Data Science
21CSS303T

Unit I

Unit-1: INTRODUCTION TO DATA SCIENCE 10 hours

Benefits and uses of Data science, Facets of data, The data science process

Introduction to Numpy: **Numpy, creating array, attributes, Numpy Arrays objects: Creating Arrays, basic operations (Array Join, split, search, sort), Indexing, Slicing and iterating, copying arrays, Arrays shape manipulation, Identity array, eye function Exploring Data using Series, Exploring Data using Data Frames, Index objects, Re-index, Drop Entry, Selecting Entries, Data Alignment, Rank and Sort, Summary Statistics, Index Hierarchy**

Data Acquisition: **Gather information from different sources, Web APIs, Open Data Sources, Web Scrapping.**

Big Data vs Data Science

- *Big data* is a blanket term for any collection of data sets so large or complex that it becomes difficult to process them using traditional data management techniques such as, for example, the RDBMS (relational database management systems).

- *Data science* involves using methods to analyze massive amounts of data and extract the knowledge it contains.

You can think of the relationship between big data and data science as being like the relationship between crude oil and an oil refinery.

Characteristics of Big Data

- *Volume*—How much data is there?
- *Variety*—How diverse are different types of data?
- *Velocity*—At what speed is new data generated?

Benefits and uses of data science and big data

1. It’s in Demand
2. Abundance of Positions
3. A Highly Paid Career
4. Data Science is Versatile
5. Data Science Makes Data Better
6. Data Scientists are Highly Prestigious
7. No More Boring Tasks
8. Data Science Makes Products Smarter
9. Data Science can Save Lives

Facets of data

- Structured
- Unstructured
- Natural language
- Machine-generated
- Graph-based
- Audio, video, and images
- Streaming

Structured Data

- **Structured data** is data that depends on a data model and resides in a fixed field within a record.

1	Indicator ID	Dimension List	Timeframe	Numeric Value	Missing Value Flag	Confidence Inte
2	214390830	Total (Age-adjusted)	2008	74.6%		73.8%
3	214390833	Aged 18-44 years	2008	59.4%		58.0%
4	214390831	Aged 18-24 years	2008	37.4%		34.6%
5	214390832	Aged 25-44 years	2008	66.9%		65.5%
6	214390836	Aged 45-64 years	2008	88.6%		87.7%
7	214390834	Aged 45-54 years	2008	86.3%		85.1%
8	214390835	Aged 55-64 years	2008	91.5%		90.4%
9	214390840	Aged 65 years and over	2008	94.6%		93.8%
10	214390837	Aged 65-74 years	2008	93.6%		92.4%
11	214390838	Aged 75-84 years	2008	95.6%		94.4%
12	214390839	Aged 85 years and over	2008	96.0%		94.0%
13	214390841	Male (Age-adjusted)	2008	72.2%		71.1%
14	214390842	Female (Age-adjusted)	2008	76.8%		75.9%
15	214390843	White only (Age-adjusted)	2008	73.8%		72.9%
16	214390844	Black or African American only (Age-adjusted)	2008	77.0%		75.0%
17	214390845	American Indian or Alaska Native only (Age-adjusted)	2008	66.5%		57.1%
18	214390846	Asian only (Age-adjusted)	2008	80.5%		77.7%
19	214390847	Native Hawaiian or Other Pacific Islander only (Age-adjusted)	2008	DSU		
20	214390848	2 or more races (Age-adjusted)	2008	75.6%		69.6%

Unstructured data

- Unstructured data is data that isn't easy to fit into a data model because the content is context-specific or varying.

Natural language

- Natural language is a special type of unstructured data; it’s challenging to process because it requires knowledge of specific data science techniques and linguistics.

- The natural language processing community has had success in entity recognition, topic recognition, summarization, text completion, and sentiment analysis, but models trained in one domain don’t generalize well to other domains.

Machine-generated data

- Machine-generated data is information that's automatically created by a computer, process, application, or other machine without human intervention.
- Machine-generated data is becoming a major data resource and will continue to do so.

Machine-generated data

```
CSIPERF:TXCOMMIT;313236
2014-11-28 11:36:13, Info                  CSI    00000153 Creating NT transaction (seq
69), objectname [6]"(null)"
2014-11-28 11:36:13, Info                  CSI    00000154 Created NT transaction (seq 69)
result 0x00000000, handle @0x4e54
2014-11-28 11:36:13, Info                  CSI    00000155@2014/11/28:10:36:13.471
Beginning NT transaction commit...
2014-11-28 11:36:13, Info                  CSI    00000156@2014/11/28:10:36:13.705 CSI perf
trace:
CSIPERF:TXCOMMIT;273983
2014-11-28 11:36:13, Info                  CSI    00000157 Creating NT transaction (seq
70), objectname [6]"(null)"
2014-11-28 11:36:13, Info                  CSI    00000158 Created NT transaction (seq 70)
result 0x00000000, handle @0x4e5c
2014-11-28 11:36:13, Info                  CSI    00000159@2014/11/28:10:36:13.764
Beginning NT transaction commit...
2014-11-28 11:36:14, Info                  CSI    0000015a@2014/11/28:10:36:14.094 CSI perf
trace:
CSIPERF:TXCOMMIT;386259
2014-11-28 11:36:14, Info                  CSI    0000015b Creating NT transaction (seq
71), objectname [6]"(null)"
2014-11-28 11:36:14, Info                  CSI    0000015c Created NT transaction (seq 71)
result 0x00000000, handle @0x4e5c
2014-11-28 11:36:14, Info                  CSI    0000015d@2014/11/28:10:36:14.106
Beginning NT transaction commit...
2014-11-28 11:36:14, Info                  CSI    0000015e@2014/11/28:10:36:14.428 CSI perf
trace:
CSIPERF:TXCOMMIT;375581
```

Graph-based or network data

- “Graph data” can be a confusing term because any data can be shown in a graph.
- “Graph” in this case points to mathematical *graph theory*.
- In graph theory, a graph is a mathematical structure to model pair-wise relationships between objects.
- **Graph or network data** is, in short, data that focuses on the relationship or adjacency of objects.
- The graph structures use nodes, edges, and properties to represent and store graphical data.
- Graph-based data is a natural way to represent social networks, and its structure allows you to calculate specific metrics such as the influence of a person and the shortest path between two people.

Graph-based or network data

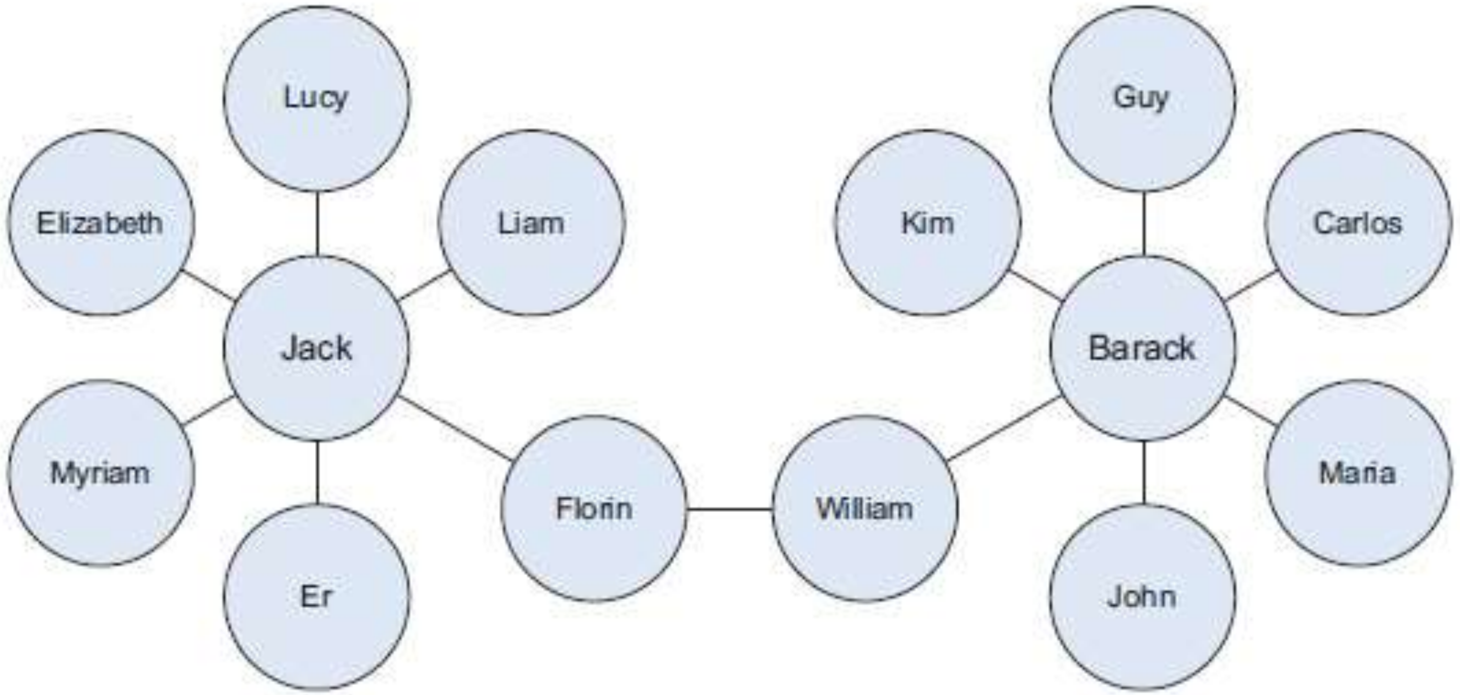


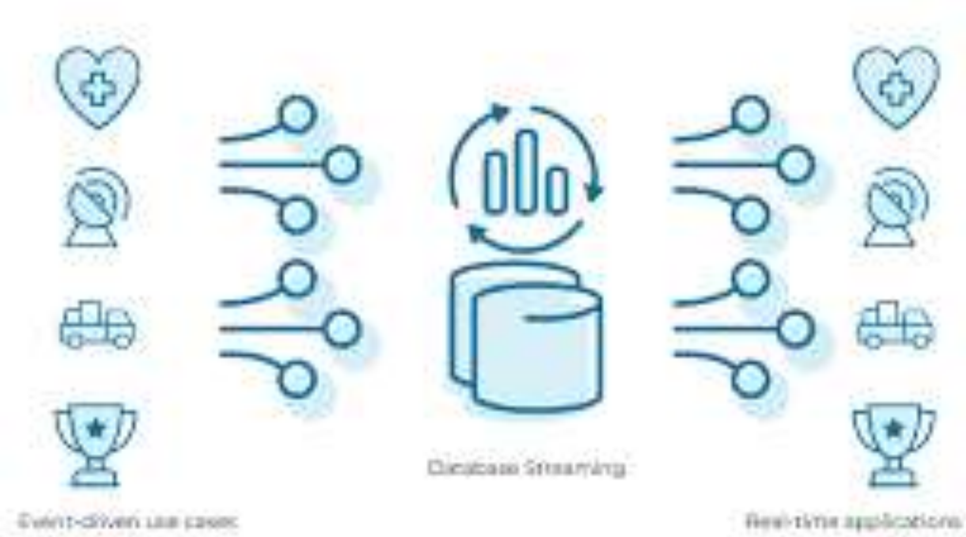
Figure 1.4 Friends in a social network are an example of graph-based data.

Audio, video and image

- Audio, image, and video are data types that pose specific challenges to a data scientist.

Streaming

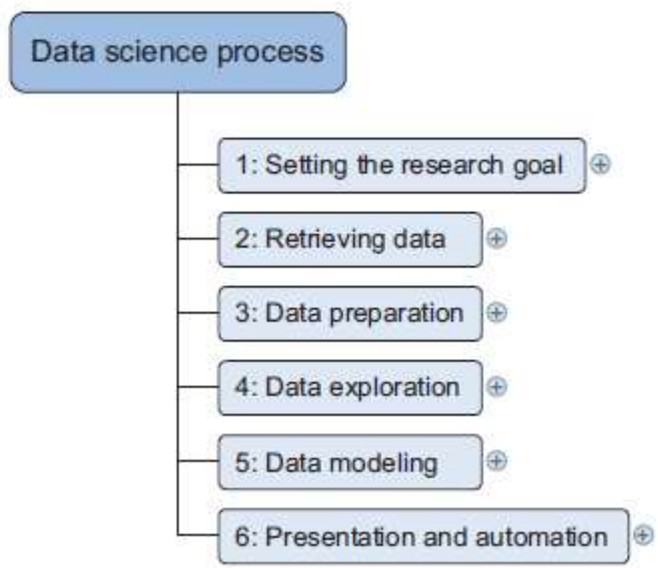
- While streaming data can take almost any of the previous forms, it has an extra property.
- The data flows into the system when an event happens instead of being loaded into a data store in a batch.



The Data Science Process

The Data Science Process

- The data science process typically consists of six steps, as you can see in the mind map



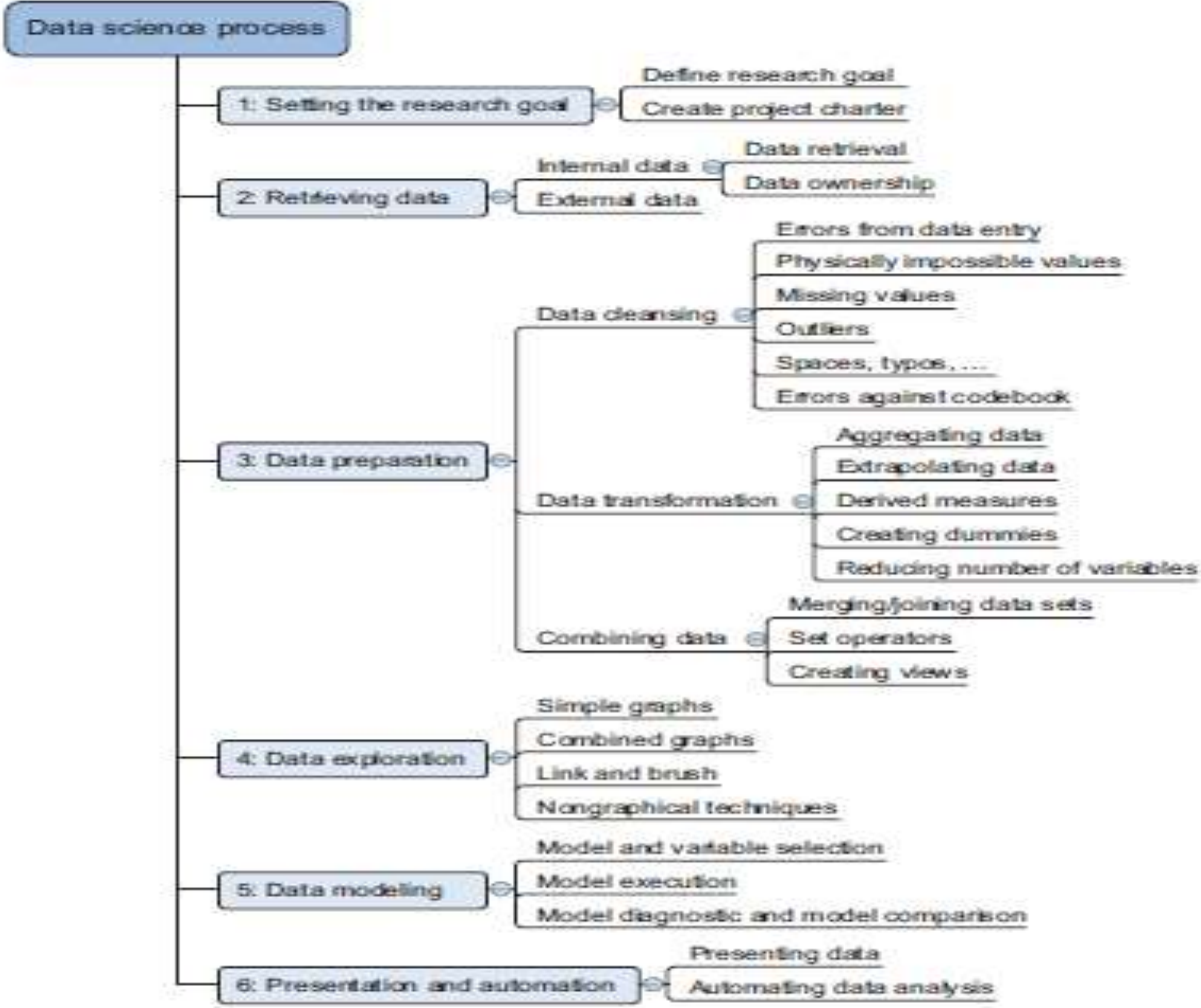


Figure 2.1 The six steps of the data science process

Setting the research goal

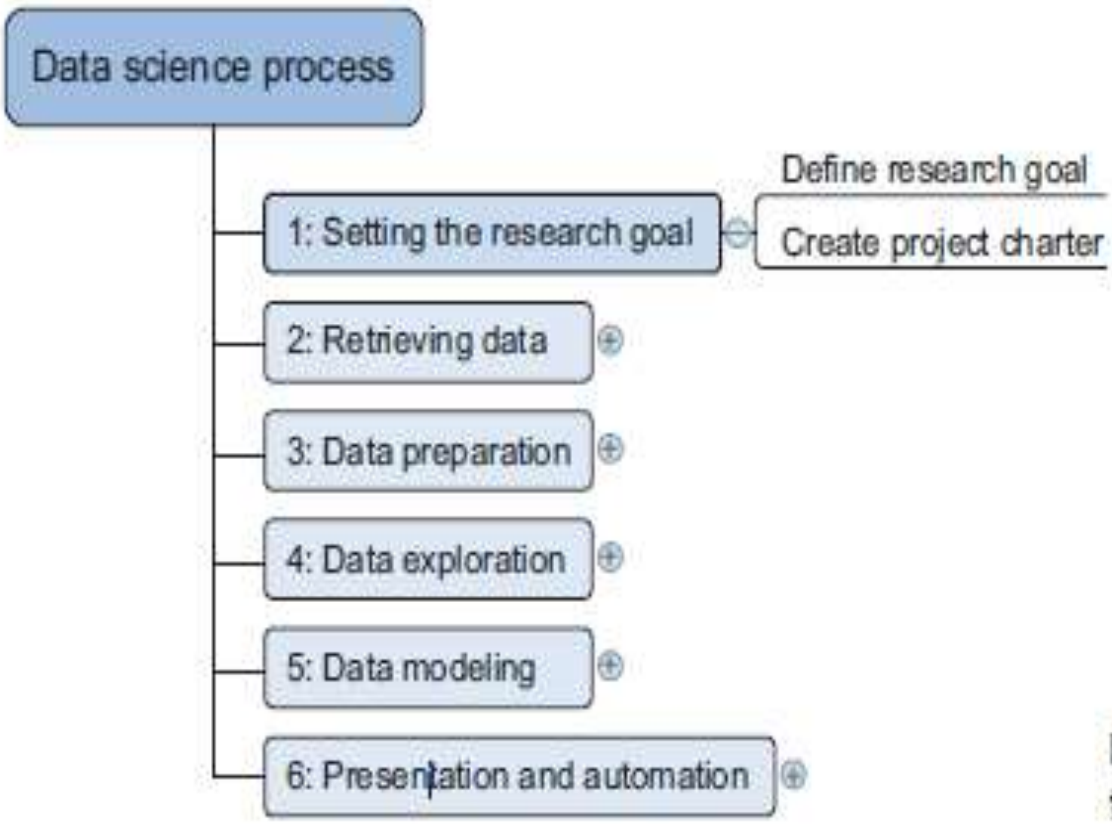


Figure 2.2 Step 1: Setting the research goal

Setting the research goal

- Data science is mostly applied in the context of an organization.
  - **A clear research goal**
  - **The project mission and context**
  - **How you’re going to perform your analysis**
  - **What resources you expect to use**
  - **Proof that it’s an achievable project, or proof of concepts**
  - **Deliverables and a measure of success**
  - **A timeline**

Retrieving data

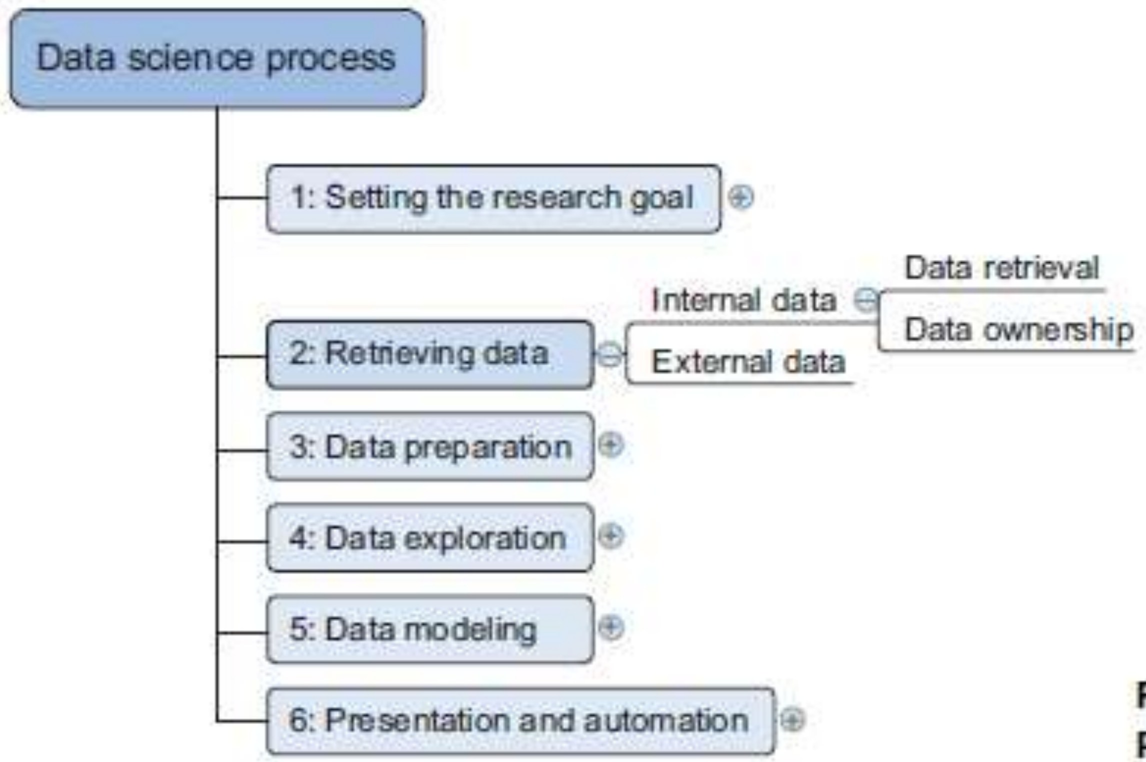


Figure 2.3 Step 2: Retrieving data

Retrieving data

- Data can be stored in many forms, ranging from simple text files to tables in a database.
- The objective now is acquiring all the data you need.

- ***Start with data stored within the company***
  - ***Databases***
  - ***Data marts***
  - ***Data warehouses***
  - ***Data lakes***

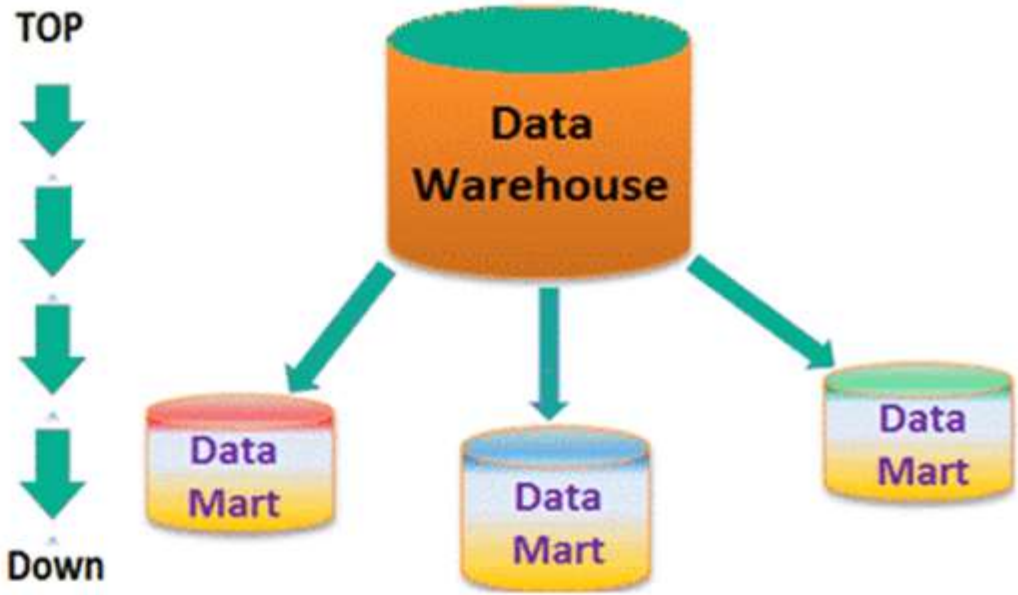
Data Lakes

- A data lake is a centralized storage repository that holds a massive amount of structured and unstructured data.
- According to Gartner, "it is a collection of storage instances of various data assets additional to the originating data sources."

Data warehouse

- Data warehousing is about the collection of data from varied sources for meaningful business insights.
- An electronic storage of a massive amount of information, it is a blend of technologies that enable the strategic use of data!

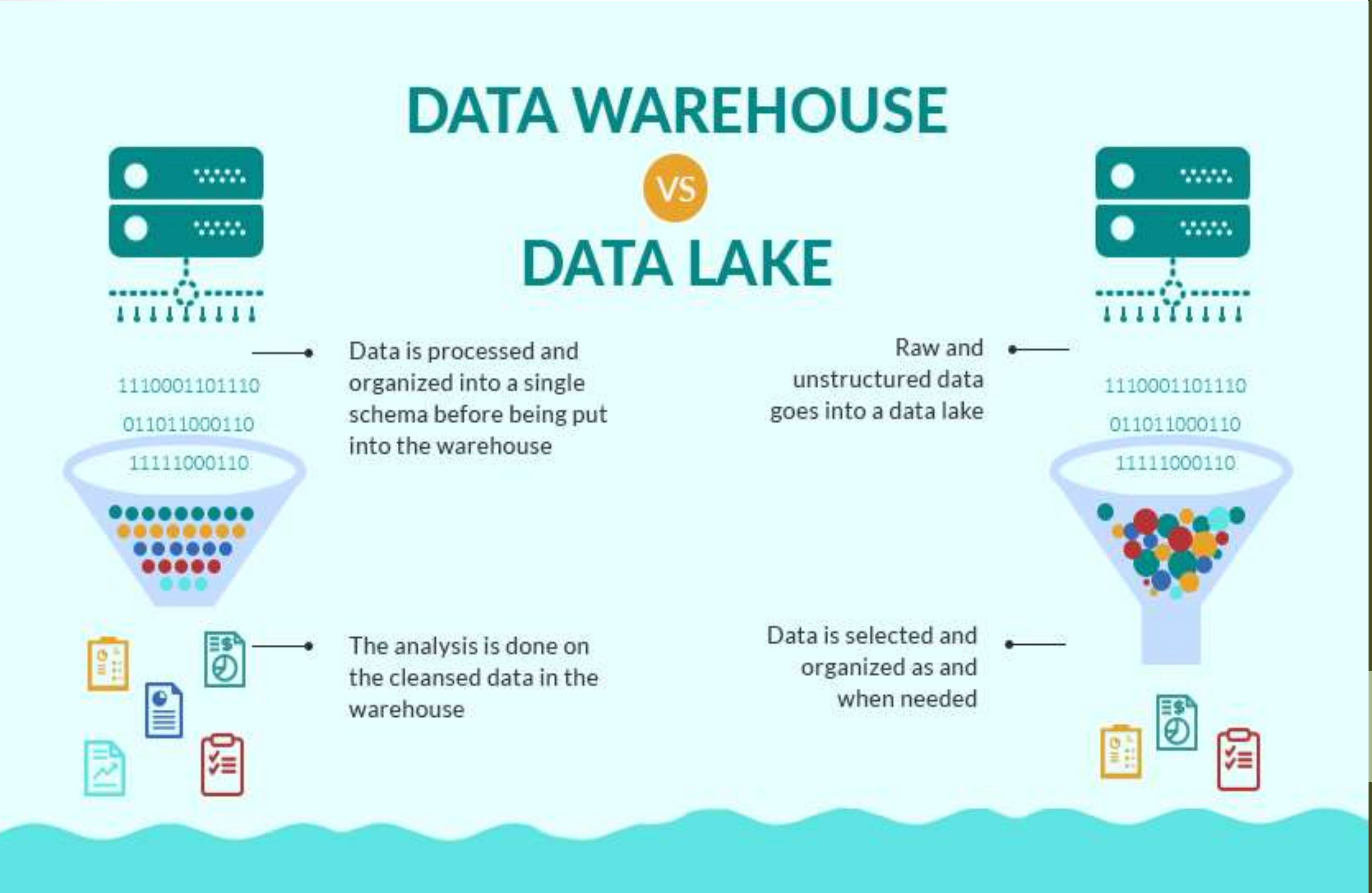
Data Mart



DWH vs DM

- Data Warehouse is a **large repository of data** collected from different sources whereas Data Mart is only subtype of a data warehouse.
- Data Warehouse is focused on **all departments** in an organization whereas Data Mart focuses on a **specific group**.
- Data Warehouse designing process is **complicated** whereas the Data Mart process is **easy** to design.
- Data Warehouse takes a **long time** for data handling whereas Data Mart takes a **short time** for data handling.
- Comparing Data Warehouse vs Data Mart, Data Warehouse size range is **100 GB to 1 TB+** whereas Data Mart size is less than **100 GB.**
- When we differentiate Data Warehouse and Data Mart, Data Warehouse implementation process takes **1 month to 1 year** whereas Data Mart takes a **few months** to complete the implementation process.

DWH vs DL



DATA WAREHOUSE
VS
DATA LAKE
Data is processed and organized into a single schema before being put into the warehouse
1110001101110
011011000110
11111000110
The analysis is done on the cleansed data in the warehouse
Raw and unstructured data goes into a data lake
1110001101110
011011000110
11111000110
Data is selected and organized as and when needed

Parameters	Data Lake	Data Warehouse
Data Structure	Data is raw and all types—structured, semi-structured, or unstructured—is captured in its original form.	Data is processed and only structured information is captured and organized in schemas.
Users	Ideal for users who carry out deep analysis such as data scientists and need advanced analytical tools.	Ideal for operational users such as business professionals and moguls since the data is structured and easy to use.
Storage Costs	Storing data is relatively inexpensive.	Storing data is time-consuming and costly.
Accessibility	Updates can be made quickly thus making it highly accessible	Costly to make changes, thereby quite complicated
Position of Schema	Schema is defined after data is stored, thus making it highly agile.	Schema is defined before data is stored, thus offering performance and security.
Data Processing	Uses ELT (Extract Load Transform) process.	Uses ETL (Extract Transform Load) process.

Data Lakes

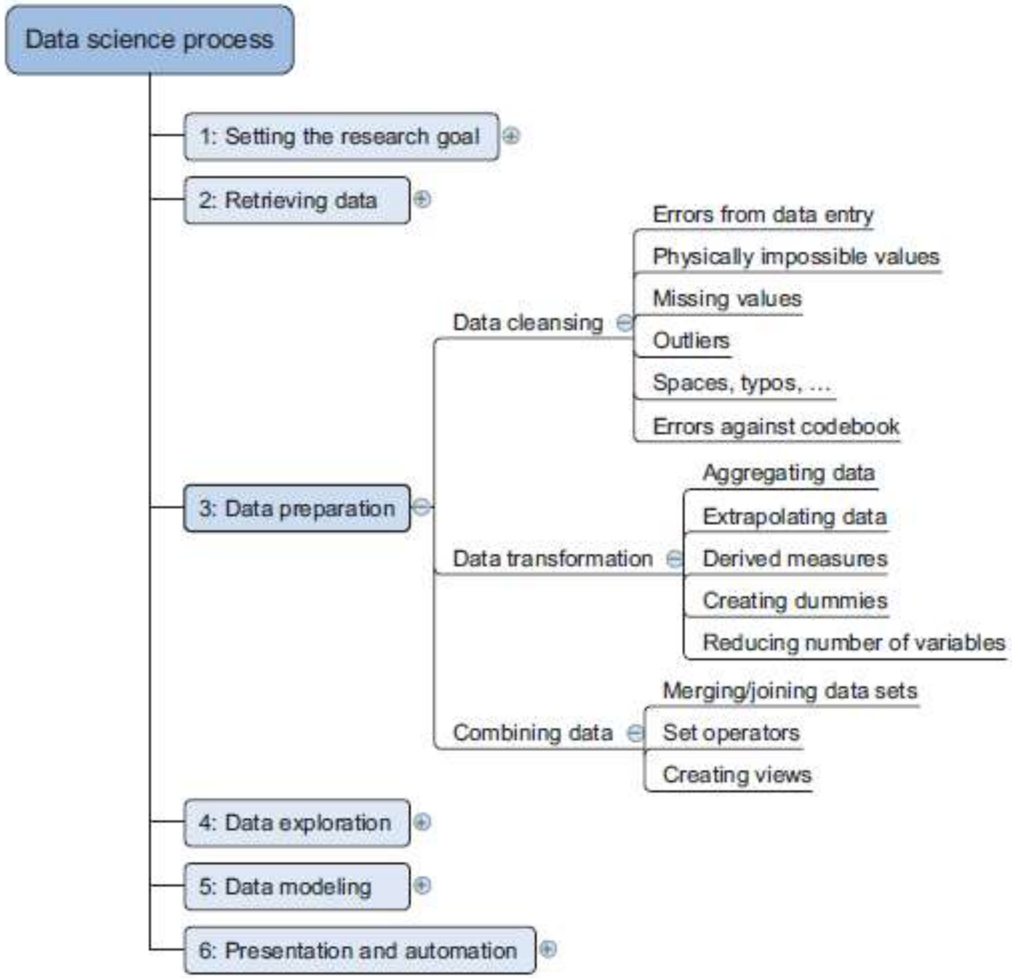
- Data lakes are a fairly new concept and experts have predicted that it might cause the death of data warehouses and data marts.
- Although with the increase of unstructured data, data lakes will become quite popular. But you will probably prefer keeping your structured data in a data warehouse.

Data Providers

Table 2.1 A list of open-data providers that should get you started

Open data site	Description
Data.gov	The home of the US Government's open data
https://open-data.europa.eu/	The home of the European Commission's open data
Freebase.org	An open database that retrieves its information from sites like Wikipedia, MusicBrains, and the SEC archive
Data.worldbank.org	Open data initiative from the World Bank
Aiddata.org	Open data for international development
Open.fda.gov	Open data from the US Food and Drug Administration

Cleansing, integration and transformation



Cleansing data

- **Data cleansing is a sub process of the data science process that focuses on removing errors in your data so your data becomes a true and consistent representation of the processes it originates from.**
- **True and consistent representation**
  - *interpretation error*
  - *inconsistencies*

Outliers

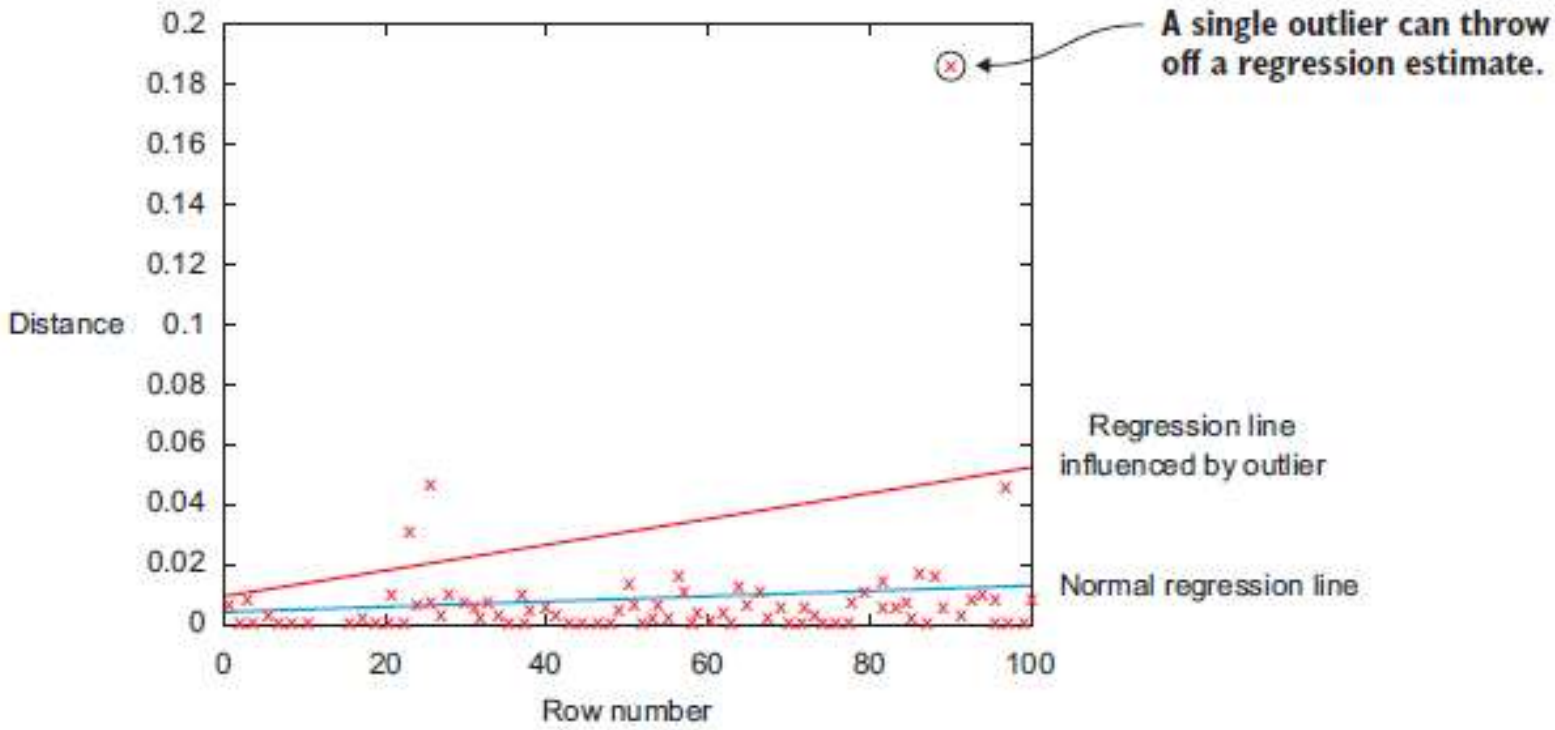


Figure 2.5 The encircled point influences the model heavily and is worth investigating because it can point to a region where you don't have enough data or might indicate an error in the data, but it also can be a valid data point.

Data Entry Errors

- Data collection and data entry are error-prone processes.
- They often require human intervention, and because humans are only human, they make **typos or lose their concentration for a second** and introduce an error into the chain. But data collected by machines or computers isn't free from errors either.
- Errors can arise from **human** sloppiness, whereas others are due to **machine or hardware** failure.

Data Entry Errors

Table 2.3 Detecting outliers on simple variables with a frequency table

Value	Count
Good	1598647
Bad	1354468
Godo	15
Bade	1

Redundant Whitespaces

- Whitespaces tend to be hard to detect but cause errors like other redundant characters would.
- Capital letter mismatches are common.
- Most programming languages make a distinction between “Brazil” and “brazil”. In this case you can solve the problem by applying a function that returns both strings in lowercase, such as .lower() in Python. “Brazil”.lower() == “brazil”.lower() should result in true.

Impossible values and Sanity checks

- Sanity checks are another valuable type of data check.
- Sanity checks can be directly expressed with rules:

    check = 0 <= age <= 120

Outliers

- An outlier is an observation that seems to be distant from other observations or, more specifically, one observation that follows a different logic or generative process than the other observations.
- Find outliers ➔ Use a plot or table

Outliers

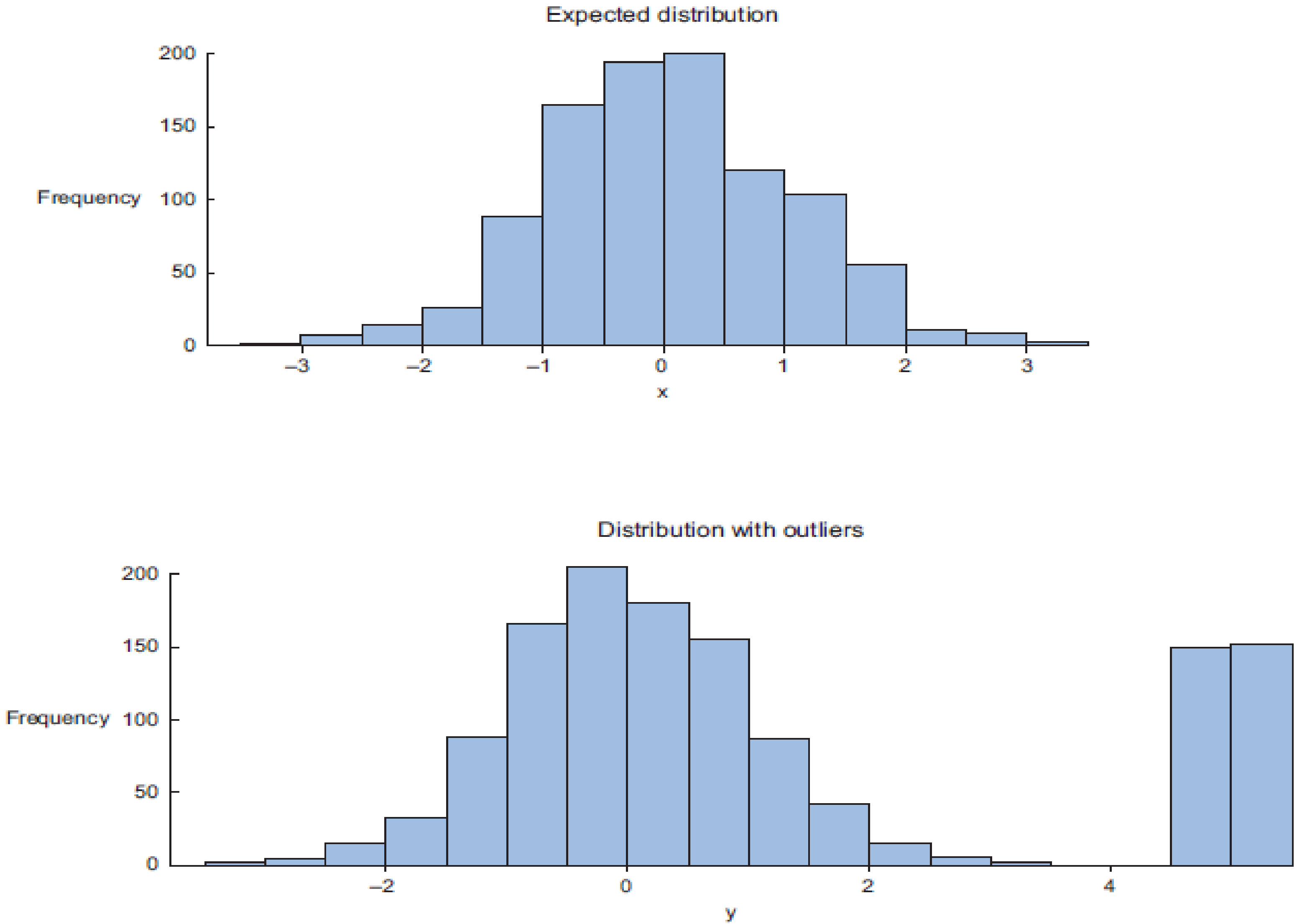


Figure 2.6 **Distribution plots are helpful in detecting outliers and helping you understand the variable.**

Handle missing data

Table 2.4 An overview of techniques to handle missing data

Technique	Advantage	Disadvantage
Omit the values	Easy to perform	You lose the information from an observation
Set value to `null`	Easy to perform	Not every modeling technique and/or implementation can handle `null` values
Impute a static value such as 0 or the mean	Easy to perform You don't lose information from the other variables in the observation	Can lead to false estimations from a model
Impute a value from an estimated or theoretical distribution	Does not disturb the model as much	Harder to execute You make data assumptions
Modeling the value (nondependent)	Does not disturb the model too much	Can lead to too much confidence in the model Can artificially raise dependence among the variables Harder to execute You make data assumptions

Deviations from a code book

- A code book is a **description of your data**, a form of metadata.
- It contains things such as the number of variables per observation, the number of observations, and what each encoding within a variable means.(For instance “0” equals “negative”, “5” stands for “very positive”.)

Combining data from different data sources

- **Joining** ➔ enriching an observation from one table with information from another table
- **Appending or Stacking** ➔adding the observations of one table to those of another table.

Joining

- **Joining ➔ focus on enriching a single observation**
- To join tables, you use variables that represent the same object in both tables, such as a date, a country name, or a Social Security number. These common fields are known as keys.
- When these keys also uniquely define the records in the table they are called **Primary Keys**

Figure 2.7 Joining two tables on the Item and Region keys

Appending

- **Appending ➔** effectively adding observations from one table to another table.

Client	Item	Month
John Doe	Coca-Cola	January
Jackie Qi	Pepsi-Cola	January

Client	Item	Month
John Doe	Zero-Cola	February
Jackie Qi	Maxi-Cola	February

Client	Item	Month
John Doe	Coca-Cola	January
Jackie Qi	Pepsi-Cola	January
John Doe	Zero-Cola	February
Jackie Qi	Maxi-Cola	February

Figure 2.8 Appending data from tables is a common operation but requires an equal structure in the tables being appended.

Views

- To avoid duplication of data, you virtually combine data with views
- Existing ➔ needed more storage space
- A view behaves as if you're working on a table, but this table is nothing but a virtual layer that combines the tables for you.

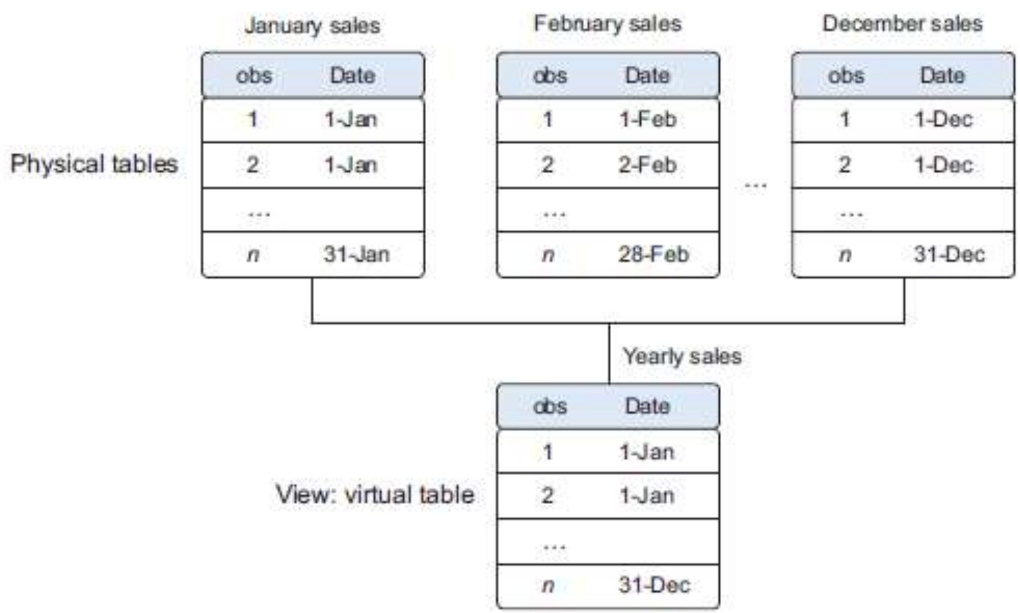


Figure 2.9 A view helps you combine data without replication.

Views

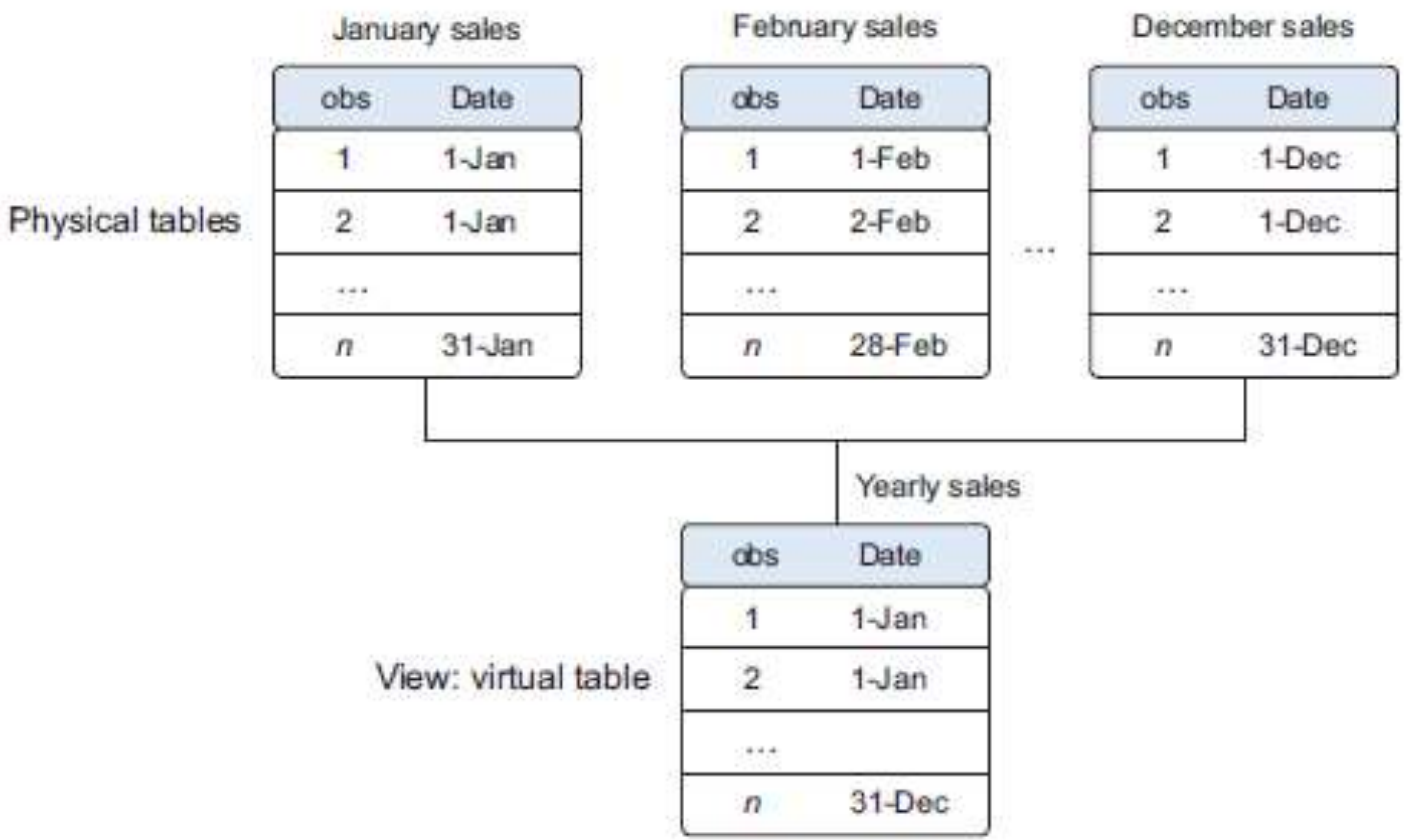


Figure 2.9 **A view helps you combine data without replication.**

Enriching aggregated measures

- Data enrichment can also be done by adding calculated information to the table, such as the total number of sales or what percentage of total stock has been sold in a certain region

Product class	Product	Sales in $	Sales t-1 in $	Growth	Sales by product class	Rank sales
A	B	X	Y	(X-Y) / Y	AX	NX
Sport	Sport 1	95	98	–3.06%	215	2
Sport	Sport 2	120	132	–9.09%	215	1
Shoes	Shoes 1	10	6	66.67%	10	3

Figure 2.10 Growth, sales by product class, and rank sales are examples of derived and aggregate measures.

Transforming data

- Certain models require their data to be in a certain shape.
- Transforming your data so it takes a suitable form for data modeling.

x	1	2	3	4	5	6	7	8	9	10
log(x)	0.00	0.43	0.68	0.86	1.00	1.11	1.21	1.29	1.37	1.43
y	0.00	0.44	0.69	0.87	1.02	1.11	1.24	1.32	1.38	1.46

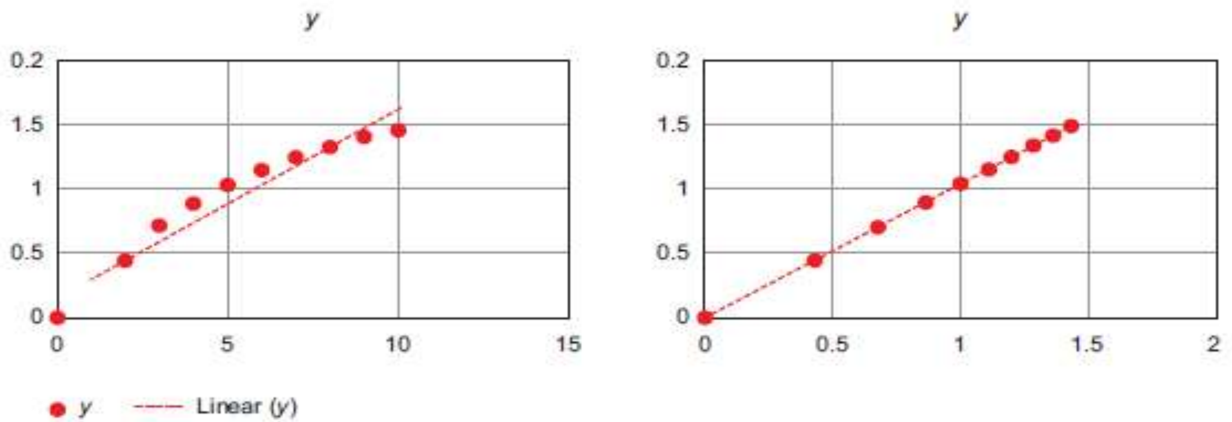


Figure 2.11 Transforming x to log x makes the relationship between x and y linear (right), compared with the non-log x (left).

Reducing the number of variables

- Too many variables
  - ➔don’t add new information to the model
  - ➔model difficult to handle
  - ➔certain techniques don’t perform well when you overload them with too many input variables
- Data scientists use special methods to reduce the number of variables but retain the maximum amount of data.

Turning variables into dummies

- *Dummy variables* can only take two values: true(1) or false(0). They're used to indicate the absence of a categorical effect that may explain the observation.

Customer	Year	Gender	Sales
1	2015	F	10
2	2015	M	8
1	2016	F	11
3	2016	M	12
4	2017	F	14
3	2017	M	13

M F

Customer	Year	Sales	Male	Female
1	2015	10	0	1
1	2016	11	0	1
2	2015	8	1	0
3	2016	12	1	0
3	2017	13	1	0
4	2017	14	0	1

Figure 2.13 Turning variables into dummies is a data transformation that breaks a variable that has multiple classes into multiple variables, each having only two possible values: 0 or 1.

Data Exploration

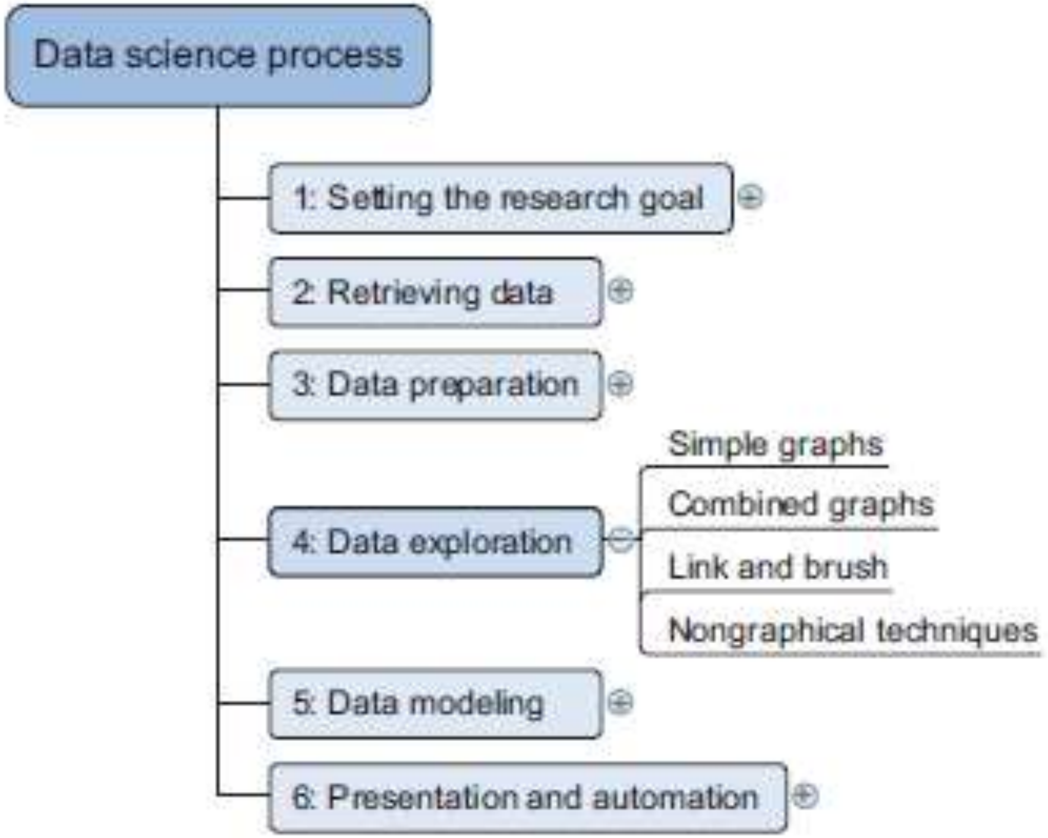
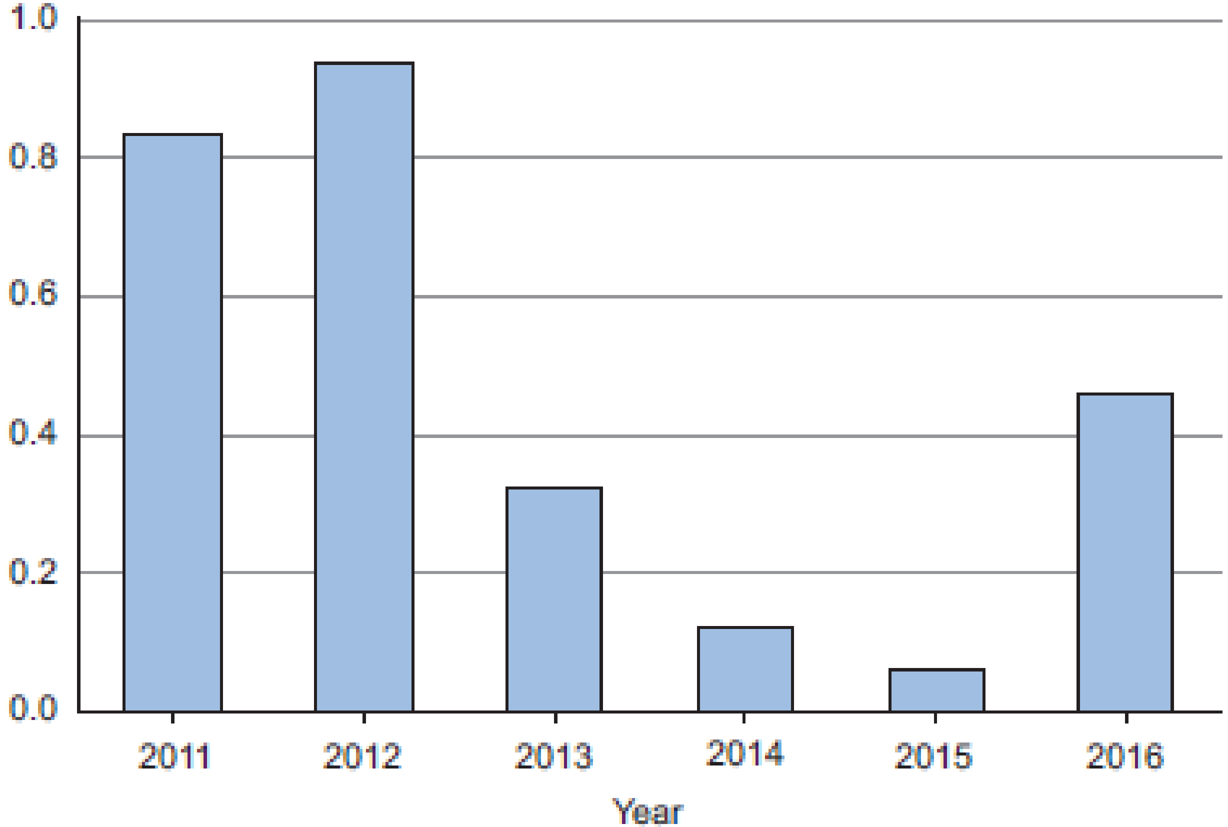


Figure 2.14 Step 4: Data exploration

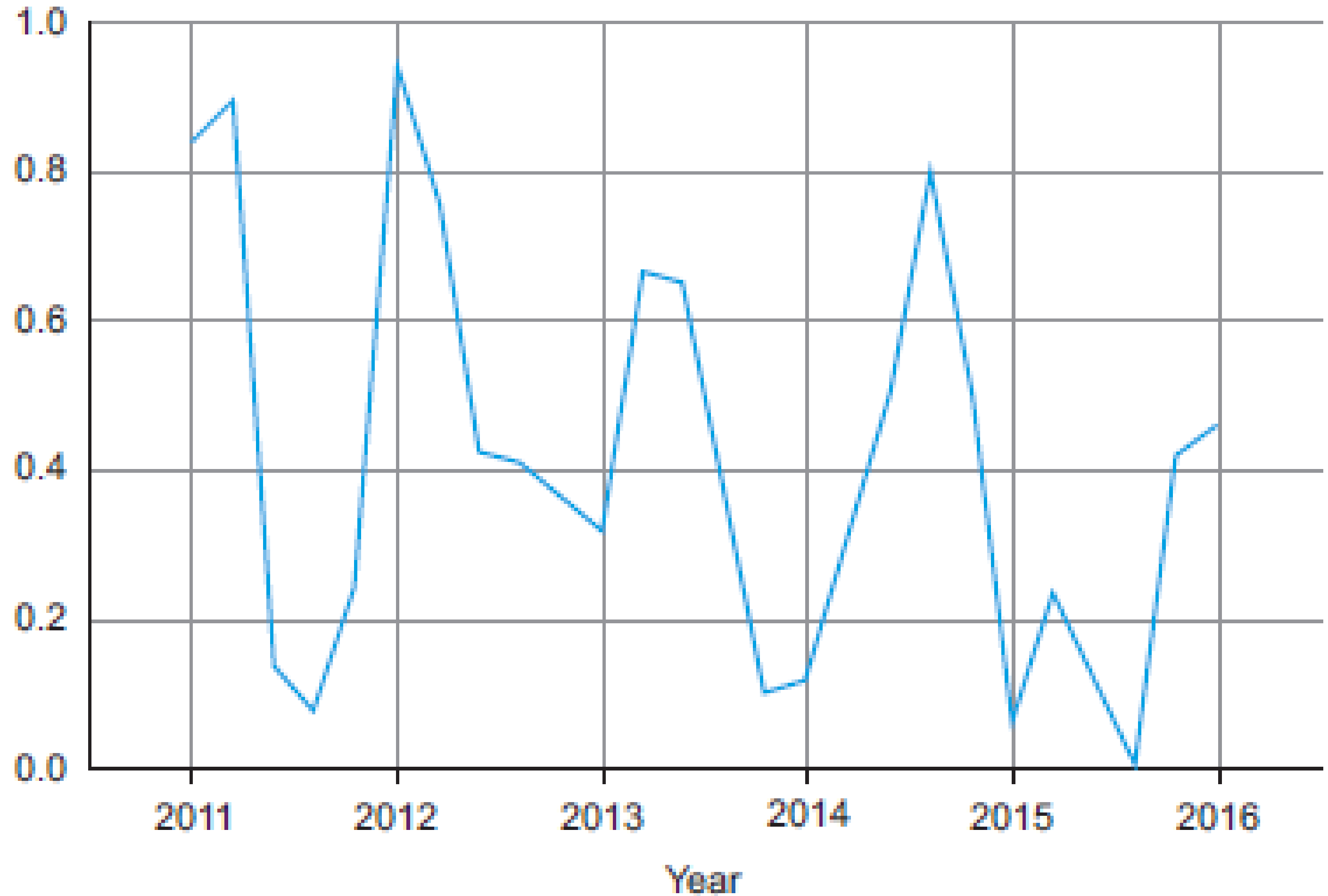
Data Exploration

- Information becomes much easier to grasp when shown in a picture, therefore you mainly use graphical techniques to gain an understanding of your data and the interactions between variables.
- Visualization Techniques
  - Simple graphs
  - Histograms
  - Sankey
  - Network graphs

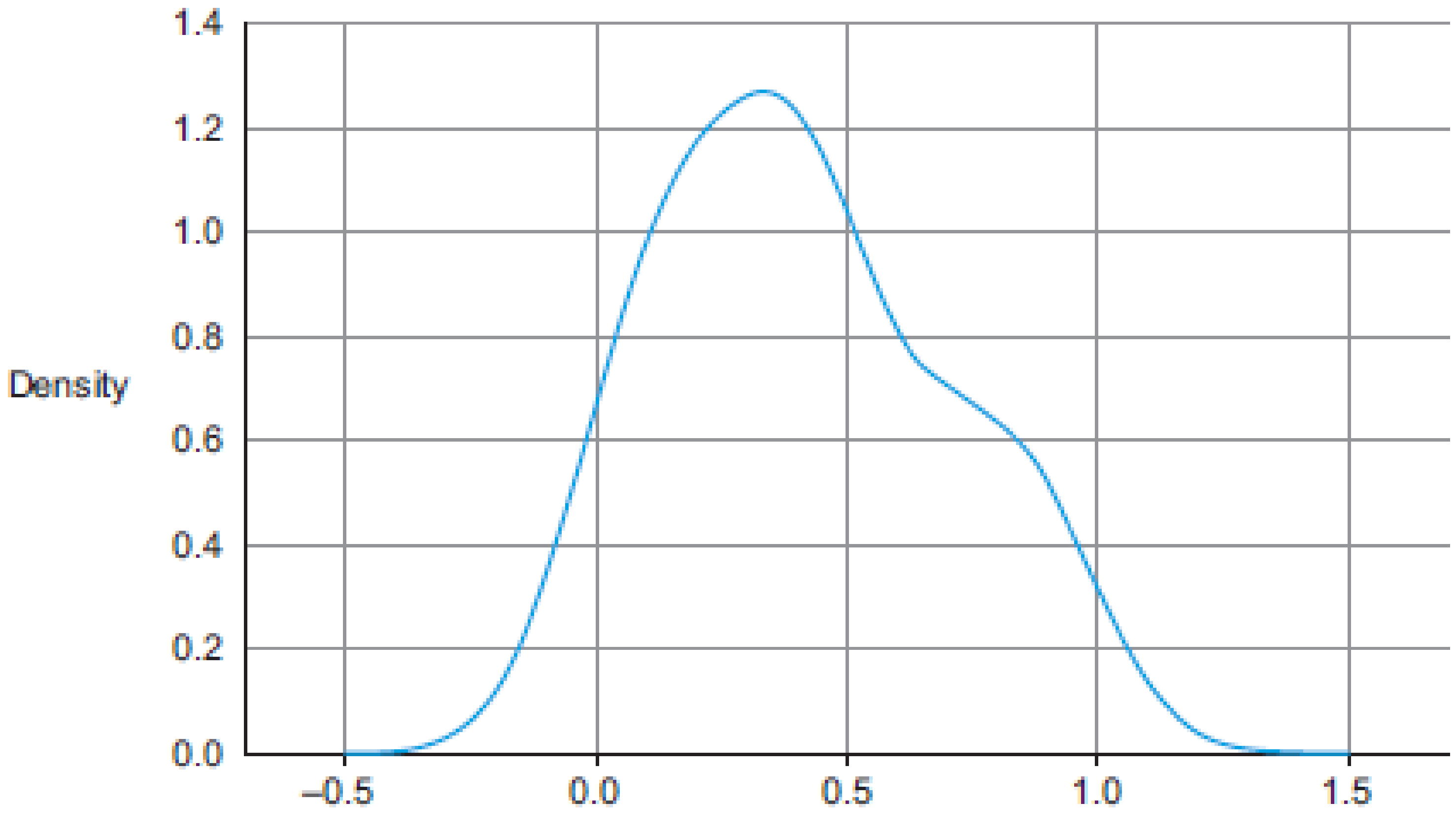
Bar Chart



Line Chart



Distribution



Overlaying

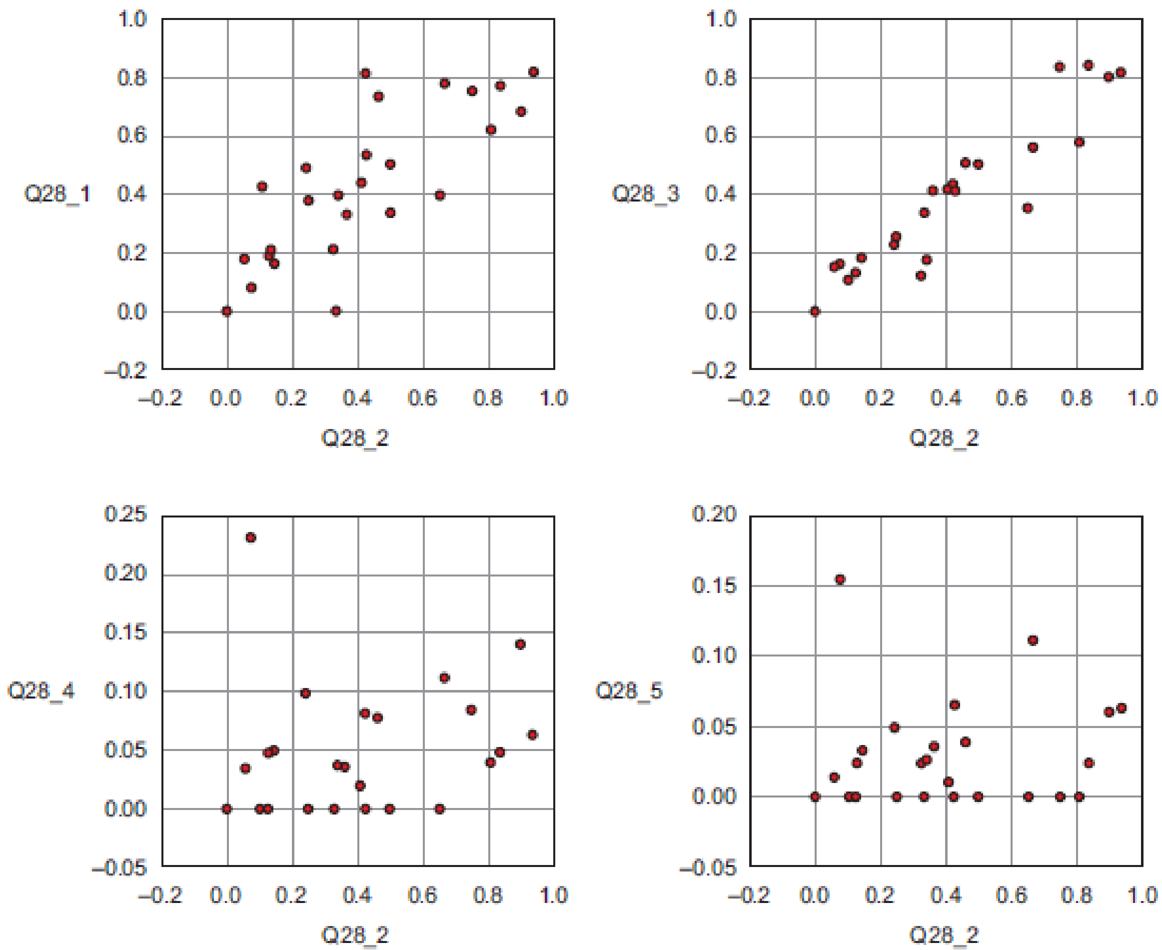


Figure 2.16 Drawing multiple plots together can help you understand the structure of your data over multiple variables.

Brushing and Linking

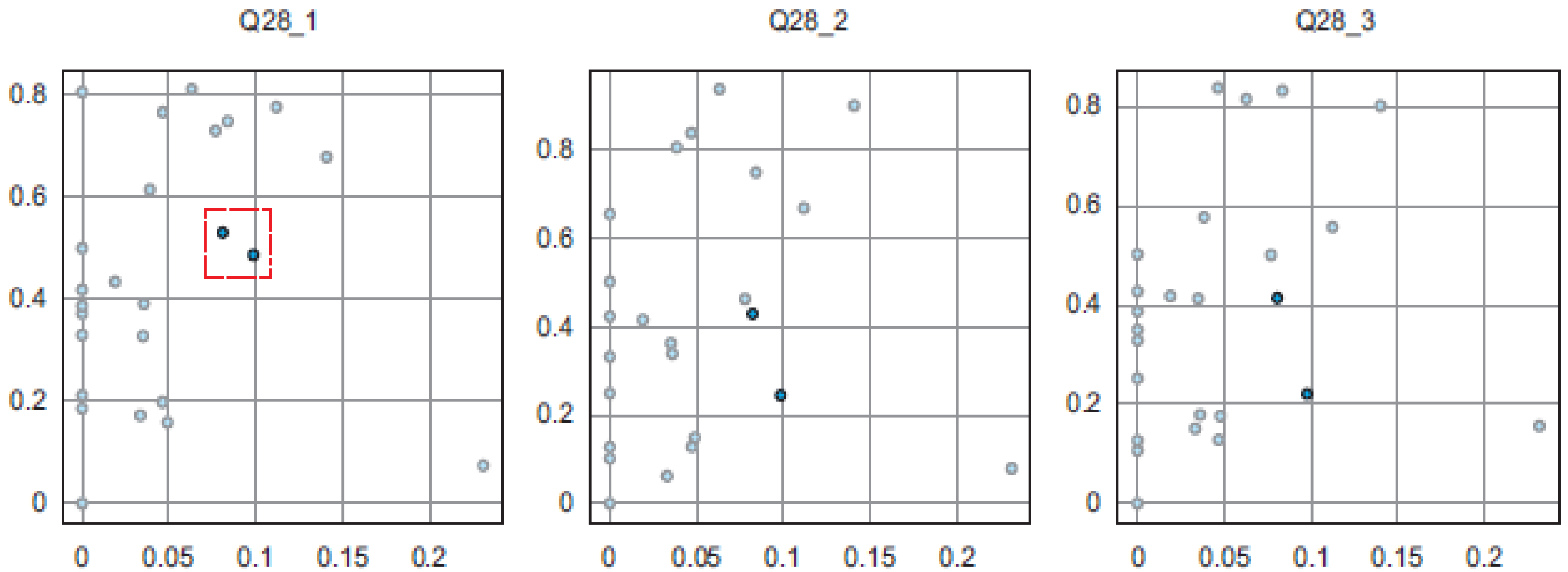


Figure 2.18 Link and brush allows you to select observations in one plot and highlight the same observations in the other plots.

STEP 5: BUILD THE MODELS

Data modeling

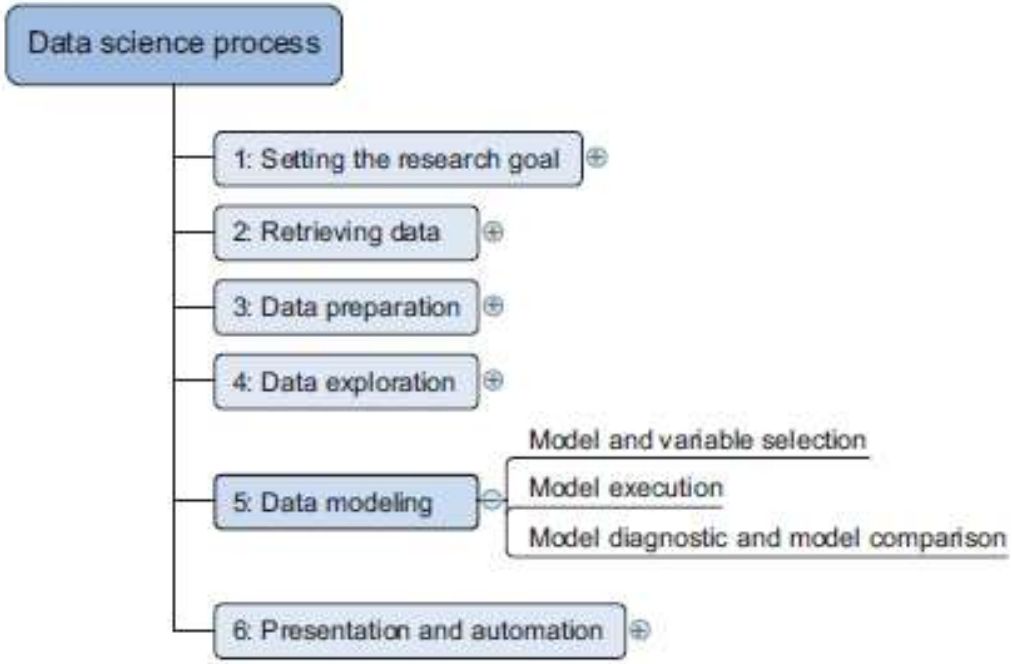


Figure 2.21 Step 5: Data modeling

Data modeling

- Building a model is an iterative process.
- The way you build your model depends on whether you go with classic statistics or the somewhat more recent machine learning school, and the type of technique you want to use.
- Models consist of the following main steps:
  - **1 Selection of a modeling technique and variables to enter in the model**
  - **2 Execution of the model**
  - **3 Diagnosis and model comparison**

Model and variable selection

- Must the model be moved to a production environment and, if so, would it be easy to implement?
- How difficult is the maintenance on the model: how long will it remain relevant if left untouched?
- Does the model need to be easy to explain?

Model execution

Listing 2.1 Executing a linear prediction model on semi-random data

```
import statsmodels.api as sm
import numpy as np
predictors = np.random.random(1000).reshape(500,2)
target = predictors.dot(np.array([0.4, 0.6])) + np.random.random(500)
lmRegModel = sm.OLS(target,predictors)
result = lmRegModel.fit()
result.summary()
```

Imports required Python modules.

Creates random data for predictors (x-values) and semi-random data for the target (y-values) of the model. We use predictors as input to create the target so we infer a correlation here.

Fits linear regression on data.

Shows model fit statistics.

Model execution

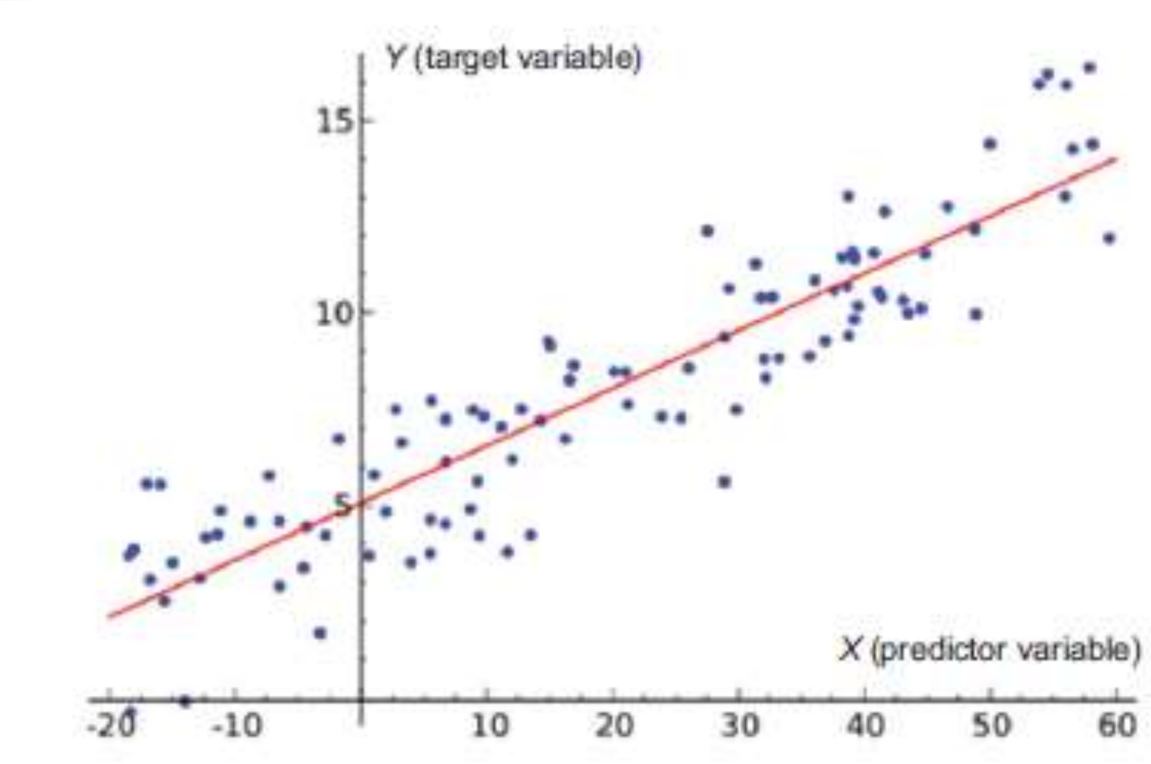


Figure 2.22 Linear regression tries to fit a line while minimizing the distance to each point

Model execution

Dep. Variable:	y	R-squared:	0.893
Model:	OLS	Adj. R-squared:	0.893
Method:	Least Squares	F-statistic:	2088.
Date:	Fri, 30 Oct 2015	Prob (F-statistic):	7.13e-243
Time:	12:44:31	Log-Likelihood:	-176.74
No. Observations:	500	AIC:	357.5
Df Residuals:	498	BIC:	365.9
Df Model:	2		
Covariance Type:	nonrobust		

Model fit: higher is better but too high is suspicious.

	coef	std err	t	P>|t|	[95.0% Conf. Int.]
x1	0.7658	0.040	19.130	0.000	0.687 0.844
x2	1.1252	0.039	28.603	0.000	1.048 1.202

p-value to show whether a predictor variable has a significant influence on the target. Lower is better and <0.05 is often considered "significant."

Omnibus:	34.269	Durbin-Watson:	1.943
Prob(Omnibus):	0.000	Jarque-Bera (JB):	13.480
Skew:	-0.125	Prob(JB):	0.00118
Kurtosis:	2.235	Cond. No.	2.51

Linear equation coefficients.
$y = 0.7658x1 + 1.1252x2$.

Figure 2.23 Linear regression model information output

Introduction to Numpy

NumPy Arrays

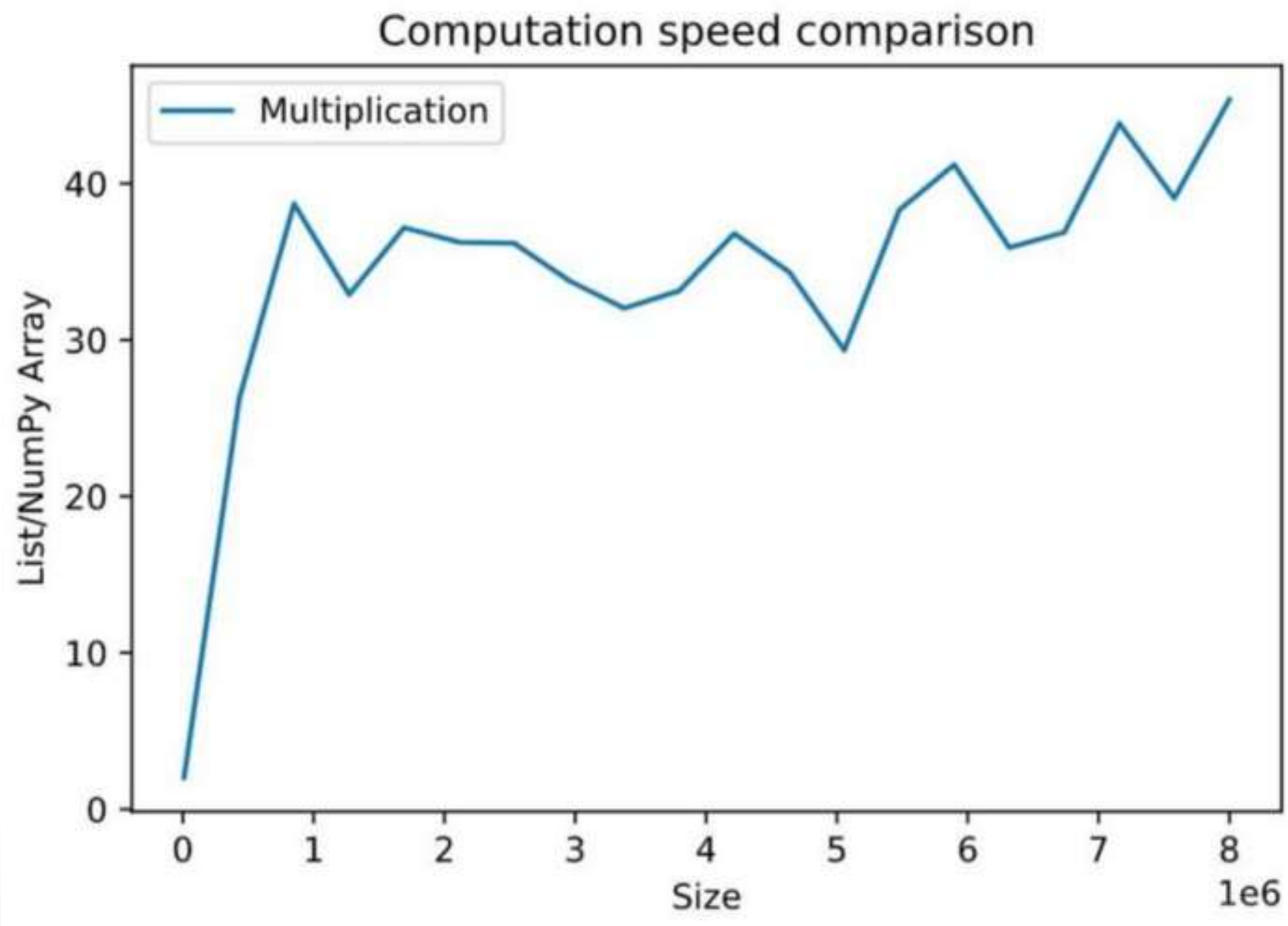
NumPy

- **Num**erical **Py**thon
- General-purpose array-processing package.
- High-performance multidimensional array object, and tools for working with these arrays.
- Fundamental package for scientific computing with Python.
- It is open-source software.

NumPy - Features

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Choosing NumPy over Python list



Array

- **An array is a data type used to store multiple values using a single identifier (variable name).**
- **An array contains an ordered collection of data elements where each element is of the same type and can be referenced by its index (position)**

Array

- **Similar to the indexing of lists**
- **Zero-based indexing**
  - **[10, 9, 99, 71, 90]**

NumPy Array

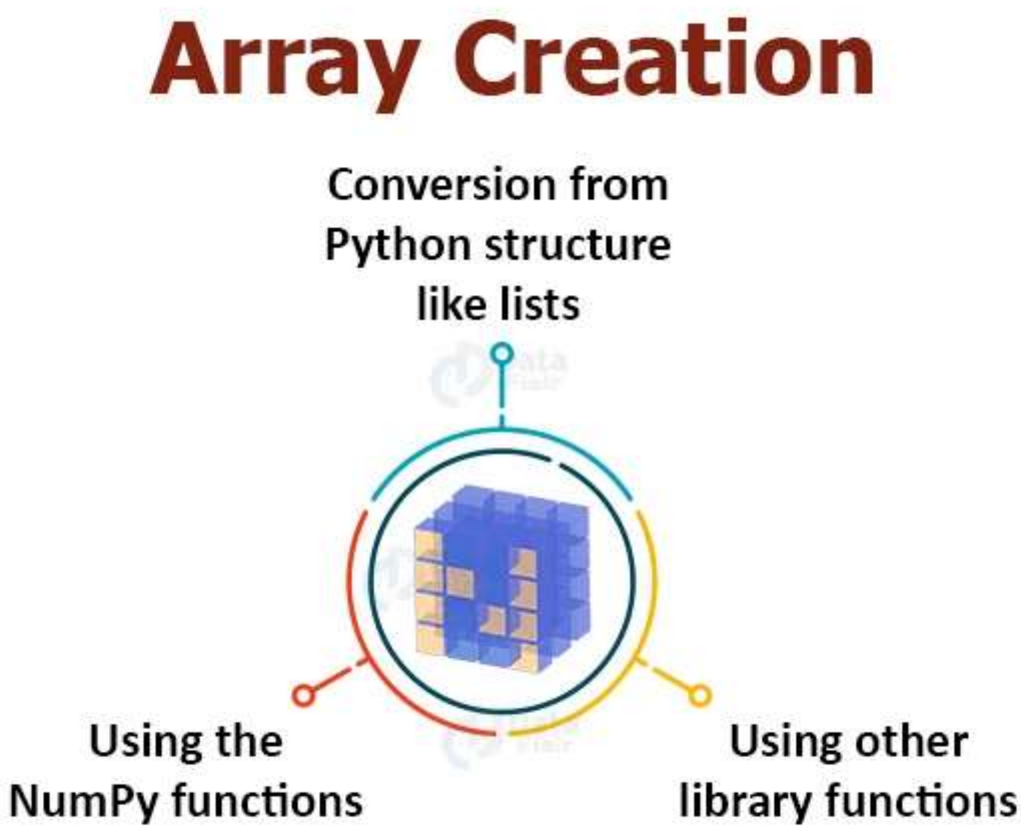
- Store lists of numerical data, vectors and matrices
- Large set of routines (built-in functions) for creating, manipulating, and transforming NumPy arrays.
- NumPy array is officially called ndarray but commonly known as array

Creation of NumPy Arrays from List

- **First we need to import the NumPy library**

  import numpy as np

Creation of Arrays



1. Using the NumPy functions

a. Creating one-dimensional array in NumPy

```
import numpy as np
array=np.arange(20)
array
```

Output:

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12, 13, 14, 15, 16, 17, 18, 19])

1. Using the NumPy functions

a. check the dimensions by using array.shape.

(20,)

Output:

array([0 1 2 3 4 5 6 7 8 9 10 1112 13 14,15, 16, 17, 18, 19])

1. Using the NumPy functions

b. Creating two-dimensional arrays in NumPy

array=np.arange(20).reshape(4,5)

Output:

array([[0, 1, 2, 3, 4],
[5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]
[15, 16, 17, 18, 19]])

1. Using the NumPy functions

c. Using other NumPy functions

np.zeros((2,4))

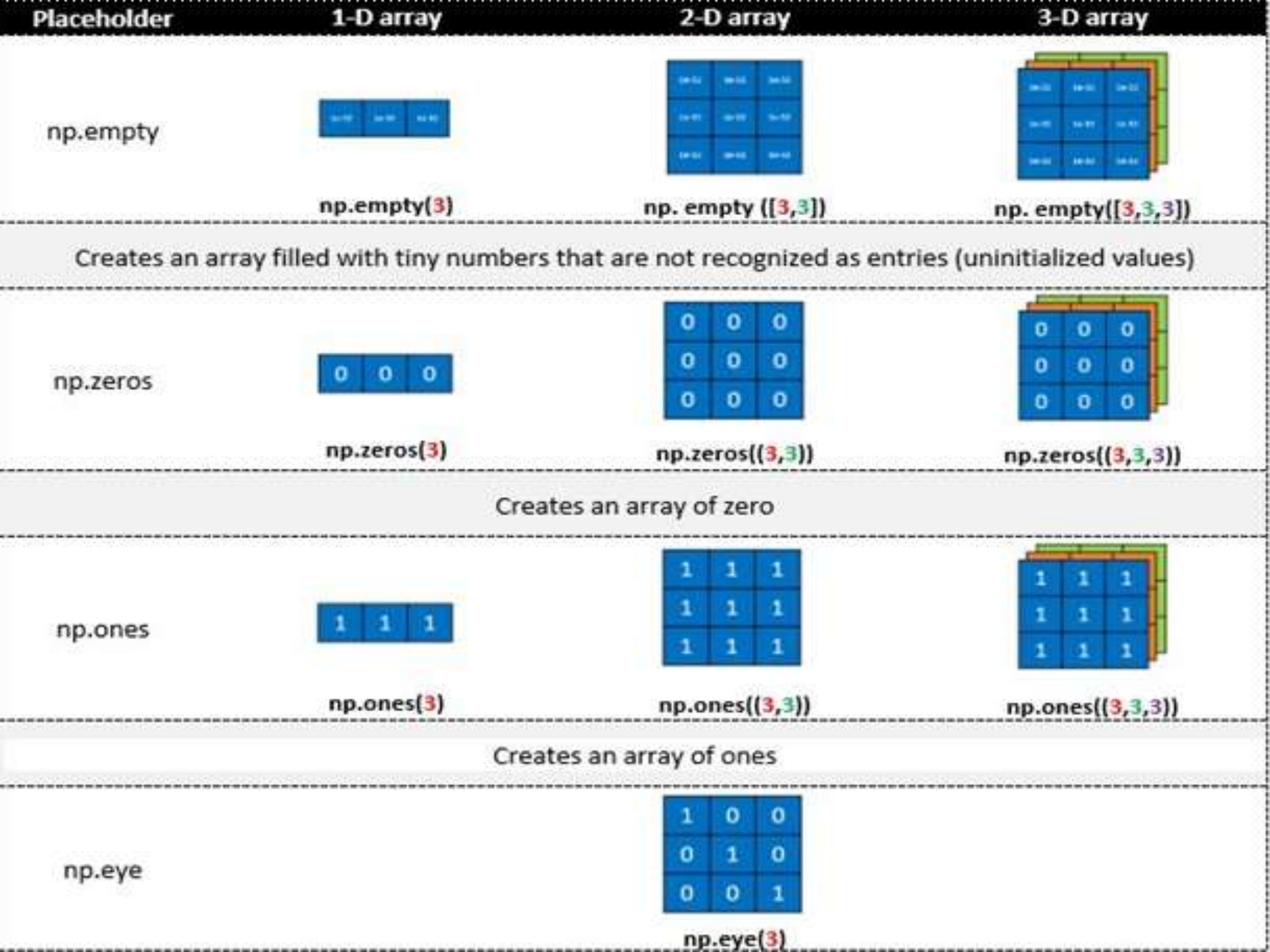
np.ones((3,6))

np.full((2,2), 3)

Output:

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

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



Placeholder
1-D array
2-D array
3-D array
np.empty
np.empty(3)
np. empty ([3,3])
np. empty([3,3,3])
Creates an array filled with tiny numbers that are not recognized as entries (uninitialized values)
np.zeros
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
np.zeros(3)
np.zeros((3,3))
np.zeros((3,3,3))
Creates an array of zero
np.ones
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
np.ones(3)
np.ones((3,3))
np.ones((3,3,3))
Creates an array of ones
np.eye
1 0 0
0 1 0
0 0 1
np.eye(3)

Placeholder	1-D array	2-D array	3-D array
np.full	$\begin{bmatrix}2 & 2 & 2\end{bmatrix}$ **np.full((3),2)**	$\begin{bmatrix}2 & 2 & 2\\2 & 2 & 2\\2 & 2 & 2\end{bmatrix}$ **np.full((3,3),2)**	$\begin{bmatrix}2 & 2 & 2\\2 & 2 & 2\\2 & 2 & 2\end{bmatrix}$ **np.full((3,3,3),2)**
Creates an array of with desired value. In this case array of 2			
np.random.rand	$\begin{bmatrix}0 & 0 & 1\end{bmatrix}$ **np.random.rand(3)**	$\begin{bmatrix}1 & 0 & 0\\1 & 1 & 1\\0 & 0 & 1\end{bmatrix}$ **np.random.rand(3,3)**	$\begin{bmatrix}0 & 0 & 1\\0 & 0 & 1\\1 & 1 & 0\end{bmatrix}$ **np.random.rand(3,3,3)**
Creates an array with random values			

1. Using the NumPy functions

c. Using other NumPy functions

```
import numpy as np
a=np.zeros((2,4))
b=np.ones((3,6))
c=np.empty((2,3))
d=np.full((2,2), 3)
e= np.eye(3,3)
f=np.linspace(0, 10, num=4)

print(a)
print(b)
print(c)
print(d)
```

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

[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]

[[1.14137702e-316 0.00000000e+000
6.91583610e-310]
 [6.91583609e-310 6.91583601e-310
6.91583601e-310]]

[[3 3]
 [3 3]]

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

[ 0.         3.33333333  6.66666667 10.
]
```

1. Using the NumPy functions

Sr No.	Function	Description
1	empty_like()	Return a new array with the same shape and type
2	ones_like()	Return an array of ones with the same shape and type.
3	zeros_like()	Return an array of zeros with the same shape and type
4	full_like()	Return a full array with the same shape and type
5	asarray()	Convert the input to an array.
6	geomspace()	Return evenly spaced numbers on a log scale.
7	copy()	Returns a copy of the given object

1. Using the NumPy functions

Sr No.	Function	Description
8	diag()	a diagonal array
9	frombuffer()	buffer as a 1-D array
10	fromfile()	Construct an array from text or binary file
11	bmat()	Build a matrix object from a string, nested sequence, or array
12	mat()	Interpret the input as a matrix
13	vander()	Generate a Vandermonde matrix
14	triu()	Upper triangle of array

1. Using the NumPy functions

Sr No.	Function	Description
15	tril()	Lower triangle of array
16	tri()	An array with ones at & below the given diagonal and zeros elsewhere
17	diagflat()	two-dimensional array with the flattened input as a diagonal
18	fromfunction()	executing a function over each coordinate
19	logspace()	Return numbers spaced evenly on a log scale
20	meshgrid()	Return coordinate matrices from coordinate vectors

2. Conversion from Python structure like lists

```
import numpy as np
array=np.array([4,5,6])
print(array)
list=[4,5,6]
print(list)
```

```
[4 5 6]
[4, 5, 6]
```

Working with Ndarray

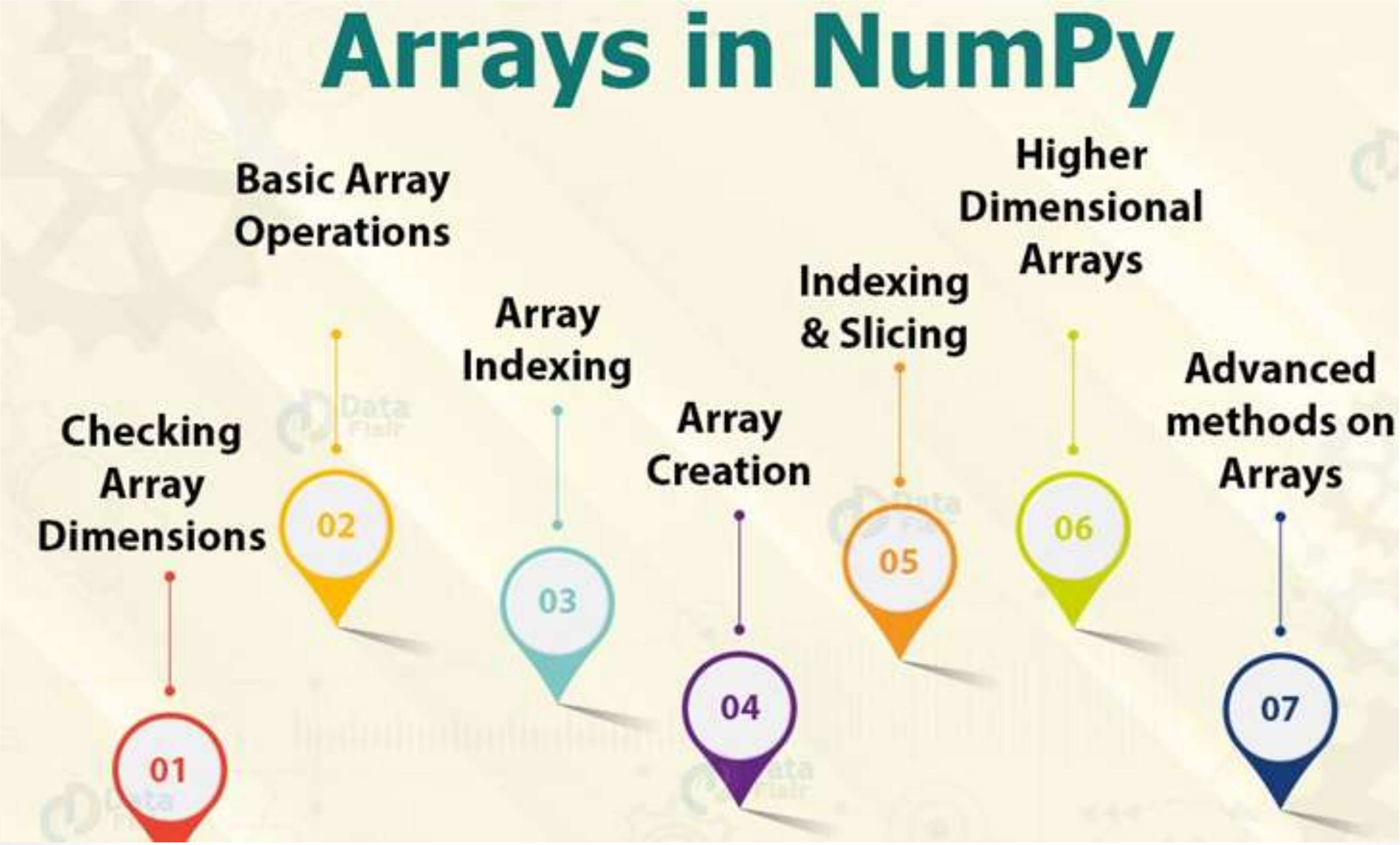
- **np.ndarray(shape, type)**
  - **Creates an array of the given shape with random numbers.**
- **np.array(array_object)**
  - **Creates an array of the given shape from the list or tuple.**
- **np.zeros(shape)**
  - **Creates an array of the given shape with all zeros.**
- **np.ones(shape)**
  - **Creates an array of the given shape with all ones.**
- **np.full(shape,array_object, dtype)**
  - **Creates an array of the given shape with complex numbers.**
- **np.arange(range)**
  - **Creates an array with the specified range.**

NumPy Basic Array Operations

There is a vast range of built-in operations that we can perform on these arrays.

1. **ndim** – It returns the dimensions of the array.
2. **itemsize** – It calculates the byte size of each element.
3. **dtype** – It can determine the data type of the element.
4. **reshape** – It provides a new view.
5. **slicing** – It extracts a particular set of elements.
6. **linspace** – Returns evenly spaced elements.
7. max/min , sum, sqrt
8. **ravel** – It converts the array into a single line.

Arrays in NumPy



Checking Array Dimensions in NumPy

```
import numpy as np
a = np.array(10)
b = np.array([1,1,1,1])
c = np.array([[1, 1, 1], [2,2,2]])
d = np.array([[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]])
print(a.ndim)  #0
print(b.ndim)  #1
print(c.ndim)  #2
print(d.ndim) #3
```

Higher Dimensional Arrays in NumPy

```
import numpy as np
arr = np.array([1, 1, 1, 1, 1], ndmin=10)
print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[[[[[[1 1 1 1 1]]]]]]]]]]
number of dimensions : 10
```

Indexing and Slicing in NumPy

Indexing & Slicing

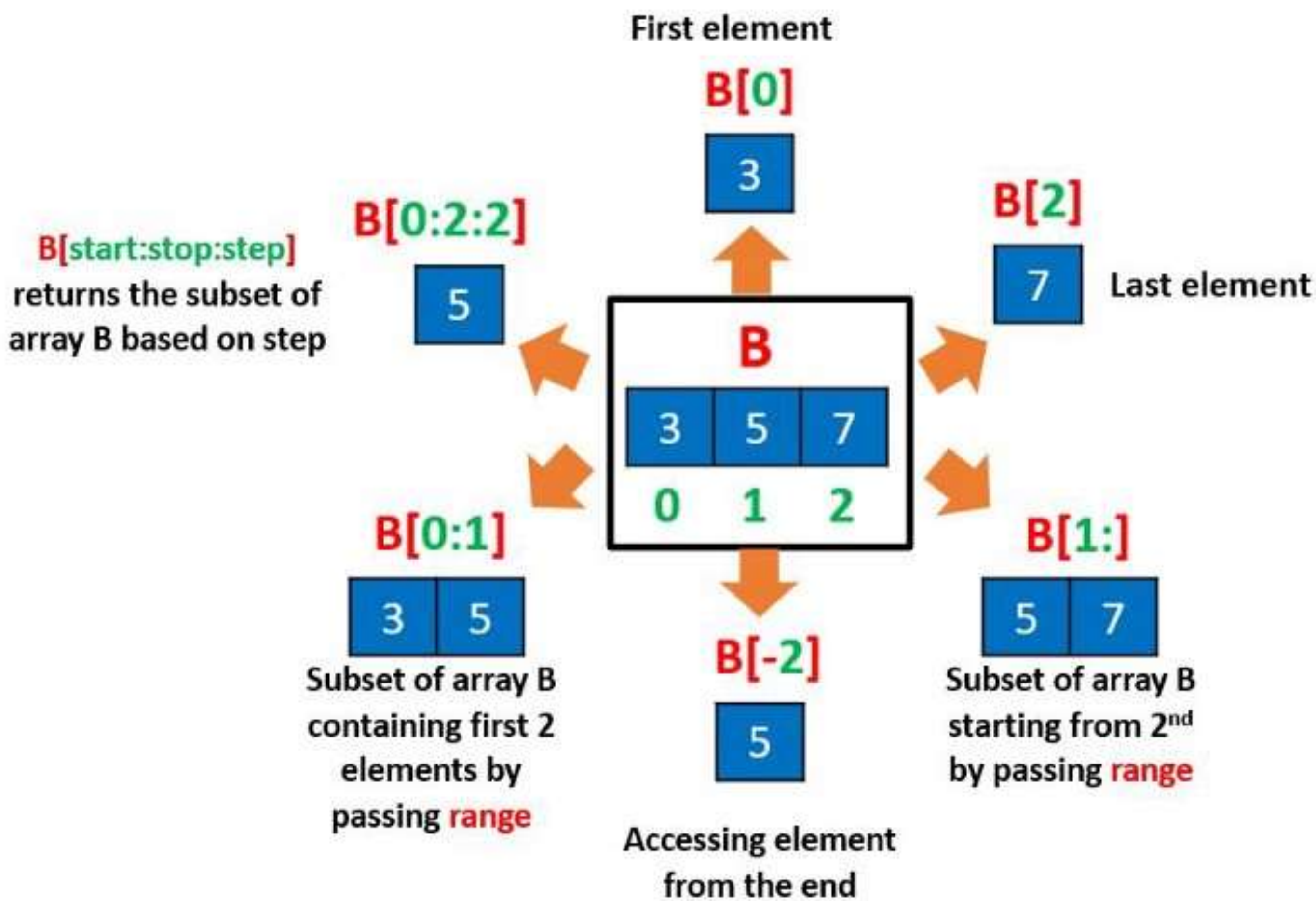
Indexing

```
import numpy as np
arr=([1,2,5,6,7])
print(arr[3]) #6
```

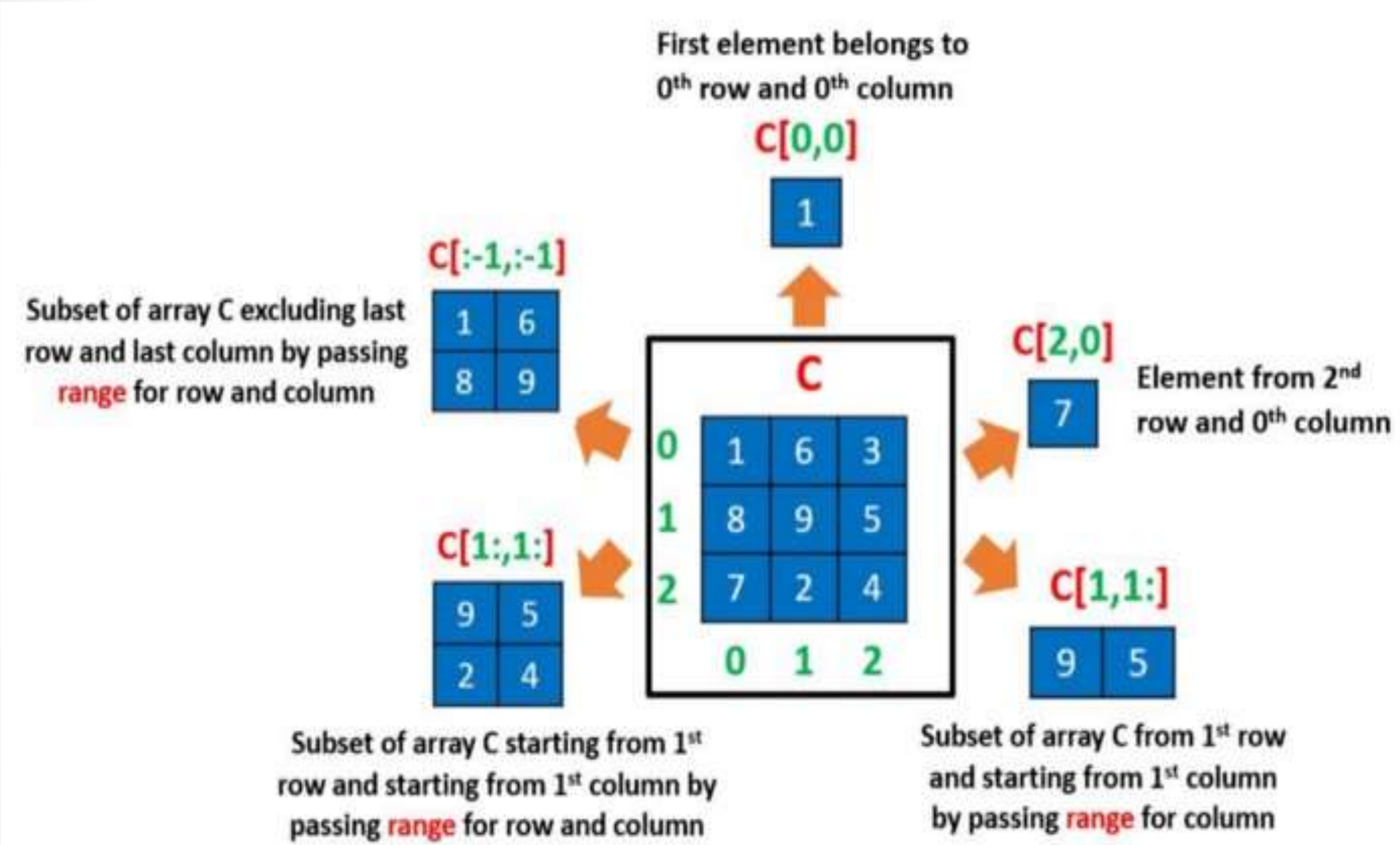
Slicing

```
import numpy as np
arr=([1,2,5,6,7])
print(arr[2:5])  #[5, 6, 7]
```

Indexing and Slicing



Indexing and Slicing in 2-D



Copying Arrays

Copy from one array to another

- **Method 1:** Using np.empty_like() function
- **Method 2:** Using np.copy() function
- **Method 3:** Using **Assignment** Operator

Using np.empty_like( )

- This function returns a new array with the same shape and type as a given array.

  Syntax:

  - **numpy.empty_like(a, dtype = None, order = ‘K’, subok = True)**

Using np.empty_like( )

- import numpy as np
- ary=np.array([13,99,100,34,65,11,66,81,632,44])
- print("Original array: ")
- # printing the Numpy array
- print(ary)
- # Creating an empty Numpy array similar to ary
- copy=np.empty_like(ary)
- # Now assign ary to copy
- copy=ary
- print("\nCopy of the given array: ")
- # printing the copied array
- print(copy)

Using np.empty_like( )

```
import numpy as np

# Creating a numpy array using np.array()
ary = np.array([13, 99, 100, 34, 65, 11,
                66, 81, 632, 44])

print("Original array: ")

# printing the Numpy array
print(ary)

# Creating an empty Numpy array similar
# to ary
copy = np.empty_like(ary)

# Now assign ary to copy
copy = ary

print("\nCopy of the given array: ")

# printing the copied array
print(copy)
```

```
Original array:
[ 13  99 100  34  65  11  66  81 632  44]

Copy of the given array:
[ 13  99 100  34  65  11  66  81 632  44]
```

Using np.copy() function

- This function returns an array copy of the given object.
  Syntax :
  - numpy.copy(a, order='K', subok=False)

```
# importing Numpy package
import numpy as np
org_array = np.array([1.54, 2.99, 3.42, 4.87, 6.94, 8.21, 7.65, 10.50,
77.5])
print("Original array: ")
print(org_array)
# Now copying the org_array to copy_array using np.copy() function
copy_array = np.copy(org_array)
print("\nCopied array: ")
# printing the copied Numpy array
print(copy_array)
```

Using np.copy() function

```
# importing Numpy package
import numpy as np
org_array = np.array([1.54, 2.99, 3.42, 4.87, 6.94, 8.21, 7.65, 10.50,
77.5])
print("Original array: ")
print(org_array)
copy_array = np.copy(org_array)
print("\nCopied array: ")
# printing the copied Numpy array
print(copy_array)
```

```
Original array:
[ 1.54  2.99  3.42  4.87  6.94  8.21  7.65 10.5   77.5 ]

Copied array:
[ 1.54  2.99  3.42  4.87  6.94  8.21  7.65 10.5   77.5 ]
```

Using Assignment Operator

```
import numpy as np
org_array = np.array([[99, 22, 33],[44, 77, 66]])
# Copying org_array to copy_array  using Assignment operator
copy_array = org_array

# modifying org_array
org_array[1, 2] = 13

# checking if copy_array has remained the same

# printing original array
print('Original Array: \n', org_array)

# printing copied array
print('\nCopied Array: \n', copy_array)
```

```
Original Array:
 [[99 22 33]
 [44 77 13]]

Copied Array:
 [[99 22 33]
 [44 77 13]]
```

Iterating Arrays

- Iterating means going through elements one by one.
- As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.
- If we iterate on a 1-D array it will go through each element one by one.
- Iterate on the elements of the following 1-D array:

```
import numpy as np

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

for x in arr:
    print(x)
```

Output:

```
1
2
3
```

Iterating Arrays

- **Iterating 2-D Arrays**
  - In a 2-D array it will go through all the rows.
  - If we iterate on a *n*-D array it will go through (n-1)th dimension one by one.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
  print(x)
```

Output:

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

Iterating Arrays

- To return the actual values, the scalars, we have to iterate the arrays in each dimension.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  for y in x:
    print(y)
```

```
1
2
3
4
5
6
```

Iterating Arrays

- **Iterating 3-D Arrays**
  - In a 3-D array it will go through all the 2-D arrays.

- ```
  import numpy as np

  arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

  for x in arr:
    print(x)
  ```

[[1 2 3] [4 5 6]]
[[7 8 9] [10 11 12]]
  ```

Iterating Arrays

- **Iterating 3-D Arrays**
  - To return the actual values, the scalars, we have to iterate the arrays in each dimension.

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
  for y in x:
    for z in y:
      print(z)
```

1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

Iterating Arrays Using nditer()

- The function nditer() is a helping function that can be used from very basic to very advanced iterations.

- **Iterating on Each Scalar Element**
  - In basic for loops, iterating through each scalar of an array we need to use *n* for loops which can be difficult to write for arrays with very high dimensionality.

```
import numpy as np

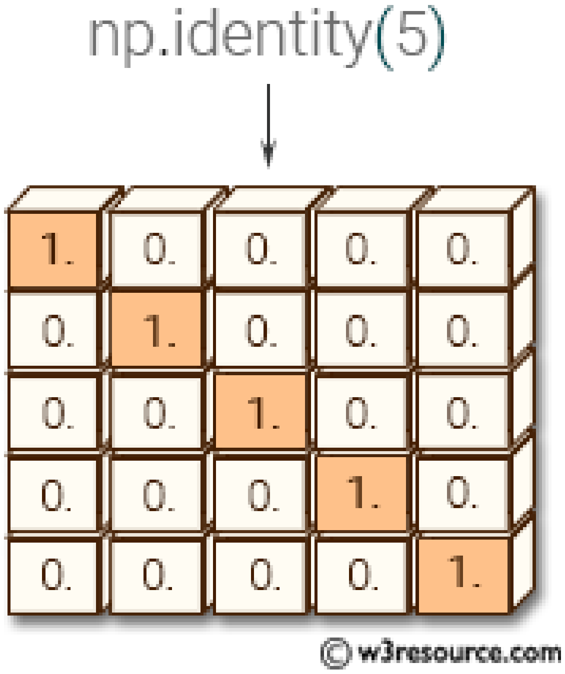
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

for x in np.nditer(arr):
  print(x)
```

```
1
2
3
4
5
6
7
8
```

Identity array

- The identity array is a square array with ones on the main diagonal.
- The identity() function return the identity array.



Identity

- **numpy.identity(n, dtype = None) :** Return a identity matrix i.e. a square matrix with ones on the main daignol

- **Parameters:**
  - **n :** [int] Dimension n x n of output array
  - **dtype :** [optional, float(by Default)] Data type of returned array

```
import numpy as np

z=np.identity(4)
print(z)
```

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

Identity array

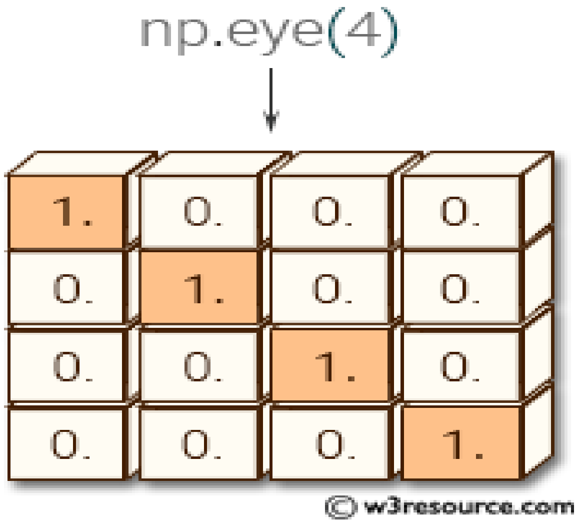
```
# 2x2 matrix with 1's on main diagonal
b = np.identity(2, dtype = float)
print("Matrix b : \n", b)
 a = np.identity(4)
print("\nMatrix a : \n", a)
```

Output:

```
Matrix b :
[[ 1. 0.]
 [ 0. 1.]]
Matrix a :
[[ 1. 0. 0. 0.]
[ 0. 1. 0. 0.]
 [ 0. 0. 1. 0.]
[ 0. 0. 0. 1.]]
```

eye( )

- **numpy.eye(R, C = None, k = 0, dtype = type <‘float’>)** **:** Return a matrix having 1’s on the diagonal and 0’s elsewhere w.r.t. **k**.
- **R :** Number of rows
  C : [optional] Number of columns; By default M = N
  k : [int, optional, 0 by default]
  Diagonal we require; k>0 means diagonal above main diagonal or vice versa.
  dtype : [optional, float(by Default)] Data type of returned array.



eye( )

```
import numpy as np
print(np.eye(4))
print(np.eye(3,2))
print(np.eye(3,3,1))
print(np.eye(3,2,-1))
```

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

Identity( ) vs eye( )

- **np.identity** returns a **square matrix** (special case of a 2D-array) which is an identity matrix with the main diagonal (i.e. 'k=0') as 1's and the other values as 0's. you can't change the diagonal k here.
- **np.eye** returns a **2D-array**, which fills the diagonal, i.e. 'k' which can be set, with 1's and rest with 0's.
- So, the main advantage depends on the requirement. If you want an identity matrix, you can go for identity right away, or can call the np.eye leaving the rest to defaults.
- But, if you need a 1's and 0's matrix of a particular shape/size or have a control over the diagonal you can go for eye method.

Identity( ) vs eye( )

```
import numpy as np
print(np.eye(3,5,1))
print(np.eye(8,4,0))
print(np.eye(8,4,-1))
print(np.eye(8,4,-2))
Print(np.identity(4)
```

Shape of an Array

- import numpy as np

  arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

  print(arr.shape)

- Output: (2,4)

Reshaping arrays

- Reshaping means changing the shape of an array.
- The shape of an array is the number of elements in each dimension.
- By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

- import numpy as np

  arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

  newarr = arr.reshape(4, 3)

  print(newarr)

- Output:
- **[[1 2 3]**
- **[4 5 6]**
- **[7 8 9]**
- **[10 11 12]]**

Reshape From 1-D to 3-D

- The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements
- import numpy as np

  arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

  newarr = arr.reshape(2, 3, 2)

  print(newarr)

Output:

[[[1 2]
[3 4]
[5 6]]

[[7 8]
[9 10]
[11 12]]]

Can we Reshape into any Shape?

- Yes, as long as the elements required for reshaping are equal in both shapes.
- We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(3, 3)

print(newarr)
```

- **Traceback (most recent call last): File "demo_numpy_array_reshape_error.py", line 5, in <module> ValueError: cannot reshape array of size 8 into shape (3,3)**

Flattening the arrays

- Flattening array means converting a multidimensional array into a 1D array.
- import numpy as np

  arr = np.array([[1, 2, 3], [4, 5, 6]])

  newarr = arr.reshape(-1)

  print(newarr)
- Output: [1 2 3 4 5 6]
- There are a lot of functions for changing the shapes of arrays in numpy **flatten**, **ravel** and also for rearranging the elements **rot90**, **flip**, **fliplr**, **flipud** etc. These fall under Intermediate to Advanced section of numpy.

Introduction to Pandas

Pandas

- **Pandas is a popular open-source data manipulation and analysis library for Python.**
- **It provides easy-to-use data structures like DataFrame and Series, which are designed to make working with structured data fast, easy, and expressive.**
- **Pandas are widely used in data science, machine learning, and data analysis for tasks such as data cleaning, transformation, and exploration.**

Series

- A Pandas Series is a one-dimensional array-like object that can hold data of any type (integer, float, string, etc.).
- It is labelled, meaning each element has a unique identifier called an index.
- Series is defined as a column in a spreadsheet or a single column of a database table.
- Series are a fundamental data structure in Pandas and are commonly used for data manipulation and analysis tasks.
- They can be created from lists, arrays, dictionaries, and existing Series objects.
- Series are also a building block for the more complex Pandas DataFrame, which is a two-dimensional table-like structure consisting of multiple Series objects.

Series

```
import pandas as pd

# Initializing a Series from a list
data = [1, 2, 3, 4, 5]
series_from_list = pd.Series(data)
print(series_from_list)

# Initializing a Series from a dictionary
data = {'a': 1, 'b': 2, 'c': 3}
series_from_dict = pd.Series(data)
print(series_from_dict)

# Initializing a Series with custom index
data = [1, 2, 3, 4, 5]
index = ['a', 'b', 'c', 'd', 'e']
series_custom_index = pd.Series(data, index=index)
print(series_custom_index)
```

Output

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
a    1
b    2
c    3
dtype: int64
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

Series - Indexing

- **Each element in a Series has a corresponding index, which can be used to access or manipulate the data.**

```
print(series_from_list[0])
print(series_from_dict['b’])
```

Output

1

2

Series – Vectorized Operations

- **Series supports vectorized operations, allowing you to perform arithmetic operations on the entire series efficiently.**

```
series_a = pd.Series([1, 2, 3])
series_b = pd.Series([4, 5, 6])
sum_series = series_a + series_b
print(sum_series)
```

Output

```
0    5
1    7
2    9
dtype: int64
```

Series – Alignment

- **When performing operations between two Series objects, Pandas automatically aligns the data based on the index labels.**

```
series_a = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
series_b = pd.Series([4, 5, 6], index=['b', 'c', 'd'])
sum_series = series_a + series_b
print(sum_series)
```

Output

```
a    NaN
b    6.0
c    8.0
d    NaN
dtype: float64
```

Series – NaN Handling

- **Missing values, represented by NaN (Not a Number), can be handled gracefully in Series operations.**

```
series_a = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
series_b = pd.Series([4, 5], index=['b', 'c'])
sum_series = series_a + series_b
print(sum_series)
```

Output

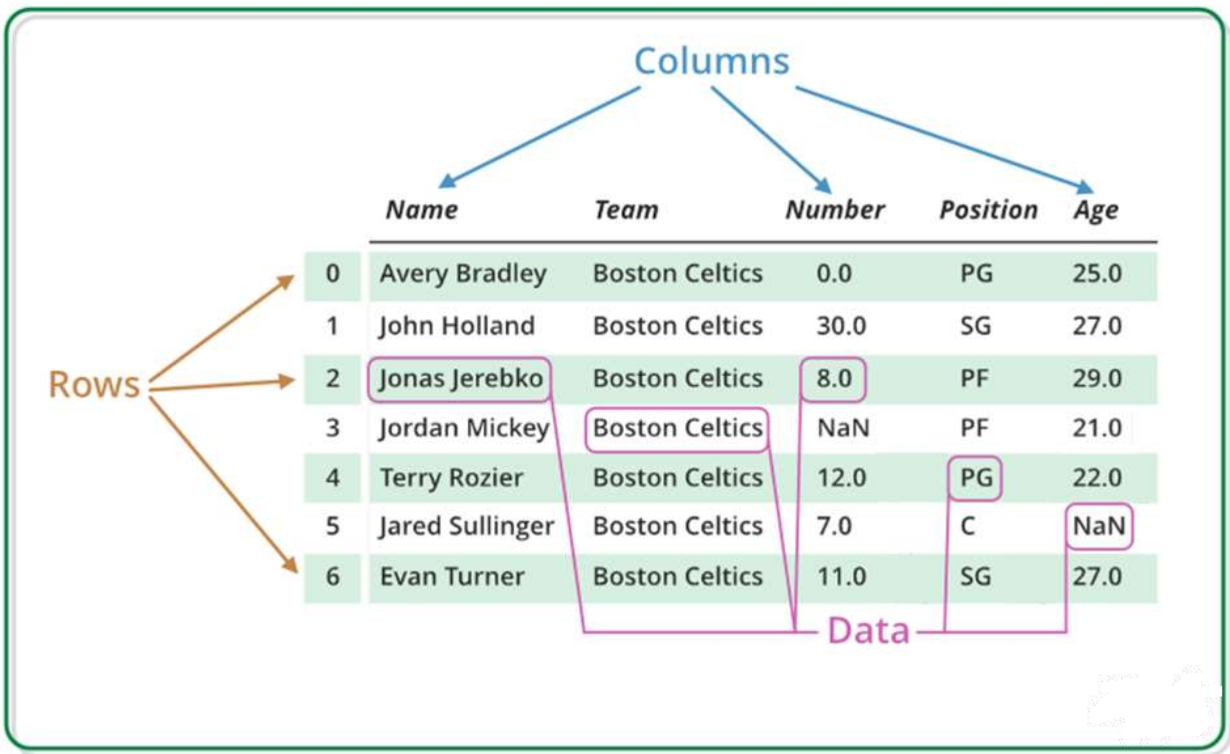
```
a    NaN
b    6.0
c    8.0
dtype: float64
```

DataFrame

- A Pandas DataFrame is a two-dimensional, tabular data structure with rows and columns.
- It is similar to a spreadsheet or a table in a relational database.
- The DataFrame has three main components:
  - data, which is stored in rows and columns;
  - rows, which are labeled by an index;
  - columns, which are labeled and contain the actual data.

DataFrame

- **The DataFrame has three main components:**
  - **data, which is stored in rows and columns;**
  - **rows, which are labeled by an index;**
  - **columns, which are labeled and contain the actual data.**



DataFrames

```
import pandas as pd

# Initializing a DataFrame from a dictionary
data = {'Name': ['John', 'Alice', 'Bob'],
      'Age': [25, 30, 35],
      'City': ['New York', 'Los Angeles', 'Chicago']}
df = pd.DataFrame(data)
print(df)

# Initializing a DataFrame from a list of lists
data = [['John', 25, 'New York'],
      ['Alice', 30, 'Los Angeles'],
      ['Bob', 35, 'Chicago']]
columns = ['Name', 'Age', 'City']
df = pd.DataFrame(data, columns=columns)
print(df)
```

```
    Name  Age         City
0   John   25     New York
1  Alice   30  Los Angeles
2    Bob   35      Chicago
    Name  Age         City
0   John   25     New York
1  Alice   30  Los Angeles
2    Bob   35      Chicago
```

DataFrames - Indexing

- DataFrame provides flexible indexing options, allowing access to rows, columns, or individual elements based on labels or integer positions.

```python
# Accessing a column
print(df['Name'])

# Accessing a row by label
print(df.loc[0])

# Accessing a row by integer position
print(df.iloc[0])

# Accessing an individual element
print(df.at[0, 'Name'])
```

```
0    John
1   Alice
2     Bob
Name: Name, dtype: object
Name       John
Age          25
City   New York
Name: 0, dtype: object
Name       John
Age          25
City   New York
Name: 0, dtype: object
John
```

DataFrame – Column Operations

- **Columns in a DataFrame are Series objects, enabling various operations such as arithmetic operations, filtering, and sorting.**

```
# Adding a new column
df['Salary'] = [50000, 60000, 70000]

# Filtering rows based on a condition
high_salary_employees = df[df['Salary'] &gt; 60000]
print(high_salary_employees)

# Sorting DataFrame by a column
sorted_df = df.sort_values(by='Age', ascending=False)
print(sorted_df)
```

DataFrames – Column Operations

- **Columns in a DataFrame are Series objects, enabling various operations such as arithmetic operations, filtering, and sorting.**

```
# Adding a new column
df['Salary'] = [50000, 60000, 70000]

# Filtering rows based on a condition
high_salary_employees = df[df['Salary'] &gt; 60000]
print(high_salary_employees)
# Sorting DataFrame by a column
sorted_df = df.sort_values(by='Age', ascending=False)
print(sorted_df)
```

```
Name  Age    City  Salary
2  Bob   35  Chicago   70000
   Name  Age        City  Salary
2    Bob   35     Chicago   70000
1  Alice   30  Los Angeles   60000
0   John   25    New York   50000
```

DataFrames – Handling NaN

- DataFrames provide methods for handling missing or NaN values, including dropping or filling missing values.

```
# Dropping rows with missing values
df.dropna()
print(df)

# Filling missing values with a specified value
df.fillna(0)
print(df)
```

```
    Name  Age         City  Salary
0   John   25     New York   50000
1  Alice   30  Los Angeles   60000
2    Bob   35      Chicago   70000
    Name  Age         City  Salary
0   John   25     New York   50000
1  Alice   30  Los Angeles   60000
2    Bob   35      Chicago   70000
```

DataFrames – Grouping and Aggregation

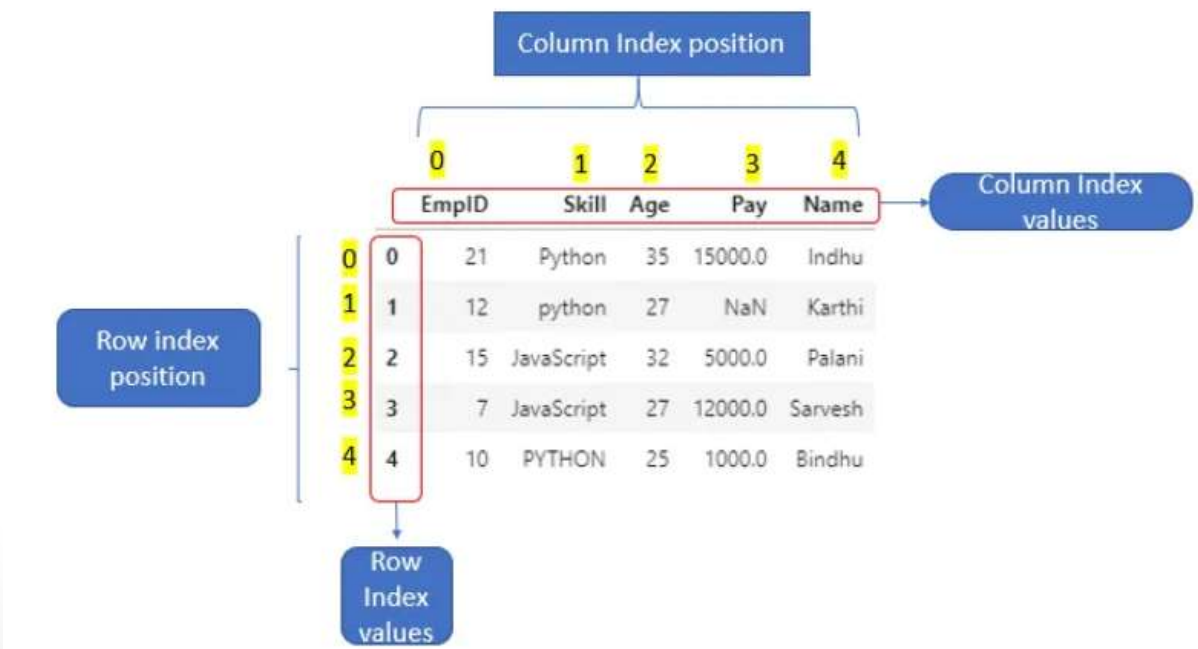
- **DataFrames support group-by operations for summarizing data and applying aggregation functions.**

```
# Grouping by a column and calculating mean
avg_age_by_city = df.groupby('City')['Age'].mean()
print(avg_age_by_city)
```

```
City
Chicago       35.0
Los Angeles   30.0
New York      25.0
Name: Age, dtype: float64
```

Indexing

- **I**ndexing is a fundamental operation for accessing and manipulating data efficiently.
- It involves assigning unique identifiers or labels to data elements, allowing for rapid retrieval and modification.



Indexing - Features

- **Immutability:** Once created, an index cannot be modified.
- **Alignment:** Index objects are used to align data structures like Series and DataFrames.
- **Flexibility:** Pandas offers various index types, including integer-based, datetime, and custom indices.

Index - Creation

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
df = pd.DataFrame(data, index=['A', 'B', 'C'])
```

	Name	Age
A	Alice	25
B	Bob	30
C	Charlie	35

Re-index

- **Reindexing is the process of creating a new DataFrame or Series with a different index.**
- **The reindex() method is used for this purpose.**

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
      'Age': [25, 30, 35]}
df = pd.DataFrame(data, index=['A', 'B', 'C'])
# Create a new index
new_index = ['A', 'B', 'D', 'E']

# Reindex the DataFrame
df_reindexed = df.reindex(new_index)

df_reindexed
```

	Name	Age
A	Alice	25.0
B	Bob	30.0
D	NaN	NaN
E	NaN	NaN

Drop Entry

- **Dropping entries in data science refers to removing specific rows or columns from a dataset.**
- **This is a common operation in data cleaning and preprocessing to handle missing values, outliers, or irrelevant information.**

Drop Entry

```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)
df
# Drop column
newdf = df.drop("Age", axis='columns')

newdf
```

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35

	Name
0	Alice
1	Bob
2	Charlie

Selecting Entries – Selecting by Position

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
      'Age': [25, 30, 35],
      'City': ['New York', 'Los Angeles',  'Chicago']}
df = pd.DataFrame(data)
# Select the second row
df.iloc[1]
```

Created DataFrame

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

Selecting data by Position

	1
Name	Bob
Age	30
City	Los Angeles

dtype: object

Selecting Entries – Selecting by Condition

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'Los Angeles', 'Chicago']}
df = pd.DataFrame(data)
# Select rows where Age is greater than 30
df[df['Age'] > 30]
```

Created DataFrame

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

Selecting data by Condition

	Name	Age	City
2	Charlie	35	Chicago

Data Alignment

- **Data alignment is intrinsic, which means that it's inherent to the operations you perform.**
- **Align data in them by their labels and not by their position**
- **align( ) function is used to align**
  - **Used to align two data objects with each other according to their labels.**
  - **Used on both Series and DataFrame objects**
  - **Returns a new object of the same type with labels compared and aligned.**

Data Alignment

```
import pandas as pd
import numpy as np
df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9] })
df2 = pd.DataFrame({
    'A': [10, 11],
    'B': [12, 13],
    'D': [14, 15] })
```

	A	B	C
0	1	4	7
1	2	5	8
2	3	6	9

	A	B	D
0	10	12	14
1	11	13	15

Data Alignment

```
import pandas as pd
import numpy as np
df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9] })
df2 = pd.DataFrame({
    'A': [10, 11],
    'B': [12, 13],
    'D': [14, 15] })
df1_aligned, df2_aligned = df1.align(df2, fill_value=np.nan)
```

	A	B	C	D
0	1	4	7	NaN
1	2	5	8	NaN
2	3	6	9	NaN

	A	B	C	D
0	10.0	12.0	NaN	14.0
1	11.0	13.0	NaN	15.0
2	NaN	NaN	NaN	NaN

Rank

- **Ranking is assigning ranks or positions to data elements based on their values.**
- **Rank is returned based on position after sorting.**
- **Used when analyzing data with repetitive values or when you need to identify the top or bottom entries.**

Rank

```
import numpy as np

import pandas as pd

df = pd.DataFrame(data={'Animal': ['fox', 'Kangaroo’,

'deer','spider', 'snake’],

'Number_legs': [4, 2, 4, 8, np.nan]})

df
```

	Animal	Number_legs
1	Fox	4.0
2	Kangaroo	2.0
3	Deer	4.0
4	Spider	8.0
5	Snake	NaN

Rank

```
pd.DataFrame ( data = { 'Animal' : [ 'fox', 'Kangaroo', 'deer',
                'spider', 'snake' ],
      'Number_legs' : [ 4, 2, 4, 8, np.nan ] } )
```

↓

DataFrame	Animal	Number_legs
0	Fox	4.0
1	Kangaroo	2.0
2	Deer	4.0
3	Spider	8.0
4	Snake	NaN

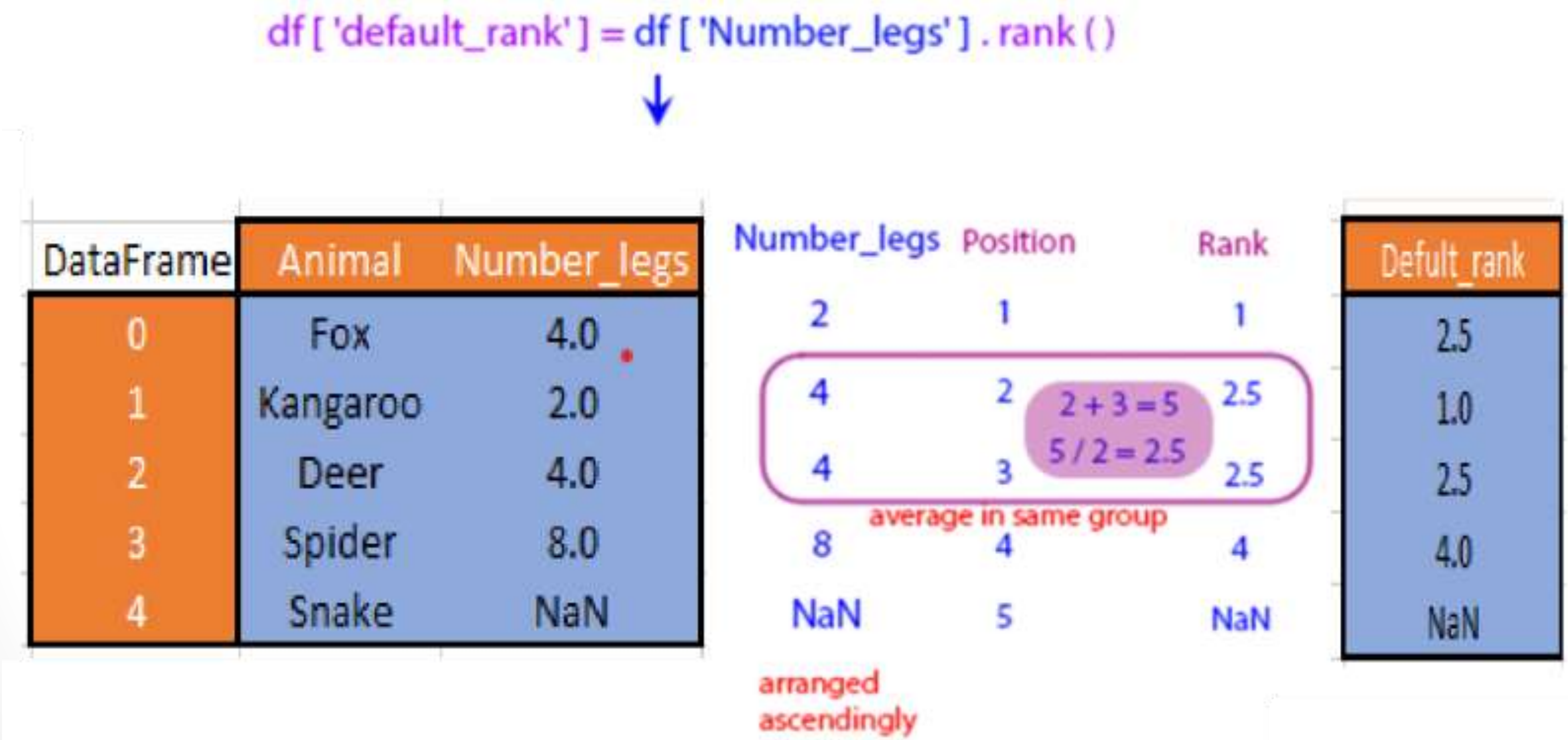
	Animal	Number_legs
1	Fox	4.0
2	Kangaroo	2.0
3	Deer	4.0
4	Spider	8.0
5	Snake	NaN

Rank

```
df['default_rank'] = df['Number_legs'].rank()

df['max_rank'] = df['Number_legs'].rank(method='max’)

df['NA_bottom’]= df['Number_legs'].rank(na_option='bottom’)

df['pct_rank'] = df['Number_legs'].rank(pct=True)

df
```

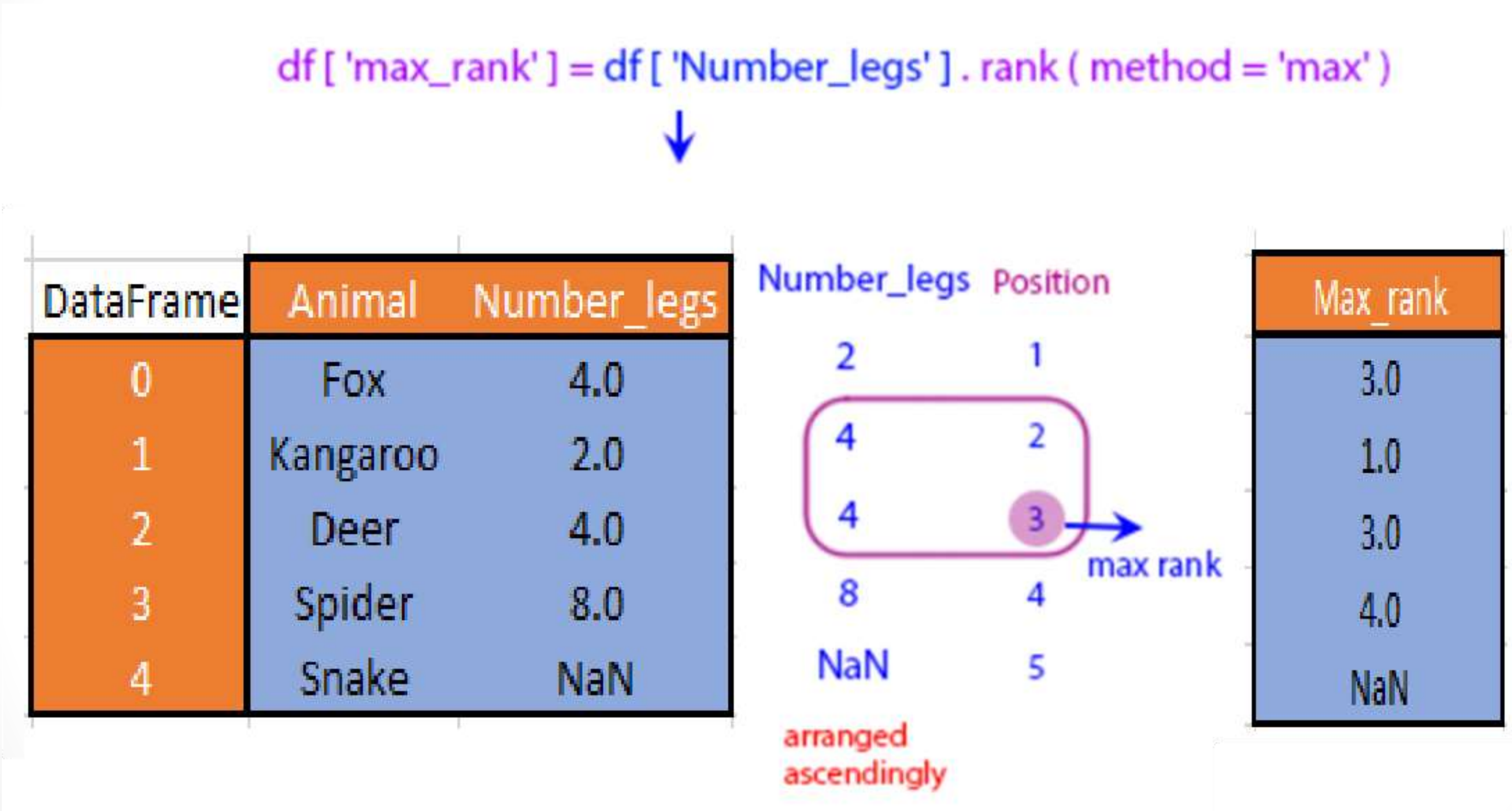
	Animal	Number_legs	default_rank	max_rank	NA_bottom	pct_rank
0	fox	4.0	2.5	3.0	2.5	0.625
1	Kangaroo	2.0	1.0	1.0	1.0	0.250
2	deer	4.0	2.5	3.0	2.5	0.625
3	spider	8.0	4.0	4.0	4.0	1.000
4	snake	NaN	NaN	NaN	5.0	NaN

Rank

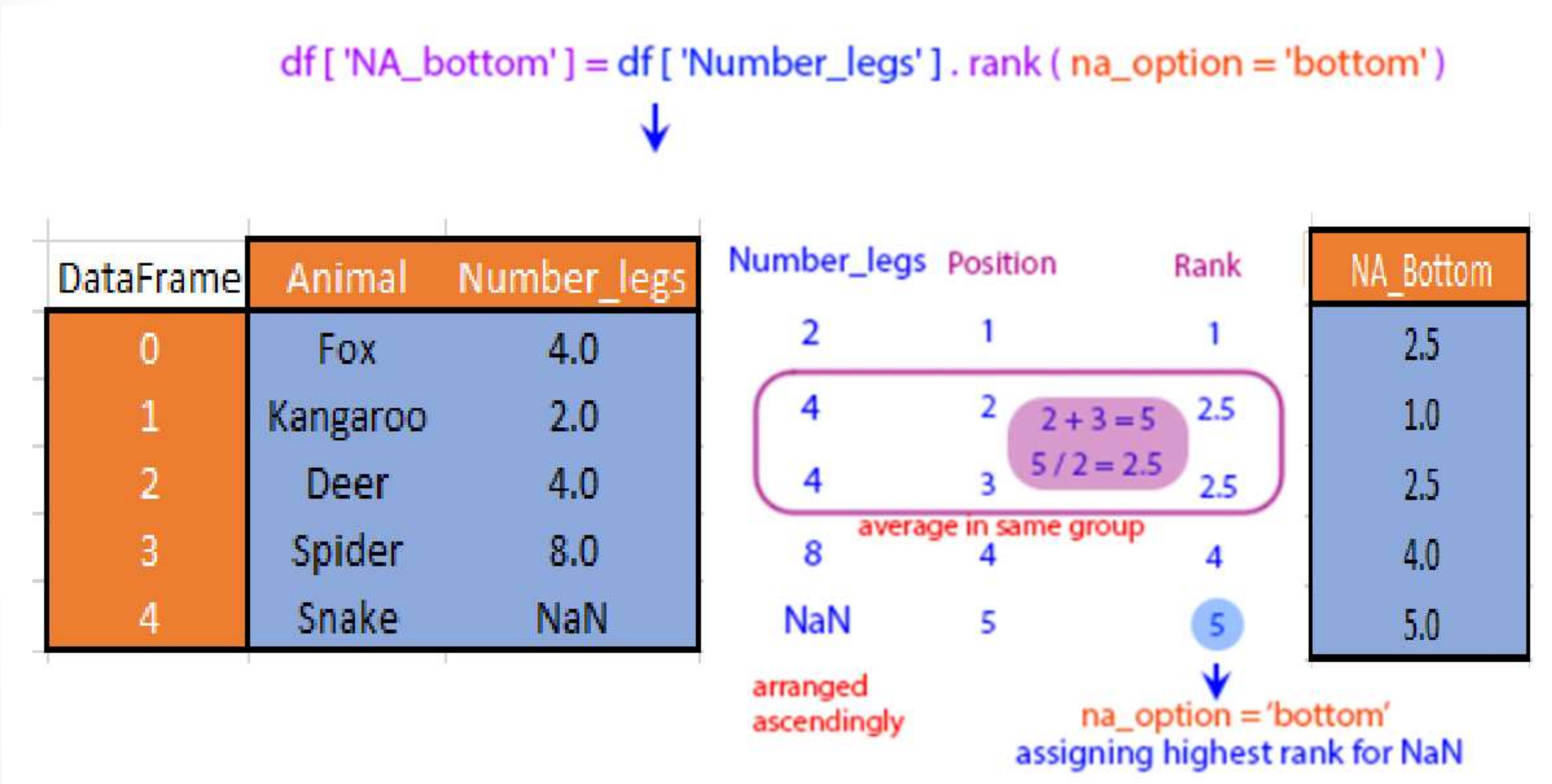


df ['default_rank'] = df ['Number_legs'] . rank ( )
DataFrame
Animal
Number_legs
0
1
2
3
4
Fox
Kangaroo
Deer
Spider
Snake
4.0
2.0
4.0
8.0
NaN
Number_legs
Position
Rank
2
1
1
4
2
2.5
4
3
2.5
2 + 3 = 5
5 / 2 = 2.5
average in same group
8
4
4
NaN
5
NaN
arranged ascendingly
Defult_rank
2.5
1.0
2.5
4.0
NaN

Rank


df ['max_rank'] = df ['Number_legs'] . rank (method = 'max')
DataFrame
Animal
Number_legs
0
1
2
3
4
Fox
Kangaroo
Deer
Spider
Snake
4.0
2.0
4.0
8.0
NaN
Number_legs
Position
2
4
4
8
NaN
1
2
3
4
5
max rank
arranged ascendingly
Max_rank
3.0
1.0
3.0
4.0
NaN

Rank


df ['NA_bottom'] = df ['Number_legs'] . rank (na_option = 'bottom')
DataFrame
Animal
Number_legs
0
1
2
3
4
Fox
Kangaroo
Deer
Spider
Snake
4.0
2.0
4.0
8.0
NaN
Number_legs
Position
Rank
2
1
1
4
2
2.5
4
3
2.5
2 + 3 = 5
5 / 2 = 2.5
average in same group
8
4
4
NaN
5
5
arranged
ascendingly
na_option = 'bottom'
assigning highest rank for NaN
NA_Bottom
2.5
1.0
2.5
4.0
5.0

Rank

```
df [ 'pct_rank' ] = df [ 'Number_legs' ] . rank (pct = True )
```

↓

DataFrame	Animal	Number_legs
0	Fox	4.0
1	Kangaroo	2.0
2	Deer	4.0
3	Spider	8.0
4	Snake	NaN

Number_legs	Position	Rank	pct_rank
2	1	1	1 / 4
4	2	2.5	2.5 / 4
4	3	2.5	2.5 / 4
8	4	4	4 / 4
NaN	5	5	NaN
arranged ascendingly			present rank / highest rank (4)

Pct_rank
0.625
0.25
0.625
1.0000
NaN

Sort

- Sort by the values along the axis
- Sort a pandas DataFrame by the values of one or more columns
- Use the ascending parameter to change the sort order
- Sort a DataFrame by its index using .sort_index()
- Organize missing data while sorting values
- Sort a DataFrame in place using inplace set to True

Sort

```
import pandas as pd
age_list = [['Afghanistan', 1952, 8425333, 'Asia'],
        ['Australia', 1957, 9712569, 'Oceania'],
        ['Brazil', 1962, 76039390, 'Americas'],
        ['China', 1957, 637408000, 'Asia'],
        ['France', 1957, 44310863, 'Europe'],
        ['India', 1952, 3.72e+08, 'Asia'],
        ['United States', 1957, 171984000, 'Americas']]
df = pd.DataFrame(age_list, columns=['Country', 'Year',
                        'Population', 'Continent'])
df
```

Sort

```
import pandas as pd
age_list = [['Afghanistan', 1952, 8425333, 'Asia'],
        ['Australia', 1957, 9712569, 'Oceania'],
        ['Brazil', 1962, 76039390, 'Americas'],
        ['China', 1957, 637408000, 'Asia'],
        ['France', 1957, 44310863, 'Europe'],
        ['India', 1952, 3.72e+08, 'Asia'],
        ['United States', 1957, 171984000, 'Americas']]
df = pd.DataFrame(age_list, columns=['Country', 'Year', 'Population', 'Continent'])
df
```

	Country	Year	Population	Continent
0	Afghanistan	1952	8425333.0	Asia
1	Australia	1957	9712569.0	Oceania
2	Brazil	1962	76039390.0	Americas
3	China	1957	637408000.0	Asia
4	France	1957	44310863.0	Europe
5	India	1952	372000000.0	Asia
6	United States	1957	171984000.0	Americas

Sort by Ascending Order

```
import pandas as pd
age_list = [['Afghanistan', 1952, 8425333, 'Asia'],
        ['Australia', 1957, 9712569, 'Oceania'],
        ['Brazil', 1962, 76039390, 'Americas'],
        ['China', 1957, 637408000, 'Asia'],
        ['France', 1957, 44310863, 'Europe'],
        ['India', 1952, 3.72e+08, 'Asia'],
        ['United States', 1957, 171984000, 'Americas']]
df = pd.DataFrame(age_list, columns=['Country', 'Year', 'Population', 'Continent'])
df.sort_values(by=['Country']) # sorting in Ascending Order
df
```

	Country	Year	Population	Continent
0	Afghanistan	1952	8425333.0	Asia
1	Australia	1957	9712569.0	Oceania
2	Brazil	1962	76039390.0	Americas
3	China	1957	637408000.0	Asia
4	France	1957	44310863.0	Europe
5	India	1952	372000000.0	Asia
6	United States	1957	171984000.0	Americas

Sort by Descending Order

```
import pandas as pd
age_list = [['Afghanistan', 1952, 8425333, 'Asia'],
    ['Australia', 1957, 9712569, 'Oceania'],
    ['Brazil', 1962, 76039390, 'Americas'],
    ['China', 1957, 637408000, 'Asia'],
    ['France', 1957, 44310863, 'Europe'],
    ['India', 1952, 3.72e+08, 'Asia'],
    ['United States', 1957, 171984000, 'Americas']]
df = pd.DataFrame(age_list, columns=['Country', 'Year', 'Population', 'Continent'])
df.sort_values(by=['Population'], ascending=False) # sorting in Descending Order
df
```

	Country	Year	Population	Continent
3	China	1957	637408000.0	Asia
5	India	1952	372000000.0	Asia
6	United States	1957	171984000.0	Americas
2	Brazil	1962	76039390.0	Americas
4	France	1957	44310863.0	Europe
1	Australia	1957	9712569.0	Oceania
0	Afghanistan	1952	8425333.0	Asia

Sort by Descending Order

```
import pandas as pd
age_list = [['Afghanistan', 1952, 8425333, 'Asia'],
        ['Australia', 1957, 9712569, 'Oceania'],
        ['Brazil', 1962, 76039390, 'Americas'],
        ['China', 1957, 637408000, 'Asia'],
        ['France', 1957, 44310863, 'Europe'],
        ['India', 1952, 3.72e+08, 'Asia'],
        ['United States', 1957, 171984000, 'Americas']]
df = pd.DataFrame(age_list, columns=['Country', 'Year', 'Population', 'Continent'])
df.sort_values(by=['Population'], ascending=False) # sorting in Descending Order
df
```

	Country	Year	Population	Continent
3	China	1957	637408000.0	Asia
5	India	1952	372000000.0	Asia
6	United States	1957	171984000.0	Americas
2	Brazil	1962	76039390.0	Americas
4	France	1957	44310863.0	Europe
1	Australia	1957	9712569.0	Oceania
0	Afghanistan	1952	8425333.0	Asia