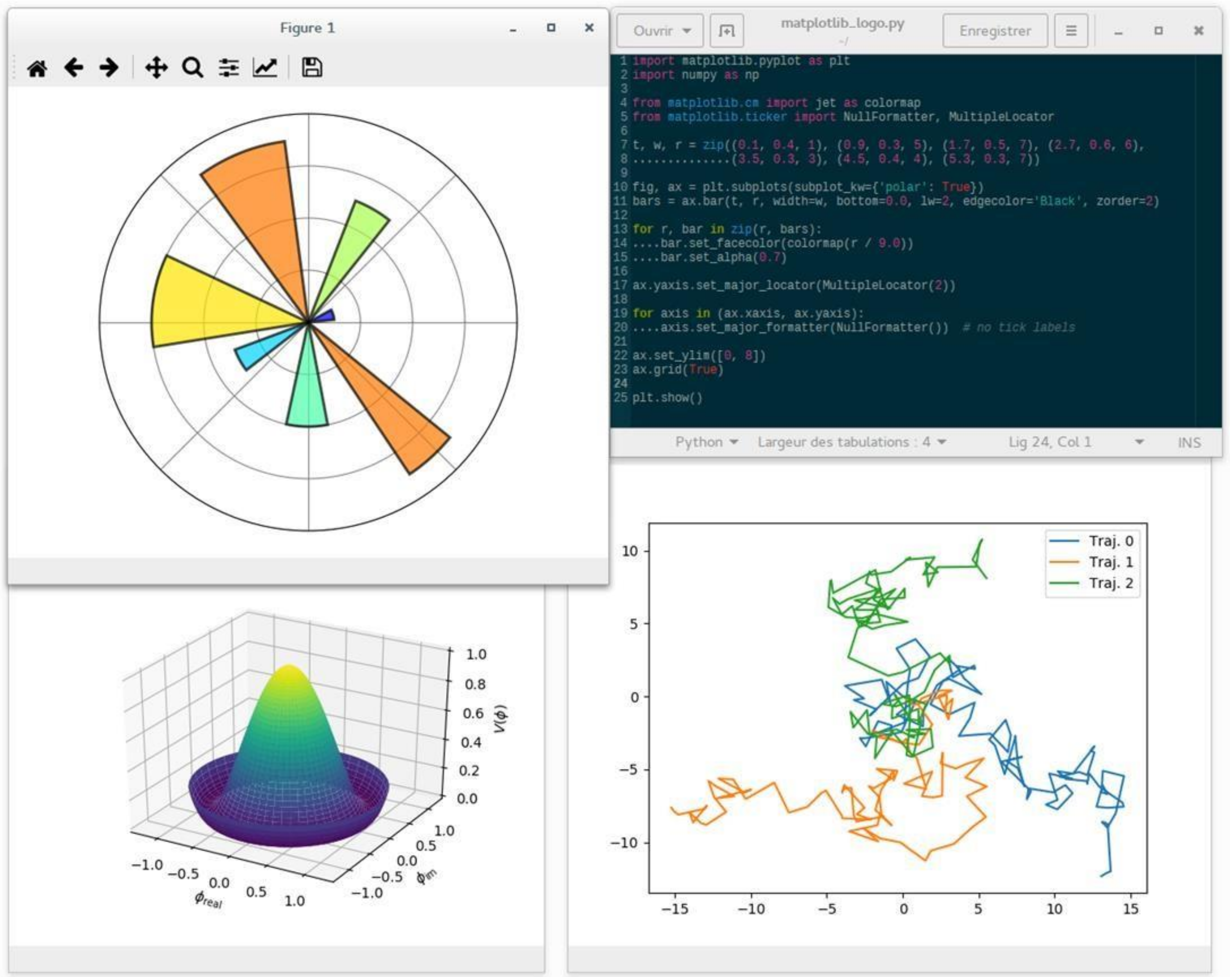
- EDA & Data Understanding
- Data Manipulation
- Visual Data Mining
- Models validation
- Analytics & reporting

Python Libraries for Data Science

- matplotlib:
  - python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
  - a set of functionalities similar to those of MATLAB
  - line plots, scatter plots, barcharts, histograms, pie charts etc.
  - relatively low-level; some effort needed to create advanced visualization
  - Link: https://matplotlib.org/

Matplotlib

- #pip install matplotlib



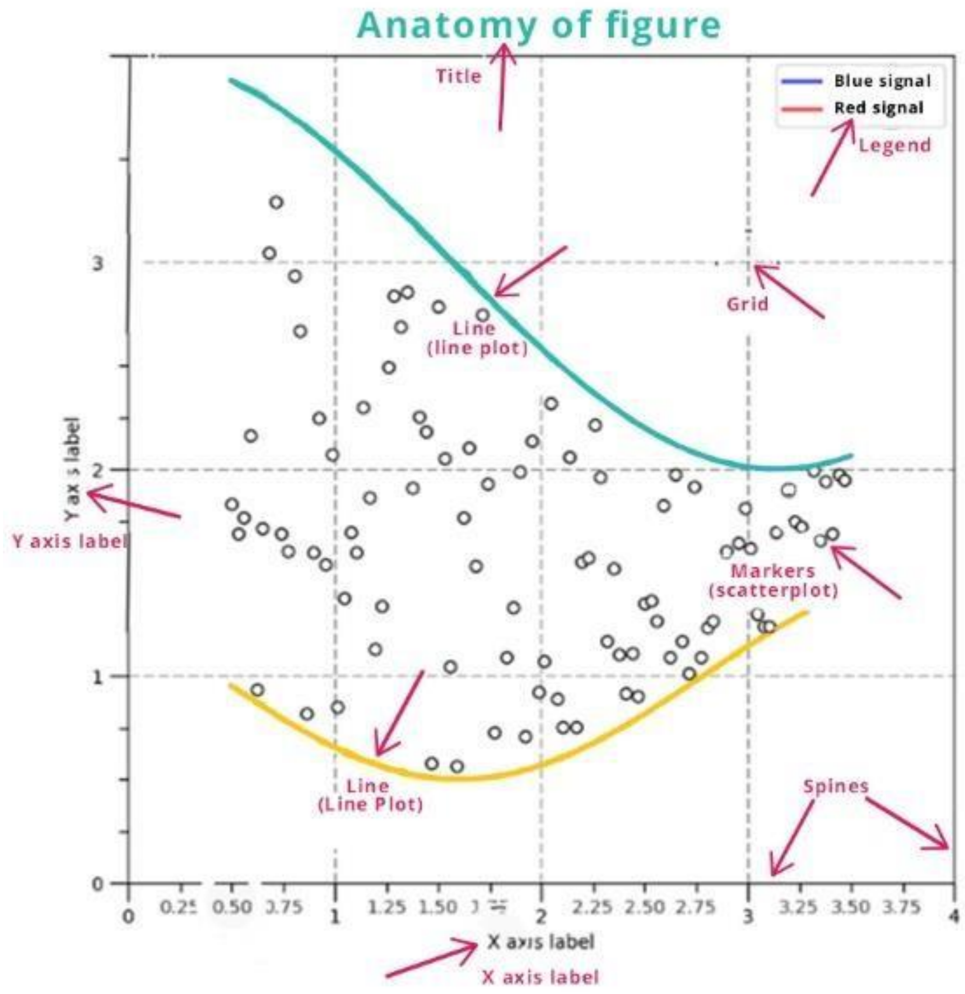
```python
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.cm import jet as colormap
from matplotlib.ticker import NullFormatter, MultipleLocator

t, w, r = zip((0.1, 0.4, 1), (0.9, 0.3, 5), (1.7, 0.5, 7), (2.7, 0.6, 6),
..............(3.5, 0.3, 3), (4.5, 0.4, 4), (5.3, 0.3, 7))

fig, ax = plt.subplots(subplot_kw={'polar': True})
bars = ax.bar(t, r, width=w, bottom=0.0, lw=2, edgecolor='Black', zorder=2)

for r, bar in zip(r, bars):
....bar.set_facecolor(colormap(r / 9.0))
....bar.set_alpha(0.7)

ax.yaxis.set_major_locator(MultipleLocator(2))

for axis in (ax.xaxis, ax.yaxis):
....axis.set_major_formatter(NullFormatter())  # no tick labels

ax.set_ylim([0, 8])
ax.grid(True)

plt.show()
```

Basic visualization rules

- Select plot type gives insight about case.
- Label axis.
- Add title.
- Add labels for each categories.
- Optionally text or arrow can be added at interesting data points.
- Use sizes and colors of the data to make plot more appealing.

Parts of a Plot

- There are different parts of a plot, are denoted in the following figure:



Structure for Plotting

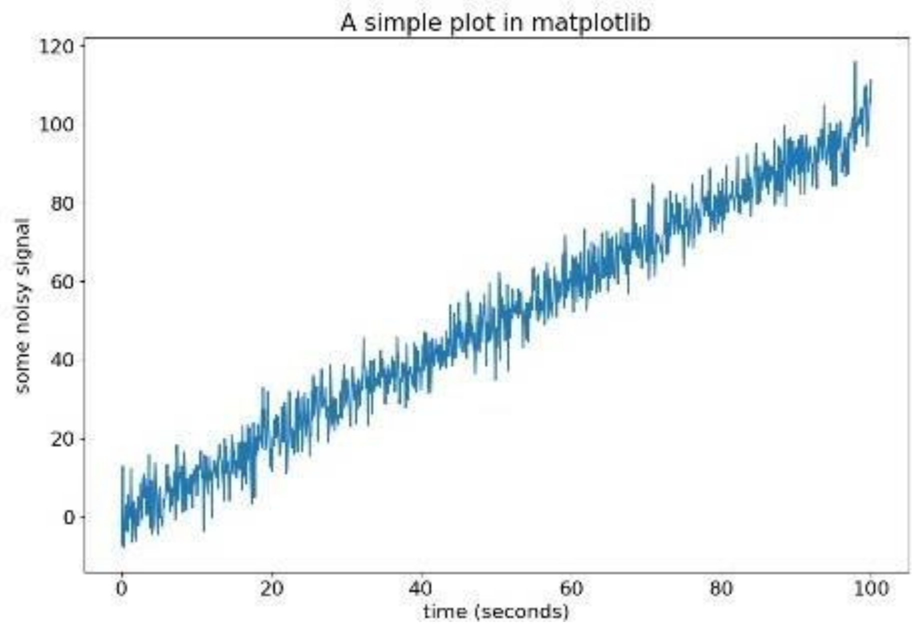
In [ ]:

```
plt.plot(x, y) #plot x and t using default line style and  color

plt.xlabel('comment') #set a label that will be displayed in thelegend

plt.ylabel('comment') #set a label that will be displayed in thelegend

plt.title('comment') #available titles are positioned above the axes in thecenter

plt.legend() #elements to be added to the legend are automaticallydetermined

plt.show() #display a figure
```

Example of Plot

In [ ]:

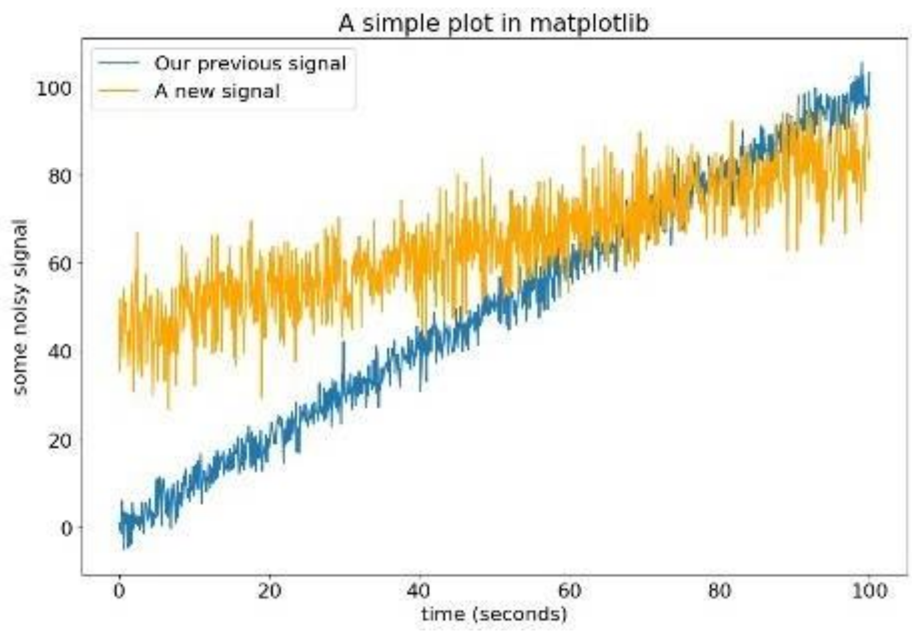
```
import numpy

from matplotlib import pyplot

x = numpy.linspace(0., 100., 1001)

y = x + numpy.random.randn(1001) * 5

pyplot.plot(x, y)

pyplot.xlabel('time (seconds)')

pyplot.ylabel('some noisy signal')

pyplot.title('A simple plot in matplotlib')
```



Example of Plot

In [ ]:

```
import numpy
from matplotlib import pyplot
x = numpy.linspace(0., 100., 1001)
y1 = x + numpy.random.randn(1001) * 3
y2 = 45 + x * .4 + numpy.random.randn(1001) * 7
pyplot.plot(x, y1, label = 'Our previous signal')
pyplot.plot(x, y2, color = 'orange', label = 'A new signal')
pyplot.xlabel('time (seconds)')
pyplot.ylabel('some noisy signal')
pyplot.title('A simple plot in matplotlib')
pyplot.legend()
```



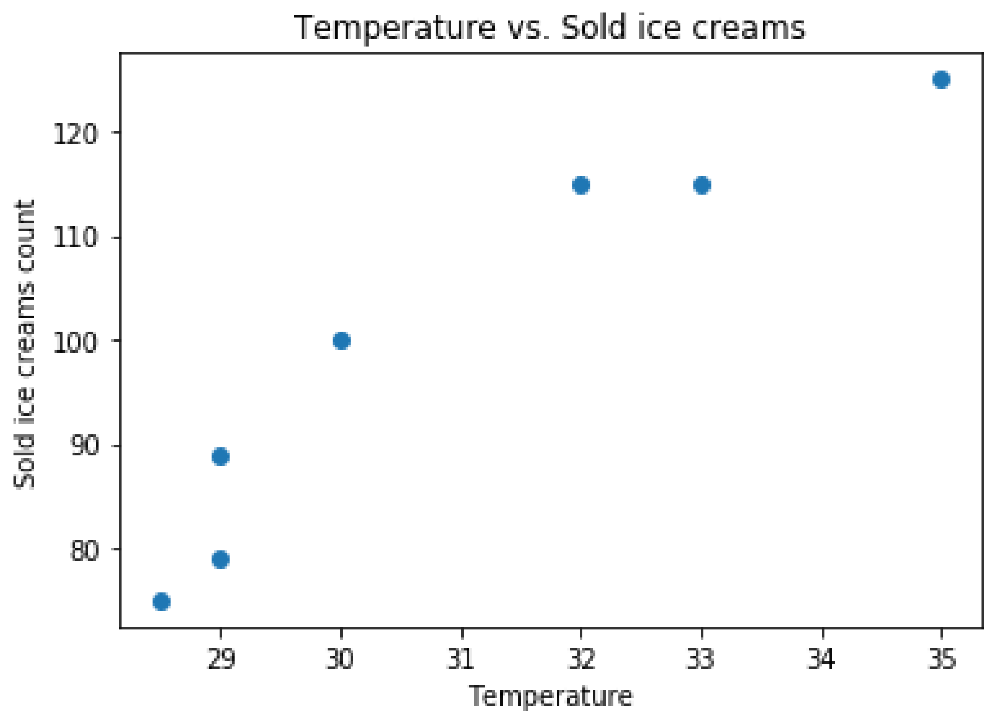
Scatter

- Scatter shows all individual data points, they aren’t connected with lines.
- Each data point has the value of the x-axis value and the value from the y-axis values.
- Scatter can be used to display trends or correlations.
- In data science, it shows how 2 variables compare.
- To make a scatter plot with Matplotlib, we can use the *plt.scatter()* function.
- Again, the first argument is used for the data on the horizontal axis, and the second - for the vertical axis.

Example of Plot

In [ ]:

```
import matplotlib.pyplot as plt
temp = [30, 32, 33, 28.5, 35, 29, 29]
ice_creams_count = [100, 115, 115, 75, 125, 79,89]

plt.scatter(temp, ice_creams_count)

plt.title('Temperature vs. Sold ice creams')

plt.xlabel('Temperature')

plt.ylabel('Sold ice creams count')

plt.show()
```



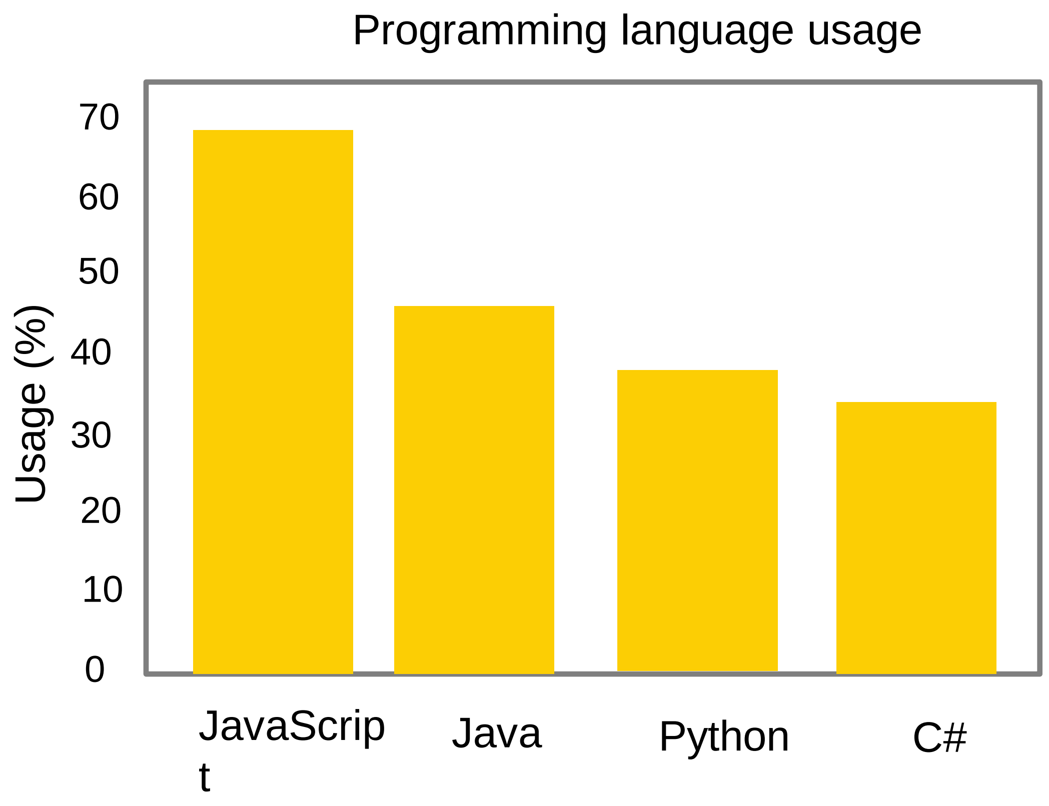
Bar chart

- Represents categorical data with rectangular bars.
- Each bar has a height corresponds to the value it represents.
- It’s useful when we want to compare a given numeric value on different categories.
- It can also be used with 2 data series.
- To make a bar chart with Maplotlib, we’ll need the *plt.bar()* function.

Example Bar chart

In [ ]:

```
import matplotlib.pyplot asplt

labels = ['JavaScript', 'Java', 'Python', 'C#']

usage = [69.8, 45.3, 38.8,34.4]

y_positions =range(len(labels))
plt.bar(y_positions,usage)
plt.xticks(y_positions,labels)
plt.ylabel('Usage (%)')
plt.title('Programming languageusage')
plt.show()
```



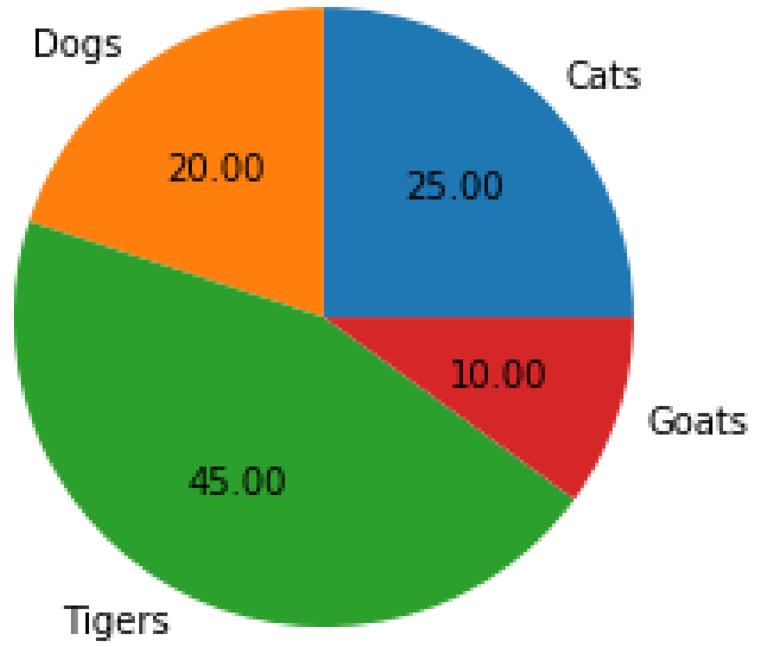
Pie chart

- Pie chart a circular plot, divided into slices to show numerical proportion.
- They are widely used in the business world. However, many experts recommend to avoid them.
- The main reason is that it’s difficult to compare the sections of a given pie chart.
- Also, it’s difficult to compare data across multiple pie charts.
- They can be replaced by a bar chart.

Example Pie Chart

In [ ]:

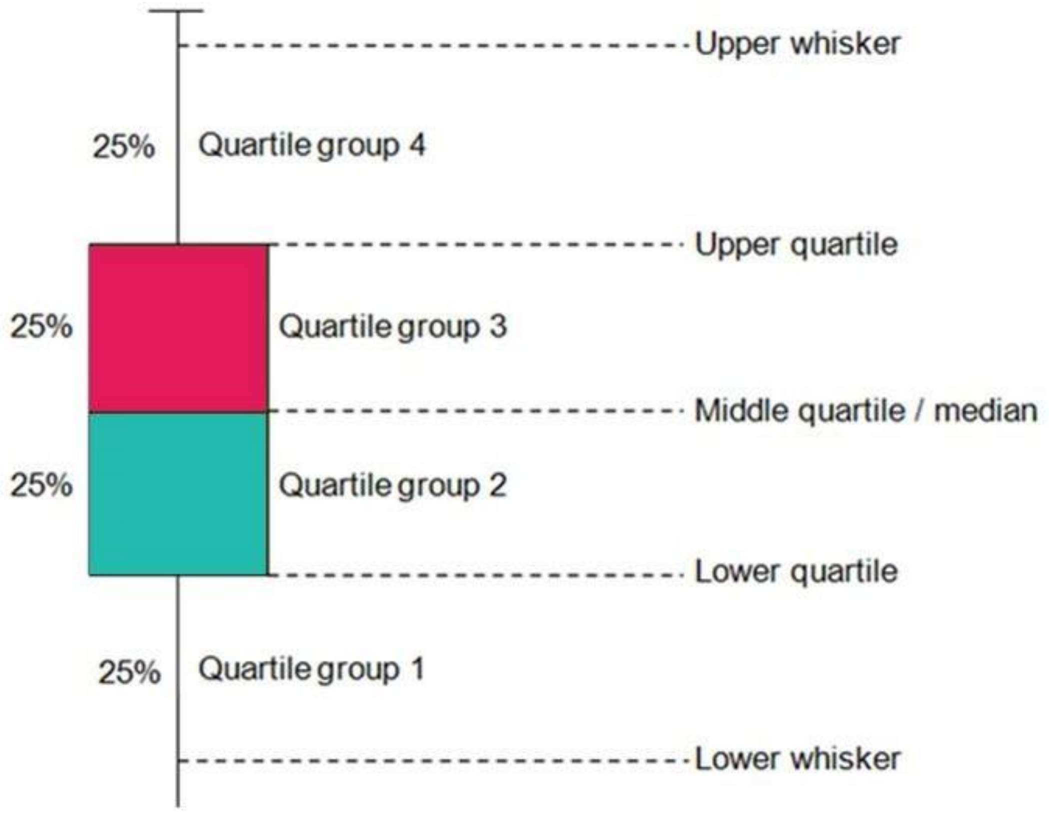
```
import matplotlib.pyplot as plt

sizes = [25, 20, 45, 10]
labels = ["Cats", "Dogs", "Tigers", "Goats"]

plt.pie(sizes, labels = labels, autopct = "%.2f")

#float and percentage value

plt.axes().set_aspect("equal")

plt.show()
```



Box Plot

- A box plot is a graphical rendition of statistical data based on the minimum, first quartile, median, third quartile, and maximum.

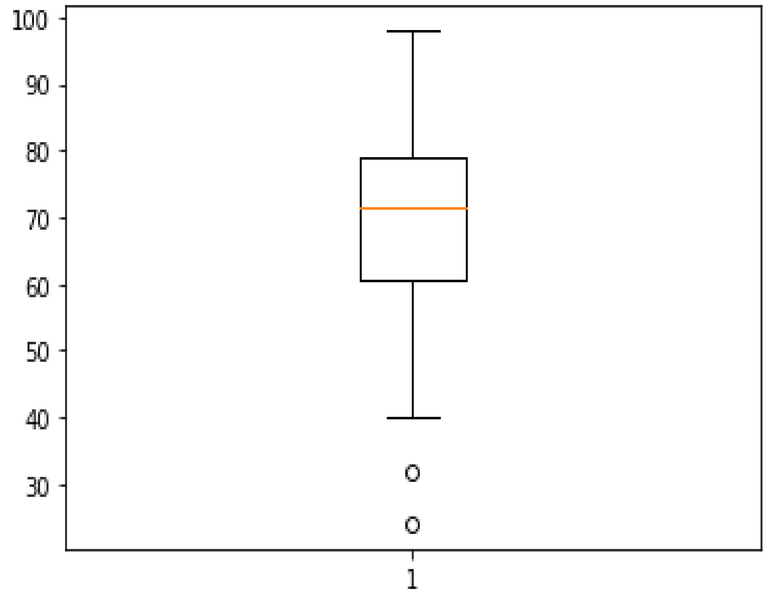
- The term "box plot" comes from the fact that the graph looks like a rectangle with lines extending from the top and bottom.



Example Box Plot

In [ ]:

```
import matplotlib.pyplot as plt

value1 = [82,76,24,40,67,62,75,78,71,32,98,89,78,67,72,82,87,66,56,52]

plt.boxplot(value1)
plt.show()
```



Python Libraries for Data Science

Seaborn:

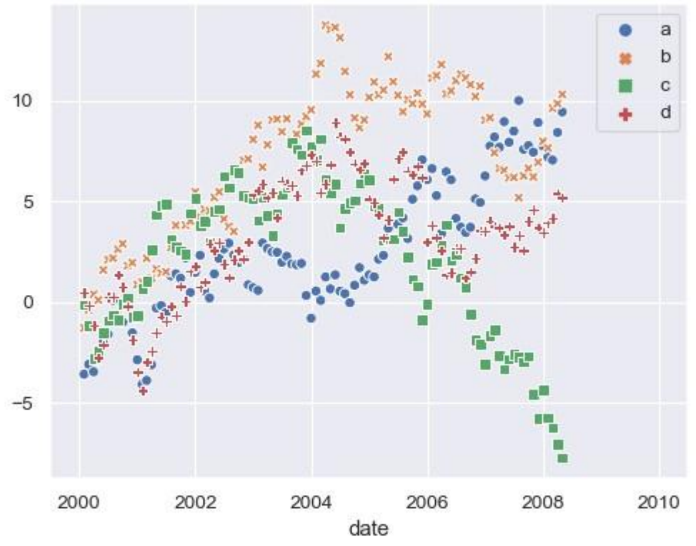
- Data Analysis and Visualization using Python Sept 2017 seaborn - harnesses the power of matplotlib to create beautiful charts in a few lines of code.
- based on matplotlib
- provides high level interface for drawing attractive statistical graphics
- Similar (in style) to the popular ggplot2 library in R

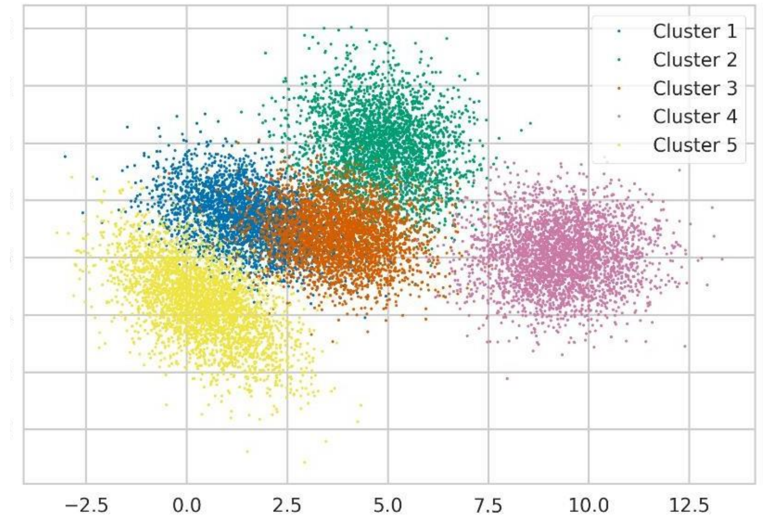
Link: https://seaborn.pydata.org/

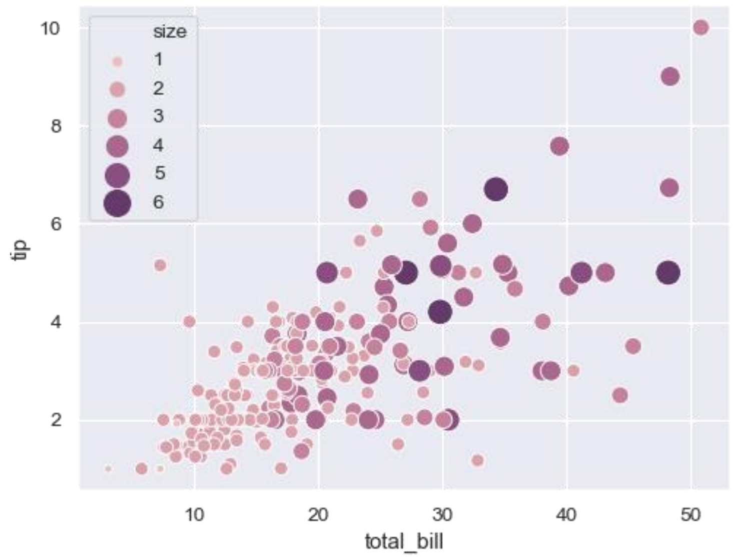
Seaborn Plots

Graphics	Description
distplot	histogram
barplot	estimate of central tendency for a numeric variable
violinplot	similar to boxplot, also shows the probability density of the data
jointplot	Scatterplot
regplot	Regression plot
pairplot	Pairplot
boxplot	boxplot
swarmplot	categorical scatterplot
factorplot	General categorical plot

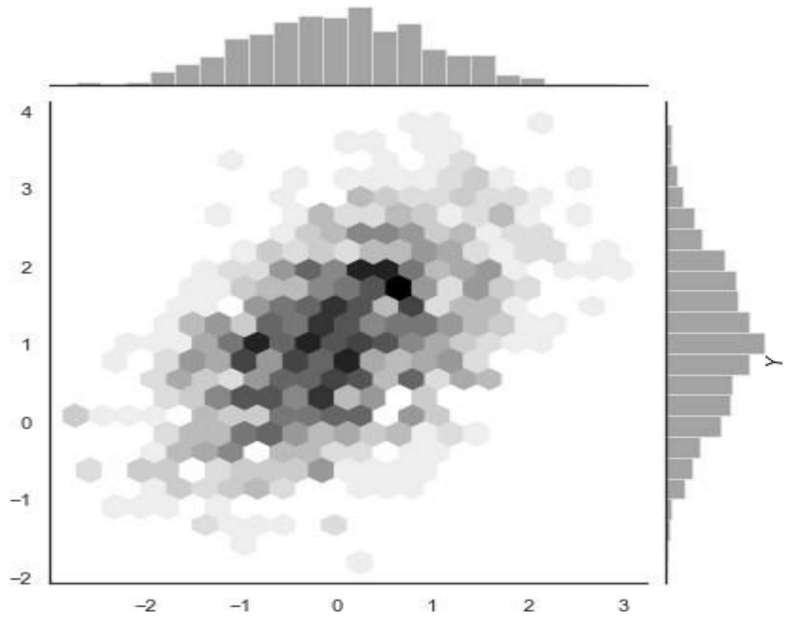
Example of Scatter Plot Plots

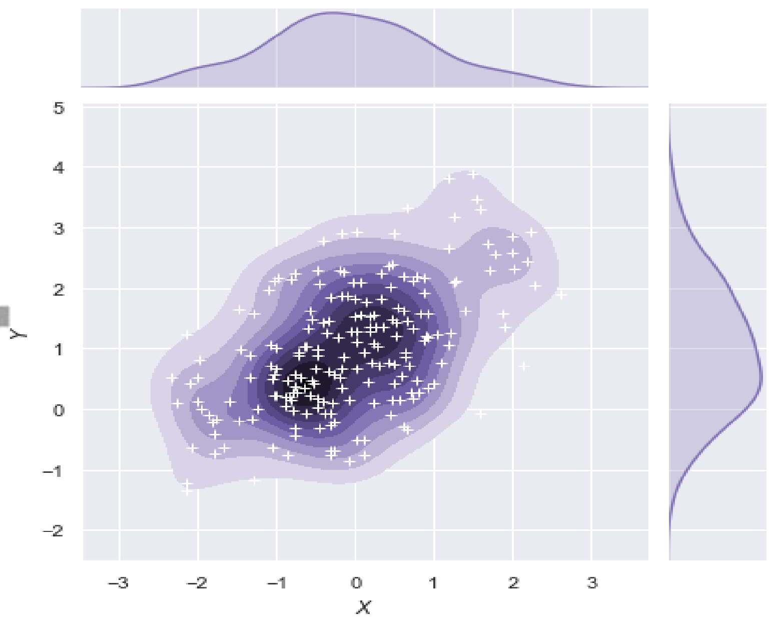


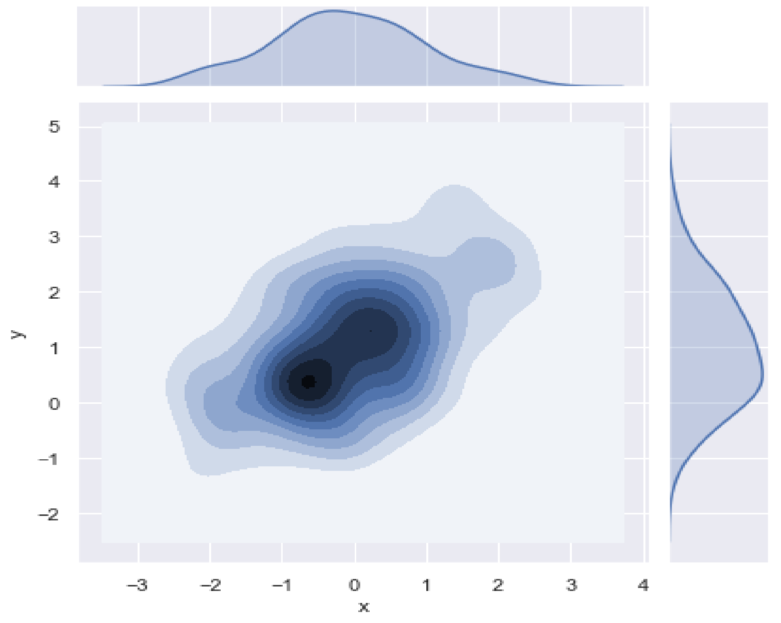




Example of Density Plot



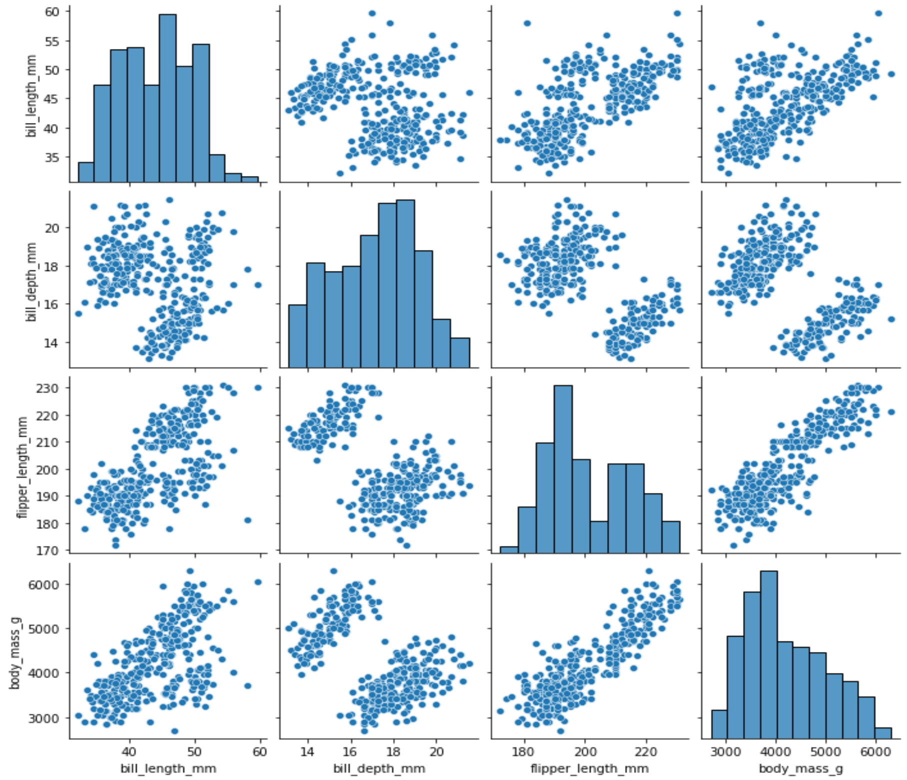




Example Pair Plot

In [ ]:

```
import seaborn as sns
penguins = sns.load_dataset("penguins")
sns.pairplot(penguins)
```



Thank You !