Fundamentals of Data Science
21CSS202T

Unit I

Unit-1: INTRODUCTION TO DATA SCIENCE 9 hours

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

Introduction to Python Libraries: 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, Universal function, Linear algebra with Numpy, eigen values and eigen vectors with Numpy, Numpy Random: Data Distribution, Normal, Exponential, Binomial, Poisson, Uniform and ChiSaquare distributions.

T1: Using Numpy implement Array Indexing and slicing

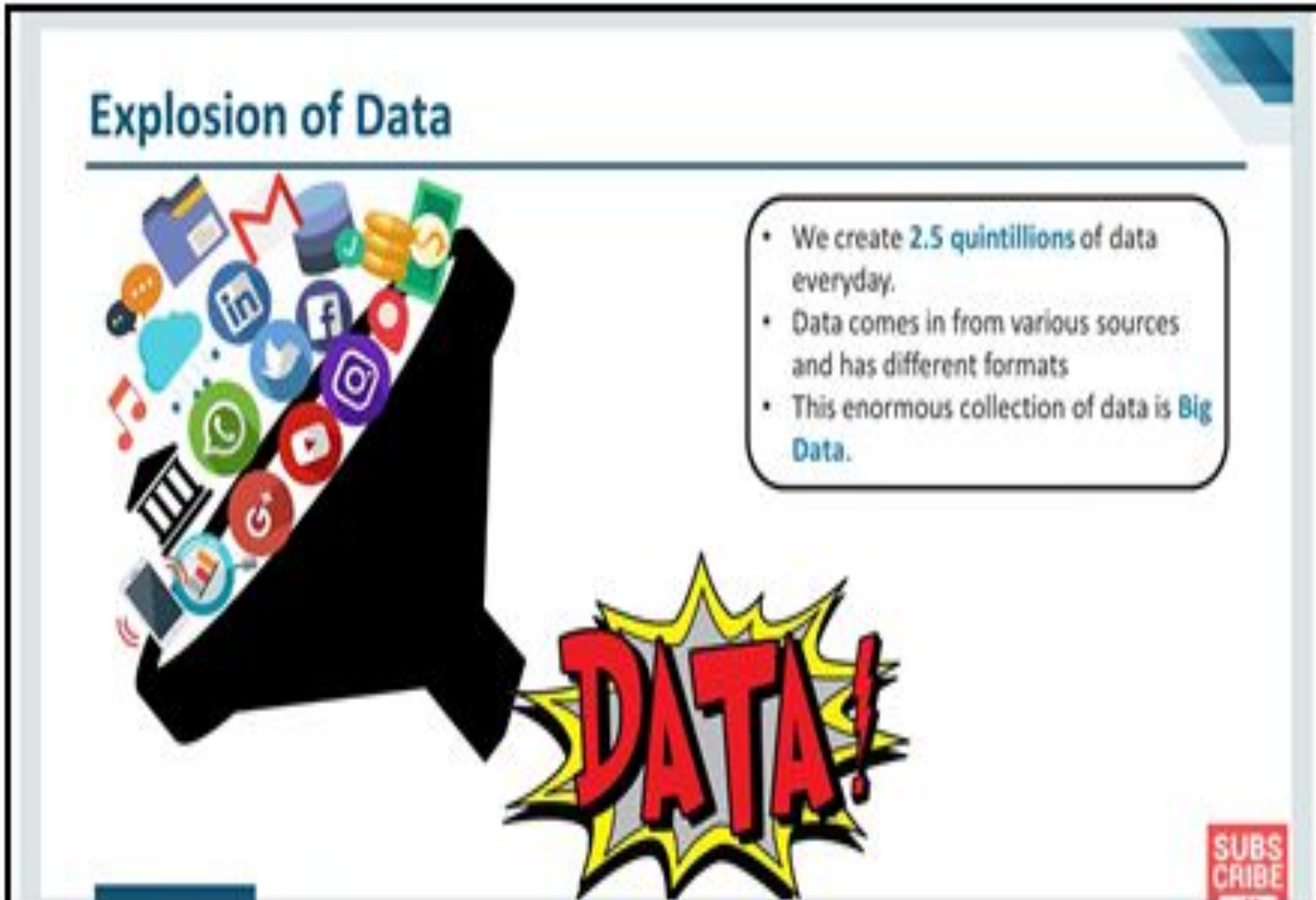
T2: Using Numpy implement Array basic operations

T3: Using Numpy implement Linear algebra and Random package

Data Science

- **Data Science is the study of data to extract meaningful insights for business.**

Why Big Data


Explosion of Data
We create 2.5 quintillions of data everyday.
Data comes in from various sources and has different formats
This enormous collection of data is Big Data.
DATA!
SUBS CRIBE

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 analyse 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?

Characteristics of Big data

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

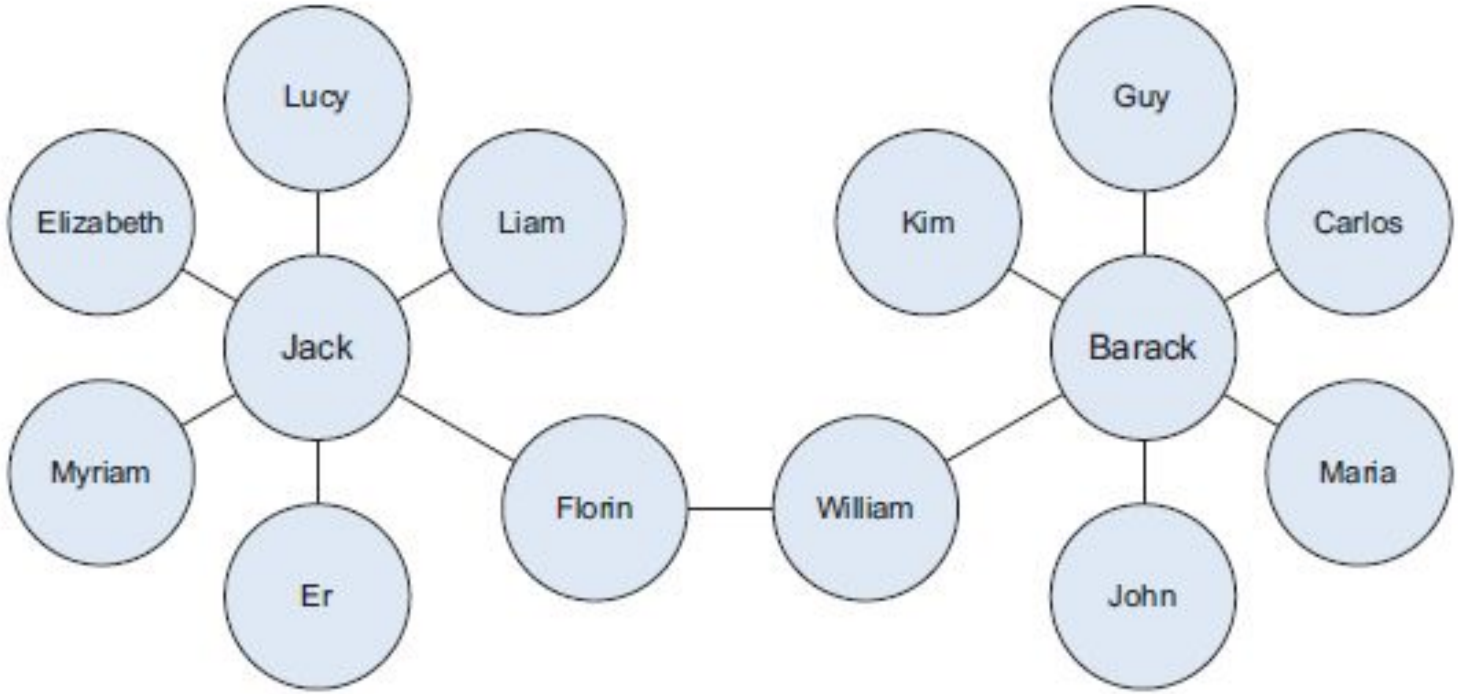


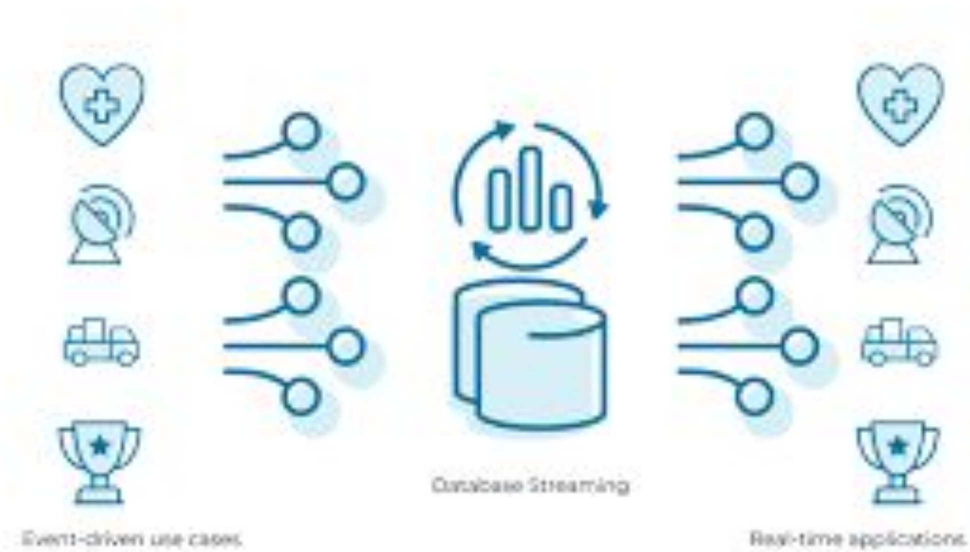
Figure 1.4 **Friends in a social network are an example of graph-based data.**

Audio, video and image

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

Streaming

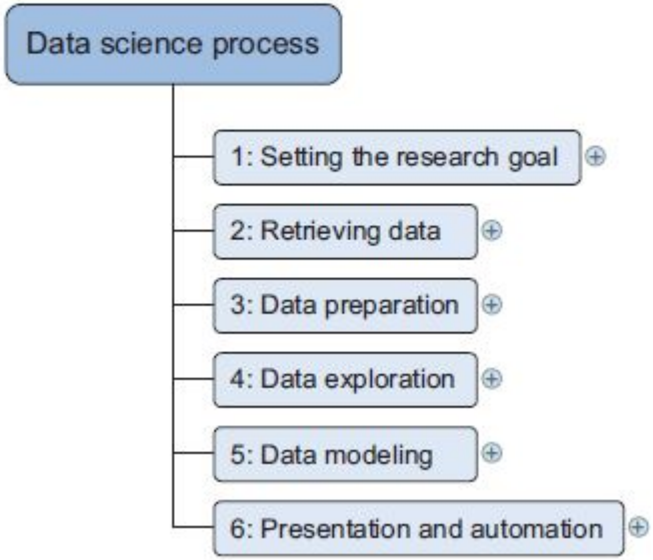
- While streaming data can take almost any of the previous forms, it has an extra property.
- The data flows into the system when an event happens instead of being loaded into a data store in a batch.



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



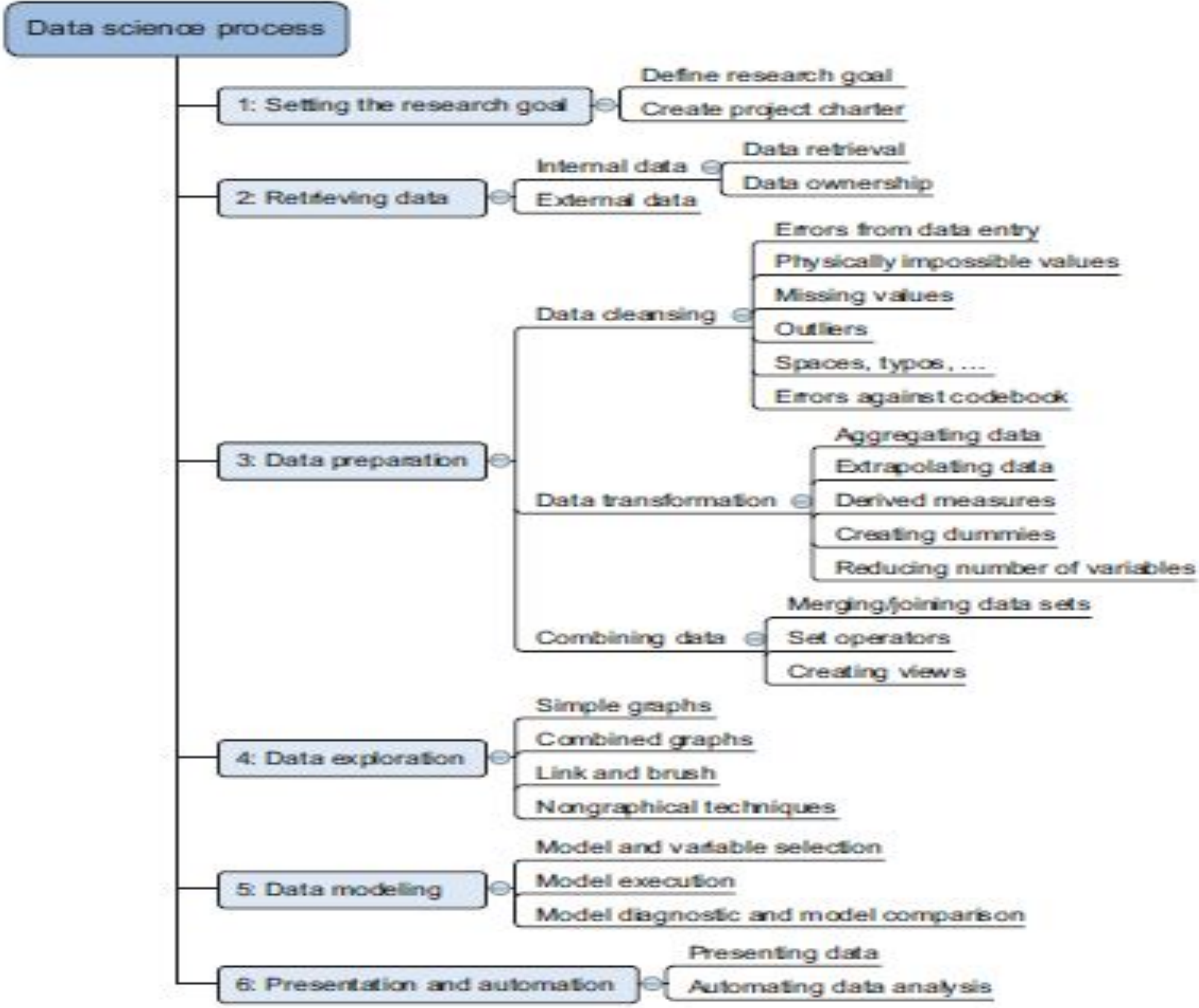


Figure 2.1 The six steps of the data science process

Setting the research goal

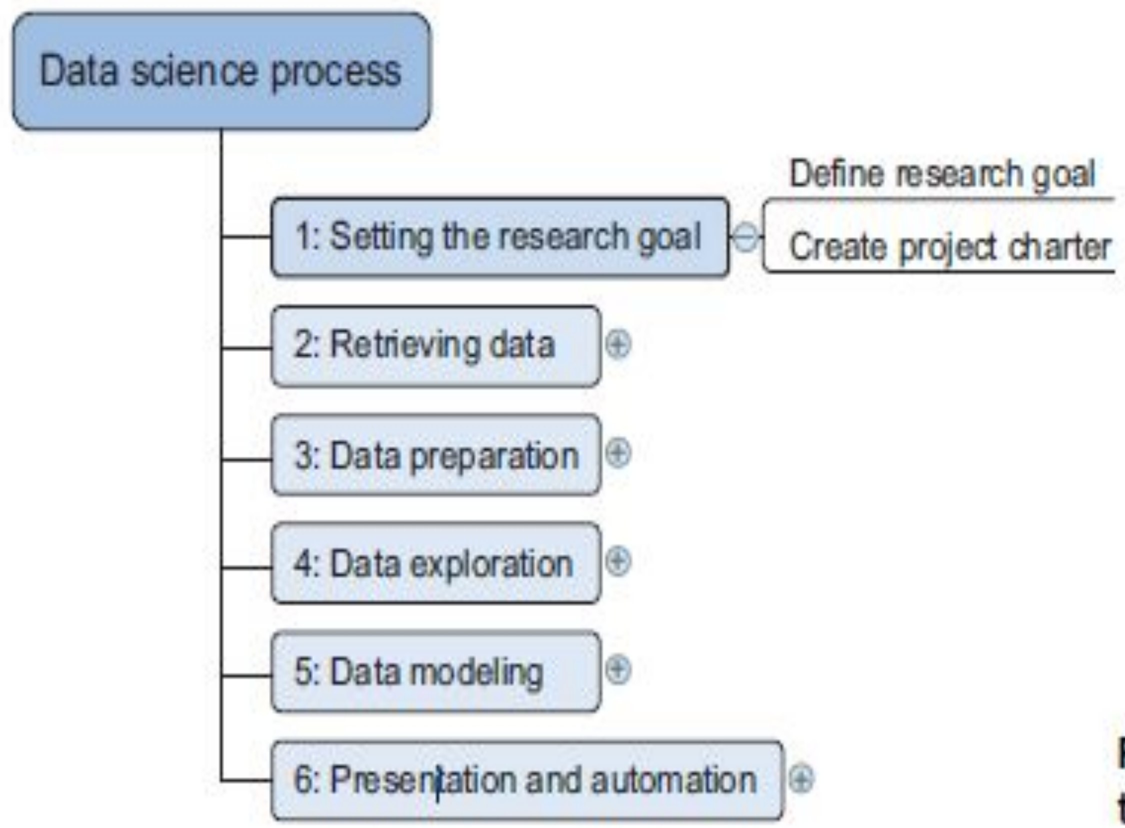


Figure 2.2 Step 1: Setting the research goal

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

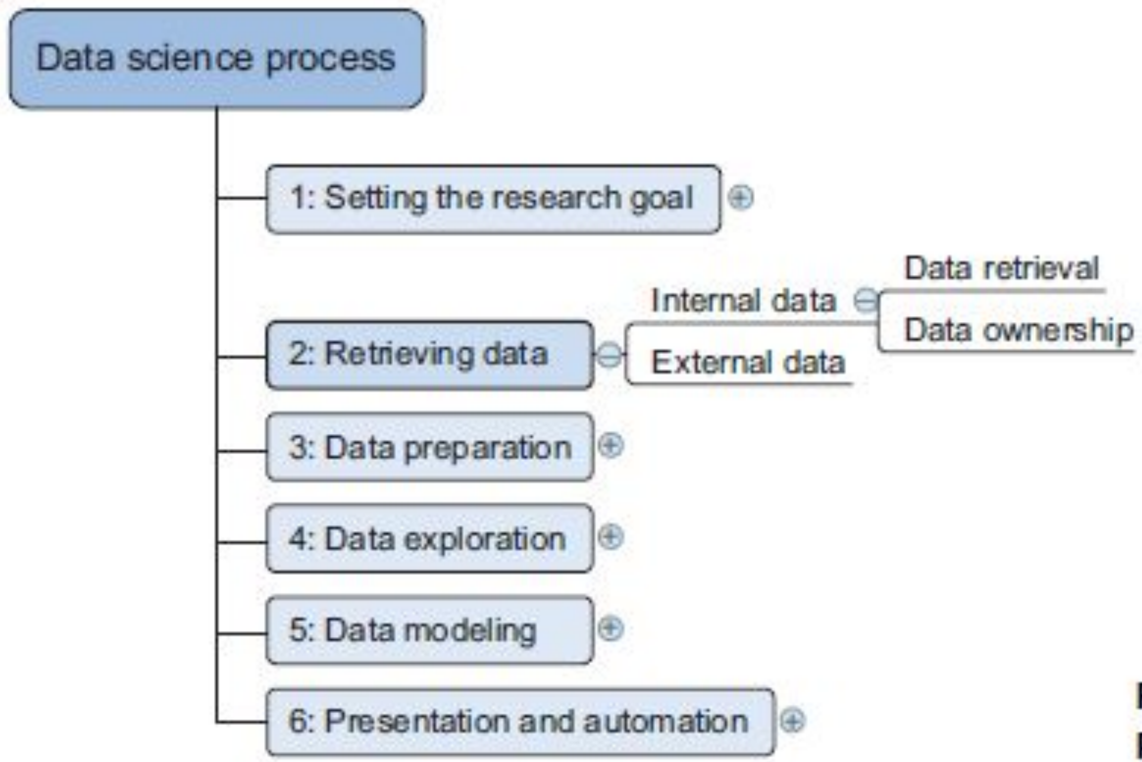


Figure 2.3 Step 2: Retrieving data

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***

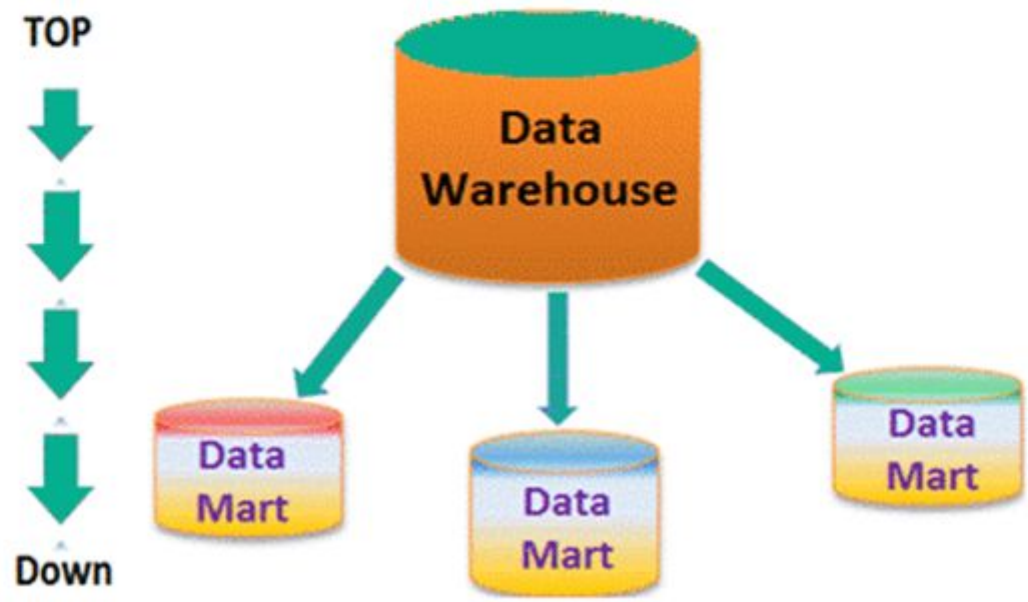
Data Lakes

- 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!

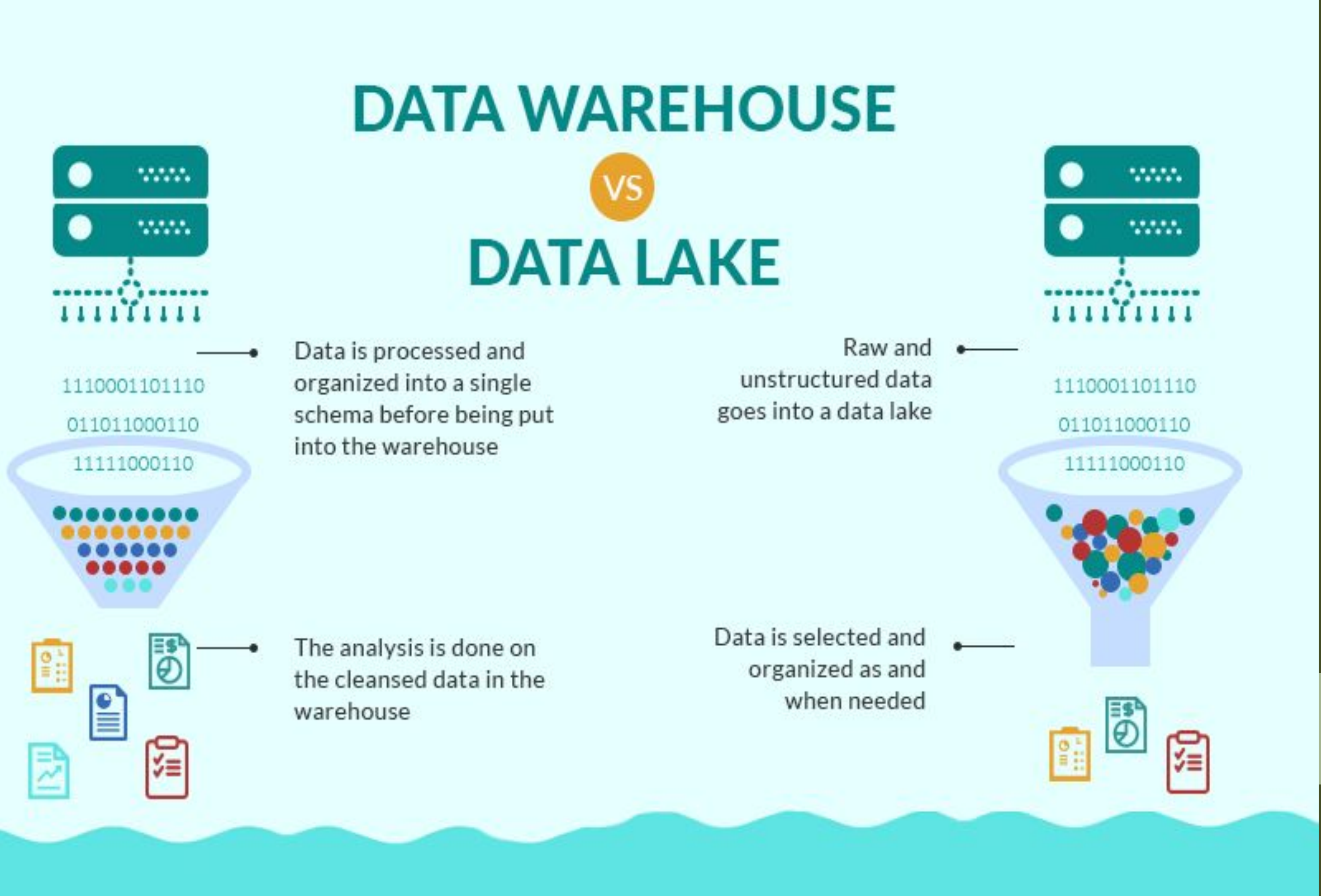
Data Mart



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
1110001101110
011011000110
11111000110
Data is processed and organized into a single schema before being put into the warehouse
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

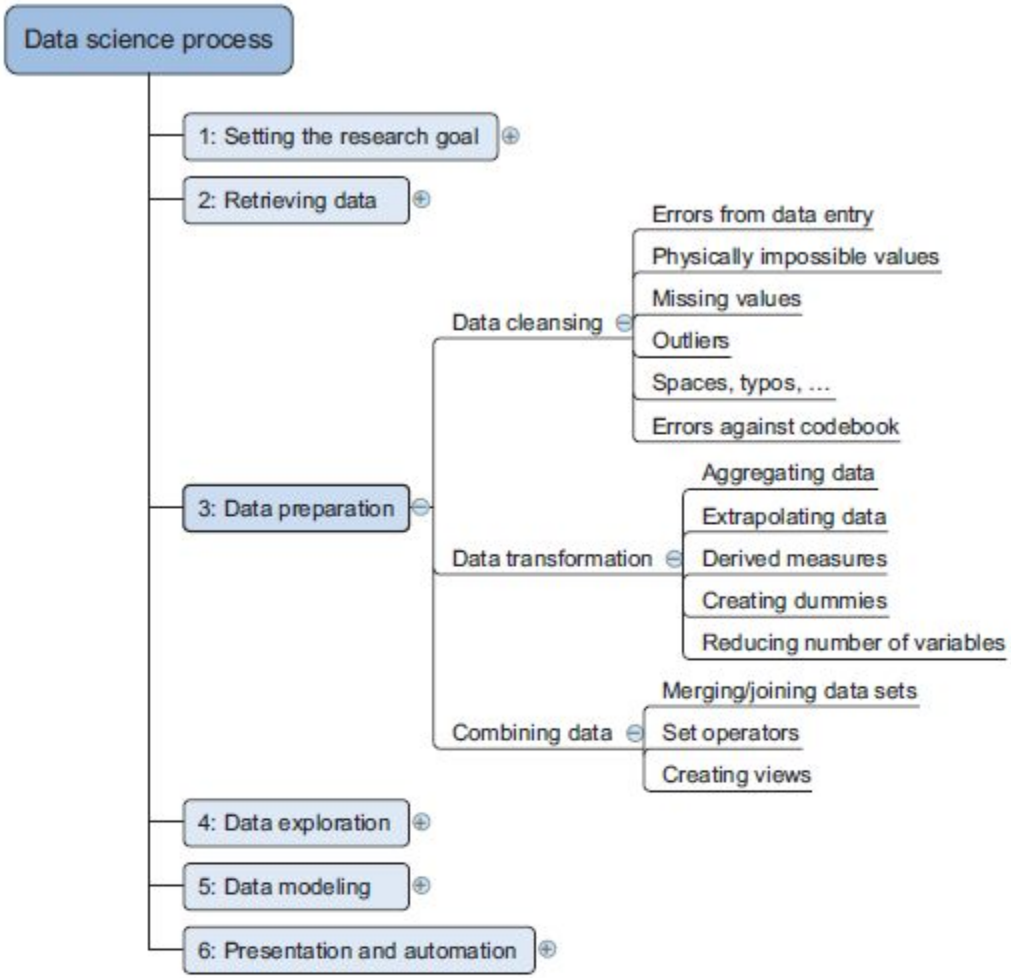
- Data lakes are a fairly new concept and experts have predicted that it might cause the death of data warehouses and data marts.
- Although with the increase of unstructured data, data lakes will become quite popular. But you will probably prefer keeping your structured data in a data warehouse.

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

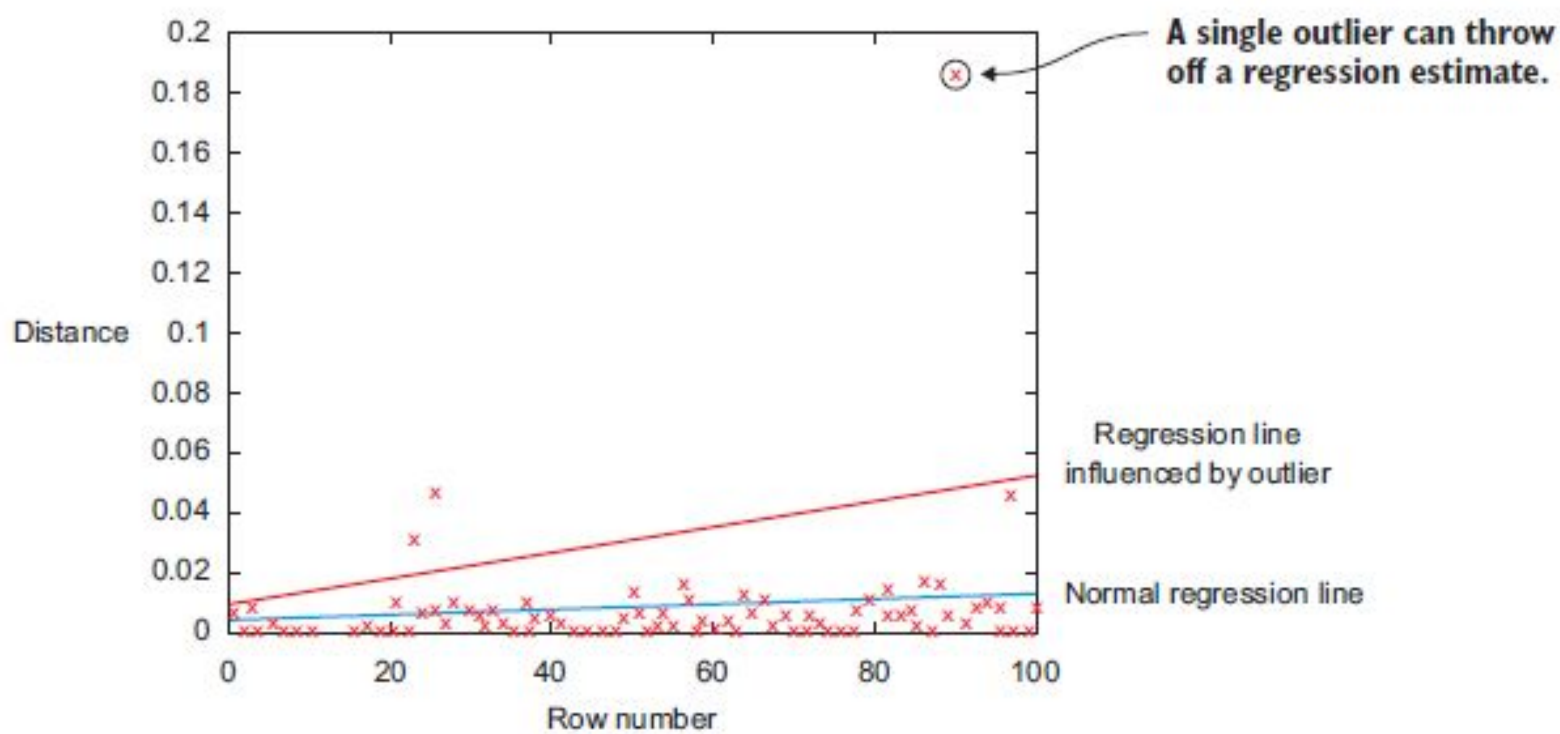


Figure 2.5 The encircled point influences the model heavily and is worth investigating because it can point to a region where you don't have enough data or might indicate an error in the data, but it also can be a valid data point.

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

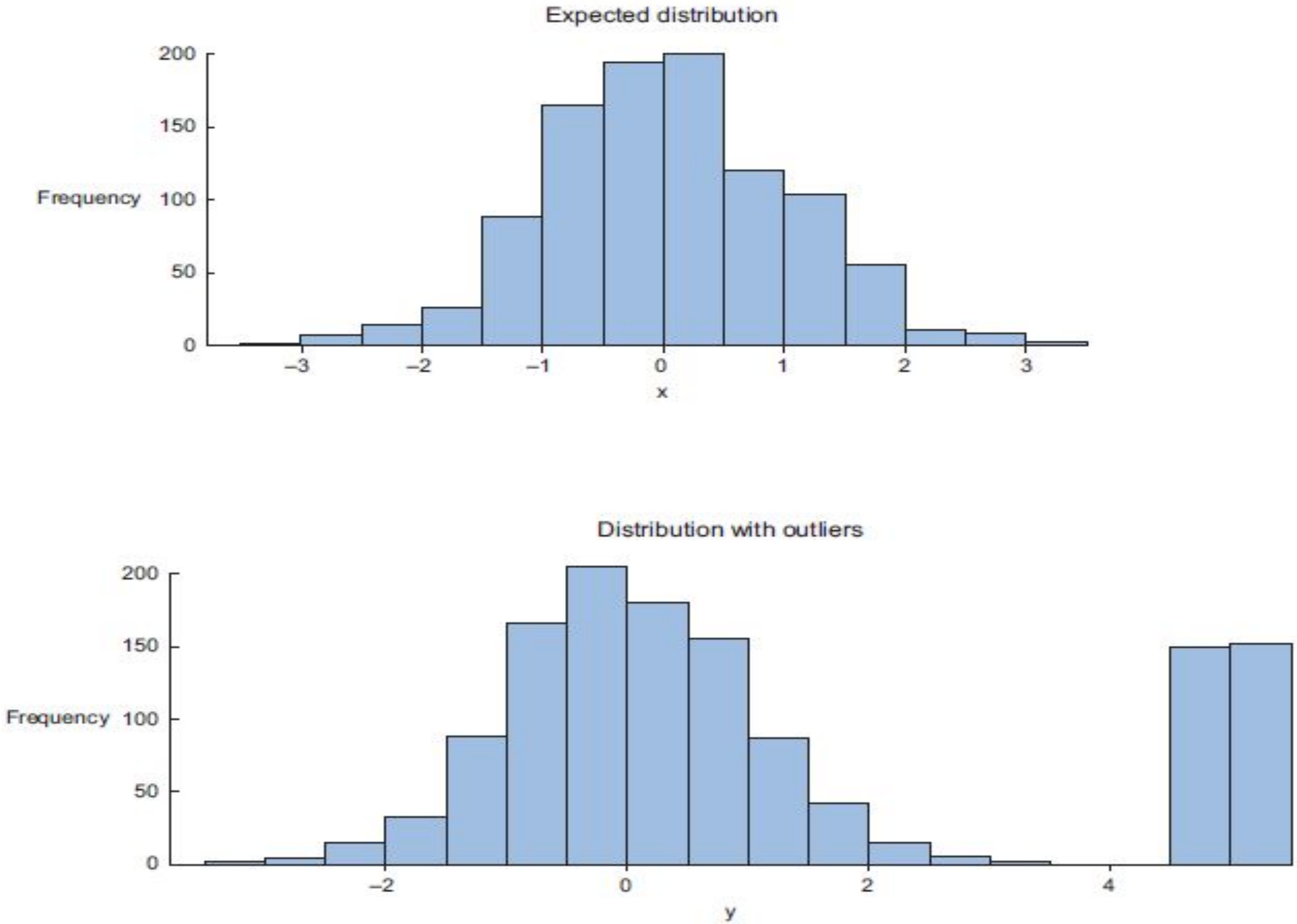


Figure 2.6 Distribution plots are helpful in detecting outliers and helping you understand the variable.

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**

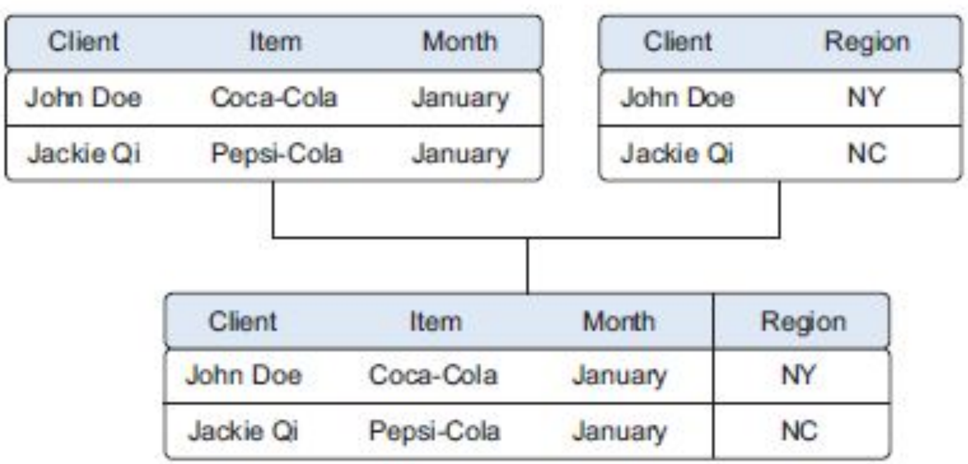


Figure 2.7 **Joining two tables on the Item and Region keys**

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.

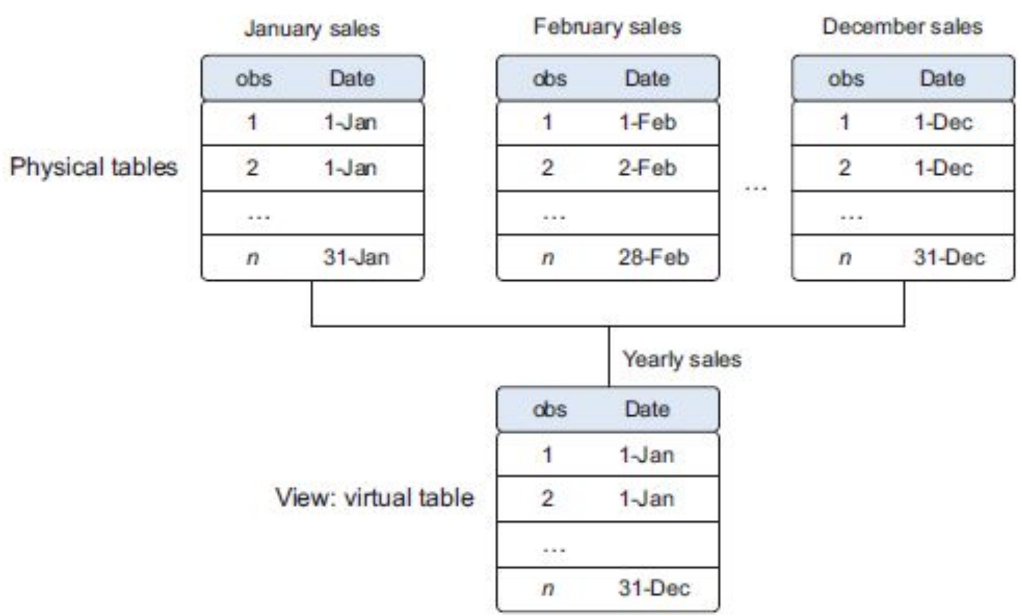


Figure 2.9 A view helps you combine data without replication.

Views

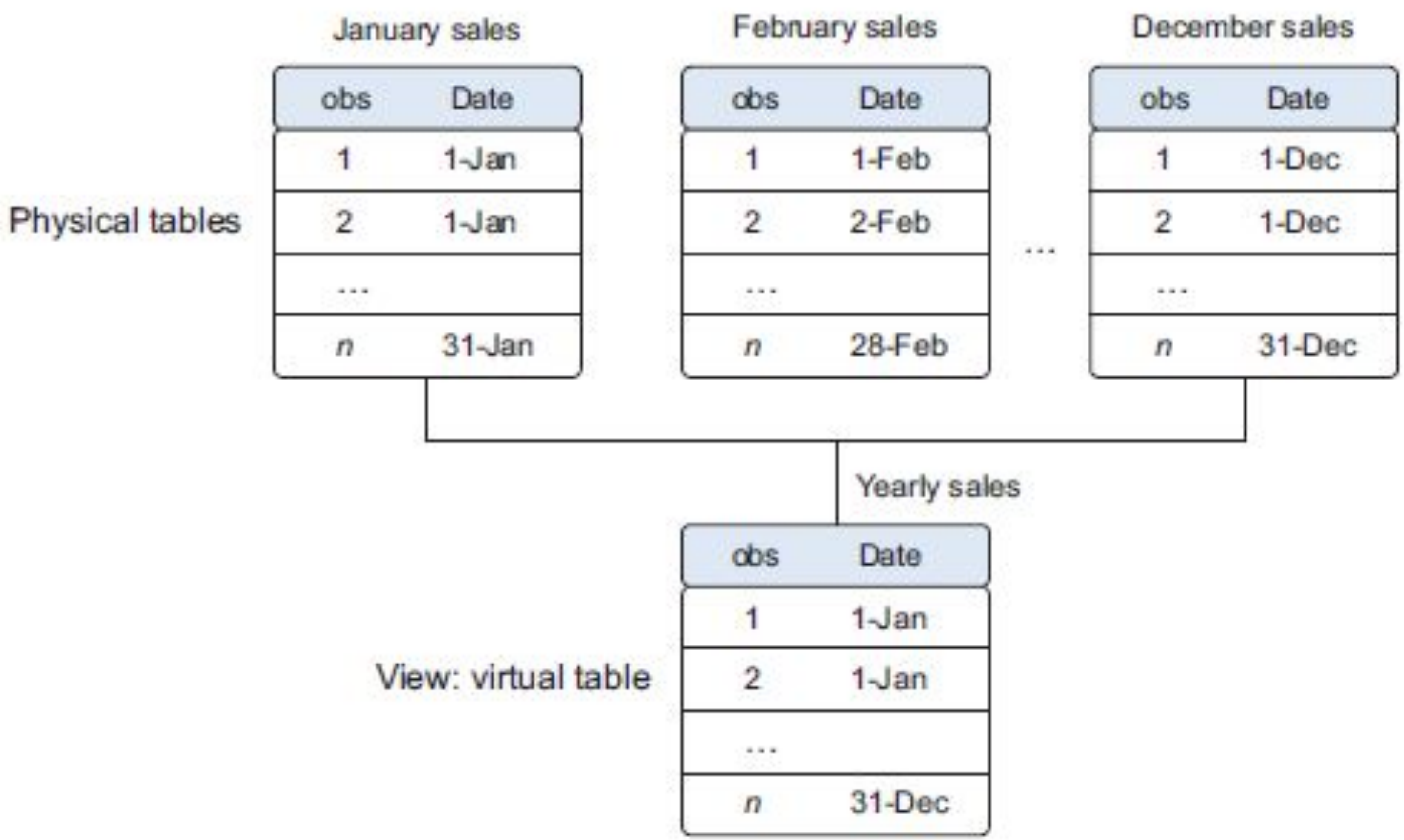


Figure 2.9 A view helps you combine data without replication.

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

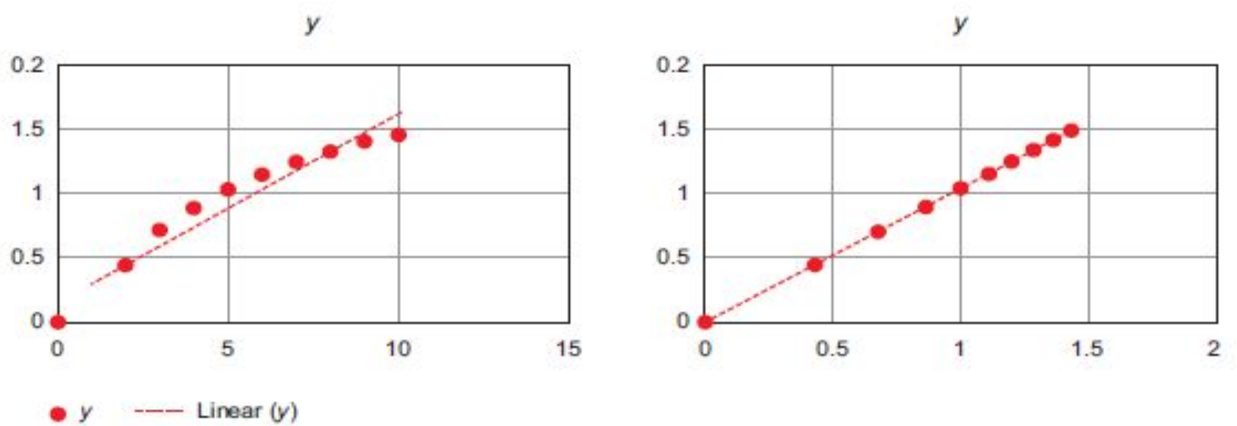


Figure 2.11 **Transforming x to log x makes the relationship between x and y linear (right), compared with the non-log x (left).**

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

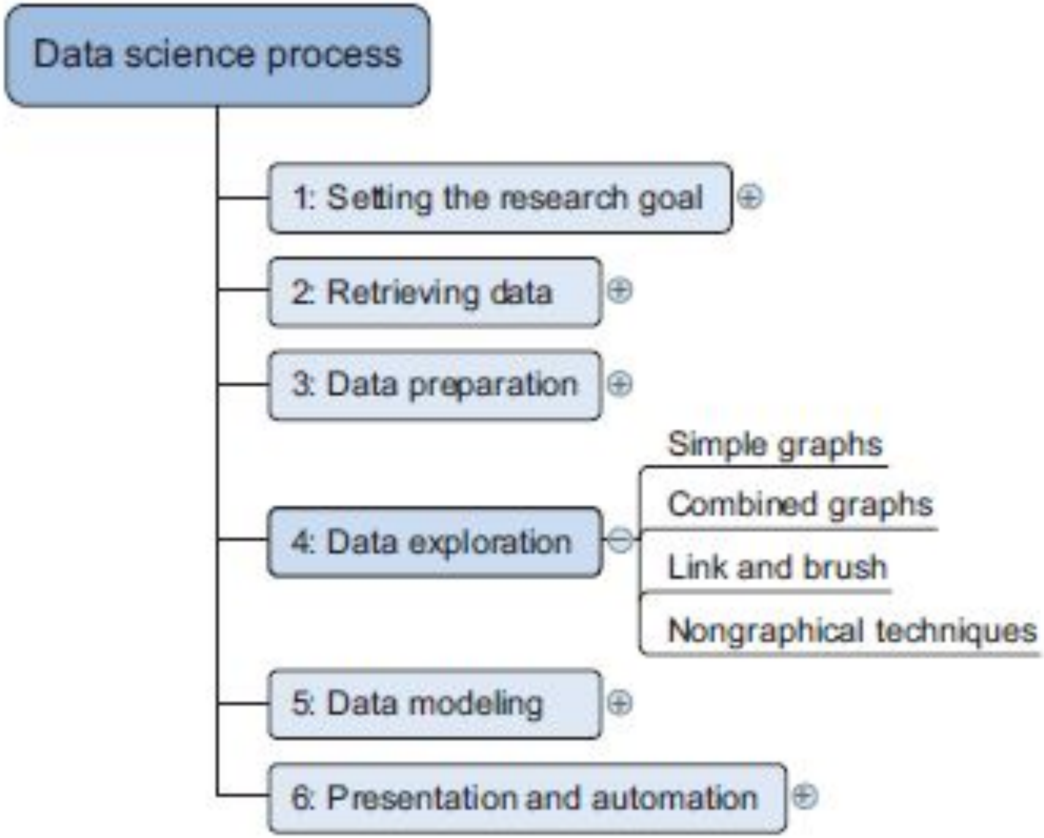
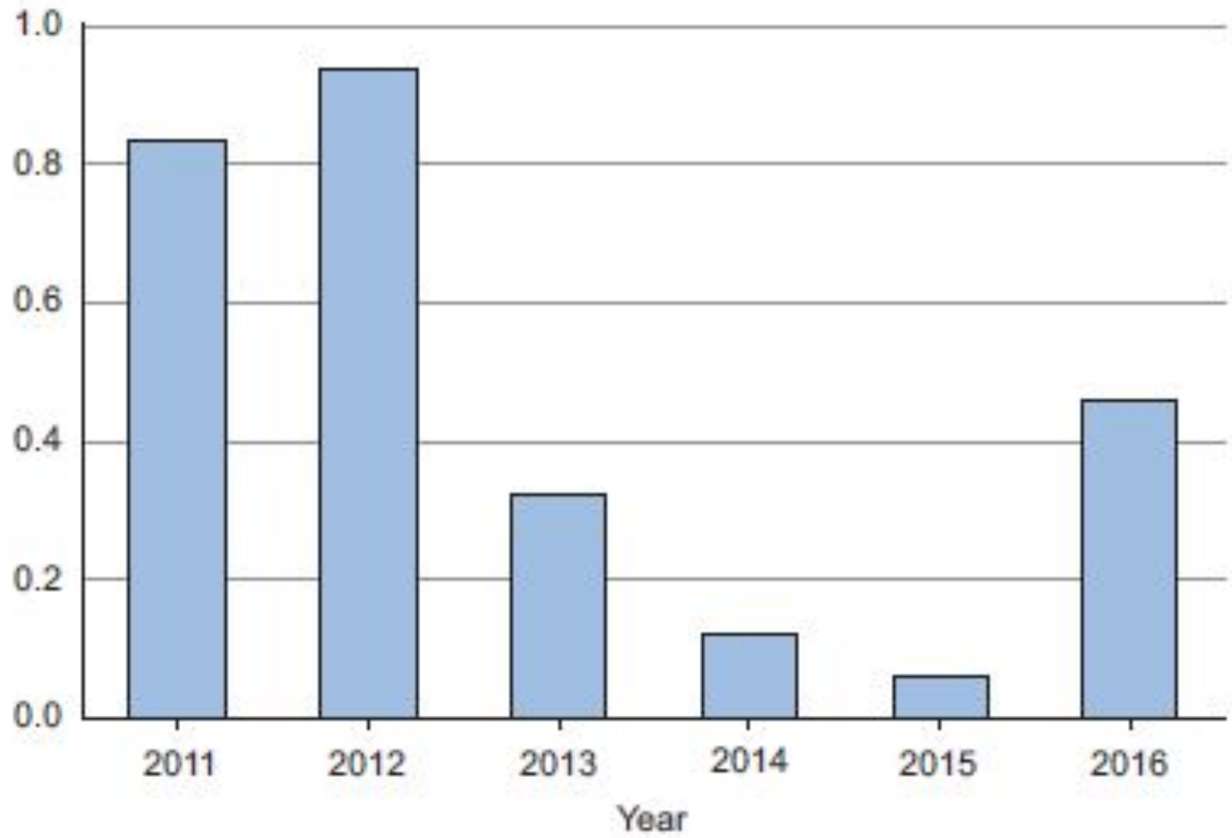


Figure 2.14 Step 4: Data exploration

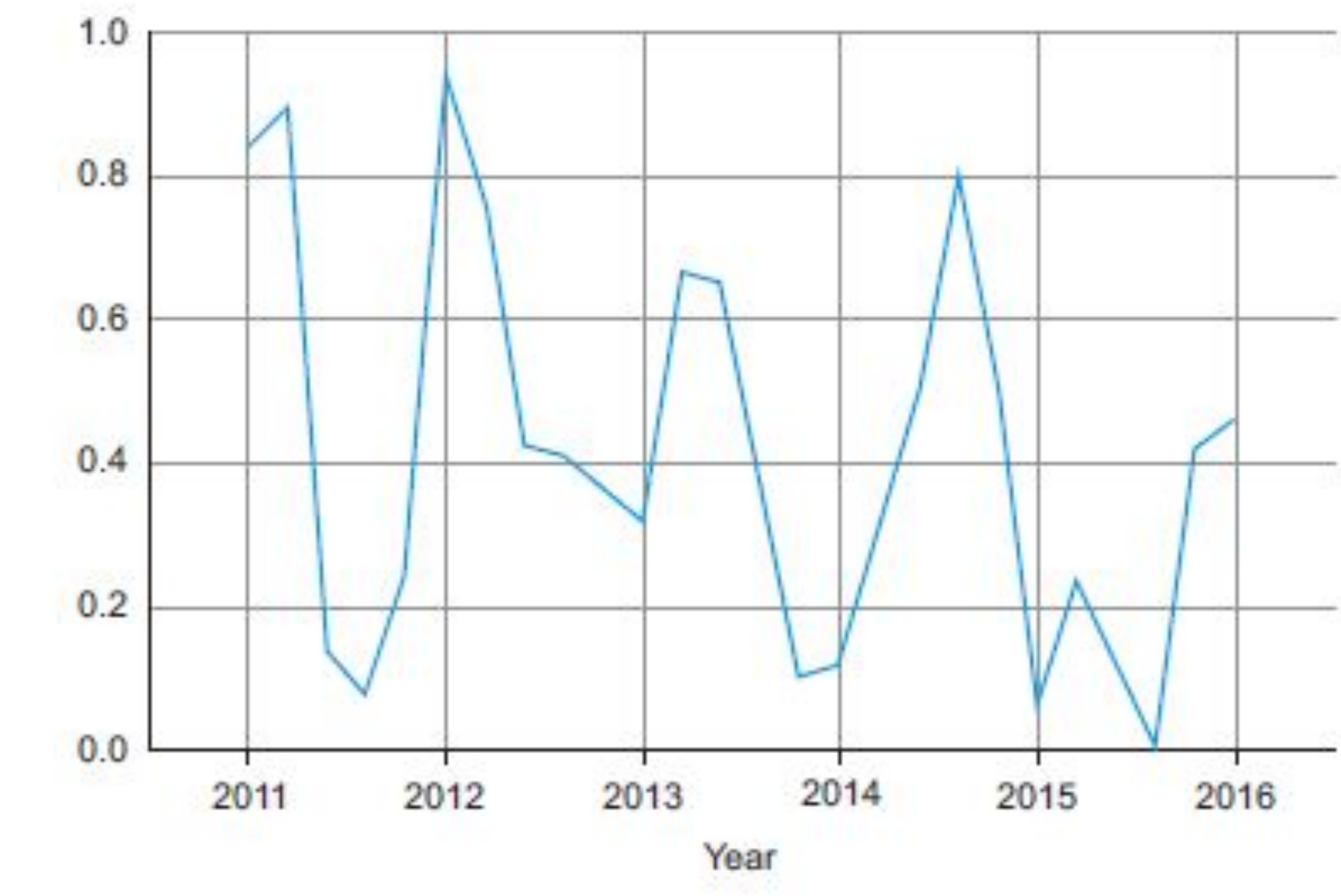
Data Exploration

- 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

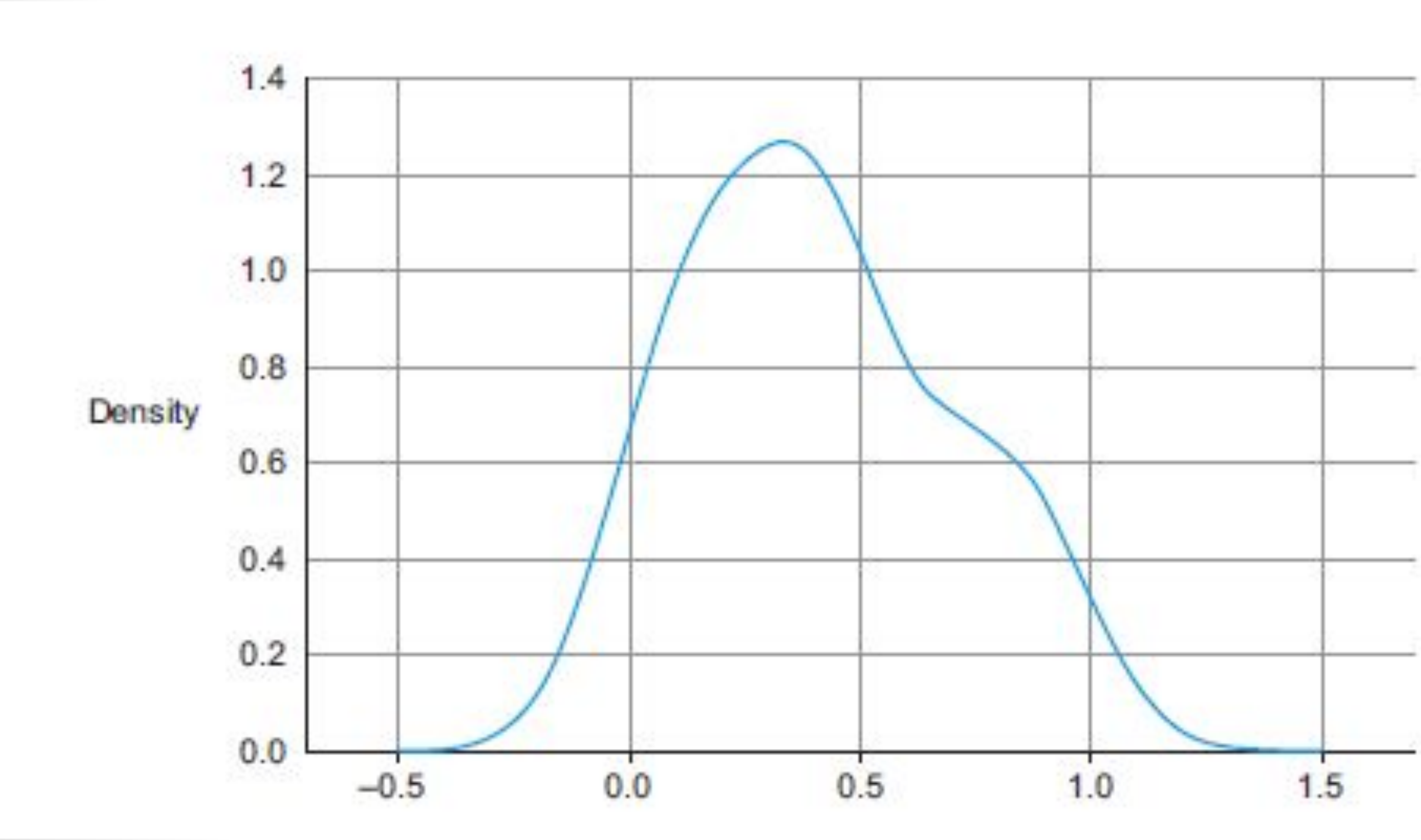
Bar Chart



Line Chart



Distribution



Overlaying

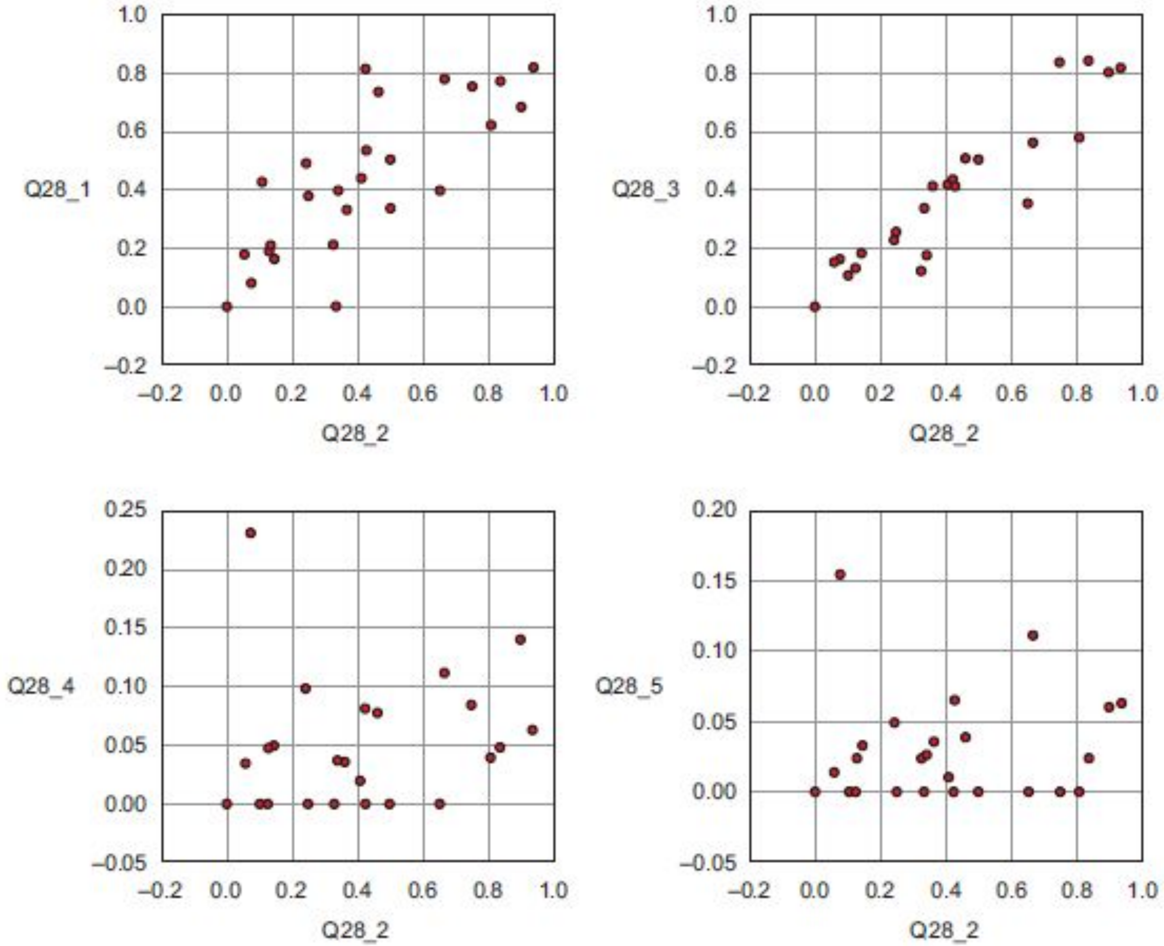


Figure 2.16 Drawing multiple plots together can help you understand the structure of your data over multiple variables.

Brushing and Linking

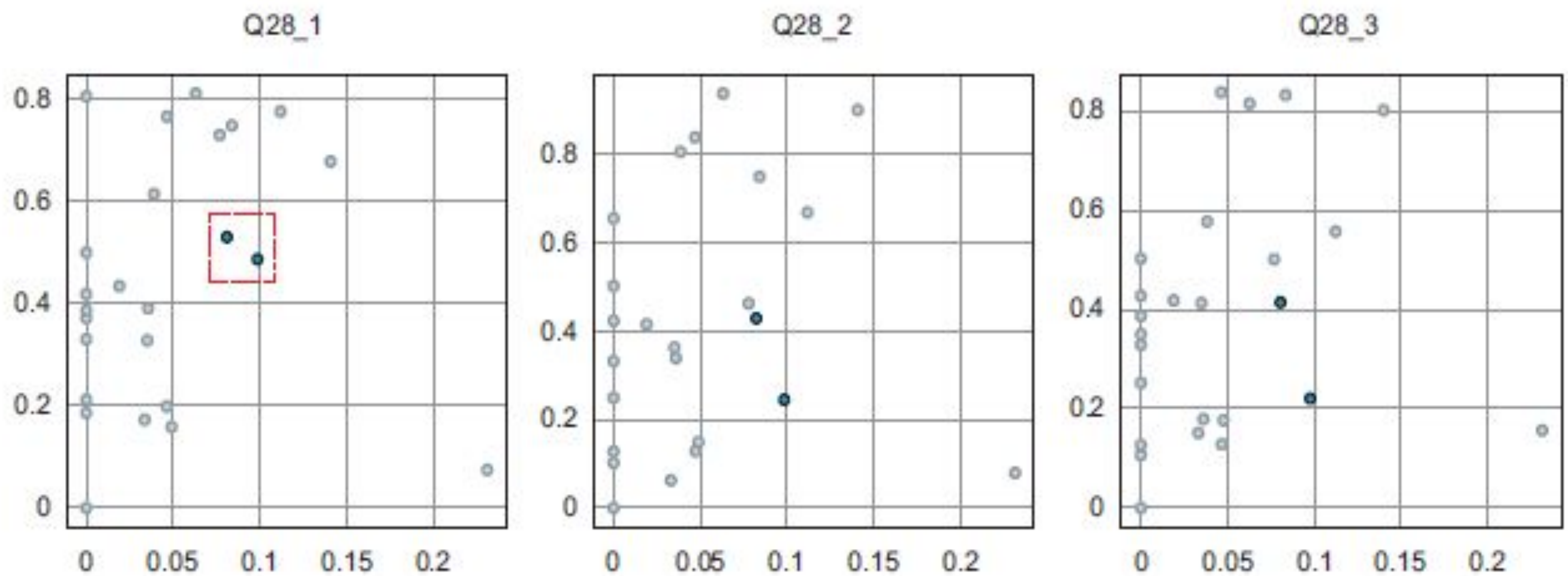


Figure 2.18 Link and brush allows you to select observations in one plot and highlight the same observations in the other plots.

STEP 5: BUILD THE MODELS

Data modeling

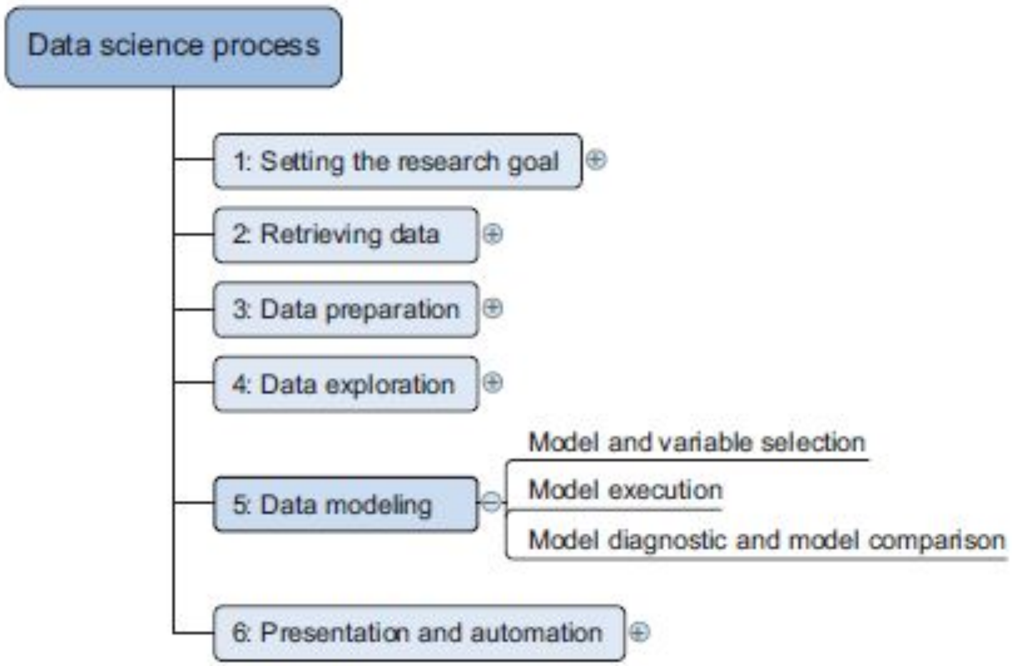


Figure 2.21 Step 5: Data modeling

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

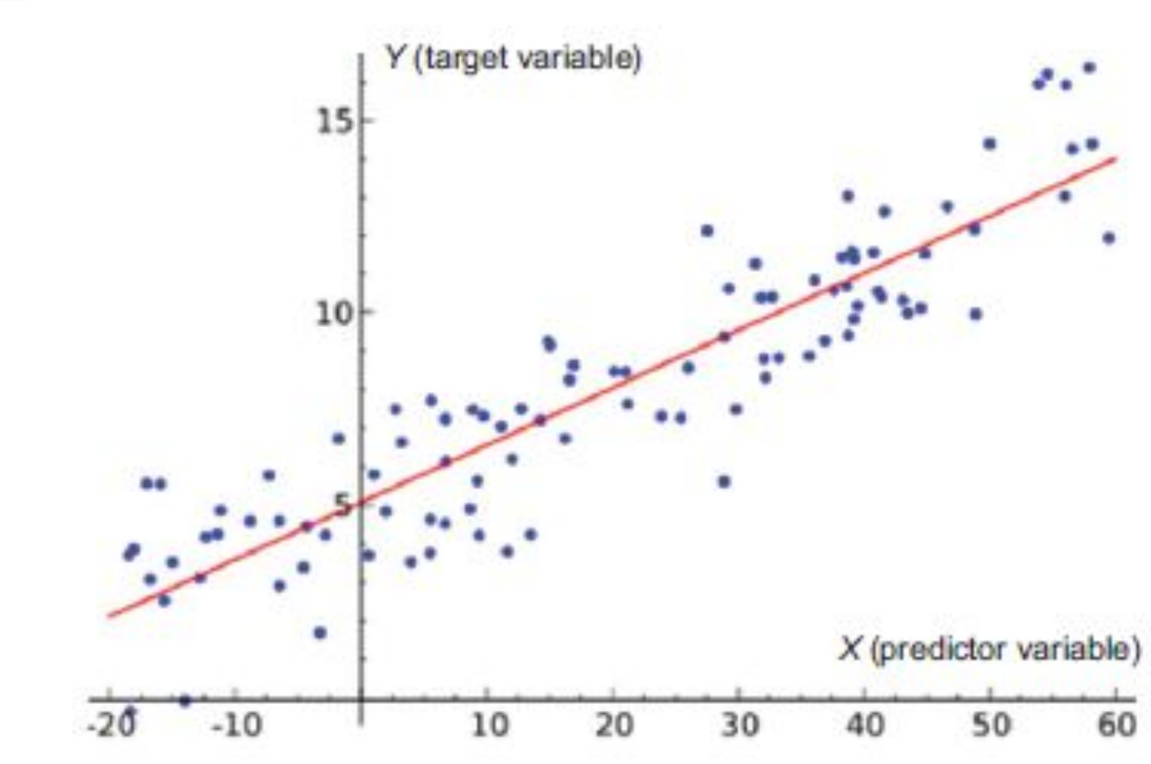


Figure 2.22 Linear regression tries to fit a line while minimizing the distance to each point

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 Python Libraries

NumPy Arrays

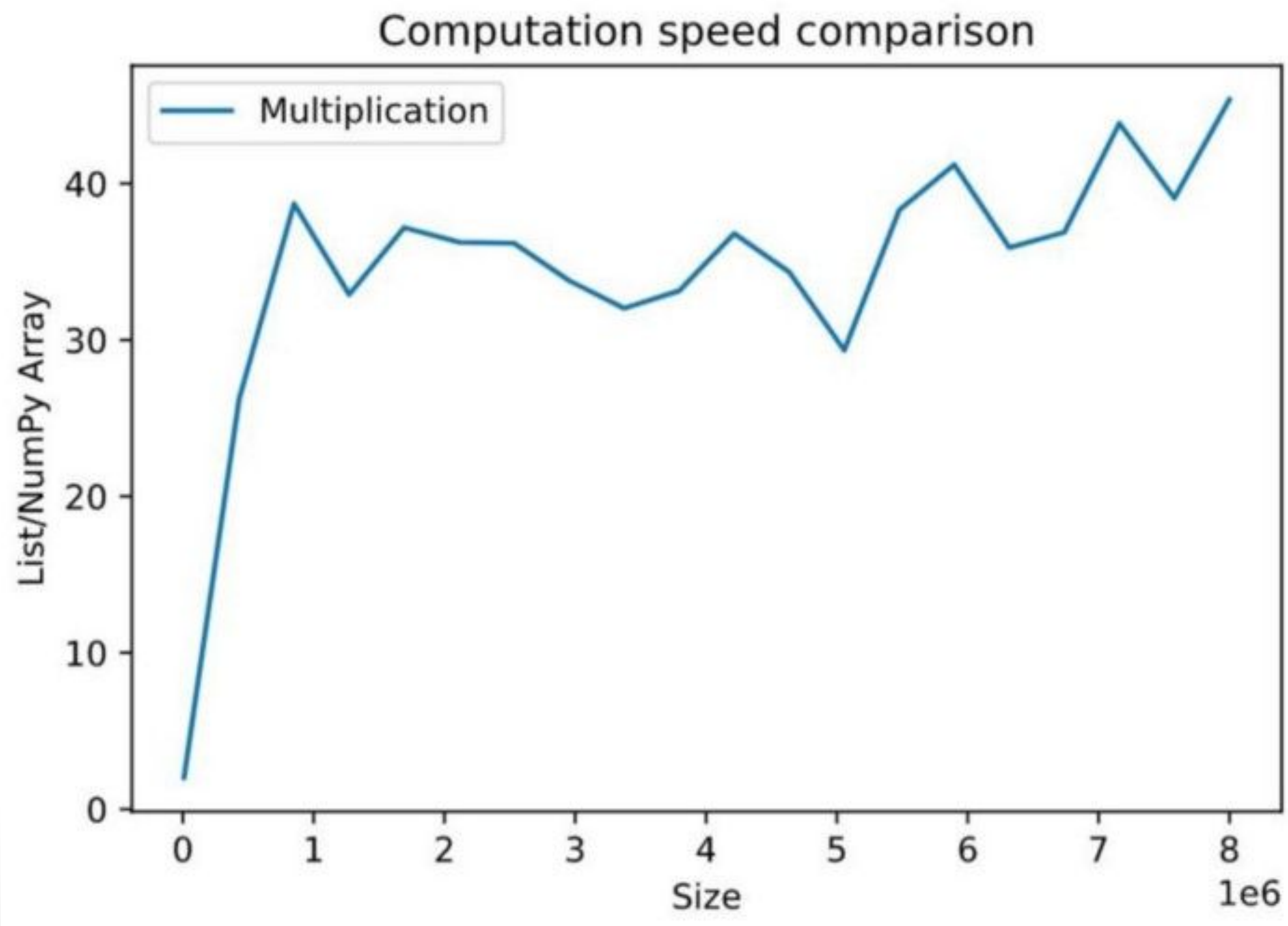
NumPy

- **Numerical Python**
- **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

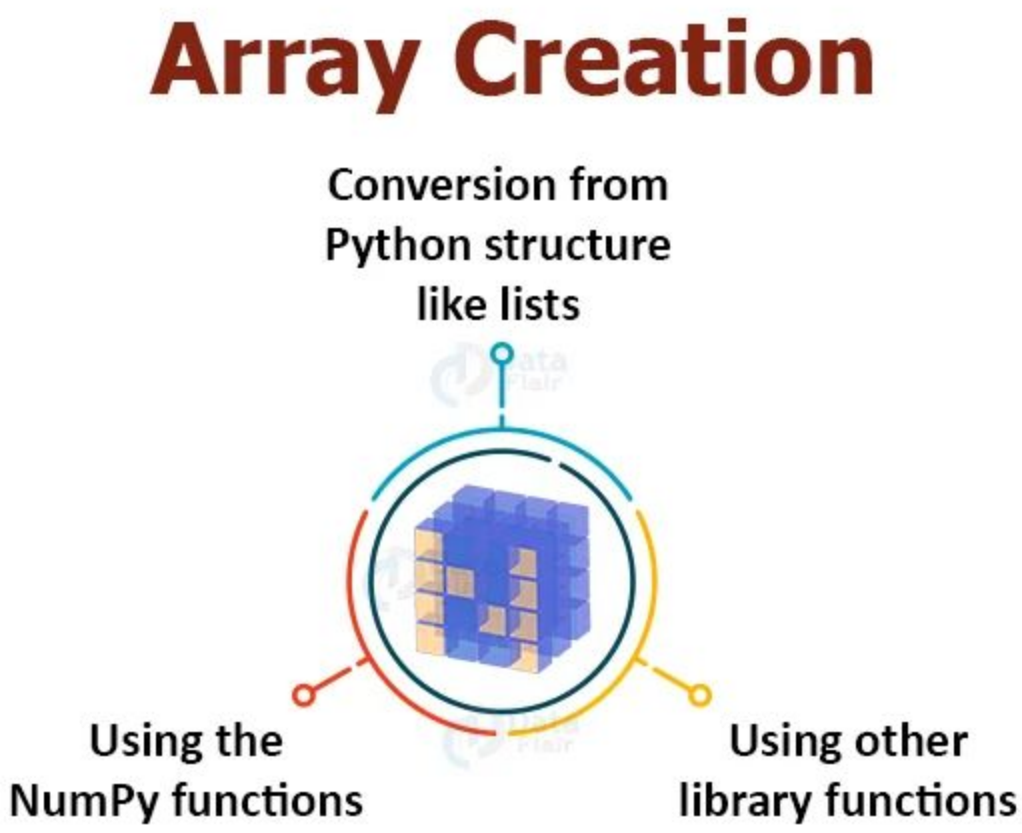
- **Store lists of numerical data, vectors and matrices**
- **Large set of routines (built-in functions) for creating, manipulating, and transforming NumPy arrays.**
- **NumPy array is officially called ndarray but commonly known as array**

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]])

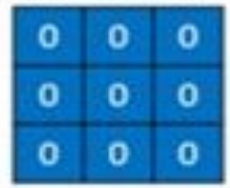
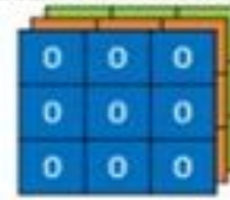
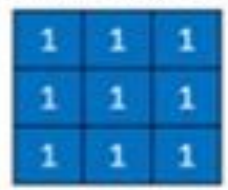
1. Using the NumPy functions

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	 np.zeros(3)	 np.zeros((3,3))	 np.zeros((3,3,3))
Creates an array of zero			
np.ones	 np.ones(3)	 np.ones((3,3))	 np.ones((3,3,3))
Creates an array of ones			
np.eye		 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

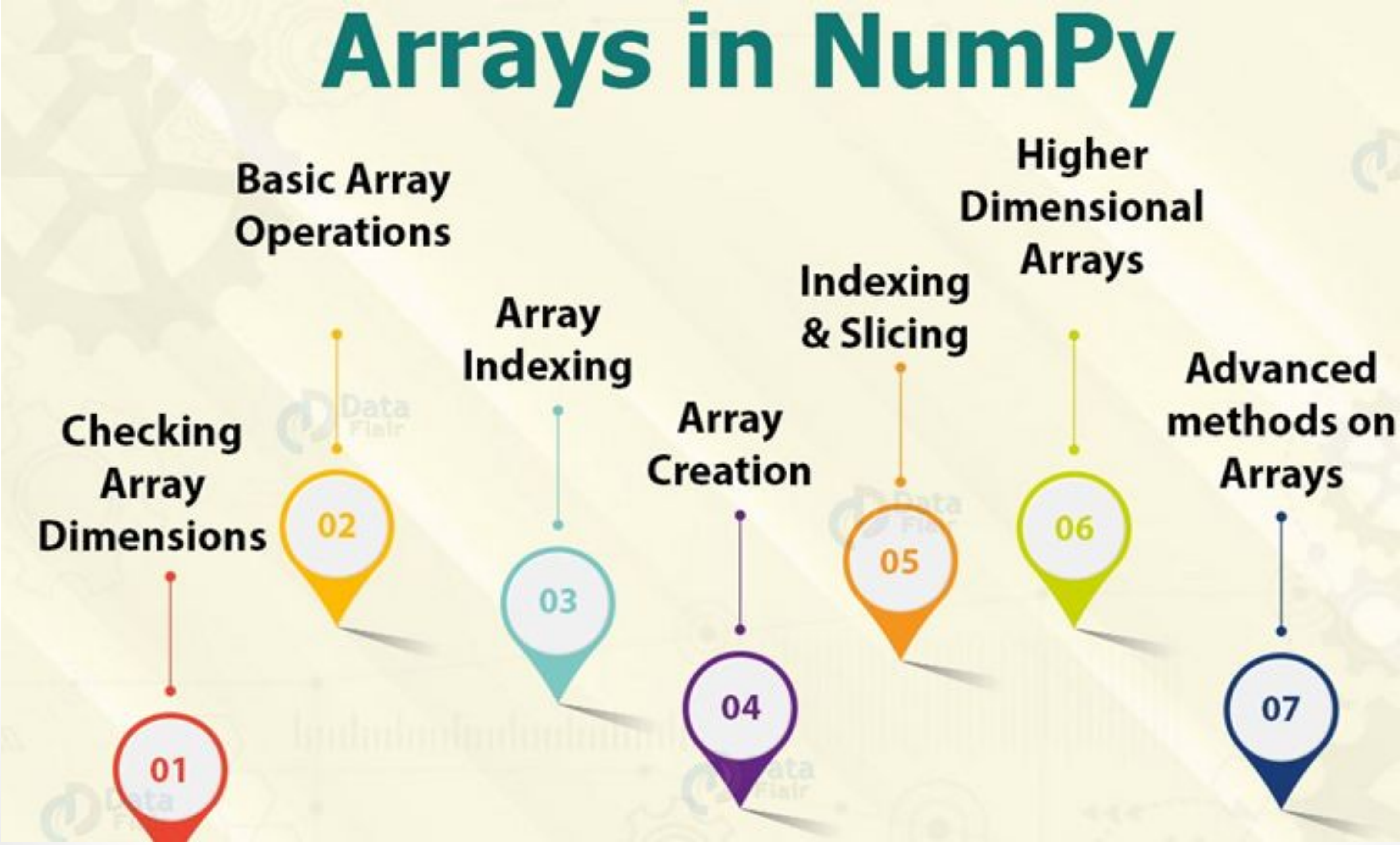
- **np.ndarray(shape, type)**
  - **Creates an array of the given shape with random numbers.**
- **np.array(array_object)**
  - **Creates an array of the given shape from the list or tuple.**
- **np.zeros(shape)**
  - **Creates an array of the given shape with all zeros.**
- **np.ones(shape)**
  - **Creates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

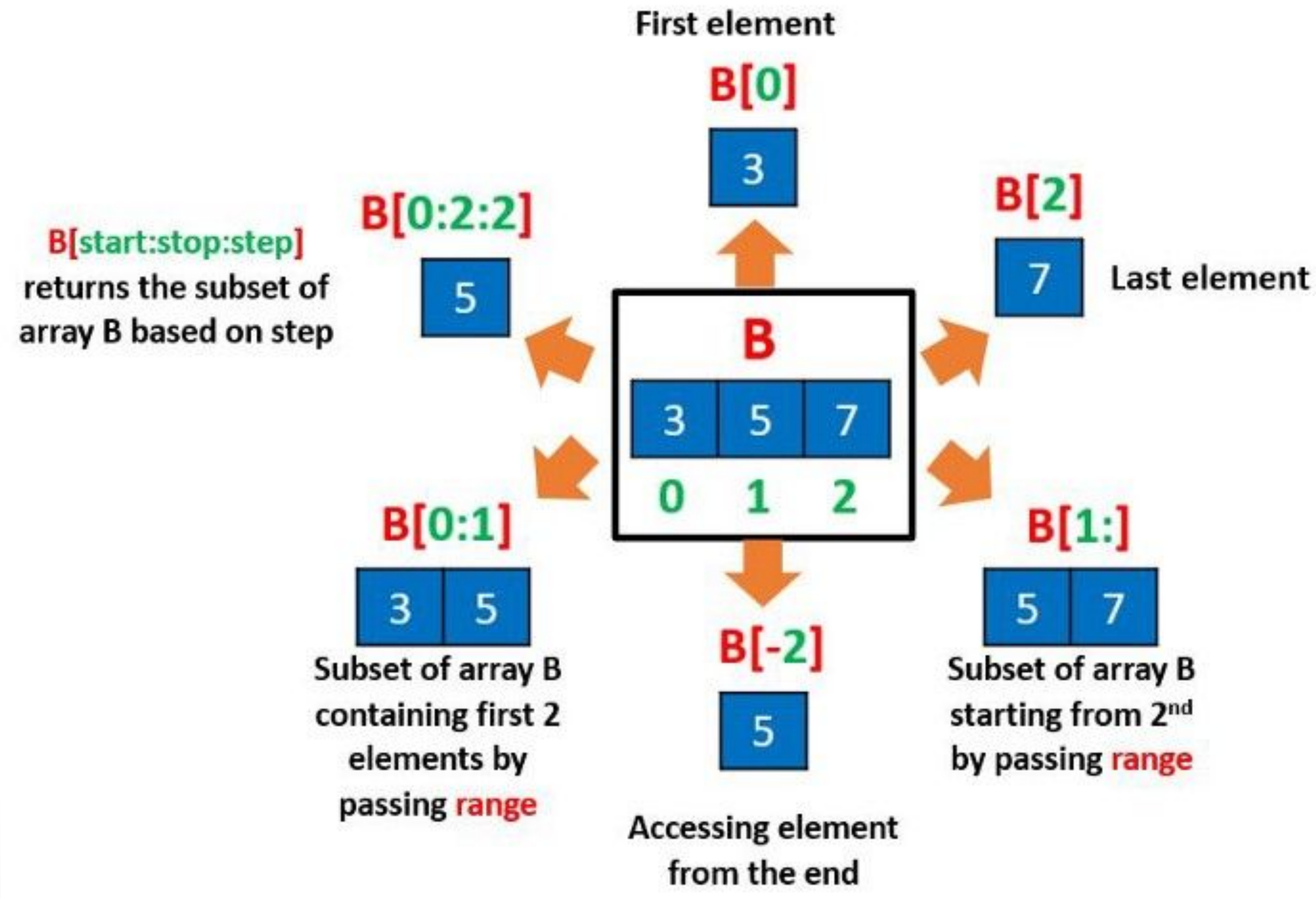
Indexing

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

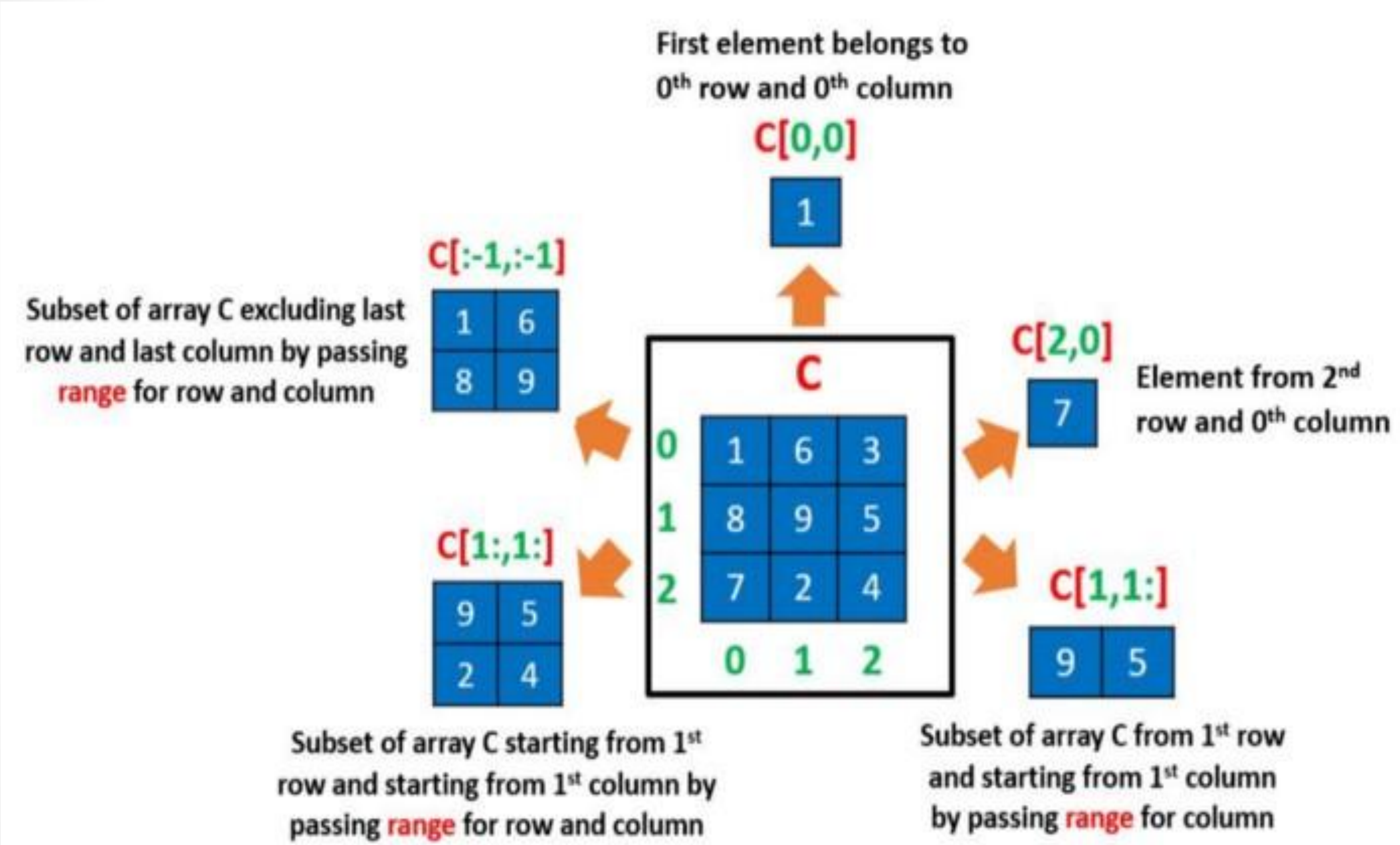
Slicing

```
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.

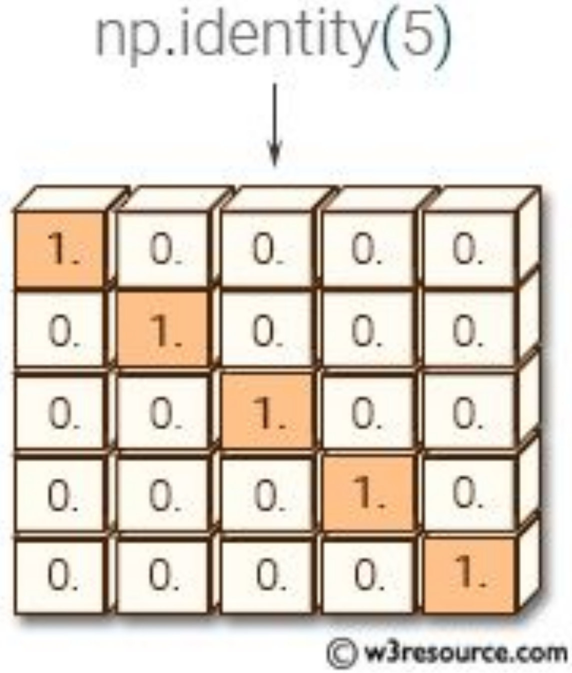
- **Iterating on Each Scalar Element**
  - In basic for loops, iterating through each scalar of an array we need to use *n* for loops which can be difficult to write for arrays with very high dimensionality.

```
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

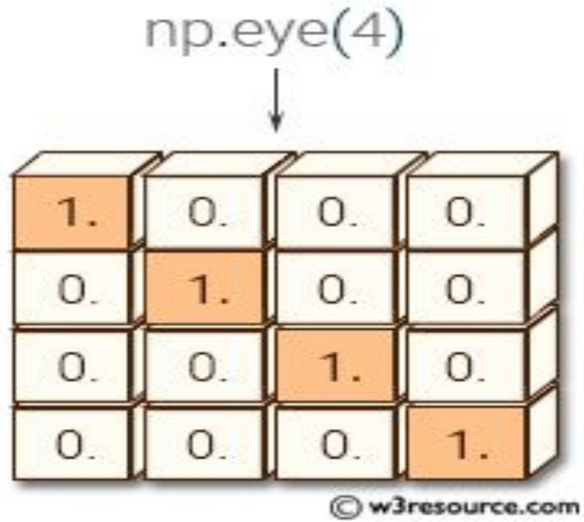
```
# 2x2 matrix with 1's on main diagonal
b = geek.identity(2, dtype = float)
print("Matrix b : \n", b)
 a = geek.identity(4)
print("\nMatrix a : \n", a)
```

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

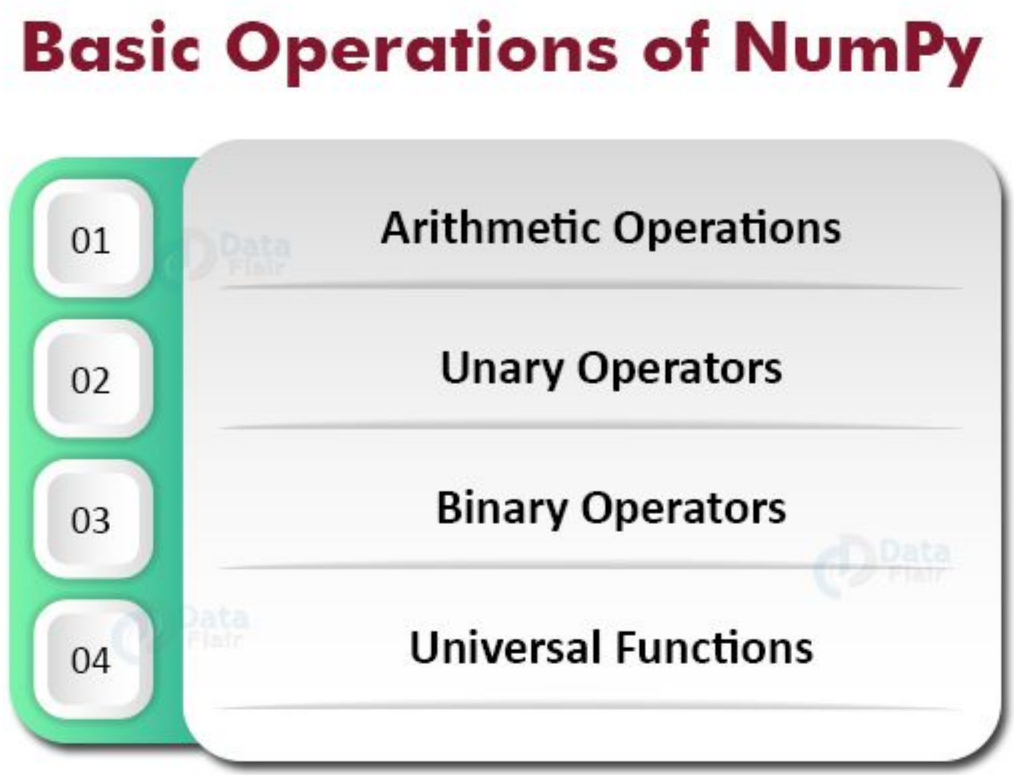
- Flattening array means converting a multidimensional array into a 1D array.
- import numpy as np

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

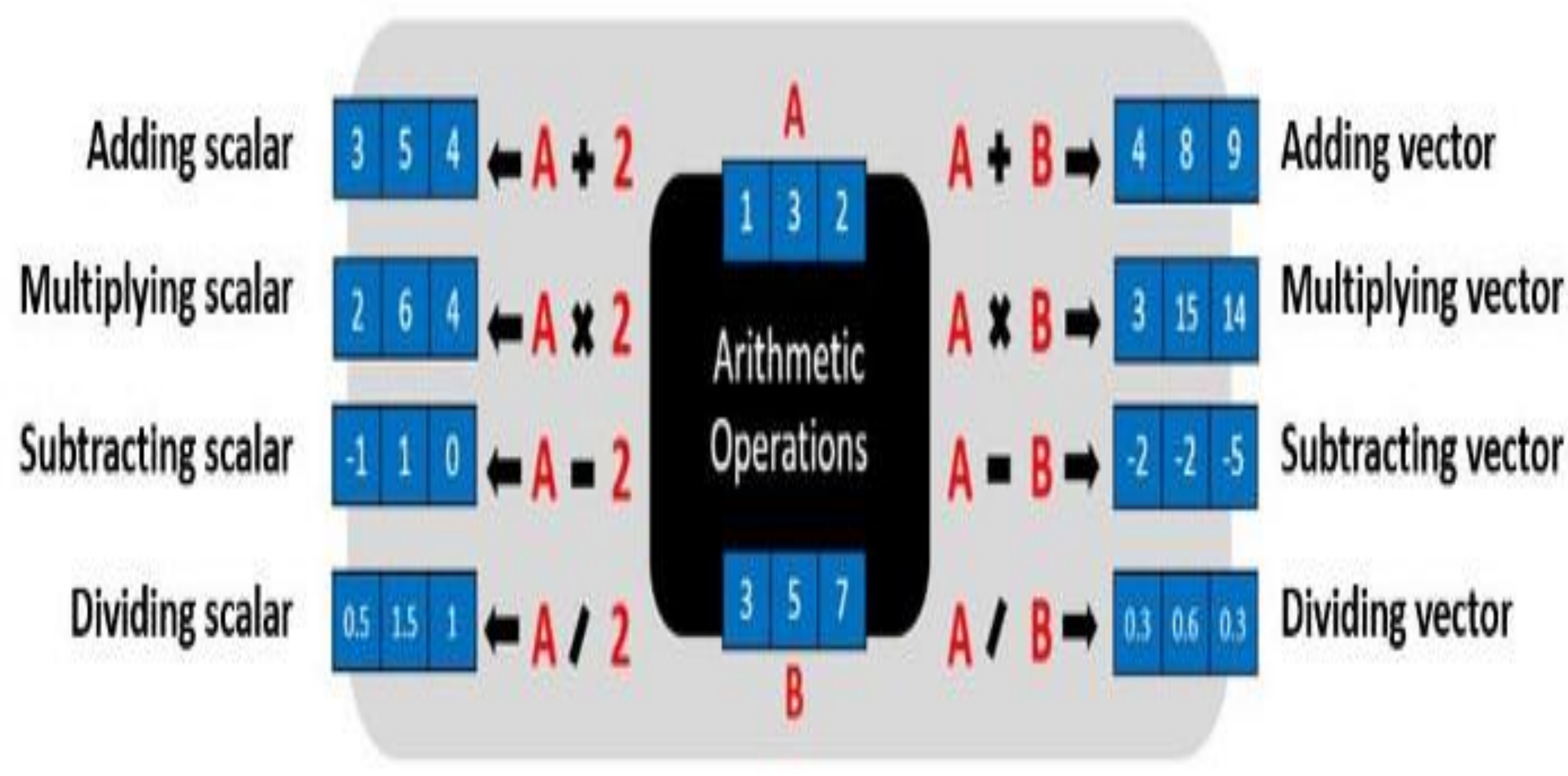
  newarr = arr.reshape(-1)

  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.

Operations on NumPy



1.NumPy Arithmetic Operations



1.NumPy Arithmetic Operations

```
import numpy as np
a = np.array([1, 2, 3, 4])
# add 5 to every element
print ( a+5)
# subtract 2 from each element
print ( a-2)
# multiply each element by 5
print (a*10)
# divide each element by 2
print ( a/2)
```

```
[6 7 8 9]
[-1  0  1  2]
[10 20 30 40]
[0.5 1.  1.5 2. ]
```

2. NumPy Unary Operators

```
import numpy as np
arr = np.array([[1,5, 12], [2,32, 20], [3, 40, 13]])
print(arr.max(axis = 1))
print(arr.max(axis = 0))
print (arr.min(axis = 0))
print(arr.min(axis = 1))
print (arr.sum( ))
print ( arr.sum(axis=0))
print( arr.sum(axis=1))
```

```
[12 32 40]
[ 3 40 20]
[ 1  5 12]
[1 2 3]
128
[6 77 45]
[18 54 56]
```

3. NumPy Binary Operators

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[4, 3], [2, 1]])
print (a + b)
print (a*b)
```

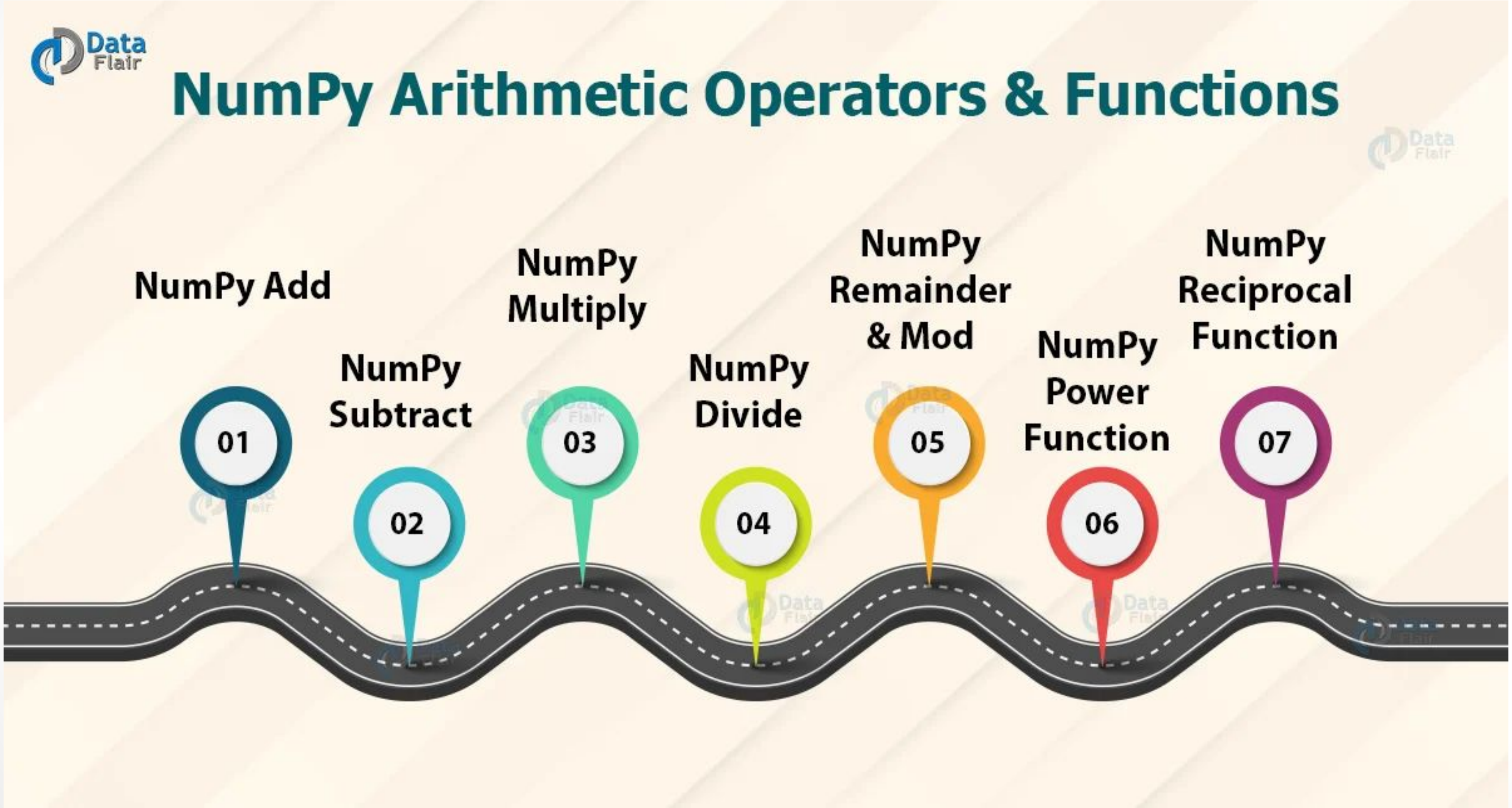
```
[[5 5]
 [5 5]]

[[4 6]
 [6 4]]
```

NumPy Universal Functions

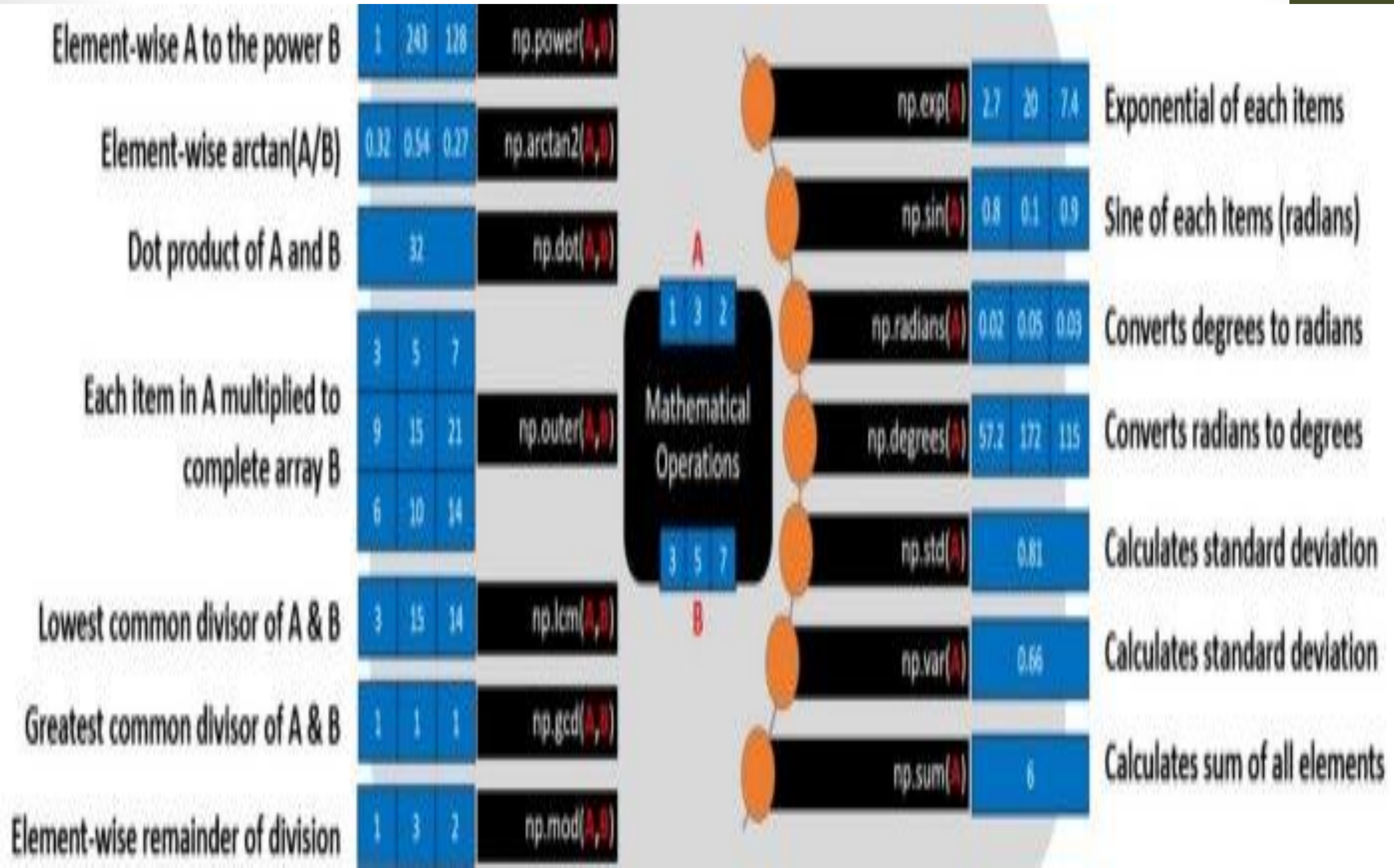
```
import numpy as np
a = np.array([0, np.pi/2, np.pi])
print ( np.sin(a))
a = np.array([0, 1, 2, 3])
print ( np.exp(a))
print ( np.sqrt(a))
```

```
[0.0000000e+00 1.0000000e+00 1.2246468e-16]

[ 1. 2.71828183 7.3890561 20.08553692]

[0. 1. 1.41421356 1.73205081]
```

Arithmetic Operators & Functions



Mathematical Functions



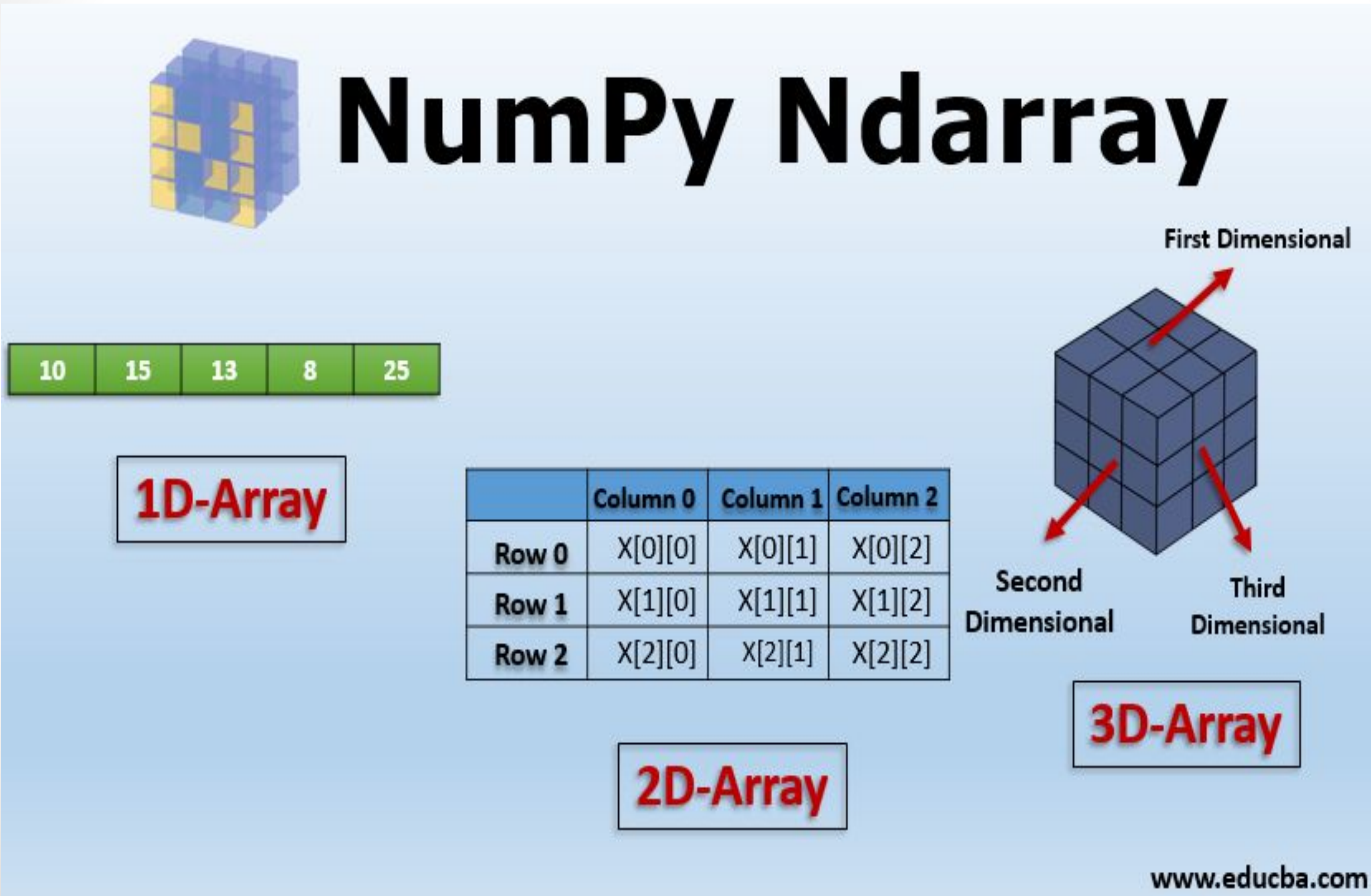
NumPy functions

```
import numpy as np
a = np.array([7,3,4,5,1])
b = np.array([3,4,5,6,7])
np.add(a, b)  #([ 10, 7, 9, 11, 8])
np.subtract(a,b) #[ 4,-1,-1,-1,-6]
np.multiply(a, b) #[21, 12, 20, 30, 7]
np.divide(a, b) #[2.33333333, 0.75 , 0.8 , 0.83333333, 0.14285714]
np.remainder(a,b) #[1, 3, 4, 5, 1]
np.mod(a,b) #[1, 3, 4, 5, 1]
np.power(a,b) #[ 343, 81, 1024, 15625, 1]
np.reciprocal(a) #[0, 0, 0, 0, 1]
```

NumPy Add Operator

```
import numpy as np
a = np.array([10,20,100,200,500])
b = np.array([3,4,5,6,7])
print(a+b) #[ 13 24 105 206 507]
print(a-b)
print(a*b)
print(a/b)
```

Types of Array



Creating a 1-D Array

```
import numpy as np
#creating an array to understand its attributes
A = np.array([[1,2,3],[1,2,3],[1,2,3]])
print("Array A is:\n",A)
#type of array
print("Type:", type(A))
#Shape of array
print("Shape:", A.shape)
#no. of dimensions
print("Rank:", A.ndim)
#size of array
print("Size:", A.size)
#type of each element in the array
print("Element type:", A.dtype)
```

Creating a 1-D Array

```
import numpy as np
#creating an array to understand its attributes
A = np.array([[1,2,3],[1,2,3],[1,2,3]])
print("Array A is:\n",A)
#type of array
print("Type:", type(A))
#Shape of array
print("Shape:", A.shape)
#no. of dimensions
print("Rank:", A.ndim)
#size of array
print("Size:", A.size)
#type of each element in the array
print("Element type:", A.dtype)
```

Output:

```
Array A is:
 [[1 2 3]
 [1 2 3]
 [1 2 3]]
Type: <class 'numpy.ndarray'>
Shape: (3, 3)
Rank: 2
Size: 9
Element type: int32
```

Creation of a multidimensional array(ndarray)

```
import numpy as np
#creating array using ndarray
A = np.ndarray(shape=(2,2), dtype=float)
print("Array with random values:\n", A)
# Creating array from list
B = np.array([[1, 2, 3], [4, 5, 6]])
print ("Array created with list:\n", B)
# Creating array from tuple
C = np.array((1 , 2, 3))
print ("Array created with tuple:\n", C)
# Creating array with all ones
D = np.ones((3, 3))
print ("Array with all ones:\n", D)
# Creating array with all zeros
E = np.zeros((3, 3))
print ("Array with all zeroes:\n",E)
# Creating an array with complex data type
F = np.full((3, 3), 1, dtype = 'complex')
print ("Array of complex data type:\n", F)
```

Creation of a multidimensional array(ndarray)

```
F = np.full((3, 3), 1, dtype = 'complex')
print ("Array of complex data type:\n", F)
#creating an array with buffer
G = np.ndarray((2,), buffer=np.array([1,2,3]),dtype=int)
print ("Array with buffer specified:\n", G)
#creating an array with range
H = np.arange(10)
print ("Array with range specified:\n", H)
```

Creation of a multidimensional array(ndarray)

```
Array with random values:
 [[3.22651327e-307 1.37962320e-306]
 [1.78019761e-306 3.11524091e-307]]
Array created with list:
 [[1 2 3]
 [4 5 6]]
Array created with tuple:
 [1 2 3]

Array with all ones:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
Array with all zeroes:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
Array of complex data type:
 [[1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j]]
Array with buffer specified:
 [1 2]
Array with range specified:
 [0 1 2 3 4 5 6 7 8 9]
```

Program to illustrate Indexing in Ndarrays

```
#creating an array to understand indexing
A = np.array([[1,2,1],[7,5,3],[9,4,8]])
print("Array A is:\n",A)
B = A[[0, 1, 2], [0, 1, 2]]
print ("Elements at indices (0, 0),(1, 1), (2, 2) are : \n",B)
#changing the value of elements at a given index
A[0,0] = 12
A[1,1] = 4
A[2,2] = 7
print("Array A after change is:\n", A)
```

Program to illustrate Indexing in Ndarrays

Output:

```
Array A is:
 [[1 2 1]
 [7 5 3]
 [9 4 8]]
Elements at indices (0, 0),(1, 1), (2, 2) are :
 [1 5 8]
Array A after change is:
 [[12  2  1]
 [ 7  4  3]
 [ 9  4  7]]
```

Program to illustrate Indexing in a 3D array

```
#creating a 3d array to see indexing in a 3D array.
import numpy as np
I = np.array([[[ 0,  1,  2,  3],
[ 4,  5,  6,  7],
[ 8,  9, 10, 11]],
[[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23]]])
print("3D Array is:\n", I)
print("Elements at index (0,0,1):\n", I[0,0,1])
print("Elements at index (1,0,1):\n", I[1,0,1])
#changing the value of elements at a given index
I[1,0,2] = 31
print("3D Array after change is:\n", I)
```

Program to illustrate Indexing in a 3D array

Output:

```
3D Array is:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
Elements at index (0,0,1):
 1
Elements at index (1,0,1):
 13
3D Array after change is:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 31 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Operations on Ndarray

```
import numpy as np
A = np.array([[1, 2, 3],
[4,5,6],[7,8,9]])
B = np.array([[1, 2, 3],
[4,5,6],[7,8,9]])
# adding arrays A and B
print ("Elementwise sum of array A and B is :\n", A + B)
# multiplying arrays A and B
print ("Elementwise multiplication of array A and B:\n", A*B)
```

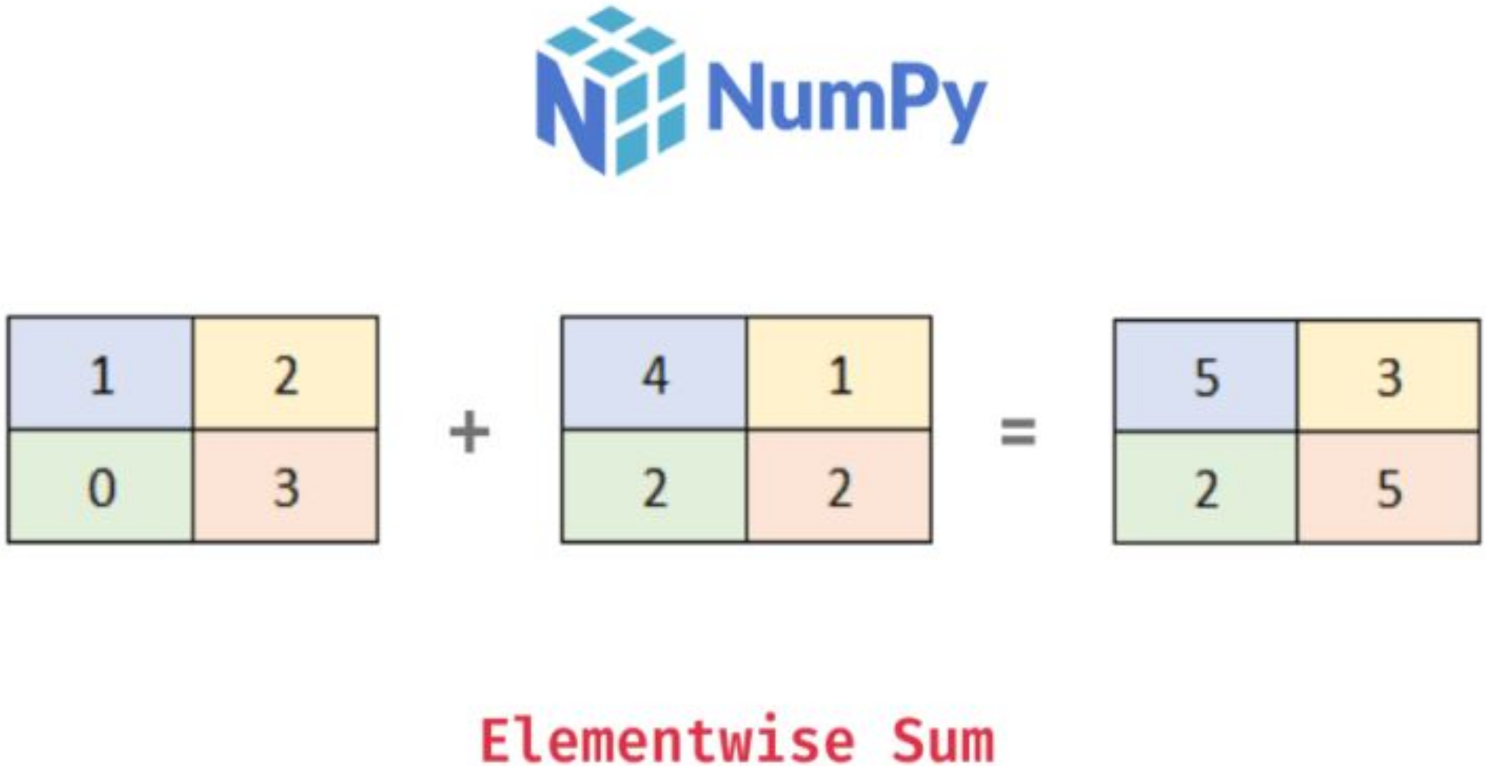
Operations on Ndarray

```
import numpy as np
A = np.array([[1, 2, 3],
[4,5,6],[7,8,9]])
B = np.array([[1, 2, 3],
[4,5,6],[7,8,9]])
# adding arrays A and B
print ("Elementwise sum of array A and B is :\n", A + B)
# multiplying arrays A and B
print ("Elementwise multiplication of array A and B:\n", A*B)
```

Output:

```
Elementwise sum of array A and B is :
 [[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
Elementwise multiplication of array A and B:
 [[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

Add two 1d arrays element-wise



Add two 1d arrays element-wise

```
import numpy as np

# create numpy arrays x1 and x2
x1 = np.array([1, 3, 0, 7])
x2 = np.array([2, 0, 1, 1])
# elementwise sum with np.add()
x3 = np.add(x1, x2)
# display the arrays
print("x1:", x1)
print("x2:", x2)
print("x3:", x3)
```

Add two 1d arrays element-wise

```
import numpy as np

# create numpy arrays x1 and x2
x1 = np.array([1, 2, 0, 3])
x2 = np.array([4, 1,2, 2])
# elementwise sum with np.add()
x3 = np.add(x1, x2)
# display the arrays
print("x1:", x1)
print("x2:", x2)
print("x3:", x3)
x3 = x1 + x2
# display the arrays
print("x1:", x1)
print("x2:", x2)
print("x3:", x3)
```

```
x1: [1 2 0 3]
x2: [4 1 2 2]
x3: [5 3 2 5]
```

Add two 2d arrays elementwise

```
# create 2d arrays x1 and x2
x1 = np.array([[1, 0, 1], [2, 1, 1], [3, 0, 3]])
x2 = np.array([[2, 2, 0], [1, 0, 1], [0, 1, 0]])
# elementwise sum with np.add()
x3 = np.add(x1, x2)
# display the arrays print("x1:\n", x1)
print("x2:\n", x2)
print("x3:\n", x3)
```

```
x1:
[[1 0 1]
[2 1 1]
[3 0 3]]
x2:
[[2 2 0]
[1 0 1]
[0 1 0]]
x3:
[[3 2 1]
[3 1 2]
[3 1 3]]
```

Add more than two arrays elementwise

```
# create numpy arrays x1, x2, and x3
x1 = np.array([1, 3, 0, 7])
x2 = np.array([2, 0, 1, 1])
x3 = np.array([0, 1, 3, 1])
# elementwise sum with +
x4 = x1+x2+x3
# display the arrays
print("x1:", x1)
print("x2:", x2)
print("x3:", x3)
print("x4:", x4)
```

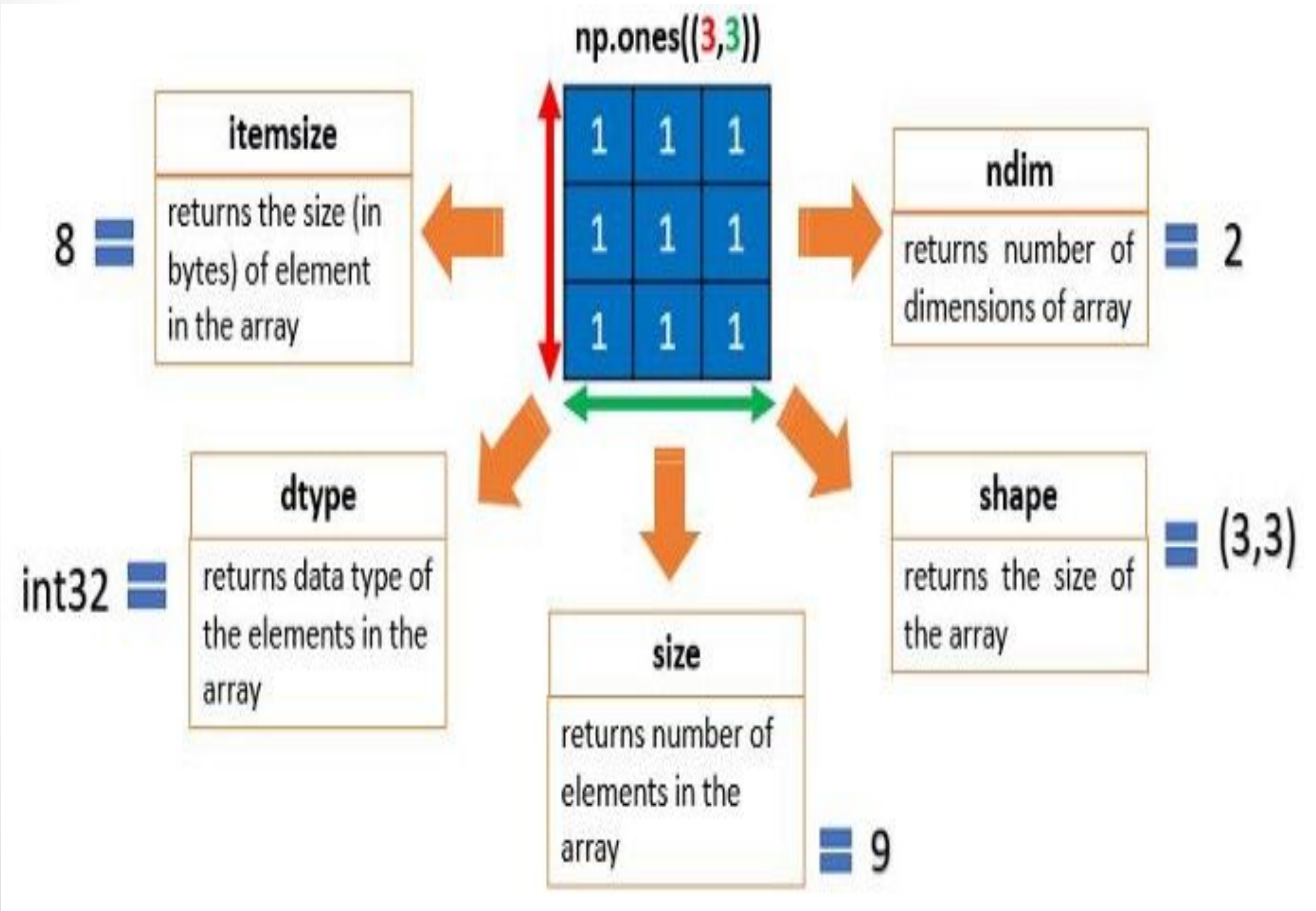
```
x1: [1 3 0 7]
x2: [2 0 1 1]
x3: [0 1 3 1]
x4: [3 4 4 9]
```

NumPy Array attributes

1. **ndarray.flags-** It provides information about memory layout
2. **ndarray.shape-** Provides array dimensions
3. **ndarray.strides-** Determines step size while traversing the arrays
4. **ndarray.ndim-** Number of array dimensions
5. **ndarray.data-** Points the starting position of array
6. **ndarray.size-** Number of array elements
7. **ndarray.itemsize-** Size of individual array elements in bytes
8. **ndarray.base-** Provides the base object, if it is a view
9. **ndarray.nbytes-** Provides the total bytes consumed by the array
10. **ndarray.T-** It gives the array transpose
11. **ndarray.real-** Separates the real part
12. **ndarray.imag-** Separates the imaginary

NumPy Array attributes

1. **ndarray.flags-** It provides information about memory layout
2. **ndarray.shape-** Provides array dimensions
3. **ndarray.strides-** Determines step size while traversing the arrays
4. **ndarray.ndim-** Number of array dimensions
5. **ndarray.data-** Points the starting position of array
6. **ndarray.size-** Number of array elements
7. **ndarray.itemsize-** Size of individual array elements in bytes
8. **ndarray.base-** Provides the base object, if it is a view
9. **ndarray.nbytes-** Provides the total bytes consumed by the array
10. **ndarray.T-** It gives the array transpose
11. **ndarray.real-** Separates the real part
12. **ndarray.imag-** Separates the imaginary



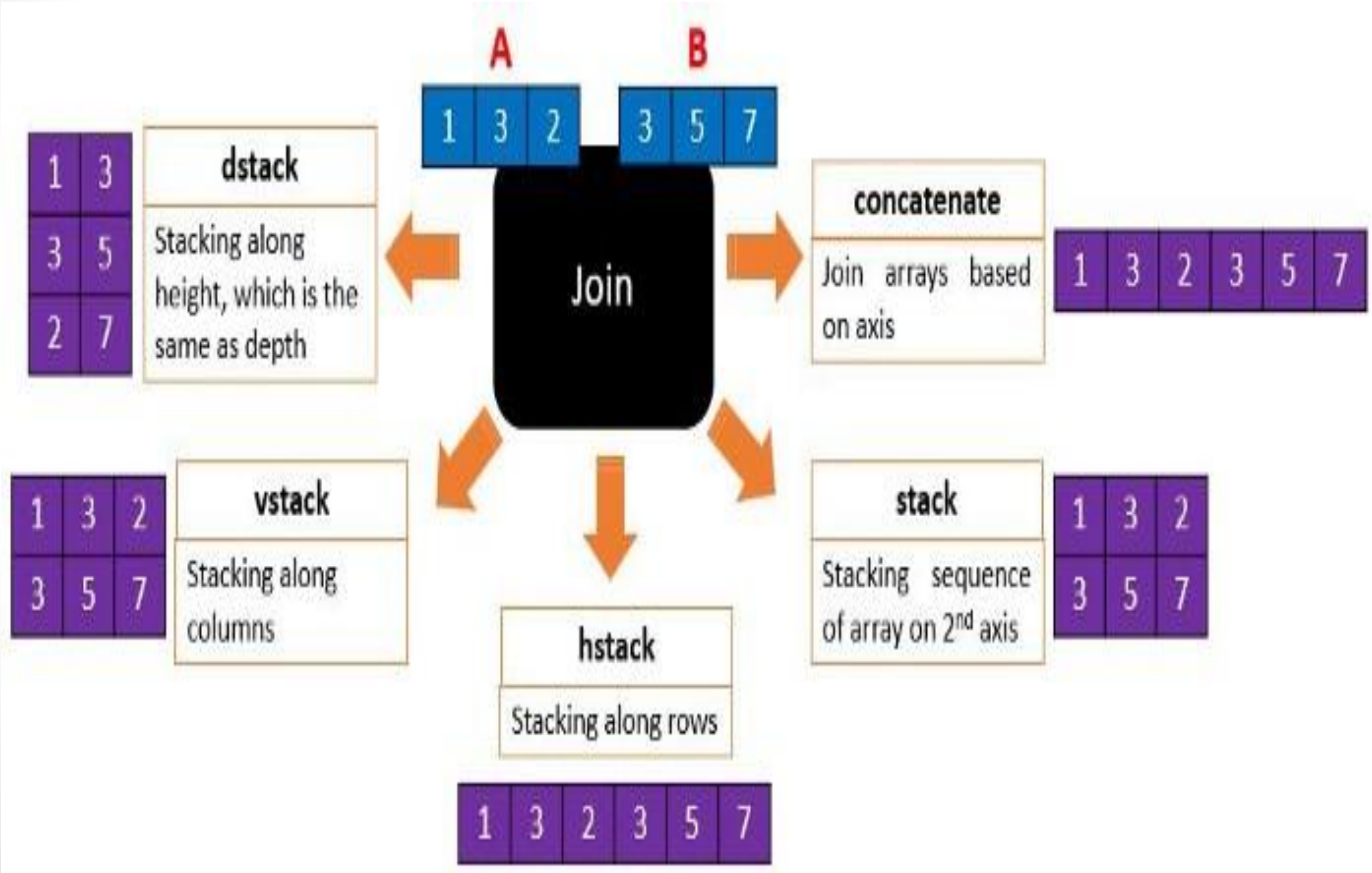
np.ones((3,3))
1 1 1
1 1 1
1 1 1
itemsize
returns the size (in bytes) of element in the array
8
ndim
returns number of dimensions of array
2
dtype
returns data type of the elements in the array
int32
size
returns number of elements in the array
9
shape
returns the size of the array
(3,3)

JOIN, SPLIT, SEARCH AND SORT

Joining

- Joining means putting contents of two or more arrays in a single array.
- In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.
- We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Join


A
B
1 3 2
3 5 7
Join
dstack
Stacking along height, which is the same as depth
1 3
3 5
2 7
concatenate
Join arrays based on axis
1 3 2 3 5 7
vstack
Stacking along columns
1 3 2
3 5 7
hstack
Stacking along rows
1 3 2 3 5 7
stack
Stacking sequence of array on 2nd axis
1 3 2
3 5 7

Join two arrays

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

Output:**[1 2 3 4 5 6]**

Join two 2-D arrays along rows (axis=1)

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

Output:**[[1 2 5 6] [3 4 7 8]]**

stack( )

- Stacking is same as concatenation, the only difference is that stacking is done along a new axis.
- We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.
- We pass a sequence of arrays that we want to join to the stack() method along with the axis. If axis is not explicitly passed it is taken as 0.

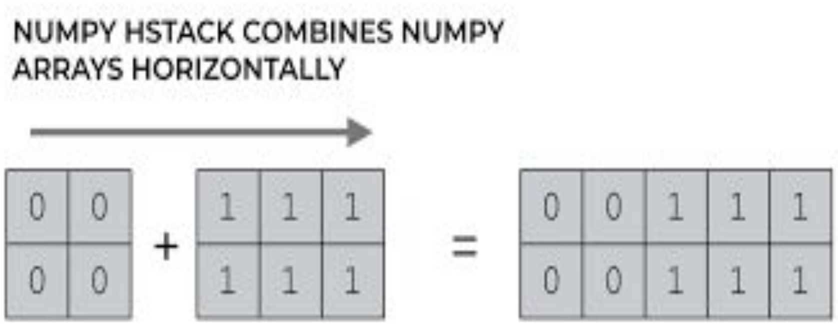
stack( )

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=1)

print(arr)
```

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

hstack( ) - Stacking Along Rows

- NumPy provides a helper function: hstack() to stack along rows.




```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.hstack((arr1, arr2))

print(arr)
```

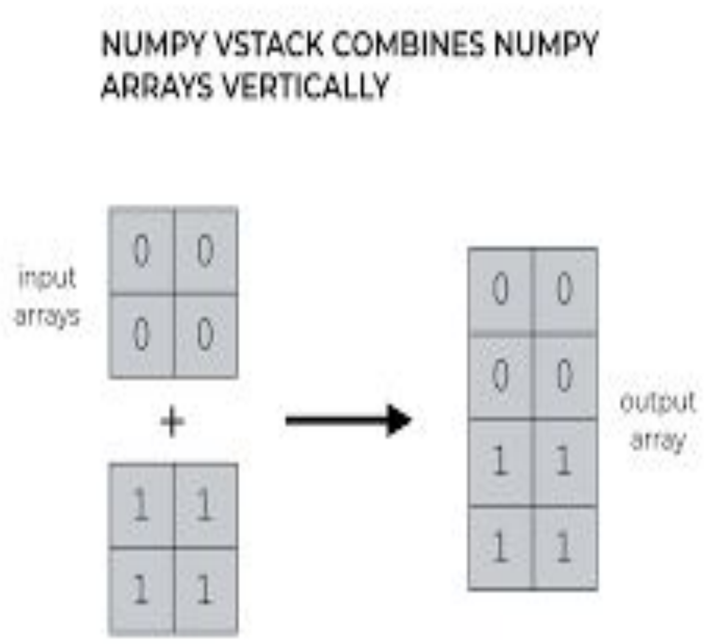
Output: **[1 2 3 4 5 6]**

vstack( ) - Stacking Along Columns

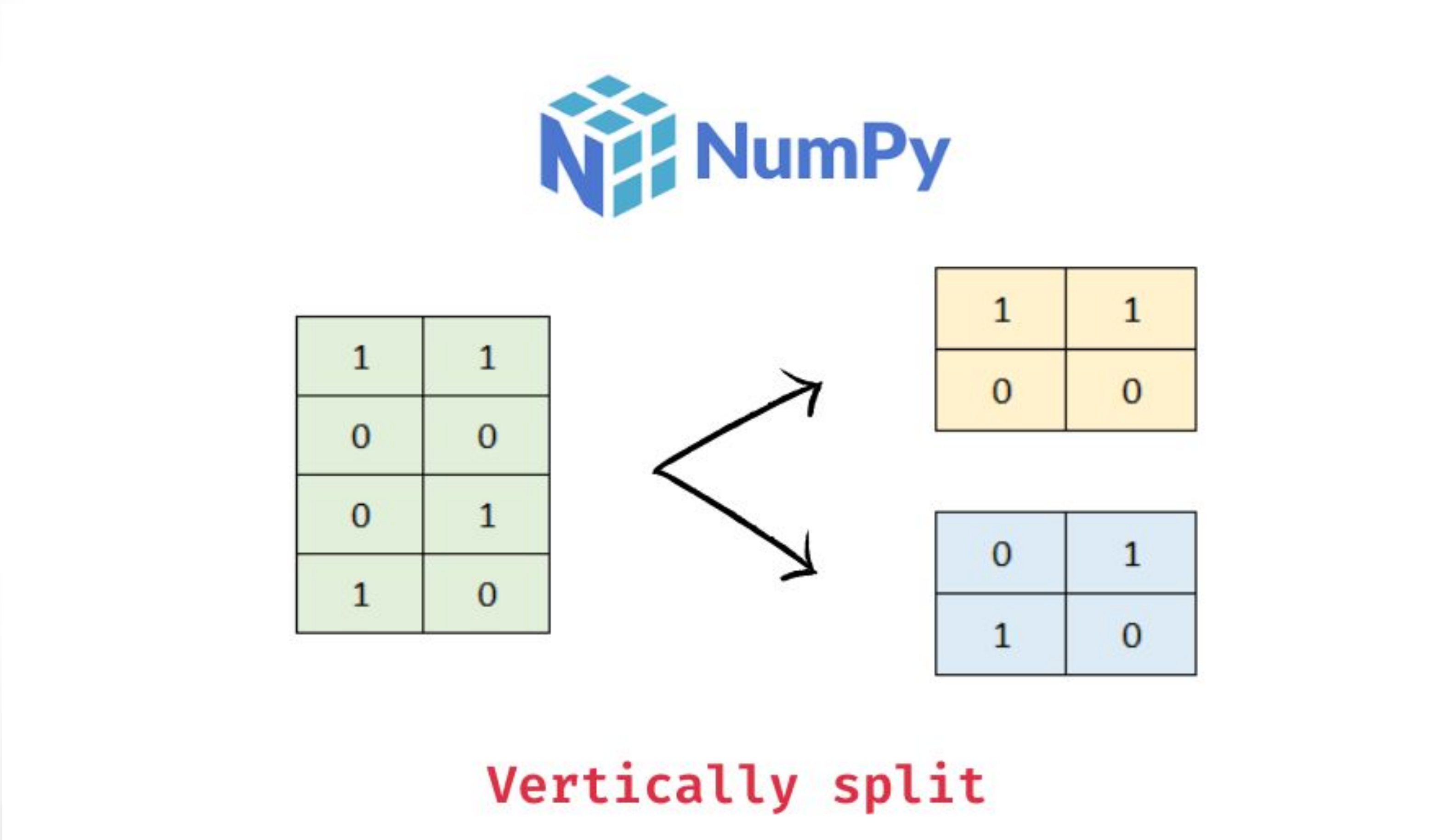
- NumPy provides a helper function: vstack() to stack along columns.

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.vstack((arr1, arr2

print(arr)
```

Output: **[[1 2 3] [4 5 6]]**



Split

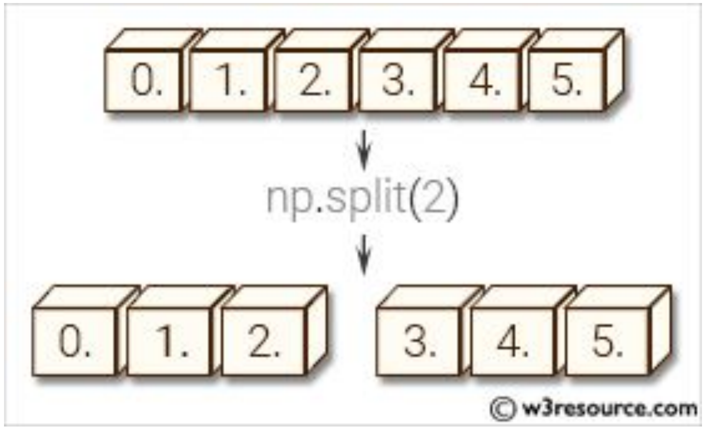


Splitting

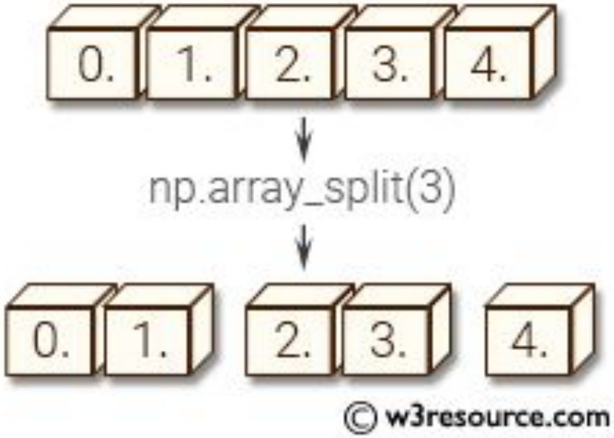
- Splitting is reverse operation of Joining.
- Joining merges multiple arrays into one and Splitting breaks one array into multiple.

Split

- **split( )**

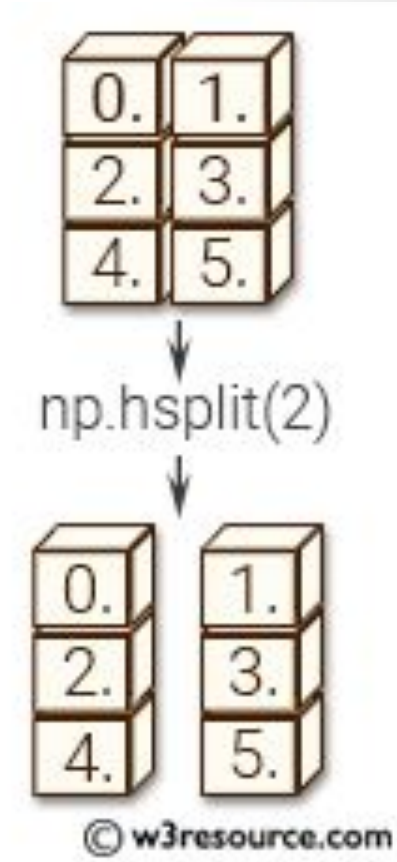


- **array_split( )**

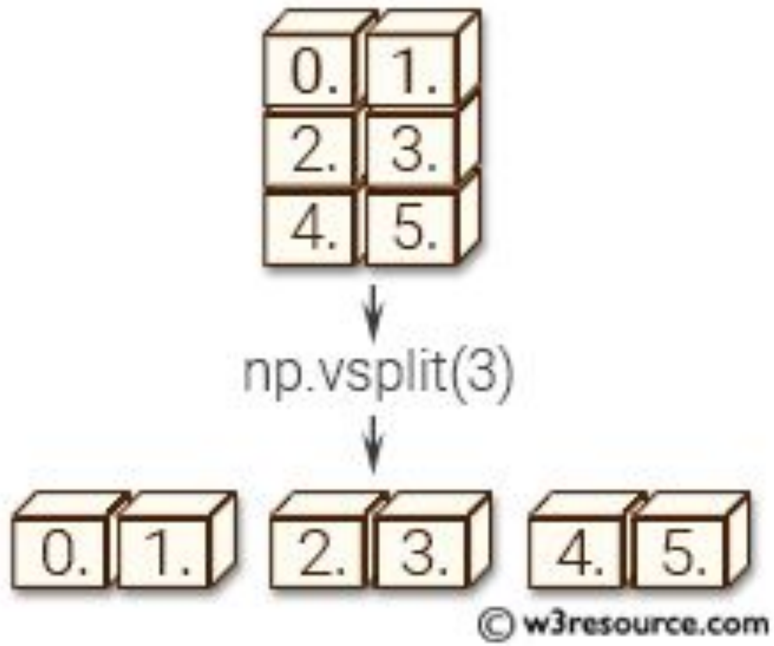


Split

- **hsplit( )**



- **vsplit( )**



Splitting

- We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.
- **Note:** The return value is an array containing three arrays.
- If the array has less elements than required, it will adjust from the end accordingly.

Note: We also have the method **split()** available but it will not adjust the elements when elements are less in source array for splitting like in example above, **array_split()** worked properly but **split()** would fail.

array_split( )

```
import numpy as np

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

newarr = np.array_split(arr, 3)

print(newarr)
```

Output:

[array([1, 2]), array([3, 4]), array([5, 6])]

split( )

```
import numpy as np

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

newarr = np.split(arr, 3)

print(newarr)
```

Output:

[array([1, 2]), array([3, 4]), array([5, 6])]

array_split( )

```
import numpy as np

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

newarr = np.array_split(arr, 4)

print(newarr)
```

Output:**[array([1, 2]), array([3, 4]), array([5]), array([6])]**

split( )

```
import numpy as np

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

newarr = np.split(arr, 4)

print(newarr)
```

Output:

Error

Split Into Arrays

- The return value of the array_split() method is an array containing each of the split as an array.

```
import numpy as np

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

newarr = np.array_split(arr, 3)

print(newarr[0])
print(newarr[1])
print(newarr[2])
```

Output:**[1 2] [3 4] [5 6]**

Splitting 2-D Arrays

- Use the same syntax when splitting 2-D arrays.
- Use the array_split() method, pass in the array you want to split and the number of splits you want to do.

```
import numpy as np

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

newarr = np.array_split(arr, 3)

print(newarr)
```

Output:**[array([[1, 2], [3, 4]]), array([[5, 6], [7, 8]]), array([[9, 10], [11, 12]])]**

hsplit( ) and vsplit( )

- The hsplit() function is used to **split an array into multiple sub-arrays horizontally** (column-wise).
- hsplit is equivalent to split with axis=1, the array is always split along the second axis regardless of the array dimension.
- The vsplit() function is used to **split an array into multiple sub-arrays vertically** (row-wise).
- Note: vsplit is equivalent to split with axis=0 (default), the array is always split along the first axis regardless of the array dimension.

hsplit

```
import numpy as np

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

newarr = np.hsplit(arr, 3)

print(newarr)
```

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

vsplit

```
import numpy as np

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

newarr = np.vsplit(arr, 3)

print(newarr)
```

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

Searching

- You can search an array for a certain value, and return the indexes that get a match.
- To search an array, use the where() method.

Find the indexes where the value is 4

```
import numpy as np

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

x = np.where(arr == 4)

print(x)
```

Output:(array([3, 5, 6]),)

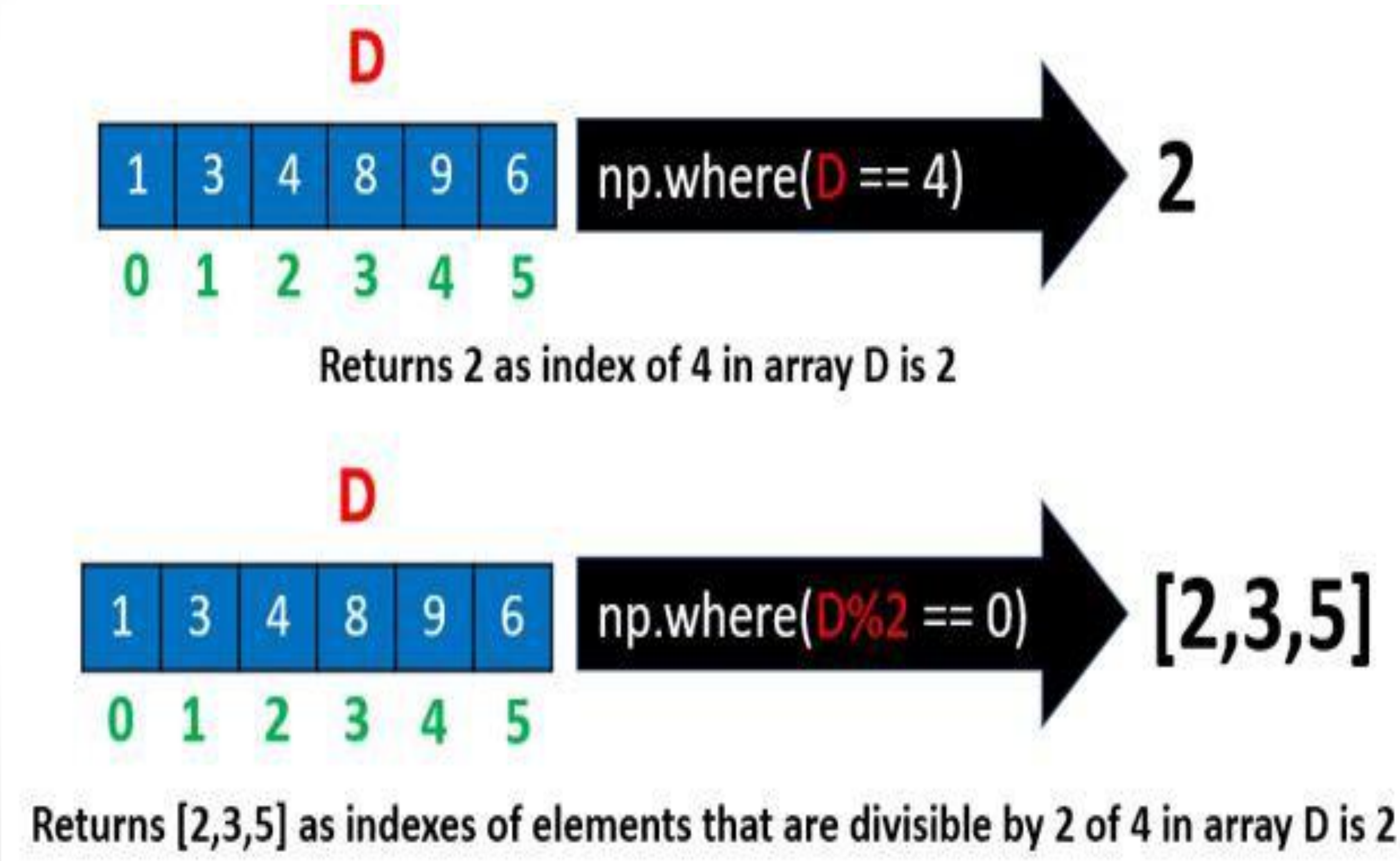
Sorting

- Sorting means putting elements in an *ordered sequence*.
- *Ordered sequence* is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
- The NumPy ndarray object has a function called sort(), that will sort a specified array.
- **Note:** This method returns a copy of the array, leaving the original array unchanged.

```
import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

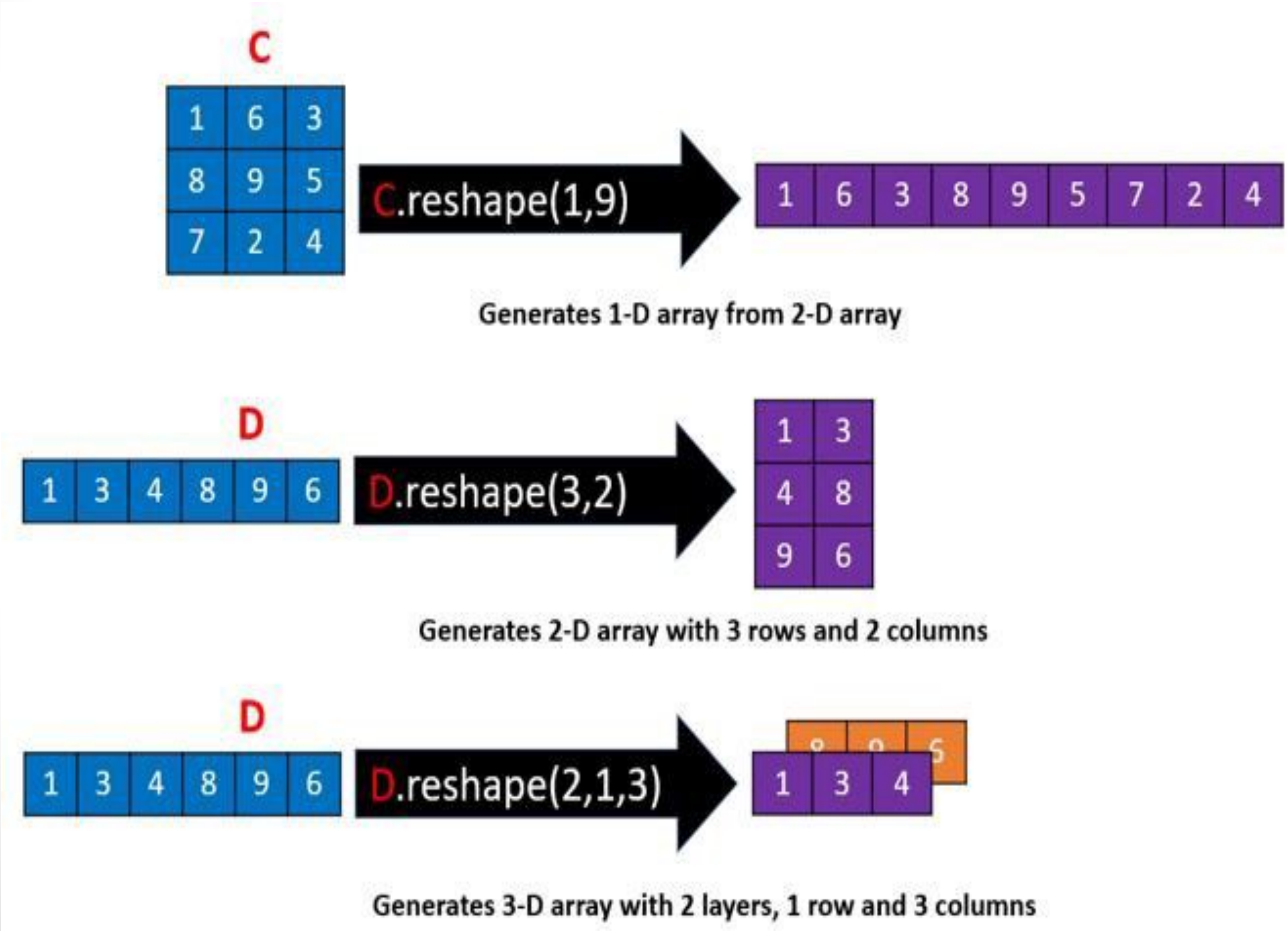
Output:[0 1 2 3]

Search


D
1 3 4 8 9 6
np.where(D == 4)
2
0 1 2 3 4 5
Returns 2 as index of 4 in array D is 2
D
1 3 4 8 9 6
np.where(D%2 == 0)
[2,3,5]
0 1 2 3 4 5
Returns [2,3,5] as indexes of elements that are divisible by 2 of 4 in array D is 2

ARRAY SHAPE MANIPULATION

Reshaping Array



List and Array

List	Array
List can have elements of different data types for example, [1,3.4, 'hello', 'a@']	All elements of an array are of same data type for example, an array of floats may be: [1.2, 5.4, 2.7]
Elements of a list are not stored contiguously in memory.	Array elements are stored in contiguous memory locations. This makes operations on arrays faster than lists.
Lists do not support element wise operations, for example, addition, multiplication, etc. because elements may not be of same type.	Arrays support element wise operations. For example, if A1 is an array, it is possible to say A1/3 to divide each element of the array by 3.
Lists can contain objects of different datatype that Python must store the type information for every element along with its element value. Thus lists take more space in memory and are less efficient.	NumPy array takes up less space in memory as compared to a list because arrays do not require to store datatype of each element separately.
List is a part of core Python.	Array (ndarray) is a part of NumPy library.

Linear Algebra with Numpy

Linear Algebra Module

- The Linear Algebra module of NumPy offers various methods to apply linear algebra on any numpy array. One can find:
  - **rank, determinant, trace, etc. of an array.**
  - **eigen values of matrices**
  - **matrix and vector products (dot, inner, outer,etc. product), matrix exponentiation**
  - **solve linear or tensor equations and much more!**

Example

```
# Importing numpy as np
import numpy as np

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

# Rank of a matrix
print("Rank of A:", np.linalg.matrix_rank(A))

# Trace of matrix A
print("\nTrace of A:", np.trace(A))

# Determinant of a matrix
print("\nDeterminant of A:", np.linalg.det(A))

# Inverse of matrix A
print("\nInverse of A:\n", np.linalg.inv(A))

print("\nMatrix A raised to power 3:\n",
      np.linalg.matrix_power(A, 3))
```

Output

Rank of A: 3
Trace of A: 11
Determinant of A: -306.0
Inverse of A:
[[0.17647059 -0.00326797 -0.02287582]
[0.05882353 -0.13071895 0.08496732]
[-0.11764706 0.1503268 0.05228758]]

Matrix A raised to power 3:
[[336 162 228]
[406 162 469]
[698 702 905]]

Eigen values and Eigen vectors

Eigen values and vectors

- **The Python Numpy linear algebra package can find the eigenvalues and eigenvectors of a matrix.**
- We calculate the eigenvalues and eigenvectors of the matrix

```
import numpy as np
from numpy import linalg as LA
A = np.array([[1,2,3],[3,2,1],[1,0,-1]])
w, v = LA.eig(A)
print(w)
[ 4.31662479e+00 -2.31662479e+00 3.43699053e-17] print(v)
[[ 0.58428153 0.73595785 0.40824829] [ 0.80407569
-0.38198836 -0.81649658] [ 0.10989708 -0.55897311
0.40824829]]
```

Eigen values and vectors

- **The numpy.linalg.eig function returns a tuple consisting of a vector and an array**
- **The vector (here w) contains the eigenvalues.**
- **The array (here v) contains the corresponding eigenvectors, one eigenvector per column.**
- **The eigenvalue w[0] goes with the 0th column of v. The eigenvalue w[1] goes with column 1, etc.**
- **To extract the *i*th column vector, we use**
    - ***u = v[:,i]***

Eigen values

```
# importing numpy library
import numpy as np

# create numpy 2d-array
m = np.array([[1, 2], [2, 3]])

print("Printing the Original square array:\n",m)

# finding eigenvalues and eigenvectors
w, v = np.linalg.eig(m)

# printing eigen values
print("Printing the Eigen values of the given square array:\n", w)

# printing eigen vectors
print("Printing Right eigenvectors of the given square array:\n“,v)
```

Eigen values

Printing the Original square array:
[[1 2]
[2 3]]

Printing the Eigen values of the given square array:
[-0.23606798 4.23606798]

Printing Right eigenvectors of the given square array:
[[-0.85065081 -0.52573111]
[0.52573111 -0.85065081]]

Eigen vectors

```
# importing numpy library
import numpy as np

# create numpy 2d-array
m = np.array([[1, 2, 3],[2, 3, 4],[4, 5, 6]])

print("Printing the Original square array:\n",m)

# finding eigenvalues and eigenvectors
w, v = np.linalg.eig(m)

# printing eigen values
print("Printing the Eigen values of the given square array:\n",w)

# printing eigen vectors
print("Printing Right eigenvectors of the given square array:\n",v)
```

Eigen vectors

Printing the Original square array:

[[1 2 3]

[2 3 4]

[4 5 6]]

Printing the Eigen values of the given square array:

[1.08309519e+01 -8.30951895e-01 1.01486082e-16]

Printing Right eigenvectors of the given square array:

[[0.34416959 0.72770285 0.40824829]

[0.49532111 0.27580256 -0.81649658]

[0.79762415 -0.62799801 0.40824829]]

Numpy Random

Data Distribution, Normal, Exponential, Binomial, Poisson, Uniform and ChiSquare distributions.

Random

- Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

- **Pseudorandom**
- Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well.
- If there is a program to generate random number it can be predicted, thus it is not truly random.
- Random numbers generated through a generation algorithm are called *pseudo random*.

Random

- Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

- **Truerandom**
- In order to generate a truly random number on our computers we need to get the random data from some outside source. This outside source is generally our keystrokes, mouse movements, data on network etc.
- We do not need truly random numbers, unless its related to security (e.g. encryption keys) or the basis of application is the randomness (e.g. Digital roulette wheels).

Generate Random number

```
from numpy import random

x = random.randint(100)

print(x)
```

Output:45

Generate Random Float

```
from numpy import random

x = random.rand()

print(x)
```

Output:0.20589891226659818

Generate Random Array

- In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

Integers

- The randint() method takes a size parameter where you can specify the shape of an array.

```
from numpy import random

x=random.randint(100, size=(5))

print(x)
```

Output:[61 66 32 13 16]

Data Distribution

- Data Distribution is a list of all possible values, and how often each value occurs.
- Such lists are important when working with statistics and data science.
- The random module offer methods that returns randomly generated data distributions.

Random Distribution

- A random distribution is a set of random numbers that follow a certain *probability density function*.
- **Probability Density Function:** A function that describes a continuous probability. i.e. probability of all values in an array.
- We can generate random numbers based on defined probabilities using the choice() method of the random module.
- The choice() method allows us to specify the probability for each value.
- The probability is set by a number between 0 and 1, where 0 means that the value will never occur and 1 means that the value will always occur.

Example

- Generate a 1-D array containing 100 values, where each value has to be 3, 5, 7 or 9.
- The probability for the value to be 3 is set to be 0.1
- The probability for the value to be 5 is set to be 0.3
- The probability for the value to be 7 is set to be 0.6
- The probability for the value to be 9 is set to be 0

```
from numpy import random

x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0],
size=(100))

print(x)
```

Example

[3 7 5 7 7 7 5 7 3 7 3 7 7 7 7 5 7 5 7 7 7 7 7 5 3 7 5 7 7 7 3 5 3 7 5 7 7 5 7 5 7 5 7 7 5 7 7 3 5 3 5 7 7 5 7 7 7 7 5 7 5 3 5 5 7 7 7 3 7 7 7 7 5 7 7 5 7 7 7 7 7 7 5 5 3 5 5 7 5 5 7 7 5 3 3 7 7 5 7 7]

Example

- You can return arrays of any shape and size by specifying the shape in the **size** parameter.
- Same example as above, but return a 2-D array with 3 rows, each containing 5 values.

```
from numpy import random
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0],
size=(3, 5))
print(x)
```

```
[[7 7 7 7 7]
 [5 3 5 7 5]
 [5 7 5 7 5]]
```

Normal Distribution

- The Normal Distribution is one of the most important distributions.
- It is also called the Gaussian Distribution after the German mathematician **Carl Friedrich Gauss**.
- It fits the probability distribution of many events, eg. IQ Scores, Heartbeat etc.
- Use the random.normal() method to get a Normal Data Distribution.
- It has three parameters:
  - loc - (Mean) where the peak of the bell exists.
  - scale - (Standard Deviation) how flat the graph distribution should be.
  - size - The shape of the returned array.

Normal Distribution

```
from numpy import random

x = random.normal(size=(2, 3))

print(x)
```

Output:

Run1:

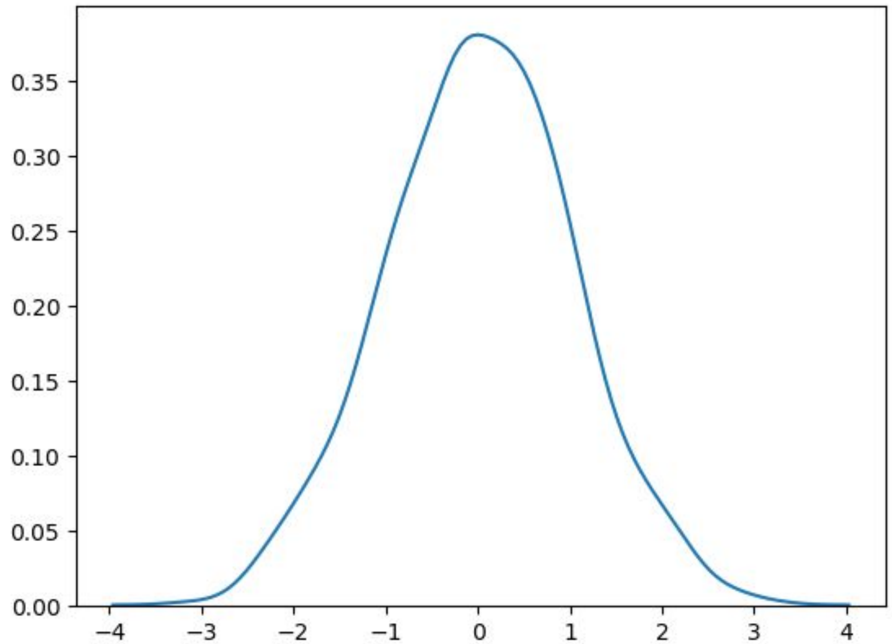
[[0.15001821 -1.31355388 -1.35020654] [-1.31067087 -0.48537757 -0.02052509]]

Run2:

[[-2.0610908 -0.3081812 0.99886608] [0.56001902 0.38363428 -0.07954767]]

Visualization of Normal Distribution

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.normal(size=1000), hist=False)
plt.show()
```



Exponential Distribution

- Exponential distribution is used for describing time till next event e.g. failure/success etc.
- It has two parameters:
  - scale - inverse of rate (see lam in poisson distribution) defaults to 1.0.
  - size - The shape of the returned array.

Exponential Distribution

Time Between Customers

- The number of minutes between customers who enter a certain shop can be modeled by the exponential distribution.
- For example, suppose a new customer enters a shop every two minutes, on average. After a customer arrives, find the probability that a new customer arrives in less than one minute.

To solve this, we can start by knowing that the average time between customers is two minutes. Thus, the rate can be calculated as:

  - $\lambda = 1/\mu$
  - $\lambda = 1/2$
  - $\lambda = 0.5$
- We can plug in $\lambda = 0.5$ and $x = 1$ to the formula for the CDF:
  - $P(X \leq x) = 1 - e^{-\lambda x}$
  - $P(X \leq 1) = 1 - e^{-0.5(1)}$
  - $P(X \leq 1) = 0.3935$

The probability that we'll have to wait less than one minute for the next customer to arrive is 0.3935.

Exponential Distribution

- Draw out a sample for exponential distribution with 2.0 scale with 2x3 size:

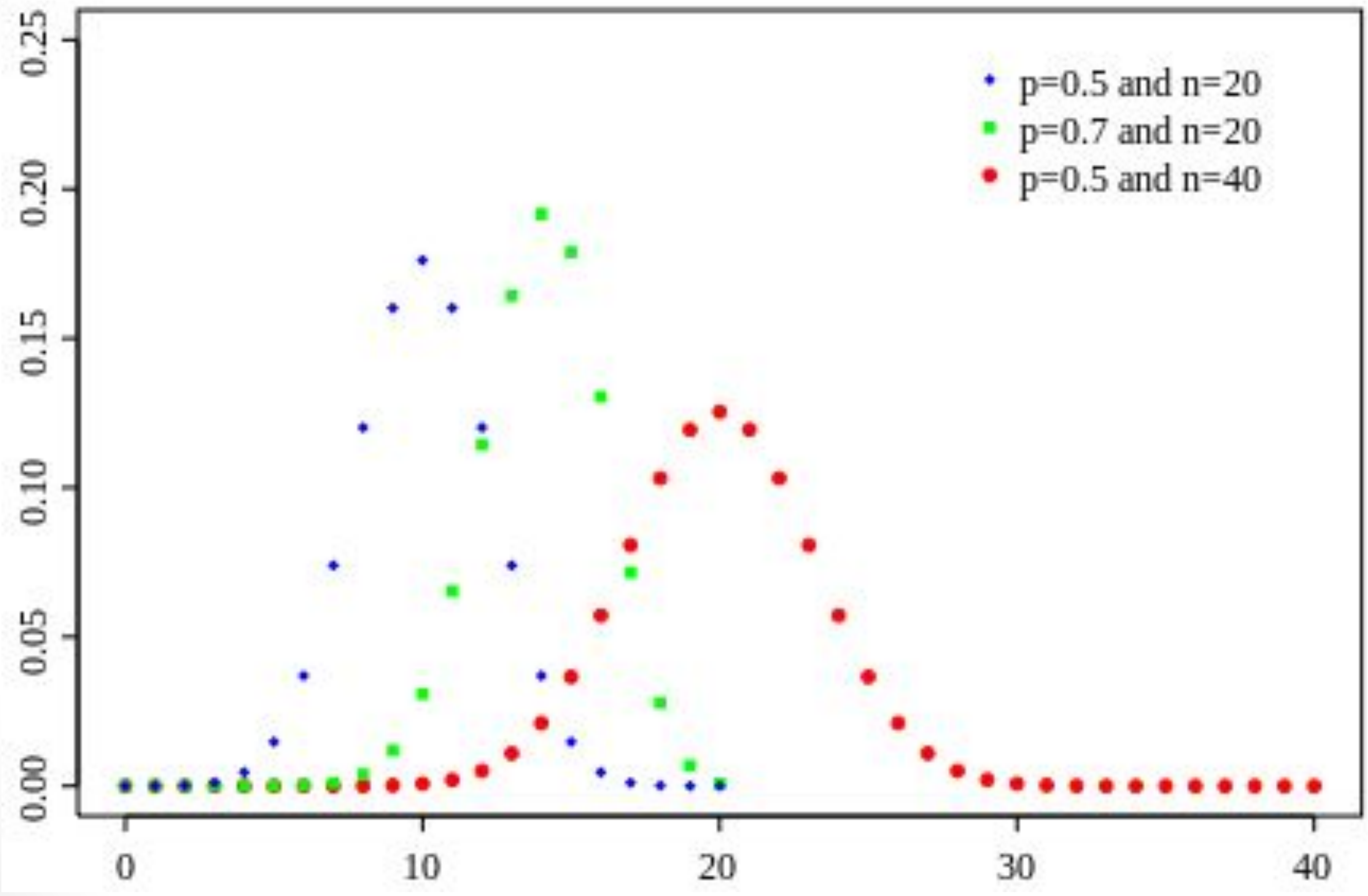
```
from numpy import random

x = random.exponential(scale=2, size=(2, 3))

print(x)
```

Output:[[0.16401759 5.71219287 1.20149124]
[2.51527074 2.13596927 1.04229153]]

Binomial Distribution

- A **binomial distribution** can be thought of as simply the probability of a **SUCCESS or FAILURE** outcome in an experiment or survey that is repeated multiple times.
- The binomial is a type of distribution that has **two possible outcomes** (the prefix "bi" means two, or twice).
  - For example, a coin toss has only two possible outcomes: **heads or tails** and taking a test could have two possible outcomes: **pass or fail**.

Binomial Distribution



Example

- For example, let's suppose you wanted to know the probability of getting a 1 on a die roll. if you were to roll a die 20 times, the probability of rolling a one on any throw is 1/6. Roll twenty times and you have a binomial distribution of (n=20, p=1/6). SUCCESS would be "roll a one" and FAILURE would be "roll anything else."
- If the outcome in question was the probability of the die landing on an even number, the binomial distribution would then become (n=20, p=1/2). That's because your probability of throwing an even number is one half.
-

Binomial Distribution

- Binomial Distribution is a *Discrete Distribution*.
- It describes the outcome of binary scenarios, e.g. toss of a coin, it will either be head or tails.
- It has three parameters:
  - n - number of trials.
  - p - probability of occurence of each trial (e.g. for toss of a coin 0.5 each).
  - size - The shape of the returned array.
- **Discrete Distribution:**The distribution is defined at separate set of events, e.g. a coin toss's result is discrete as it can be only head or tails whereas height of people is continuous as it can be 170, 170.1, 170.11

Binomial Distribution

- Given 10 trials for coin toss generate 10 data points:

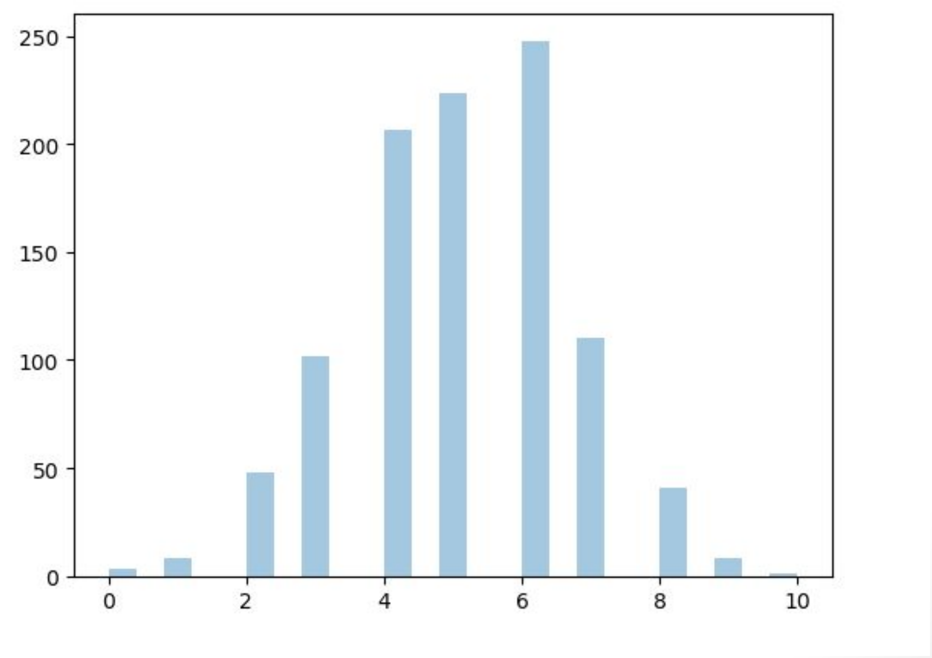
```
from numpy import random

x = random.binomial(n=10, p=0.5, size=10)

print(x)
```

Output:[5 7 6 5 4 7 5 4 6 5]

[5 3 6 4 3 3 3 5 5 5]

Visualization of Binomial Distribution

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.binomial(n=10, p=0.5, size=1000),
hist=True, kde=False)
plt.show()
```



Poisson Distribution

- Poisson Distribution is a *Discrete Distribution*.
- It estimates how many times an event can happen in a specified time. e.g. If someone eats twice a day what is probability he will eat thrice?

Poisson Probability Distribution applications

Numbers of Discrete events	Given period of time
People arriving	In an hour
Phone calls	In a day
Print jobs	In a minute

Poisson Distribution

Poisson Parameter

λ

Lambda

The rate

Number of events per time period

Mean = λ

Variance = λ

Standard deviation = $\sqrt{\lambda}$

Poisson Distribution

Poisson Distribution

Poisson Distribution

Poisson distribution

Numbers of Discrete events	Given period of time
Mice brought in	In a week

Poisson Distribution

Poisson distribution

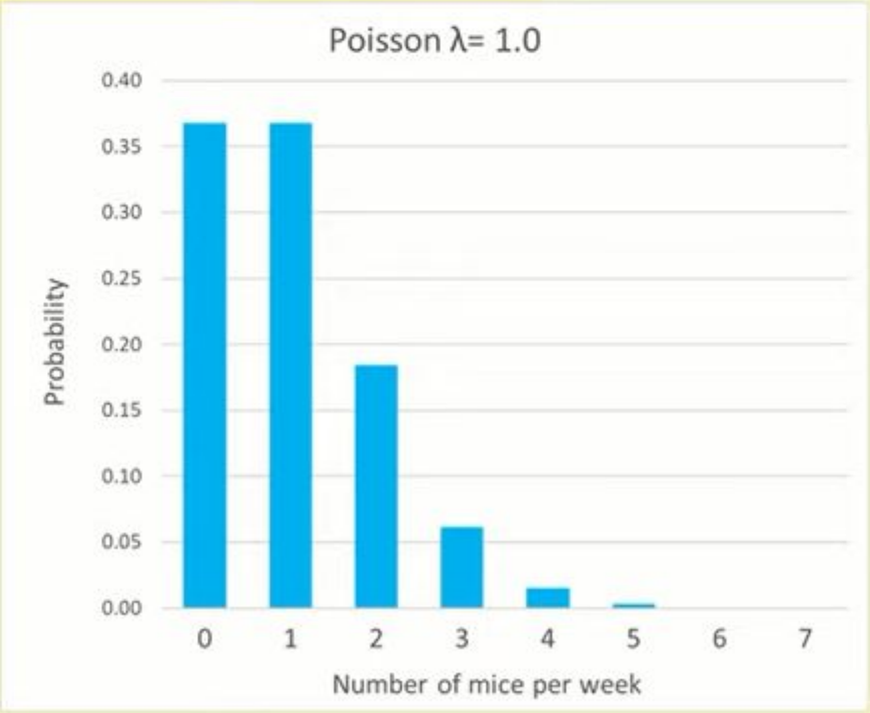
$$\lambda = 1$$

$$P(X = 4) = ?$$

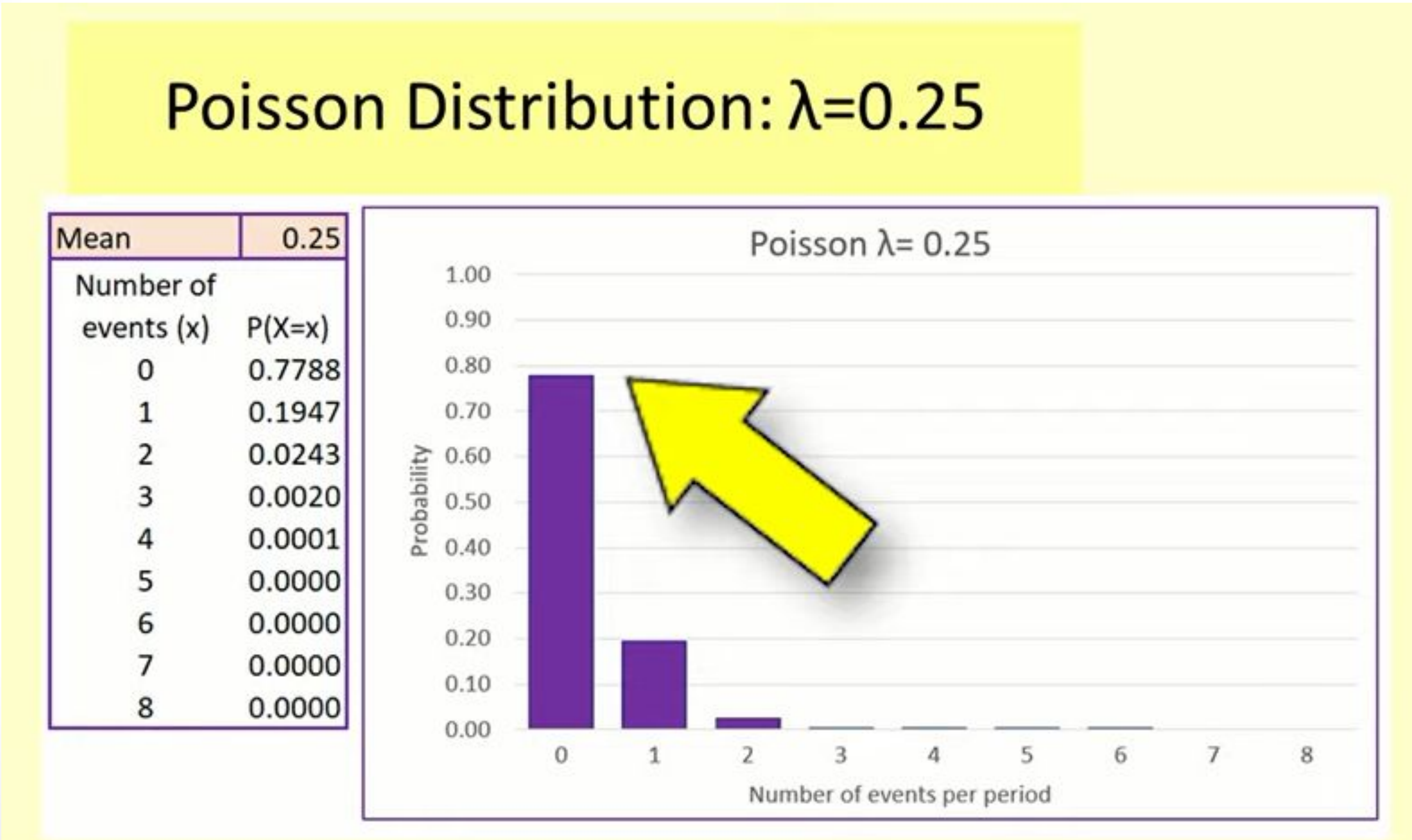
Poisson Distribution

Poisson Distribution: $\lambda=1$

Mean	1
Number of mice (x)	P(X=x)
0	0.368
1	0.368
2	0.184
3	0.061
4	0.015
5	0.003
6	0.001
7	0.000



Poisson Distribution

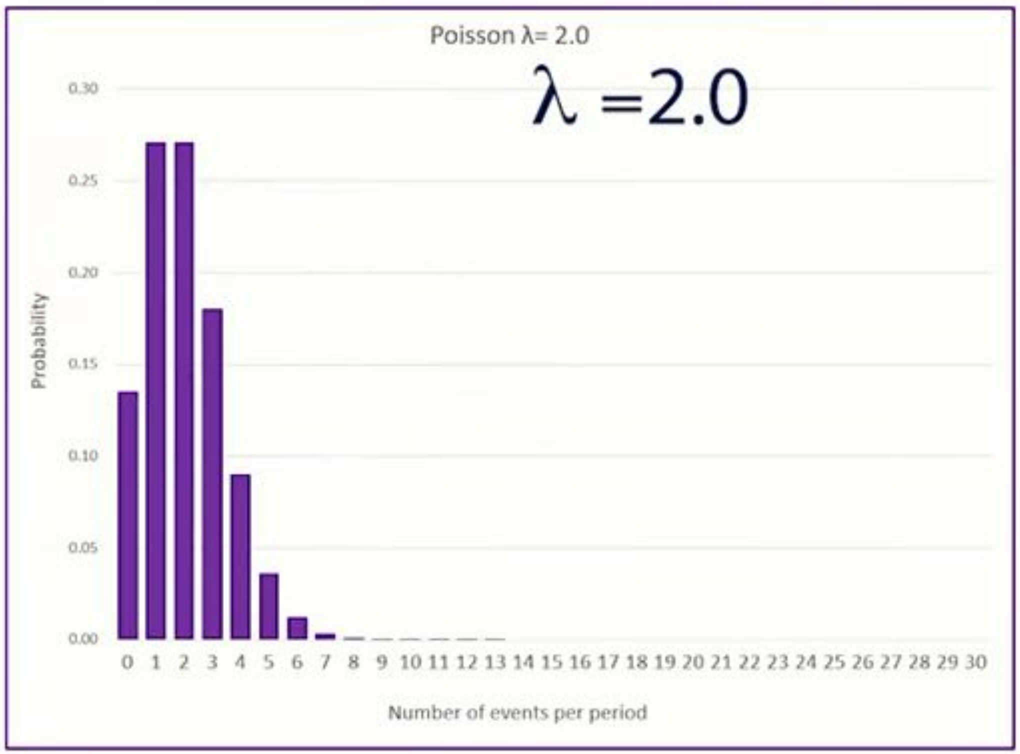


Mean	0.25
Number of events (x)	P(X=x)
0	0.7788
1	0.1947
2	0.0243
3	0.0020
4	0.0001
5	0.0000
6	0.0000
7	0.0000
8	0.0000

Poisson Distribution

Poisson Distribution: λ= Increasing

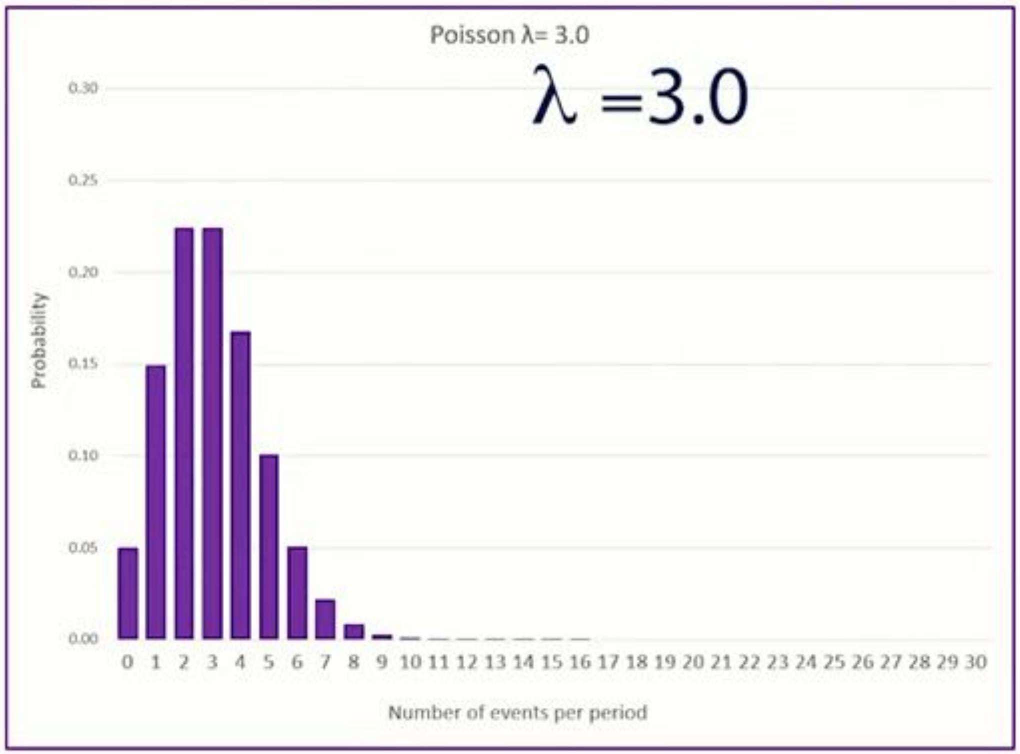
Mean	2
Number of events (x)	P(X=x)
0	0.1353
1	0.2707
2	0.2707
3	0.1804
4	0.0902
5	0.0361
6	0.0120
7	0.0034
8	0.0009
9	0.0002
10	0.0000
11	0.0000
12	0.0000
13	0.0000
14	0.0000
15	0.0000
16	0.0000
17	0.0000



Poisson Distribution

Poisson Distribution: λ= Increasing

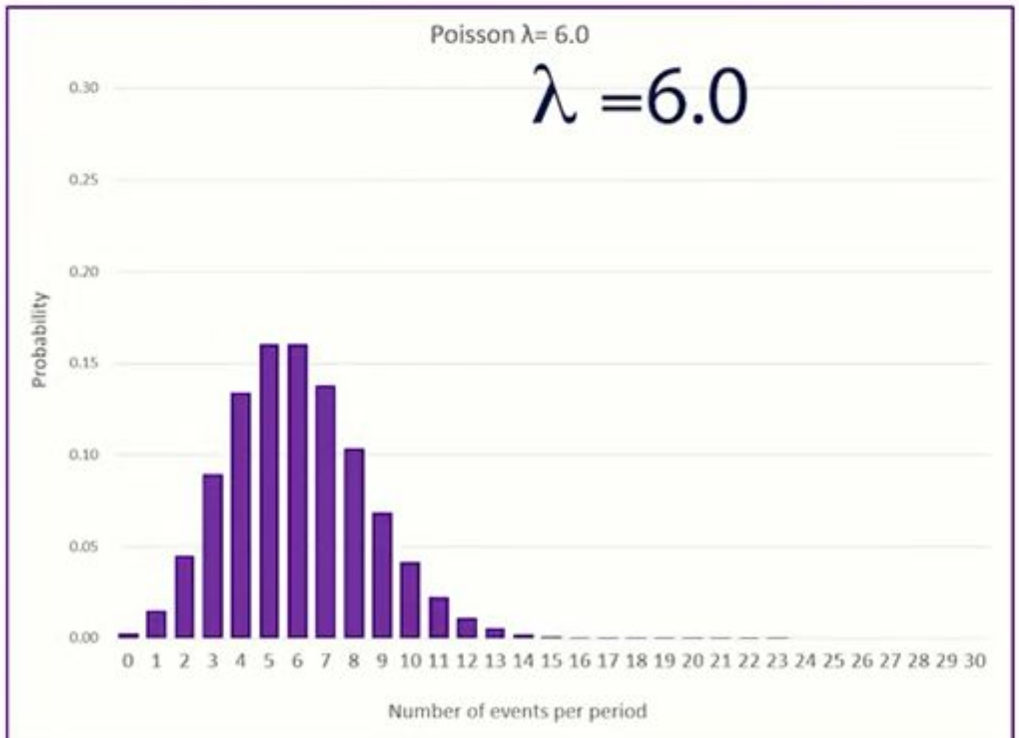
Mean	3
Number of events (x)	P(X=x)
0	0.0498
1	0.1494
2	0.2240
3	0.2240
4	0.1680
5	0.1008
6	0.0504
7	0.0216
8	0.0081
9	0.0027
10	0.0008
11	0.0002
12	0.0001
13	0.0000
14	0.0000
15	0.0000
16	0.0000
17	0.0000



Poisson Distribution

Poisson Distribution: λ= Increasing

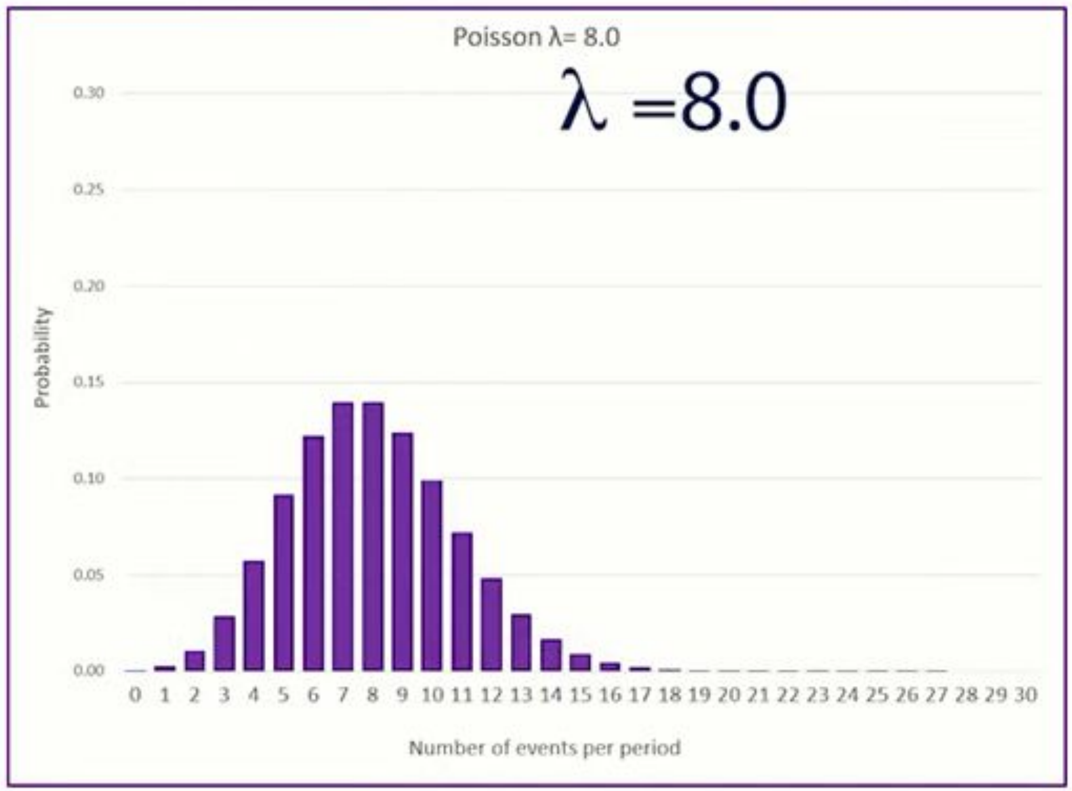
Mean	6
Number of events (x)	P(X=x)
0	0.0025
1	0.0149
2	0.0446
3	0.0892
4	0.1339
5	0.1606
6	0.1606
7	0.1377
8	0.1033
9	0.0688
10	0.0413
11	0.0225
12	0.0113
13	0.0052
14	0.0022
15	0.0009
16	0.0003
17	0.0001



Poisson Distribution

Poisson Distribution: λ= Increasing

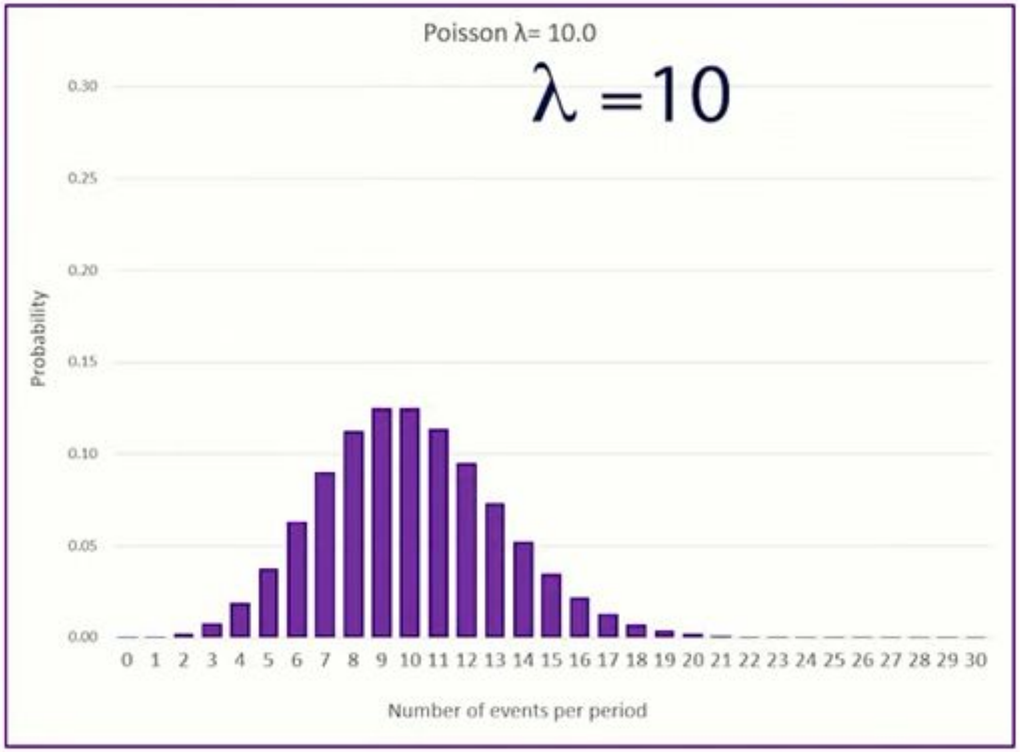
Mean	8
Number of events (x)	P(X=x)
0	0.0003
1	0.0027
2	0.0107
3	0.0286
4	0.0573
5	0.0916
6	0.1221
7	0.1396
8	0.1396
9	0.1241
10	0.0993
11	0.0722
12	0.0481
13	0.0296
14	0.0169
15	0.0090
16	0.0045
17	0.0021



Poisson Distribution

Poisson Distribution: λ= Increasing

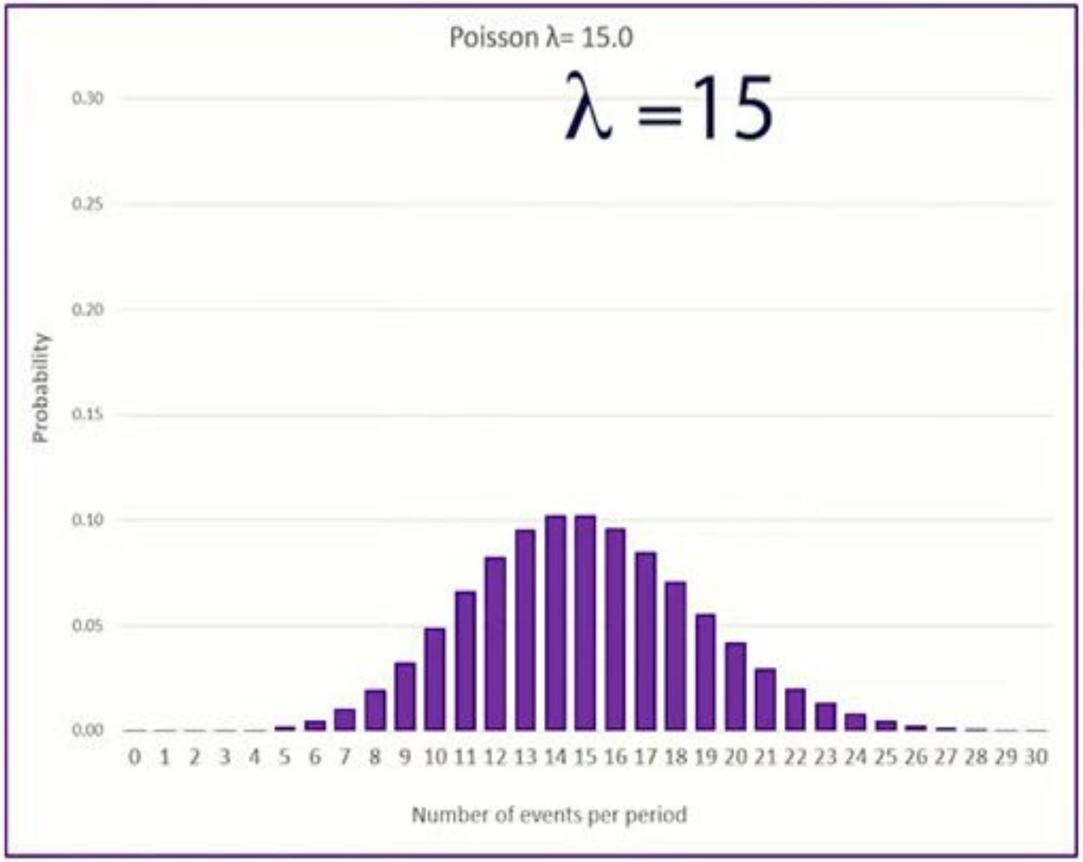
Mean	10
Number of events (x)	P(X=x)
0	0.0000
1	0.0005
2	0.0023
3	0.0076
4	0.0189
5	0.0378
6	0.0631
7	0.0901
8	0.1126
9	0.1251
10	0.1251
11	0.1137
12	0.0948
13	0.0729
14	0.0521
15	0.0347
16	0.0217
17	0.0128



Poisson Distribution

Poisson Distribution: λ= Increasing

Mean	15
Number of events (x)	P(X=x)
0	0.0000
1	0.0000
2	0.0000
3	0.0002
4	0.0006
5	0.0019
6	0.0048
7	0.0104
8	0.0194
9	0.0324
10	0.0486
11	0.0663
12	0.0829
13	0.0956
14	0.1024
15	0.1024
16	0.0960
17	0.0847



Poisson Distribution

- Poisson Distribution is a *Discrete Distribution*.
- It estimates how many times an event can happen in a specified time. e.g. If someone eats twice a day what is probability he will eat thrice?

- It has two parameters:
  - lam - rate or known number of occurences e.g. 2 for above problem.
  - size - The shape of the returned array.

Poisson Distribution

- Generate a random 1x10 distribution for occurence 2:

```
from numpy import random

x = random.poisson(lam=2, size=10)

print(x)
```

Uniform Distribution

- Used to describe probability where every event has equal chances of occurring.
- E.g. Generation of random numbers.

- It has three parameters:
    - a - lower bound - default 0 .0.
    - b - upper bound - default 1.0.
    - size - The shape of the returned array.

Uniform Distribution

- Create a 2x3 uniform distribution sample:

```
from numpy import random

x = random.uniform(size=(2, 3))

print(x)
```

Output:[[0.21295952 0.57512648 0.39384297]
[0.7543237 0.80233051 0.53264002]]

Chi Square Distribution

- Chi Square distribution is used as a basis to verify the hypothesis.
- It has two parameters:
  - df - (degree of freedom).
  - size - The shape of the returned array.

Chi Square Distribution

- Draw out a sample for chi squared distribution with degree of freedom 2 with size 2x3:

```
from numpy import random

x = random.chisquare(df=2, size=(2, 3))

print(x)
```

Output:**[[0.01738909 9.73650152 0.87953635]**
[0.14366152 0.98102103 2.72668685]]