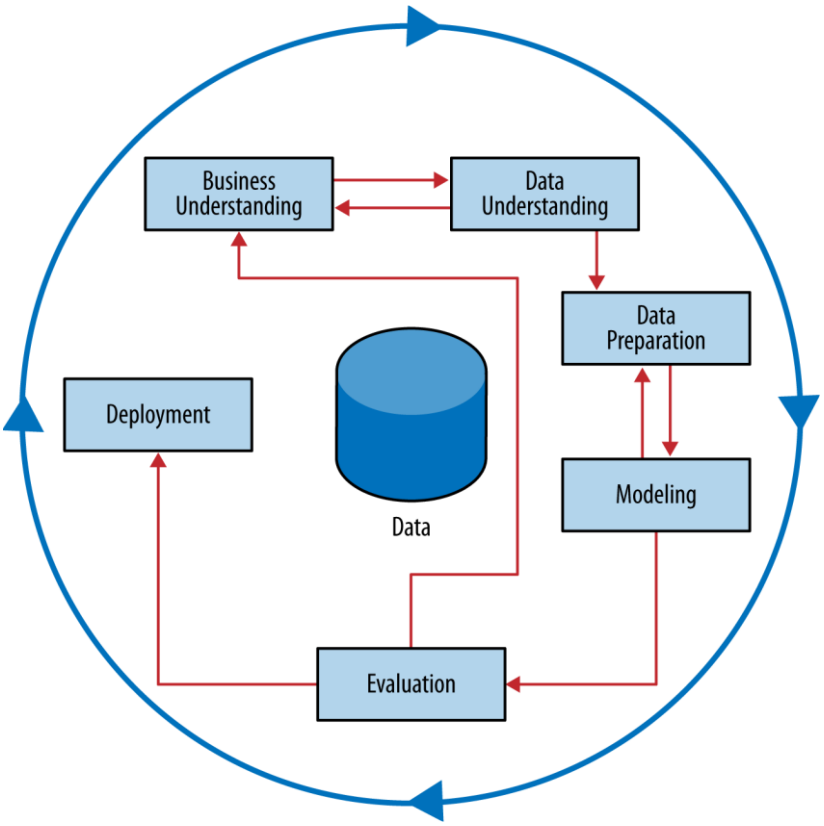
INTRODUCTION TO DATA ANALYTICS

Modern data ecosystem involves: [—*collect & store, process, and analyze data*]

1) Collect data by working with different —[Data format, Sources & Interfaces].
2) Organize, clean, and optimize data for easy access, conforming to the standards of application. (*ie*. data management & administer Enterprise Data Repository)
3) Utilize data from repository for specific usage [Interfaces /API] usage.
   (*ie*. business analysis, APPs, programmer analysis, or data science uses)



:: If there isn't a well defined operations to collect & manage data in the enterprise environment, it is hard to generate consistent data to support data science & mining activities.

Software skills vs. Analytics skills

In analytics, it's more important for individuals to be able to frame a problem *(what is best, how do I measure it & how do I find data to help me)*, to prototype solutions quickly, to make reasonable assumptions in face of ill-structured problems, to design experiments, and thus effectively translate into communication via visualizations.

… than designing clean & efficient computational instructions for a program.

Despite the large no. of data mining techniques developed over the years, there are only a handful of fundamentally different types of tasks these algorithm address: (use scenario)

1. Classification, (class probability estimation)
   - attempts to predict the class for each individual in a population based on previously labeled target data & correlated variables.

   (ie. To know if a given customer will stay for at least *6-months*, there must be a complete historical dataset that is balance & suffice, instead of an eager extrapolation from data less than *6-months*.)

   —"*Among all the customers, which are likely to respond to the given proposition?*"

2. Regression, (value estimation)
   - Likewise, it attemps to predict for each invidividual the continuous value of some variable.

   —"*How much will a given customer use the service?*"

3. Similarity matching,
   - attempts to identify similar individuals based on common features about them.

   —"*Finding people who are similar in terms of the products they have liked or purchased.*"

4. Clustering, (categorization)
   - attempts to group individuals of a population by their natural similarities, not driven by any specific purpose. (as preliminary domain exploration)

   —"*Without a desired outcome, do the customers naturally fall into different groups?*"

5. Co-occurrence grouping, (association rules)
   - attempts to find associations btw entities based on their appearing together in transactions.

   —"*What items are commonly purchased together?*"

6. Characterizing, (behavior description)
   - attempts to characterize the behavior norms / pattern from historical data, and take measures on the degree of irregularity as anomaly detection.

   —"*What is the typical behavioral norm / anomaly event? ; When / How it usually happens?* "

7. Discovering connections,
   - attempts to estimate the strength of connections btw entites, and suggest if a link should exist.

   —"*Since it is not watched, may be a customer who watched `Alien` would also like `Conjuring`?*"

8. Data reduction, (dimensionality-reduction)
   - attempts to trade-off loss of information in a large set of data for improved insight.

9. Causal inference, (A/B test)
   - conducts testing on the hypothesis made for certain pattern in data is actually correct.

   —"*When undertaking causal inference, a business needs to weigh the trade-off of increasing the investment to reduce the assumption made at higher evidence (sample size), versus deciding that the conclusions are good enough given the assumption.*" (see. A/B test)

* **In all cases, a careful DS should always incl. a causal conclusion the exact assumption that must be made in order for the causal conclusion to hold.** (eg. The information campaign is effective in promoting public health)

— A good problem-solving begins with even better questions.

" AI generally do not possess the intuition to understand the ambiguity of a problem thru context;
No matter how much information there is or advanced your tools are, it won't tell you much w/o the leading Qs. "

Effective Questioning with SMART methodology

▶ Q: " What features do people look for when buying a new car? "

Applicable to:
• **Survey** • **Data investigations**

(Refer. Business_and_DataUnderstanding_with_SMART)

(Specific), simple & focus on a single topic / feature at a time

✗ This product is too expensive, isn't it? (leading -suggests an answer as part of question)

💬 How does a car having 4-wheel drive contribute to its value, in your opinion?

(Measurable), quantified & assessed

✗ Were you satisfied with the customer trial? (close-ended -doesn't encourage extension)

💬 On a scale of 1-10, how important is your car having four-wheel drive? Explain.

(Action-oriented), encourage answers one can act on

✗ Does the feature excites you? (vague with no context -subject/diff in comparison)

💬 What features, if included with 4-wheel drive, would make you more inclined to buy the car?

(Relevant), significant to the goal && (Time-bound), limit range & meaningful time to be studied

✗ Why does it matter that a 4-wheel drive feature to be included in a car?
(it didn't help in finding ways to prevent these frogs from going extinct- project goal)

💬 What are the top five features you would like to see in a newest car package?

Note: Fairness should be ensured to avoid assumption & lead toward a certain answer [# bias].

Key Considerations in a business problem:

i. First, define the problem that needed to be solved, and collaborate with stakeholders to understand their needs &/ budget.
(eg. *Decision on best advertising method for a gaming accessories company to expand their business*)

ii. Identify company's target audience for collection of the right data to problem.
(eg. *Data of different methods to determine the most popular one with the company target audience*)

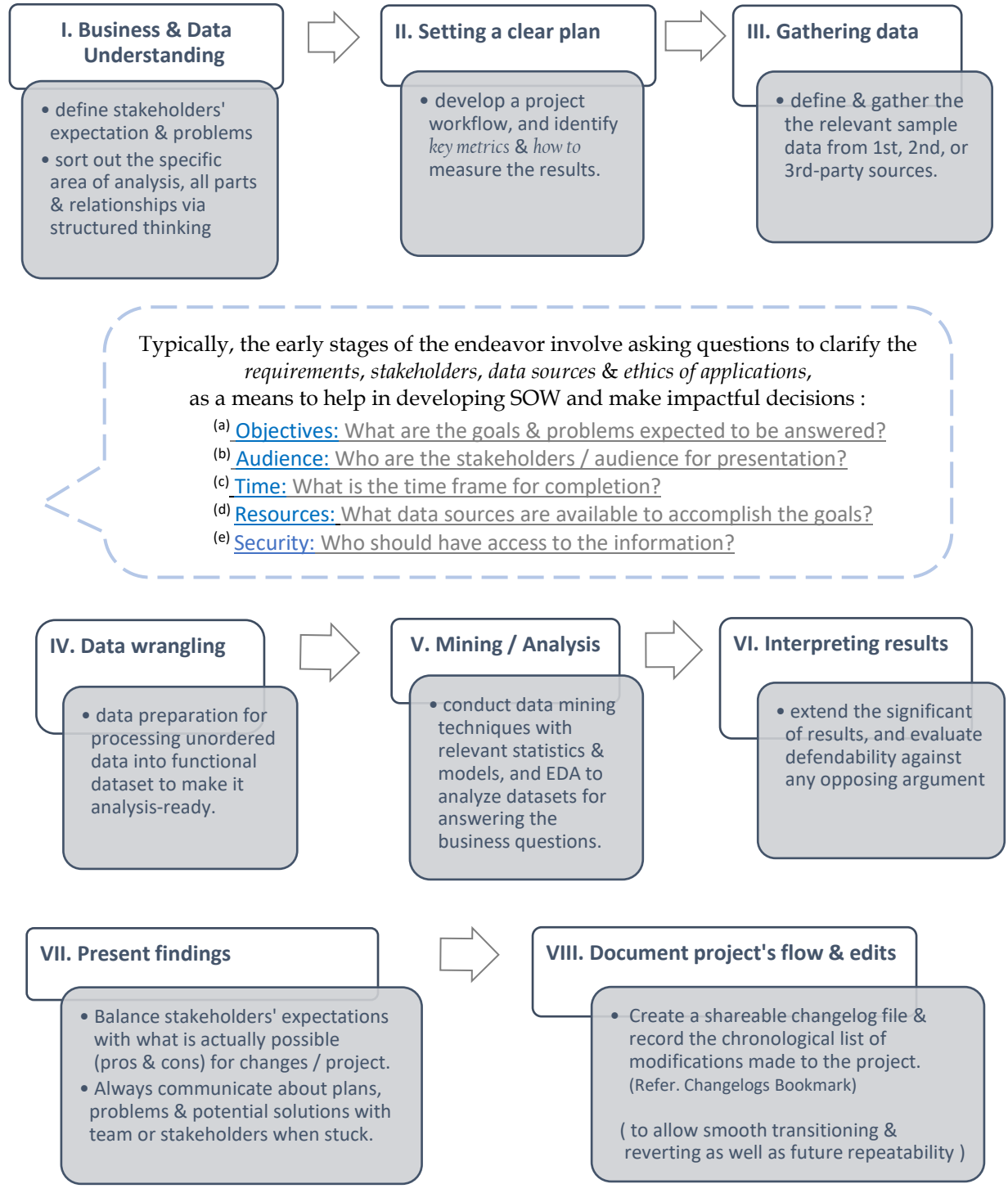
iii. Proceed with data preparation and analysis steps with creative questions formulation.
(eg. *Who & Where are these people most likely to see an advertisement and own a device... etc.*)

—Analyze with data on location, type of media, and no. of new customers acquired as a result of past ads can help predict the best placement of advertising to reach the target audience.

iv. Summarize findings using clear & compelling visuals and share the recommendations.

– Embrace inquiry, intellectual discussion, and collaboration in the data journey by reaching out to teammates & stakeholders whenever in doubt as a way to invite & assess feedback.

⋮ Detailed Analysis Process



Note: Use in conjunction with the six data analysis phases for a thorough exploration.

A data professionals typically use structured thinking of- (see. Scope of Work Examplar)

[i.] Recognizing current problem / situation, [ii.] Organizing available information

[iii.] Revealing gaps & opportunities [iv.] Identifying options

*to decompose a task into **a listing of plans & timelines** that match the known use cases.*

(Understanding Data)

- Before conducting EDA, it's important to first understand the context from its schema, what it represents in the real world, and what conditions & circumstances surrounding the data you should interpret in :

- **Who**: the person/organization that created, collected, and/or funded the data collection
- **What**: is it about? The meaning of each feature, and the things in the world that data could have an impact on.
  —eg. if sufficient & essential fields are included in the data for identifying meaningful pattern such as driving habits for determination of common causes in minor accidents.
- **Where**: was it created / collected? to whom questions regarding data quality, privacy & ethics may be referred & verified from the sources.
  —(1) Is the data originate from an experience & reliable collector? {Primary / Secondary sources}
  —(2) Check for accuracy, credibility, or bias (eg. stat bias / financial stake collector has on data results)
- **When**: was it created / collected?
  —eg. data collected awhile ago may have certain limitations or changes given present day situation.
- **How**: was it created / collected? or, does it relate to other data?
  (computer system report, selected from large databases, or manually entered table)
  —eg. the method of collection or creation of data may help explain the possible cause in an unusual event or missing value such as lagging data, system bug or a subject's group unwilling to disclose certain feature in a manual entry.
- **Why**: is it created / collected in the particular way?
  —eg. data might sometimes be collected or even made up to serve an agenda, having particular strong relationship with bias.

⁝ Usually, you may reference the data dictionary or metadata describing the dataset & data collection process and relational model for selection of fields based on the problem.

Right away you want to communicate clearly with stakeholders to answer these questions,
• *how & where if additional data were to be gathered* • *how long will it take to collect & clean*
so a realistic timeline & goals can be set after the initial data work.

Note: For fair & inclusive conclusions to be drawn using data, start with sample / dataset having ①*complete, correct & accurate representation of population*, ②*collected in most appropriate and objective manner*, and ③*incl. as many contextual points within the analysis*.

(In Coke launch failure ex, although the company successfully tested on New Coke solution over Pepsi, the decision as a replacement of the classic Coke was beyond and based on incomplete data)

There are 3 common stakeholder groups that a data analyst might find working with:

i. Executive team, senior-level professionals who set goals, develop strategy, and make sure it is executed effectively.
ii. Customer-facing team, anyone who compiles information, set expectations, and communicate customer feedback to the internal org part.
iii. Data science team, a group of data analysts:scientists:engineers collaborating to organize and explore on data.

Sorting out these questions when starting out can help one to focus and identify where best to seek guidance from leadership within organization on how to navigate.

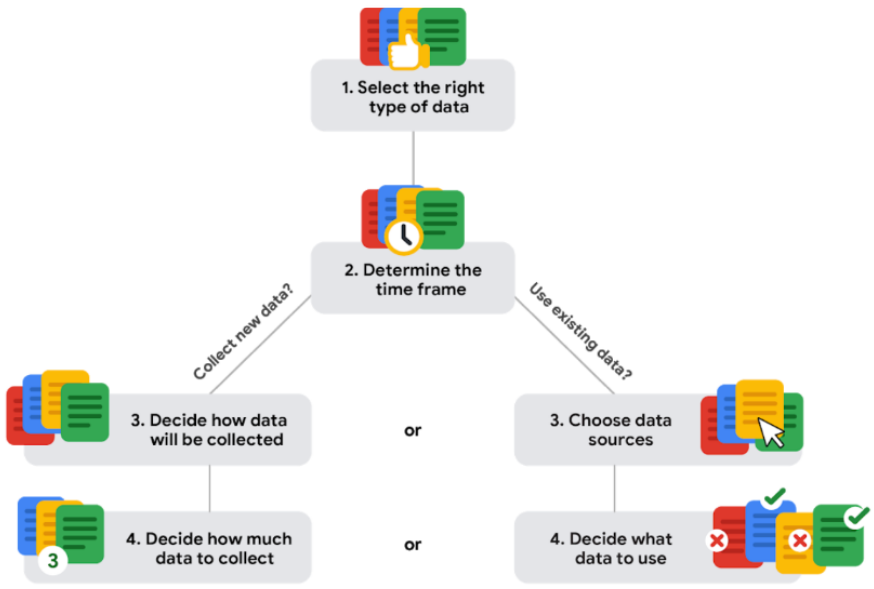
- Who are the primary & secondary stakeholders?
  - to identify main expectations & goals
- Who is managing the different set of data?
  - to allow productive work and sharing insights btw team members.
- Where can you go for help?
  - so less time will be spent searching and sorting out a confusion.

Balancing accuracy & speed (resorting to face value / detailed summary for interpretation) **can involve thinking carefully about the formulation of data mining solution and the outcome for stakeholders.**

{eg. – Does the analysis sufficient to answer the original question of the problem?
– Are there any missing angle of consideration that may worry the stakeholders? }

Data collection considerations

(the appropriate data collection relies heavily on how much time there is)



---

eg, For an immediate answer, you would need to use historical data that ald exists.

In term of data quality, there're a few standard practices that'll help in measuring the reliability of datasets before putting it into use.

R — reliable collector with accurate & unbiased data

O — original, the 2nd / 3rd party data can be validated against the original source

C — comprehensive, contain all vital information needed to answer questions

C — current, contain relevant & accurate information that fits the present context

C — cited, provide details about the history & location of origin to allow verification

When considering data ethics, it is not just about mimizing risk, but concept of beneficence —ie. *how it is going to impact people at the end of the day*.

(1) Ownership — recognize the responsibility to the people represented in the data.

(2) Transaction transparency — all data processing activities & algorithms should be completely explainable and understood by the individuals who provide them.

(3) Consent — the individuals know explicit details about how & why their data will be before agreeing to provide them.
(ie. Why is it being collected? How will it be used? How long will it be stored?)

(4) Currency — the individuals should be made aware of any financial transactions resulting from the use of their personal data & the scale these transactions.

(5) Openness — access, use, and share of open data are assented only if the data,
(i) Be available & accessible to the public as a complete set;
(ii) Be provided under terms that allow reuse & redistribution;
(iii) Allow universal participation w/ no discrimination against any disciplines, industries, persons, or groups.

(Data Privacy)

For the people whose data is being collected, this means they have the right to:

- Protection from unauthorized access to their private data
- Freedom from inappropriate use of their data
- The right to inspect, update, or correct their data
- Ability to give consent to data collection
- Legal right to access the data

⁝ Data analysts may not be responsible for data anonymization & the security systems, but they are expected to be mindful and incorporate access permissions as good practices.

When beginning a new job in the data career space, look into:
- Ask your manager for compliance guidance.
- Take time to research the data governance policies.
- Take time to research the regulatory body of particular industry and its relevant policy documents.

Data professionals need to remain aware of ethical & privacy concerns on the right ways of collect, share and use of data as well as considerations like data bias and making assumptions about data.

Name	Contact No.	Driver's License	Scoial Security	Account No.	Medical Record
Pam B.	212-456-7890	24558574	874-58-4758	Link	Link
Jim H.	234-555-1234	45157647	874-98-8745	Link	Link

i. Personal Information, PI
:: any type of information that can be traced back to a specific individual.

ii. Personally Identifiable Information, PII
:: personal information that could be used to identify an individual.

iii. Sensitive Personal Information, SPI
:: not necessary identify but contains sensitive information that could harm the individual if made public.

— WIth data privacy in mind, one can efficiently anonymize all personal info that is vulnerable to identity theft, fraud, and other issues.

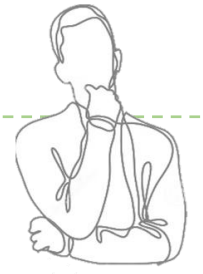
— Knowing where the data is collected is essential to identify the privacy laws & regulations that govern data.
Depending on organization's loc & industry, additional regulations or company's policies may be complied. { Such as, European Union: GDPR, Brazil: LGPD, California: CCPA, or industry specific }

[#1] Sometimes, Gender may be removed due ethical reason / bias based on context of target for analysis to avoid discrimination on the underrepresented group or skewing the result of modelling.

[#2] A common bias & consistency validation is verifying sample against ground truths / chars of the larger pop —*eg*. users aren't spending 27hrs a day / recheck field calculations, if any and identify misaligned causes.

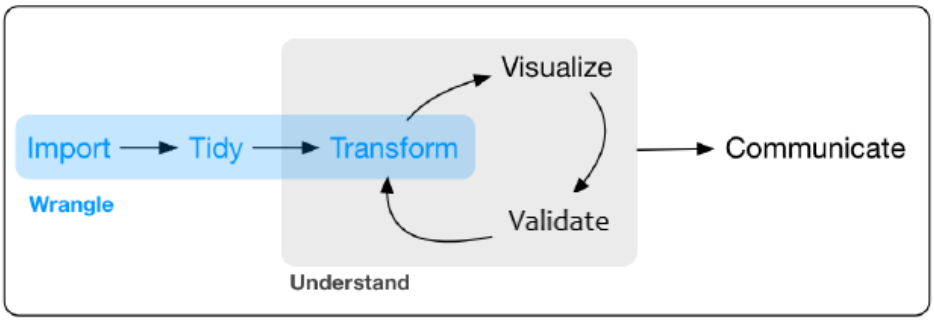
When deciding whether to use SQL, Spreadsheet, or Python for processing of the data, consider :-

- Where the data lives— in database, file, or website?
- What is the volume of data?
- Where is the data going— simple report, or prepare data for further analysis when query is too complicated
- Will it need to be updated with new data? How often?



Being able to make use multiple tools & API give you more options for how you can approach analysis to a problem efficiently after gauging its *effectiveness*, *limitations*, and *accuracy* which assumptions are based on.

Data Wrangling, is the preparation of data to fit the format required by analytic tool, both scientifically & practically as:



[1] Profiling – check dataset if the specific aspects for intended learning about business is complete, and quickly understand it w/ info, summarization, feature char (refer. RandomSampling)

[2] Cleaning : Validation – ensure integrity of data w.r.t accuracy, completeness, and consistency.

{ fixing col names, outliers, missing values, duplicate rows, or irrelevant instances (eg. past members); inconsistent format — date format/ time zone, irregular text/ empty space, unit of measurement; check if data matched the dtype defined for a field, and adding binning/ categorization. }

! It's always a good practice to make a copy of original dataset : keep track of the changes to avoid data disaster.

[3] Structuring - optimize data with format & structure of better operability and analysis.

{ data organization: ordered to facilitate usage— long, wide data, feature eng. *Dashboard, EDA,or Model*
data compatibility: different applications or systems can use the same data;
data migration: data with matching format can be moved from one system to another }

Note: Sometimes, you may come across insufficient data or the errors are time-consuming to fix. Next, your best action will be to discuss the limitations & adjust the objective, identify other alternate sources (eg. proxy variable), or research on your own. (refer. Dealing_with_insufficient_data)

To develop a deeper understanding of the business problem from large bodies of data,

i) ▸ Explore data with qualitative & quantitative Q's and draw hypotheses.

eg, a. Should the value of customer be taken into account in addition to likehood of churn?

b. Does it lead to better decision than reasonable alternative? (refer. Discovering_patterns)

:: One very general and important concern during data preparation is to beware of "leaks" information that appears in historical data but is not available when decision making.
(eg. *predicting a big spender by item purchased which can only be obtained after target is known*)

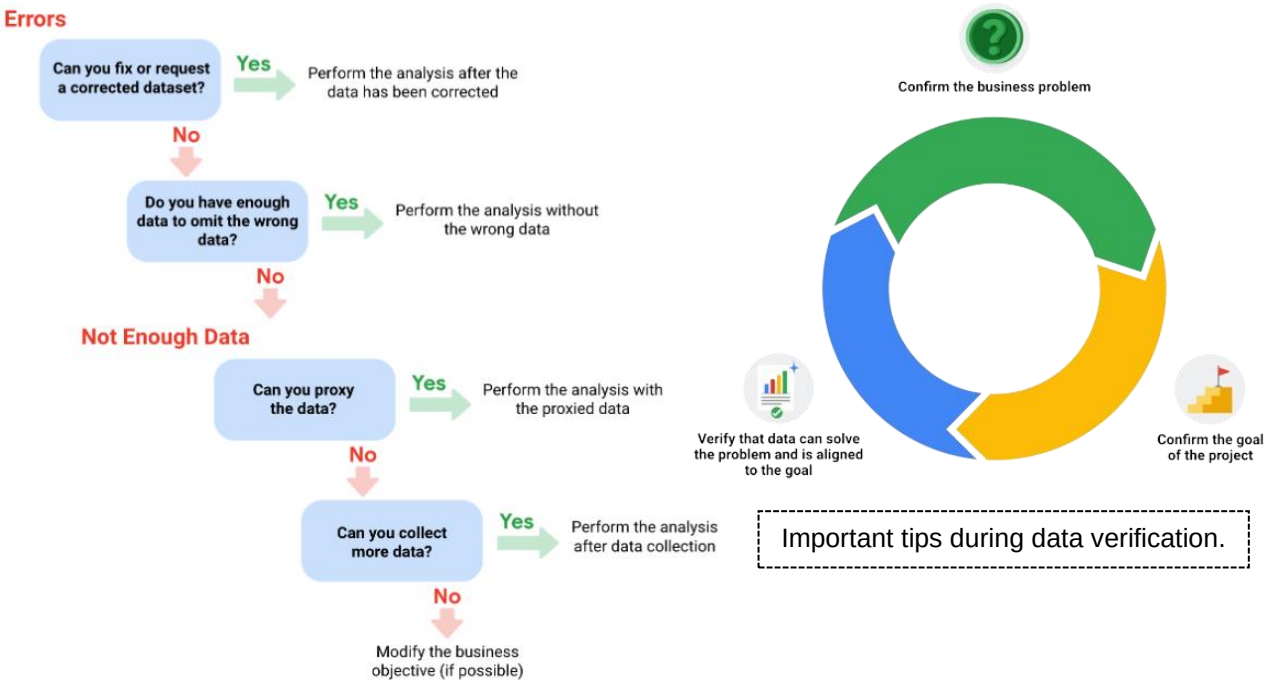
Followed, the data are manipulated & transformed into forms that yield better results.

ii) ▸ Normalize by amending homogenous / redundant dimensions, or scaling numeric values.

▸ De-normalize by conciling multiple sources / extract or split records into different cols.

▸ Enriching with feature extraction that could add value to the task.

When merging datasets, it's critical to identify the following to avoid redundancy & confirm the datasets are compatible: (1) Is it all the data that's needed? (2) Are the datasets cleaned to the same standard?



Important tips during data verification.

In the basic of analysis, the dataset might be organized with relevant data for a question by:

① Sorting — to rank items or create chronological list based on specific criteria.
(eg. Number-based numerically, Text-based alphabetically, date-based chronologically)

② Filtering — to narrow the view on specific criteria from large range of data.

③ Slicing — breaking information into smaller parts to facilitate efficient examination from different viewpoints.

④ Grouping / Pivot table — select & aggregate subset of data into groups.

* Following, you can turn it into a presentable (pivot) table and chart out exactly the cut of data that you were interested in exploring & showing in your data story.

Large datasets usually have to be broken down to be analyzed effectively.

—Often we want to calculate summary statistics / mathematical formula / trend conditionally on one or more subsets of the population / period of time.
(eg. "Does the churn rate differ btw male & female / high-income customers in a region?")

The use of AI for data work

Data professionals can leverage AI to,

① improve their analysis: *ML algorithms / aggregate large datasets,*
② perform essential tasks: *NLP / autonomous driving,*
③ streamline their workflow: *coding / formulating query statement / report writing*

Best practices for writing prompts for LLMs:

☑ Be clear & concise ☑ Be precise ☑ Incl. a description of LLM's role
☑ Provide context ☑ Try multiple prompts

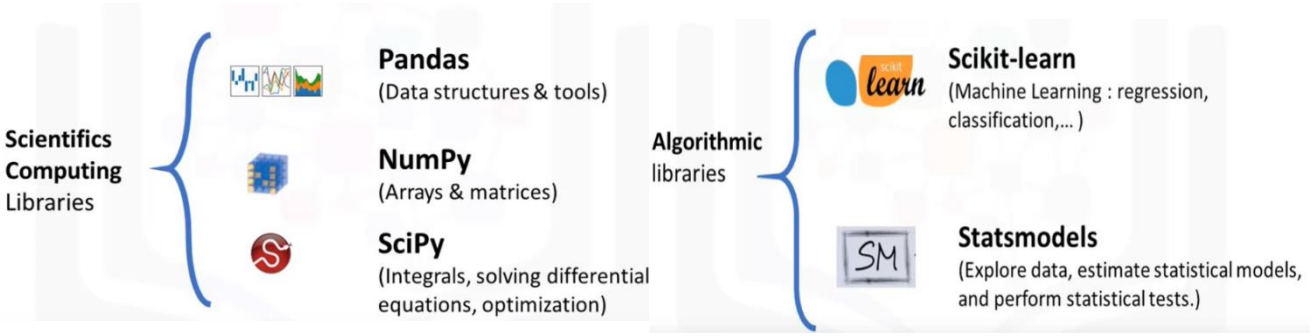
- LLM: **Gemini & ChatGPT** is a type of NLP algorithm that uses deep learning techniques to identify patterns in text and map how different words & phrases relate to each other.
- LLMs can generate insights that you may not have thought of; however, it's ultimately your responsibility to verify the results and make sure they make sense.

Note: AI yearns for well-defined instructions & clear parameters from human users, using their intuition, imagination, and unique experience to understand complex problems and address concerns about stakeholders' expectations.

(Refer: Reference Medium/AI_applications)

- " Act as a communication expert and share best practices for explaining a data science report to a business executive with no technical background. "
- " I have a dataset of customer purchases at an online retail store. Act as a data scientist and write Python code for data visualization and exploration. "
- " Create a shared project proposal to facilitate a brainstorming session for outlining the workflow and timeline of a sales prediction model development. "

Python APIs & Techniques



(*Essential Python Libraries*)

- numpy » Numerical operations optimized w/ homogeneous *ndarray* vectorization.
- pandas » Offers data structures w/ fast access, expressive functions for structured data.
- matplotlib » Plots & Visualizations lib w/ integration to rest of the ecosystem.
- seaborn » Statistical visualization on top of matplotlib.
- statsmodels » Classical and Modern statistical methods for tests & modeling.
- scipy » Advanced numerical (mathematical) operations and classical statistics module.
- scikit-learn » Machine learning models to analyze and discover patterns in data.

`scipy.integrate`
Numerical integration routines and differential equation solvers

`scipy.linalg`
Linear algebra routines and matrix decompositions extending beyond `numpy.linalg`

`scipy.optimize`
Function optimizers (minimizers) and root finding algorithms

`scipy.signal`
Signal processing tools

`scipy.special`
Wrapper around SPECFUN, a Fortran library with many common mathematical functions.

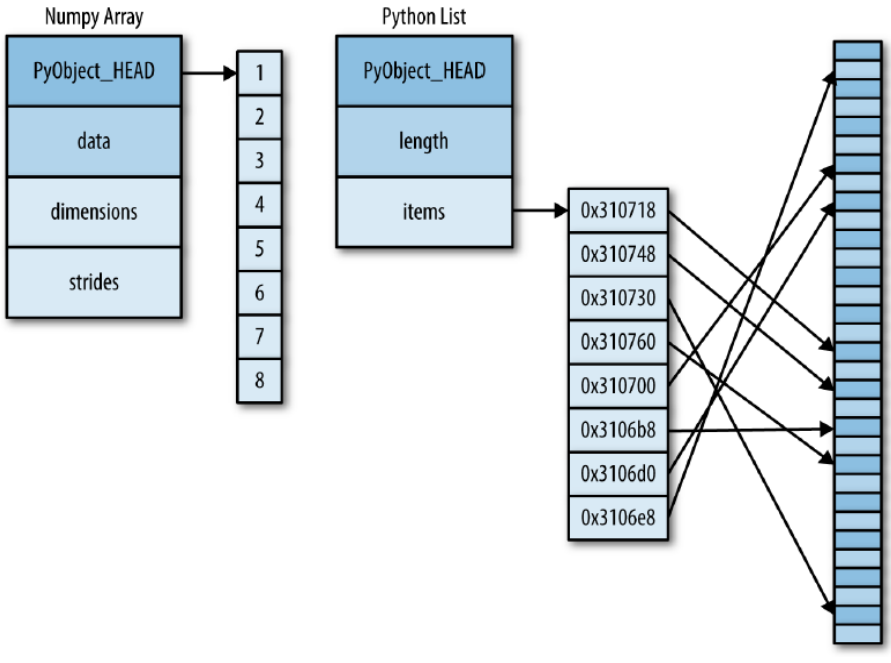
`scipy.stats`
Standard continuous & discrete probability distributions, statistical tests, and descriptive statistics.

NUMPY.NDARRAY

A single integer in Python contains:
- *ob_refcnt*— object's ref count that handle memory locations of allocation & de-allocation
- *ob_type*— encodes the type of the variable
- *ob_size*— specifies the size of following data members
- *ob_digit*— contains memory representations for values in Python variable

(1) As shown, Python object is simply a cleverly disguised C structure w/ a pointer to
—{ a position in memory containing all the Python object information }
(2) These information allows Python to be coded freely and dynamically, but also burden its memory heavily when combined many individual datatypes in one list.
(*ie. Due to consistency of variables' dtype in data analysis, much of this info is redundant*)

```
# Array                                  # List
array(['1', 's', '3'], dtype='<U11')     [1, 's', 3]
```



!As Numpy is deemed to perform numerical computation for vectorized matrix operations of the same type, the ndarray is designed so to allow efficient values storage & retrieval (processes).

For non-matching numeric datatype (*ie*. integer ↔ float), the value will be upcasted/truncated.

```
np.array([3.14, 4, 2, 3]) ;   np.array([3.14, 4, 2, 3], dtype='float32')
::array([3.14, 4, 2, 3])      ::array([3.14, 4, 2, 3], dtype='float32')
```

(Left): Conventional array constructor, (Right): Explicitly defined numeric data type for storage

[# By defining the different datatypes,
the computational performance of static & uniform type can be optimized during compilation.]

(Datatype): *dtype='int_'/ np.int_*

bool_	Boolean value: True / False stored as byte.
str_	Unicode string characters
object	Generalized Python object
int_	Default type integer that is 64-bit/32-bit.
intc	Similar to C integer of 64-bit/32-bit.
intp	Represents the integers used for indexing.
int8	1-byte integer (8-bit). Range: -128 ~ 127
int16	2-byte integer (16-bit). Range: -32768 ~ 32767
int32	4-byte integer (32-bit). Range: -2147483648 ~ 2147483647
int64	8-byte integer (64-bit). Range: -9223372036854775808 ~ 9223372036854775807
uint8	1-byte unsigned integer (8-bit)
uint16	2-byte unsigned integer (16-bit)
uint32	4-byte unsigned integer (32-bit)
uint64	8-byte unsigned integer (64-bit)
float_	Represents float64.
float16	Half-precision float. (5-bits:Exponent, 10-bits:Mantissa, 1-bit:Sign)
float32	Single-precision float. (8-bits:Exponent, 23-bits:Mantissa, 1-bit:Sign)
float64	Double-precision float. (11-bits:Exponent, 52-bits:Mantissa, 1-bit:Sign)
complex_	Represents complex128.
complex64	Complex number of real and imaginary part shares 32-bits each.
complex128	Complex number of real and imaginary part shares 64-bits each.

(Character Codes + bytesize): U1

'?'	Boolean
'B','b'	(un) signed byte
'u', 'i'	(un) signed integer
'f'	floating-point
'c'	complex-floating
'm'	time-delta
'M'	datetime
'O'	(Python) object
'U'	Unicode string
'S', 'a'	null terminated string chr
'V'	fixed chunk of memory for other type (void)

**The 5 basic numerical types that can be defined in straightforward manner

Note: *attribute-style* declaration is limited to bit precision, for unit specification (eg. 'datetime64[M]') use string format.

Array Manipulation

Attributes of arrays
Determine the size, shape, memory consumption, and data types of arrays
Indexing of arrays
Identify and allocate the value of individual array elements
Slicing of arrays
Identify and allocate smaller subarrays within a larger array
Reshaping of arrays
Alter the shape of a given array
Concatenating & splitting of arrays
Combining / splitting array(s) of data

NumPy Array attributes.

```
import numpy as np

rng = np.random.default_rng(seed=0)   #for reproducibility
x1 = rng.integers(10, size=(3, 4, 5))
print("x1 ndim: ", x1.ndim)
print("x1 shape: ", x1.shape)
print("x1 size: ", x1.size)

::x1 ndim: 3
  x1 shape: (3, 4, 5)
  x1 size: 60
```

Other attributes incl. `dtype`, `itemsize`, and `nbytes`:
—uniform datatype, byte size of each element, and total byte size of the array.

NumPy Array indexing & slicing

- array slicing return ***views*** rather than ***copies*** of the array
(*ie*. modifying subarray will alter the original array)

```
x2 = np.full([3, 3], 3)
x2_sub = x2[:2, :2]
x2_sub[0, 0] = 99
print(x2)
::[[99,  3, 3]
   [ 3,  3, 3]
   [ 3,  3, 3]]
```

```
# Fancy indexing
x = array([51, 92, 14, 71, 60, 20, 82, 86])
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
::array([[71, 86],    #1-D indexing
         [60, 20]])   ie. list of indices in specified shape
```

(1) use .copy() for duplicating the array. (2) result follows the broadcasted shape of the indexer.

*The default behavior is useful when accessing and processing pieces from large datasets w/o needing to copy the underlying data buffer.

indexing: `x[1, 2]` ; *fancy indexing*: `x[[1, 2]] == x[[1, 2], :]`
(*repeated indices assignment, x[[0, 0]] = [4, 6] will result in re-assigning an identical element*)

Note: NumPy's fancy indexing provides additional flexibility for multiple arbitrary {unordered} indexing /w rank 0: ***vector coord***.

" In Python, a copy would only be allocated if the variables change their values. "

(ie. This setting makes Python memory efficient by storing multiple copies only when required)

For modifying values with fancy indexing, use np.ufunc.at()

```
x = np.zeros(5)                 np.add.at(x, i, 1)
i = [2, 3, 3, 4, 4, 4]          print(x)
                                ::array([0, 0, 1, 2, 3])
```

Similarly, the pairing of indices follows all the broadcasting rules (see. broadcasting),

```
# Each row value-col vector with combine broadcasted shape:
X = np.arange(12).reshape((3, 4))    X[row[:, np.newaxis], col]
row = np.array([0, 1, 2])             ::array([[ 2, 1,  3],   #2-D indexing
col = np.array([2, 1, 3])                      [ 6, 5,  7],   ie. [[0], [1], [2]], [2, 1, 3]
                                               [10, 9, 11]])


# Combine fancy indexing w/ (1) slicing (2) masking
X[1:, [2, 0, 1]]          mask = np.array([1, 0, 1, 0], dtype=bool)
::array([ 6, 4, 5],              X[row[:, np.newaxis], mask]
        [10, 8, 9]])             ::array([[0,  2],
                                          [4,  6],
                                          [5, 10]])
```

eg. Compute sqr_differences in each pair of pts using broadcasting & aggregation via delegation of newaxis.

```
differences = np.sum((X[:, np.newaxis, :] - X[np.newaxis, :, :]) ** 2, axis=-1)
```

General Notes:

Each newaxis object in the selection list serves to expand the dimensions of resulting selection by 1-unit length.

eg.

```
# Add new axis after axis 1
np.arange(5)[:, np.newaxis]
:: (5,) => (5, 1)
```

```
# Add new axis in row/col position
X[:, np.newaxis]
:: (3, 4) => (3, 1, 4)
X[np.newaxis, :]
:: (3, 4) => (1, 3, 4)
```

{ *newaxis is an alias for 'None' object, and can be use in place of this with same result* }

Reshaping of Array

- flexible way of reshaping an array, *eg*. conversion from 1-D array to 2-D single row/col matrix.

Reshape the structure, or create new axis at every row/col via indexing of object

```
x = np.array([1, 2, 3])
# row vector via .reshape or, # newaxis
x.reshape((1, 3))              x[np.newaxis, :]
::array([[1, 2, 3]])           ::array([[1, 2, 3]])
```

Note: The size of the initial array must match the size of the reshape array.

'np.ufunc.reduceat()' method is similar in spirit that aggregates in the specified intervals of axis (Ref— binary ufuncs/comment)

I. NumPy's Universal Functions, <class 'np.ufunc'>

—(vectorized operation), an interface for uniform type in NumPy's arithmetic operations of which the loop is fitted into compiler layer, leading to much faster execution.

[# *Does not require sequential individual retrieval by 'for-loop', where multiple elements can be processed in a single directive manner*]

⏩ Mitigate the slowness of loops in default implementation to repetitively (1) *type-check on every value* & (2) *dispatch an identical arithmetic method* – which needlessly manifest during the many small iterations.

General rules for faster Numerical algorithm

- Pre-allocate matrices before use
- Avoid creating unnecessary variables
- Isolate repeated code with functions
- Apply vectorization than repetitive loops.

Basic 'ufunc' arithmetic operators

Operator	Equiv. ufunc
+	np.add
-	np.subtract
-	np.negative
*	np.multiply
/	np.divide
//	np.floor_divide
**	np.power
%	np.mod

Overloaded element-wise arithmetic operators

Mathematical functions

Function	Role
Trigonometric	np.sin(), np.cos(), np.tan()
Inverse-Trigo.	np.arcsin(), np.arccos(), np.arctan()
e^x ; 2^x ; n^x	np.exp() ; np.exp2() ; np.power(n, x)
Logarithmic	np.log() ; np.log2() ; np.log10()
$e^x - 1$; $\log(1+x)$	np.expm1() ; np.log1p()

Ref: NumPy documentation for more built-in available ufuncs (eg. hyperbolic trig, bitwise, comparison, rad-deg conv, rounding, etc)

Note: - The values are computed to within machine epsilon (ie. small precision roundoff error to zero is avoided at most).
- The specialized version: np.exp1 & np.log1p() is more precise than raw version at very small input.

Specialized ufuncs: scipy.special, obscure mathematical functions useful in statistics context

```
from scipy import special

# Generalized factorials & related functions
  - gamma(): special.gamma(), ln|gamma()|: special.gammaln(),
    beta(x, 2): special.beta(x, 2)
# Error function (integral of Gaussian)
  - erf(): special.erf(), erfc(): special.erfc(),
    erfinv(): special.erfinv()
```

[Refer. NumPy documentation "*gamma function python*" for more relevant information]

Aggregation functions

Function	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute median of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	`N/A`	Evaluate whether any elements are True
`np.all`	`N/A`	Evaluate whether all elements are True

The function takes directional argument (axis=0) for nd-array &/ computes results ignoring missing values

Advance Ufunc features

I, Specifying output— Rather than creating temporary array, store the result of calculation directly to specific array.

```
# (1)Write output directly to memory location (2)Used with array view
x = np.arange(5)                        y = np.zeros(10)
y = np.empty(5)                         np.power(2, x, out=y[::2])
np.multiply(x, 10, out=y)               print(y)
::array([0., 10., 20., 30., 40.])       ::[1. 0. 2. 0. 8. 0. 16. 0.]
```

If instead written as `y[::2] = 2 ** x`, the back operation will execute additional temporary array to hold the results of `2 ** x`, followed by a 2nd operation copying values into y array.

II, Binary ufuncs— Interesting aggregates that can be computed directly from the object.

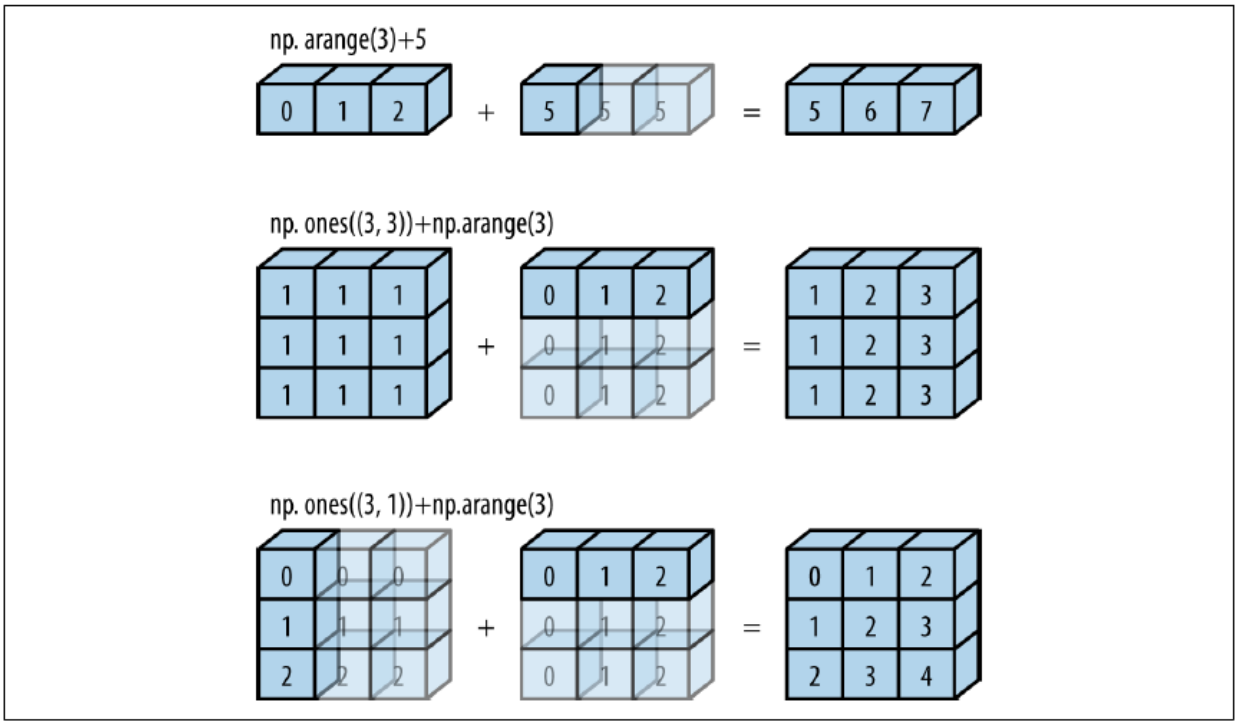
```
x = np.arange(1, 6)
#  Reduce size to 1 with iterative ufunc:
## (Default) along axis=0 ; initial=None ; element to incl. where=True
np.ufunc.reduce()
# Store all intermediate results of computation
np.add.accumulate(x),       np.multiply.accumulate(x)
::1+2, 1+2+3, 1+2+3+4, …    ::1*2, 1*2*3, 1*2*3*4, …
```

III, Outer products— compute the output of all pairs of 2-different inputs.

```
np.multiply.outer(x, x)
::array([[1,  2,  3,  4,  5],
         [2,  4,  6,  8, 10],
         [3,  6,  9, 12, 15],
         [4,  8, 12, 16, 20],
         [5, 10, 15, 20, 25]])
```

II. Broadcasting

—another means of vectorization, set of rules for *binary ufunc* on arrays of different sizes.



eg. Addition of 2-D to 1-D array (a + b),

Rule 1, pad the fewer dimension array w/ ones on left— b.shape -> (1, 3); a.shape -> (3, 1)
then by Rule 2, stretch the dim. to match other's size— b.shape -> (3, 3); a.shape -> (3, 3)

- Rule 1: If the two arrays differ in ndim, the shape of fewer dimension is padded with 1's on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array(s) shall stretch the singular {1, in the shape} dimension to match each others.
- Rule 3: If any of the dimension's size disagree and neither equal to 1, error is raised.

[Beware column slicing & fancy indexing to return 1-D array and stretch vertically to match others]

For right-side padding, reshape the array explicitly by— `a[:, np.newaxis]`
eg. *change* `M.shape ::(3, 2) + a.shape ::(3,)` *to* `(3, 2) + (3, 1)`

III. Sorting Array

Built-in | (sorting algorithm)
- `np.sort()` #ordered output ; `np.ndarray.sort()` #ordered in-place
- `np.argsort()` #return indices of sorted elements
- `np.partition` #arbitrary order of smallest K-values to the left of partition
- `np.argpartition` #aribtrary order of indices for smallest K-value on left

---

* `np.sort(x, axis=1)` : treats each row/col as independent array
[*ie*. if any relationships exist btw the row/col values will be lost!]

Note: The stretch/duplicate of values does not actually take place (no creation of temporary array), it is merely a mental model.

" **Big-O Notation:**

An algorithm efficient for large datasets will not always be the best choice for small dataset, and vice-versa.

{ *ie. Access is quick at the advantage of dynamic entries at the cost of record no ;*
Otherwise, invite unordered crowd requiring additional ordinance at the entry pt } "

IV. Comparison, Masks, and Boolean Logic

- use of Boolean masks to examine and manipulate values within NumPy arrays.
(eg. extract, modify, count, or manipulate values in an array based on some criterion)

As the case of arithmetic ufunc, it'll work on arrays of any size & shape.

Examine entries /w Comparison operators

```
# Count no. of True entries in a Boolean array:
 - np.count_nonzero(x < 6), np.sum((inches > 0.5) & (inches < 1))
# Check any / all values are True:
 - np.any(x > 8), np.all(x == 6)
```

*NumPy overloaded comparison & bitwise logic operators as *element-wise* ufuncs on array.

Masking Boolean array

```
# Extract values to the given condition {indexing by bool array}:
x ::array([[5, 0, 3, 3],      x < 5 ::array([[False,  True, True,  True],
           [7, 9, 3, 5],                    [False, False, True, False],
           [2, 4, 7, 6]])                   [ True, True, False, False]],
- x[x < 5] ::array([0, 3, 3, 2, 4])        dtype=bool)
```

Lookout—

(1) {True/False} can interpreted numerically as logical {1/0}.

(2) Python built-in functions: sum(), any(), and all() have different syntax than NumPy.

(3) In Python, all nonzero integers will be evaluated as True.

Note: writing `x < 3`, internally NumPy uses `np.less(x, 3)` [Refer. Standard array subclasses]

(Like `arr + obj` ; if `obj.__array_ufunc__` is present, `ndarray.__add__` and friends will delegate to ufunc machinery, `np.add(arr, obj)`, then invokes `obj.__array_ufunc__`)

Parallel equiv. of formal definition method on ufunc to which was delegated upon verified in Dunder method.

For more information regarding the implementation of magic method-Ufunc, refer NumPy documentation.

STRUCTURED ARRAY

- a single structure to efficiently store all of the compound, heterogenous data types.
(ie. *groups the related information together under one variable with different fieldnames*)

```
name = ['Alice', 'Bob', 'Cathy', 'Dough']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]

# Create empty Structured Arrays:
{Dict. method}
data = np.dtype({'names':('name', 'age', 'weight'),  #fieldnames
                 'formats':('U10', 'i4', 'f8')})     #values' datatype
```

For clarity, the formats can be defined more precisely at: **((np.str_, 10), int, np.float32)**

```
{List of tuples method}
       np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

Similarly, the compound datatype specification can be defined upon pre-allocated array.

```
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
::dtype[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

*If the fieldnames do not matter, one can specify the types alone—

```
      np.dtype('S10, i4, f8')
::dtype([('f0', 'S10'), ('f1', '<i4', ('f2', '<f8')))

# Assigning values:
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
::[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0) ('Doug', 19, 61.5)]
```

Structure array's *values* can be referred by index or fieldname.

```
ie. # Get all names
    data['name']
    ::array(['Alice', 'Bob', 'Cathy', 'Dough'], dtype='<U10')
    #Get first row of data
    data[0]
    ::('Alice', 25, 55.0)
    # Get name from last block      # by age is under 30
    data[-1]['name']                data[data['age'] < 30]['name']
    ::'Dough'                       ::array(['Alice', 'Dough'], dtype='<U10')
```

- First character (opt.): **>** / **<** {little / big endian}, specify the order convention for significant bit.
- Second character: **Character code**, specify the datatype.
- Last character: **Byte Size**, specify the standard memory size of object.

Advanced Compound Types

- create a type where each element contains an array or matrix of values.

```
# Create a variable of datatype /w array & floating-point matrix
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
```

Now the block in the X array is of `'id': array ; 'mat': matrix` { *fieldname-value & struct* }

```
# display the structure in the element of an array
X
::array([(0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])],
        dtype=[('id', '<i8'), ('mat', '<f8', (3, 3))])
print(X[0])
::(0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
```

However, scenario like this often lend themselves to use of Pandas `DataFrame`, which is an explicitly defined relational database / tabulated structure.

(Structure array vs. Nd-array / Dict.)

✓ Efficient storage & manipulation of dense typed array {akin to Pandas} in Python.

✓ NumPy 'dtype' directly maps onto a C structure definition.
So the buffer containing the array content can be accessed directly within a C program.
(***ie***. libraries written in a lower-level language can operate on the data stored in NumPy array w/o copying data into some other memory representation)

Useful for writing Python interface to legacy C/Fotran library that manipulates structured data.

[***eg***. Using NumPy array to map onto binary data formats in C, Fortran, or other language]

—Thus, NumPy arrays is assumed as generalized data structure with potential manipulation to seamless interoperability (if desired) by many numerical computing tools for Python.

Limitations: less flexible data manipulation than Pandas

eg. (1) attaching labels to data {index & column}, and working w/ heterogeneous data types as well as missing values.
(2) attempting operations that do not map well to element-wise broadcasting { groupby, pivot_table, merge, etc. }.

SERIES

" Just as NumPy's type-specific compiled code for efficient numerical operation. "
—*equiv.* a default sequence type/ ordered dict, mapping {*index: value*} pairs.

Type conversion from native Python structure

```
{List-style}
  data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
{Dict.-style}
  population = pd.Series({'California': 38332521, 'Texas': 26448193,
                          'New York': 19651127, 'Florida': 19552860},
                         name='pops')
  ::California  38332521
    Texas       26448193        [# To clearly identify Series from DataFrame, the column label(s)]
    New York    19651127
    Florida     19552860
    Name: pops, dtype: int64

  # Access the values      # Access the index
  data.values              data.index
```

*The constructor updates the index order w/ the latest definition. at overlapping elements:

```
pd.Series({2: 'a', 1: 'b', 3: 'c'}, index=[3, 2, 4])
:: 3  c
   2  a
   4  NaN
```

1-D [1]NumPy's array vs. [2]Pandas's series :

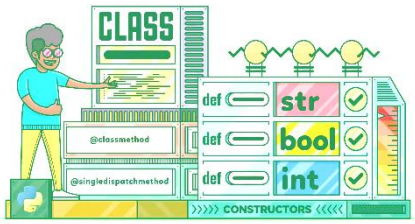
[1] prescribed sequence to the elements w/o index-association.

[2] named or, default sequence index is associated to the values as identity keys.

Note: - Unlike Python's dict., Series supports both array-type operations & key mapping.
eg. `data[1:3]` *OR* `data['Texas':'Florida'], np.exp(data)`
- For Series/DataFrame, data can be in array-like, scalar, and dictionary construct.

(**PANDAS DATATYPE**): `pd.DataFrame.astype({'col1': 'object'})`

- ✓ `float` : 'float64' *-ie.* np.float64
- ✓ `integer` : 'int64'
- ✓ `timestamp` : 'datetime64'
- ✓ `string` : object/ str
- ✓ `categorical` : 'category'
- ✓ `timezone` : 'datetime[ns, tz]'
- ✓ `timedelta` : 'timedelta64[ns]'



Pandas utilizes the numpy datatypes, except string as general Python's 'object' than more specific class.

DATAFRAME

- 2-D data structure of {Column: Series} aligned by [column > index] key from Series/ sorted dict.

Single-column DataFrame from Series/Dict.

```
states = pd.DataFrame(population)
:: California  38332521       # where index must be specified for scalar values
   Texas       26448193            California Texas    New York  Florida
   New York    19651127  =>    Pops 38332521   26448193 19651127  19552860
   Florida     19552860
   Name: Pops, dtype: int64    [# pd.DataFrame([population]) for Series has similar definition ]
```

DataFrame can be created from 'List' / 'Series', but for ease of compatibility to Pandas method,Series context is encouraged.

I, Dict. of data construct–

```
area = pd.Series({                    population = pd.Series({
    'California': 423967,                 'California': 38332521, 'Texas': 26448193,
    'Texas': 695662,                      'New York': 19651127, 'Florida': 19552860
    'Florida': 170312                     }, name='pops', dtype='int_')
    }, name='area', dtype='int_')
```

#Series' name attribute is identified as col/ind label for values when not specified (implicit definition).

```
# Explicitly defined Dict-style:
states = pd.DataFrame('Col_name': Series) / ('Col_name': Dict)
```

ie. pd.DataFrame({'area': {'California': 423967, 'Texas': 695662, 'Florida': 1730312},
'population': {'California': 38332521, 'Texas': 26448183, 'Florida': 19552860})

II, List of compound data construct–

```
# Implicitly defined Dict-style:
 - pd.DataFrame([area, population])  equiv. to
 - pd.DataFrame([{'a': 1, 'b': 2, 'c': 3}, {'a': 2, 'b': 3}])
::       a      b        c
  0    1.0    2.0      3.0
  1    2.0    3.0      NaN
```

III, (a) 2-D NumPy array,

```
# List context
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
::   foo       bar
  a  0.865257  0.213169
  b  0.442759  0.108267
  c  0.047110  0.905718
```

(b) NumPy structured array,

```
# List context
A = np.zeros(3, dtype=[('A', 'i8'),
                       ('B', 'f8')]
pd.DataFrame(A)
::   A  B
  0  0  0.0      # outer layer: records
  1  0  0.0      # inner layer: features
  2  0  0.0
```

Note: Pandas sets apart the support for geographic data in different `geopandas.GeoDataFrame` sublcass that has column with dtype: 'geometry' for geometric operations.

Pandas Index Object

- an immutable array / ordered multi-set (as `'Index'` object may contain duplicate labels)

```
ind = pd.Index([2, 3, 5, 7, 11])                    #default: pd.RangeIndex(0, 4, 1)
data = pd.Series([0, 1, 2, 3], index=ind)
::Int64Index([2, 3, 5, 7, 11], dtype='int64')

# Index object operates like list/array, but immutable:
ind[1] = 0      [# Otherwise, to match & retrieve integer indexes— ind.get_indexer(keys)]
::#error!

# Support operations on mathematical Set across datasets:
indA | indB ; indA & indB ; indA ^ indB  etc.

# Dict-like Python expressions methods can be use to examine the index:
(Refer. Python for Data Analysis, pg.136 for more useful index methods)
- 'a' in data      data.keys() / df.index ; df.columns
  ::True           ::Index(['a', 'b', 'c', 'd'], dtype='object')
- list(data.items())
  ::[('a', 0.25), ('b', 0.5), ('c', 0.75)], ('d', 1.0)]
```

*Index object's attributes: `ind.size, ind.shape, ind.ndim, ind.dtype, ind.name,`
*Other useful methods: isin(), delete(locs), drop(labels), insert(loc, items), unique(), append(items)

Selection: Indexing, Slicing & Masking

(A) Basic mechanism for indexing, slicing, masking

```
# Series {array-expr}
- data['a':'c'], data[0:2], data[(data > 0.3) & (data < 0.8)], data[['a', 'e']]
```

(indexing prioritize explicit integer index ; while slicing prioritize implicit integer index)

Recognize DataFrame as generalized dict. than generalized array, *(whether default / specified column name)*
Direct indexing : df[0]/df['zero']['a'] – follow nested keys as ***view*** on underlying data.

```
# DataFrame
Row selection :        - data.iloc[-1], data.loc['California':'Texas']
Column selection :     - data.area {attribute-style}, data['area'] {dict-style}
```

*2-D selection : `- data.iloc[:, [0, 3]], data.loc['California', :'population']`
(DataFrame returns Series -its basic unit for 1-D slicing ; List slicing- `df.loc[:,['a']]` to retain 2-D shape)

In 1-D selection, while *classic indexing* refers to columns, *slicing* refers to rows—
Also, {implicit int} `.iloc[0:2]`, excl. upper limit; {explicit key} `.loc['a':'c']`, incl. upper limit

Alternatively, Boolean mask can be used as indexer for selecting rows:

```
- data.loc[data.density > 100, ['pop', 'density']]
- data[(data.density > 100) & (data.pop > 100000)]   # row-wise direct indexing
```

```
arr[[[0], [2]], [0, 2]] == df.iloc[[0, 2], [0, 2]]   # select subset of matrix
```

(*Due to preservation & alignment of indices and columns, Pandas does not support fancy indexing from-group level individual coordinate pairs to avoid excessive obsolete values creation*)

NOTE: Column selection by attribute-style may not be viable if column label ≠ string, conflict with method or, new column.

(B) Modify by view on the object

```
- data.loc[('col1' < 0), [['col1', 'col2']]] = [1, 0]
```

* Unlike NumPy, overwrite to a view by direct indexing will not be permanent, but use- iloc[ ] / loc[ ].

** IN **Series**, index will be aligned, inserting missing values for any holes.

(C) Dropping entire row/ cols

```
- data.drop(['California', 'Texas'], inplace=True) OR del data['Texas']   —single col only
- data.drop(columns=['population', 'area'])
```

! RECOMENDED:

For DataFrame selection,

(1) Use `iloc` {implicit indexer} & `loc` {explicit indexer}
—to avoid ambiguity btw sequence & label-based indexing convention.

(2) Most Pandas methods do not modify original object (excl. inplace), use += / = assignment.

Pandas Ufuncs: Index Preservation & Alignment

- As complementary to NumPy, Pandas preserves all universal functions (np.ufunc) with the addition of index & columns alignment for all binary operations.

(ie. more flexible broadcasting shape: [5, 5] + [3, 3], returning NA for non-overlapping entries)

```
area = pd.Series({'AK': 1723337, 'TX': 695662, 'CA': 423967})
population = pd.Series({'CA': 38332521, 'TX': 26448193,'NY': 19651127})
```

```
# Preserve indexes as
  union of 2-arrays
 population / area
:: AK          NaN
   CA    90.413926
   NY          NaN
   TX    38.018740
   dtype: float64
```

Utilize arithmetic methods than shorthand operators when inherent options are required.

```
A.add(B, fill_value=A.stack().mean())
```
radd: flipped arg

[# fill missing entries with mean of all A elements]

[# any missing entry from larger data structure is marked with NaN]

EQUIV: `area.index | population.index ::Index([…], dtype='object')`

Ex. Operations btw DataFrame & Series

```
df = pd.DataFrame(np.arange(9).reshape((3, 3)), columns=list('QRS'))
df - df.iloc[:, 1]                    df - df.iloc[1, :]
::    Q   R   S   0   1   2           ::    Q  R  S
  0 NaN NaN NaN NaN NaN NaN              0 -3 -3 -3
  1 NaN NaN NaN NaN NaN NaN              1  0  0  0
  2 NaN NaN NaN NaN NaN NaN              2  3  3  3
```

[OR use- **df.sub(**series, axis=0**)**]
#for ease of reference.

! **Beware**: Arithmetic ufunc of Series resulting from slicing or, fancy indexing,
the alignment follows label of Series' indexes-to-DataFrame's columns w/ broadcasting.
ie. {DataFrame} primary keys ≃ {Series} primary keys

Note: Pandas ufuncs for arithmetic operation is similar to NumPy (eg. pd.DataFrame.multiply()), except: floordiv() {floor div}.

Apply Function and Transform

To apply a function / computation along an axis of Dataframe (default: column-wise),
[! Do not mix up with groupby( ).apply( ) which takes grouped DataFrames as input instead of Series.]

```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4], 'Qu2': [2, 3, 1, 2, 3],
                     'Qu3': [1, 5, 2, 4, 4]})
```

```
# Compute normalization
f = lambda x: x.max() - x.min()
frame.apply(f)
::b  1.802165
  d  1.684034
  e  2.689627
  dtype: float64
```

OR

```
# Compute common histogram for all columns
data.apply(pd.value_counts).fillna(0)
::    Qu1  Qu2  Qu3
  1  1.0  1.0  1.0
  2  0.0  2.0  1.0
  3  2.0  2.0  0.0
  4  2.0  0.0  2.0
  5  0.0  0.0  1.0
```

[# Instead, DataFrame.transform( ) is used to compute different feature's settings w/ list of disparate functions]

Akin to the properties of Python function, the function need not return a scalar type;

I. Return a Series with multiple values:

```
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
frame.apply(f)
::            b         d          e
  min  -0.55730  0.281746  -1.296221
  max   1.246435 1.965781   1.393406
```

II. Compute a formatted string for each floating-point value:

```
def readable_numbers(x):
    """takes a large number and formats it into K, M short readable digits."""
    if x >= 1e6:
        s = '{:1.1f}M'.format(x*1e-6)
    else:
        s = '{:1.0f}K'.format(x*1e-3)
    return s
df.map(readable_numbers)
::    freq_1  freq_2
  0   15.6M    209K
  1   44.6M   35.1M
  2   37.9M   41.6M
  3   27.6M   28.8M
```

[# When a dict or function is passed to the df.map( ), it will replace the matching data w/ the returned value, and impute NaN for any values that do not have a corresponding match in the mapping.mean]

Note: 'result_type': {'expand', 'reduce', 'broadcast'}, is also available to alter shape of return result.
(*eg*. expand / reduce list-like into columns; or broadcast to original shape retaining original index)

Sorting, Ranking & Sampling

```
table = pd.DataFrame(np.arange(8).reshape((2, 4)),
        index=['three', 'one'], columns=['d', 'a', 'b', 'c'])
```

```
# Sort labels lexicographically in row/col
table.sort_index(axis=1, ascending=False)
::       d  c  b  a
  three  0  3  2  1
  one    4  7  6  5
```

```
# Sort values by multiple cols' keys
table2.sort_values(by=['a', 'b'])
::   a  b          a  b
  0  0  4   =>   2  0 -3
  1  1  7        0  0  4
  2  0 -3        3  1  2
  3  1 -2        1  1  7
```

[# key function can be passed to sort_value(key) option which is identical to built-in sorted() function]

Rank values along the order of position & resolve tie values' rank with desired method.

```
cells = pd.Series([7, -5, 7, 4, 2, 0, 4])
struct = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
                       'c': [-2, 5, 8, -2.5]})
cells.rank(ascending=False, method='max')
::0  2.0
  1  7.0
  2  2.0
  3  4.0
  4  5.0
  5  6.0
  6  4.0
  dtype: float64
```

```
struct.rank(axis=1)
::     a    b    c
  0  2.0  3.0  1.0
  1  1.0  3.0  2.0
  2  2.0  1.0  3.0
  3  2.0  3.0  1.0
```

Method	Description
'average'	{Default} Assign the average of group's rank to each entry
'min'	Assign by the lowest rank in equal values' group along an axis
'max'	Assign by the largest rank in equal values' group along an axis
'first'	Assign by order of the integer value
'dense'	Like method='min', but rank based on absolute levels instead of position.

Note: 'ascending' option can be allocated for increasing/decreasing rank before resolving for the tie with method.

Often in statistical evaluation, data professionals will need to engage in effective sampling of a dataset in order to make it easier to analyze.

```
# Permuatation & Random Sampling
sample_ind = np.random.permutation(5)
::array([3, 1, 4, 2, 0])
```

```
# Permuted selection / draw
df.iloc[sample_ind, :]; df.take(ind)
```

```
# Sampling without / with replacement
df.sample(n=50, replace=True, random_state=31208)
```

```
# Sampling distribution
estimate_list = []
for i in range(10000):
    estimate_list.append(df.sample(n=50, replace=True).median())
sampling = pd.DataFrame({'means_of_samples': estimate_list})
```

DATA LOADING & STORING
OF DIFFERENT FILE FORMATS

The data organization (*ie*. collected raw data) can be in one of the several forms,

Structured	Semi-structured	Unstructured
eg. Transaction information, SQL databases, spreadsheets, online forms, OTP, etc.	*eg*. e-mail, XML/JSON …, binary executables, TCP/IP packets	*eg*. photo, video, audio file, documents/PDF, webpage, survey, social media post, etc.

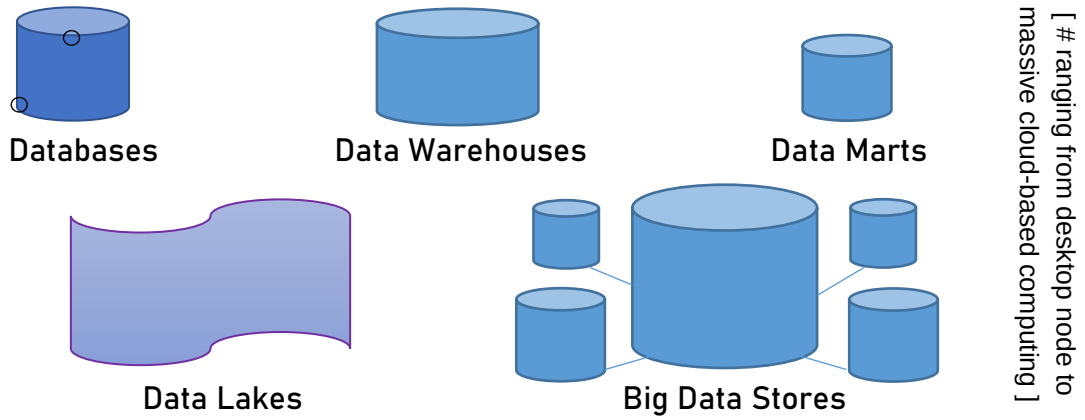
(Structured)
- Fit neatly in a rigid schema w/ defined datatypes & tabular structure.
- represented in rows & columns, but not limited to the type of database storage.

(Semi-structured)
- Partially organized & free-form.
- contains consistent identifiers (tags, elements, metadata) to group in a hierarchy, but messy content.
- require minor processing on the structured parts to allow SQL query.

(Unstructured)
- Organized loosely in an internal structure, and has no established organizational rule about how to compare two different pieces of data.
- wrap in files, document, or, softwares w/ inconsistent data types which often difficult to search, manage, and analyze (manual processing is required).

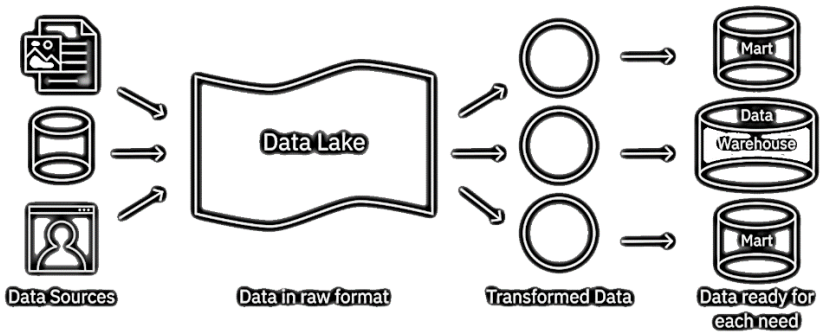
where most of the data being generated right now is unstructured, being able to understand it empowers businesses to uncover hidden info, patterns, and trends that's previously inaccessible,

Data Infrastructures,



ie. the technologies used to collect, manage, and compile data.

(Big data - Hadoop, HBase & MongoDB refers to data processing systems of datasets that are too large for traditional systems)



Relational, SQL—

- ✓ Ideal for structured storage, access, and processing of large data volume in seconds using single SQL-query. *{minimized data redundancy at relationships defined btw tables & joining tables}*
- ✓ Flexible to make changes while the database is in use.
- ✓ ACID compliance: ease of backup & disaster recovery.

- ✘ Does not work well for semi-/ unstructured data.
- ✘ Migration btw 2-RDBMS requires identical schemas & data types, whereby entering value greater than defined length of a data field will result in loss of information.

Non-relational, NoSQL—

- ✓ Built for lower cost w/ larger storage, not restricted to tabular only.
- ✓ Flexible schemas to accommodate different organization needs in modern applications.
- ✓ An efficient & cost-effective scale-out architecture that provides additional capacity and performance w/ additional of new nodes *{as distributed system scaled across multiple data centers}*.

- ✘ Not generally optimized for querying many records at once, excl. Column-store

that is,

- a data warehouse (eg. Google BigQuery) is a centralized storage for structured data from multiple individual dabasess or operating sources in an organization.
- a big data stores, has distributed computational & storage infrastructure to stock, scale, and process very large datasets.
- a data mart {subsets of data warehouse} **provides analytical capabilities for a restricted area of the data warehouse, of which is an isolated security and performance.**

- a data lake is a flexible storage for all forms of data in its native format w/ unique identifiers & metadata repository.

Data Warehouse: SQL ; Data Lake: HiveQL, depending on underlying sys.
(eg. HDFS is commonly used in conjuction with Apache Hadoop for storing & processing large volumnes of data in a distributed and scalable manner.)

Depending on the type of non-relational database, it may support SQL/ SQL-like query (eg. CQL – Cassandra ; GraphQL – Neo4J)

Due to the abundance of data collection and management as they are today,
the common data sources for any { file, web, or database } formats are—

(1) Relational Databases (2) Flat files & XML datasets (3) Web Scraping

(4) APIs & Web Services {*XML, HTML, JSON, Plain text, media file*} (5) Data Streams & Feeds {*RSS, JSON, Atom*}

Usually, data analyst will be working with structured data and deals with minimal ETL process (*eg*. when extraction from an open source is needed).

Pandas' combined loading & reading methods.

Method	Description
`read_csv`	Load delimited data from a file, URL, or file-like object {default: comma delimited}
`read_fwf`	Read data of fixed-width column format (ie. no delimiters)
`read_clipboard`	Read data from clipboard; *useful for converting tables from web pages*
`read_excel`	Read tabular data from an Excel XLS (`.xlsx`) file
`read_hdf`	Read HDF5 (`.h5`) file written by Pandas
`read_html`	Read all tables found in the given `.HTML` web content for digital viewing
`read_json`	Read JSON string representation data from a file, URL , or file-like-object
`read_feather`	Read the Feather binary file format
`read_orc`	Read the Apache ORC binary file format
`read_parquet`	Read the Apache Parquet binary file format
`read_pickle`	Read an object stored by Pandas using the Python pickle format
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_spss`	Read a data file created by SPSS program
`read_sql`	Read the results from SQL query (using SQLAlchemy, or Sqlite3 lib.)
`read_sql_table`	Read whole SQL table (using SQLAlchemy), *ie*. query that selects all by 'read_sql'
`read_stata`	Read a dataset from Stata file format
`read_xml`	Read a table of data from an XML file

! A good rule when deciding whether to import data from a source (eg. gov. academic papers, financial open data)
- import it, if the data will create value when integrated w/ data from other systems;
  else, wait until there is a stronger case.

[*# It is important to aware of the different formats, and challenges in handling them to be able to work with various project & storage type as they arrived.*]

Of the many parameters, the useful ones can be generally classified as:

(Indexing) – specify columns for the returned DataFrame, and treatments to the labels;
ie. retrieve from the file, re-define in argument, or not at all.

(Type inference & Data conversion) – the user-defined datatype conversion & missing value markers.

(Datetime parsing) – a combining capability to parse date & time information
eg. as such, to merge multiple columns into a single column.

(Iterating) – support iterative processes over chunks of very large files.

(Unclean data issues) – skipping rows, footers, comment, thousand {numeric} separator, etc.

[# Some methods perform *type inference*, because the column datatypes are not part of the data format.]

A. Text File (Plain text: unicode chrs ; Delimited text: sequence of 1-/more chrs separating values to in.each row)

A major advantage of .csv files is their widespread compatibility, where they can be imported and exported by a vast range of data analysis tools & software programs.

```
# Default delimited data
!cat examples/ex1.csv
:: a,b,c,d,message
   1,2,3,4,hello
   5,6,7,8,world
```

```
# Whitespace / Arbitrary pattern separated text
list(open('examples/ex3.txt'))
::['            A         B         C\n',
   'aaa -0.264438 -1.026059 -0.619500\n',
   'bbb  0.927272  0.302904 -0.032399\n',
   'ddd -0.871858 -0.348382  1.100491\n']
```

```
# To use .csv file, it's first downloaded to local device & imported
data = pd.read_csv(attachment link/path, index_col=0, sep="\s+")    # vertical bar, tab… regex
```

Note: The method infers first col as index when there's fewer cols than rows.

---

(1) **header** – row no.(s) to be used as column's labels ; **name** – declare new list of column's labels

(2) **index_col** – specify int. / column(s) to be used as index ; **nrows** – no. of rows to be read

(3) **sep** – chr/regexp delimitter; **keep_default_na** – verify if default `NA` representations are kept

(4) **na_values** – list/dict. mapping for column's to add to the default list of `NA` representation.

(5) **skiprows** – int / list of rows from the beginning ; **skip_footer** – no. of lines from end to be ignored

(6) **parse_dates** – if True, attempt parsing on all columns or, the int./label of column(s) to *`datetime`*.
(**Note:** if element of list is another tuple/list, will parse as combine columns)

(7) '**date_parser**' – Function to parse dates ; **on_bad_lines** –{'error'|'warn'}, on line(s) w/ too many fields

(8) '**keep_date_col**' – whether to keep the joined columns from parsed date

(9) '**converters**' - Dict. mapping the column(s)' int/label to apply function (*eg*. `{'foo': f}`)

(10) '**dayfirst**' – whether to parse potentially ambiguous dates as international format
(*eg*. 7/6/2012 → June 7, 2012)

(11) '**encoding**' – text encoding when writing or reading {default: ' utf-8'}

(12) '**comment**' – character(s) to split as comments off the end of lines

(13) '**verbose**' – print various parsing info, *(eg*. time spent in each stage / memory usage)

(14) '**thousand**' – specify thousands separator ; '**decimal**' – specify decimal separator {default: `"."` }

By default, Pandas marks empty string from file as NA / NULL from list of many default NA representations.

For reading text file in pieces,

```
pd.options.display.max_rows = 10   #alter Pandas' setting for more compact display
```

```
(A) result = pd.read_csv("ex6.csv", nrow=None)
::            one        two      three       four key
  0      0.467676  -0.038649  -0.295344  -1.824726   L
  1     -0.358893   1.404453   0.704965  -0.200638   B
  2     -0.501840   0.659254  -0.421691  -0.057688   G
  …          …         …         …         …  ..
  9997   0.523331   0.787112   0.486066   1.093156   K
  9998  -0.362559   0.598894  -1.843201   0.887292   G
  9999  -0.096376  -1.012999  -0.657431  -0.573315   0
  [1000 rows x 5 columns]
```

```
Load very large data in chunks
(B) chunk = pd.read_csv('ex6.csv', chunksize=100_000)
type(chunk)   #io interface
::pandas.io.parsers.readers.TextFileReader
tot = pd.Series([], dtype='int64')
for piece in chunk:
tot = tot.add(piece['key'].value_counts())
tot = tot.sort_values(ascending=False)
::E  368.0
  X  364.0
  L  346.0
  O  343.0
  Q  340.0
  …    …
```

Note: 'TextParser' can also equip w/ 'get_chunk( )' for reading data in pieces of arbitrary size.
[**!** When struggle to read a particular file, check Pandas documentation for the right parameter/similar Ex.]

Write & **Save** data to text format,
(ie. *file format indicates on-premise document structure along with the tools & options to interact with it*)

```
import sys
# Print content to console
data.to_csv(sys.stdout, sep= '|')
|smth|a|b|c|d|message
0|one|1|2|3|3.0|4|
1|two|5|6|8|world
2|three|9|10|11.0|12|foo
```

```
# Print content to a text file
data.to_csv(csv_path, index=True)
,smth,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,8,world
2,three,9,10,11.0,12,foo
```

[# Missing values from DataFrame appear as empty strings in output if not denoted by 'na_rep' option]

---

(1) '**index**' – whether to incl. index to file ; '**header**' – whether to incl. column's labels to file

(2) '**na_rep**' – string of sentinel value ; '**column**' – list of new column's labels ; '**encoding**' – character std

In unconventional case, when malformed lines in the file trip up 'read_csv( )',
(1) basic line-precessing program can be used to interpret the text into an iterarable reader obj.

```
!cat examples/ex7.csv
::"a","b","c"
  "1","2","3"
  "1","2","3"
```

```
import csv
with open(csv_filepath) as fid:
    lines = list(csv.reader(fid))
::[['a', 'b', 'c'],
   ['1', '2', '3'],
   ['1', '2', '3']]
```

[# Python's basic csv tool is capable of reading any file with single-character delimiter]

(2) *Subsequently,* carry out wrangling steps to rectify & organize data back to its desired form.

```
header, values = lines[0], lines[1:]
data_dict = {h: v for h, v in zip(header, zip(*values))}
::{ 'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
df = pd.DataFrame(data_dict)
```

To define a new format w/ different delimiter, string quoting convention, or line termination, a simple subclass of `csv.Dialect` could be created as followed—

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL
```

```
reader = csv.reader(fid, dialect=my_dialect)

# To define individual parameters
reader = csv.reader(fid, delimiter= '|')
```

---

(1) 'quoting' – controls when quotes is generated by the writer & recognized by the reader.
(*eg.* only quote those fields with special characters, like delimiter, quotechar, lineterminator, etc.)

(2) 'skipinitialspace' – Ignore whitespace after each delimiter {default: False}.

Note: If more complicated/ fixed multi-characters delimiters are present, line splitting and other clean up using string or Regex method may be necessary.

Otherwise, writing back to the file can be done via 'csv.writer()' & 'writer.writerow()'.

```
with open('mydata.csv', 'w') as fid:
    writer = csv.writer(fid, dialect=my_dialect)
    writer.writerow(("one", "two", "three"))
    writer.writerow(("1", "2", "3"))
    writer.writerow(("4", "5", "6"))
    writer.writerow(("7", "8", "9"))
```

B. Binary File Formats

It is referred to files that is made up of non human-readable characters, and contain formatting limited to certain applications or processors only. *{eg. Word, XLS, PDF, GIF, JPEG, Audio files, etc.}*

(Pickle file is one readable in Python only, and the format may not be stable at a later version of lib)

(i) Recommended only for short-term storage, Python has built-in 'pickle' module / Pandas 'pd.read_pickle' to serialize data in binary format. (see. SQL_relational / non-relational schema)

(ii) Alternately, HDF5, ORC & Apache Parquet are some of the well-regarded types with different serialization intended for efficiently storing & processing large quantities of data arrays.

```
frame = pd.DataFrame({'a': np.random.randn(100)})
store = pd.HDFStore('mydata.h5')
::<class 'pandas.io.pytables.HDFStore'>
  File path: examples/mydata.h5
  /obj1          frame          (shape->[100,1])
  /obj1_col      series         (shape->[100])
  /obj2          frame_table    (typ->appendable,nrows->100,ncols->1,indexers->[index])
  /obj3          frame_table    (typ->appendable,nrows->100,ncols->1,indexers->[index])
store['obj1'] = frame  ;  store['obj1_col'] = frame['a']   #work like dict; a['key']

# HDFStore supports 2-storage schemas– 'fixed' / 'tables'
#(the latter is similar to data model which supports query ops using special
# syntax, but generally slower):
store.put('obj2', frame, format='table')
store.select('obj2', where=['index >= 10 and index <=15'])
store.close
```

Conveniently, Pandas data loading method renders a shortcut to these tools:

(1) `frame.to_hdf('mydata.h5', 'obj3', format='table')`

(2) `pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])`

- Compared with simpler format, HDF5 supports on-the-fly-compression of different compression modes, enabling data w/ repeated patterns to be store efficiently.
  (ie. read & write {store} small sections to pile at bottom-up approach)
- A good choice when working w/ very large datasets that do not fit into memory since many data analysis are I/O-bound {network} rather than CPU-bound {local}.

NOTE: For different platforms, using its exclusive distributed storage like Apache Parquet for Amazon S3 may be more suitable.

C. Microsoft Excel Files

In Pandas method for modified XML-based Microsoft spreadsheet file format (.xlsx), the tool uses openpyxl module backend to read on the file.

```
xlsx = pd.ExcelFile(xlsx_path)
xlsx.sheet_names   #list all sheet names in file
::['Sheet1']
xlsx.parse(sheet_name='Sheet1', index_col=0)OR  pd.read_excel(xlsx, 'Sheet1')
::    a   b   c   d  message
  0  1   2   3   4    hello
  1  5   6   7   8    world
  2  9  10  11  12      foo
```

[# It may be passed to 'pd.read_excel' at ease, but slower when reading multiple sheets in a file]

```
# Store records to excel file
from openpyxl import Workbook

excel_wb = Workbook()
sheet = excel_wb.active   #use the active worksheet
sheet.append(['Country', 'Continent'])   #add 1st row as column labels
sheet.append('['Eygpt', 'Africa'])   #append each row of records
excel_wb.save('countries.xlsx')
```

When interacting with network server, *data retrieved* (i) or *data exchange* (ii) can be in the standard **[Document/ Program-]** type format.

i. Access to rendered webpage structure / documented packets— HTML & XML format
ii. Access to data packets from a software / server— JSON & SQL format

(*Web Scraping*)

! Before picking out data from an existing site,

(I) check if it is legal (II) examine how data is structured in content.

```
from bs4 import BeautifulSoup

URL = "https://en.wikipedia.org/wiki/World_population"
resp = request.get(URL, verify=False)
resp.raise_for_status()
soup = BeautifulSoup(resp.text, 'html.parser'/'lxml'/'html5lib')
tables = soup.find_all('table')   #scrape list of HTML tables

# To pull out the '10 most densely populated countries' table:
for index, table in enumerate(tables):
    # convert the NavigableString to a Unicode string
    if ('10 most densely populated countries' in str(table)):
    table_index = index
```

Pandas method always return a list of DataFrame(s)

```
population_data = pd.read_html(str(tables[table_index]), flavor='bs4')[0]
```

[# In a more concise manner, *match='10 most densely populated countries'* can be applied]

Use browser's developer tools, to effectively identify the desired HTML element on page before writing any code.

For downloading content(s) by following link from the most recent page to the oldest,

1. Loads the home page
2. Saves the image content on that page
3. Follows the Previous comic link
4. Repeat until it reaches the first comic {end}.

```
import requests, os, bs4

url = 'https://xkcd.com'
os.makedirs('xkcd', exist_ok=True)   # create &/ store in ./xkcd folder
while (prevLink.get('href') != '#'):
    # TODO: Download the page
    print('Downloading page %s ...' % url)
    resp = requests.get(url)
    res.raise_for_status()

    soup = bs.BeautifulSoup(resp.text, 'html.parser')

    # TODO: Find the URL of the comic image
    comicElem = soup.find(id='comic')

    if comicElem == None:
        print('Could not find comic image.')
    else:
        comicUrl = 'https:' + comicElem.get('src')
        # Download the image
        print('Dowloading image %s …' % comicUrl)
        resp = requests.get(comicUrl)
        res.raise_for_status

    # TODO: Save the image to .xkcd
    imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
    for chunk in resp.iter_content(100_000):
        imageFile.write(chunk)
    imageFile.close()

    # TODO Get the Prev button's url
    prevLink = soup.find('a', rel="prev")
    url = 'https://xkcd.com' + prevLink.get('href')

print('Download Done.')
```

Note:

#1 At times, a program that opens up links for a search term on browser's tabs may be handy.

#2 For more sophisticated harvesting, selenium than requests may be useful to bypass detection, allowing control over browser to programmatically click on links & fill in login information.

(Refer. Automate the boring stuff with Python)

URL is essentially a web address /+ key parameters, properly formatted in HTTP message for access, or update btw a client & web API.

(*eg.* https//jobs.github.com/positions.json?page=0&key=value+name) *—retrieve file in JSON & result on default page 0*

(*HTTP request & response transport*)

```
import requests::requests is simpler & more versatile for common HTTP-related tasks - network/ connection issues & data compression
resp = requests.get(URL, params={'page': 0, 'key': 'value name'}, stream=True)
resp.raise_for_status()  ; resp.request.headers()/ resp.headers['Content-Type']
```

[# Practice to check if HTTP actually works & header for metadata provided by server]

```
# Raw/Automatic-decoded content based on provided scheme in metadata
resp.content(2)/text/json()    ;     resp.encoding  – more on requests.Response properties & methods

with open(filename, 'wb') as fd:   #streaming download & write binary data to retain
    for chunk in resp.iter_content(chunk_size=100000)   # Unicode-encoding of text
    fd.write(chunk)

resp.close()
```

(1) The valid parameters are different depending on web API design, and only know when informed.
(2) For byte-like data, save it to a file for a complete inspection. (Refer. PythonStructuralProgramming)

D. JSON

Because of its small message size & employability in any software applications, this program-style format is preferred at data exchange (*ie.* in HTTP content usually contain a JSON file).

```
import json #Python std lib

info_obj = """   #all keys/ vars {dict.} must be strings
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "siblings": [{"name": "Scott", "age": 30, "hobbies": ["guitar", "soccer"]},
              {"name": "Katie", "age": 38, "hobbies": ["diving", "art"]}]}
"""
# Parsing JSON into Python dict vice-versa
pydata = json.loads(info_obj)  ;  asjson = json.dumps(pydata, indent=4)

siblings = pd.DataFrame(data['siblings'], columns=['name', 'age'])
:: # loading entire data as it is, likely to return nested struct. in some columns ]
    –use  pd.json_normalize(data['col'])
```

In Pandas API, it is defaulted to convert into a Series/DataFrame:

```
!cat examples/example.json
::[{"a": 1, "b": 2, "c": 3},
   {"a": 4, "b": 5, "c": 6},
   {"a": 7, "b": 8, "c": 9}]
```

```
data = pd.read_json(json_path)
::    a  b  c
   0  1  2  3
   1  4  5  6
   2  7  8  9
```

[# the default option assumes each var in JSON array is a row]

```
# To export data into JSON:
print(data.to_json())OR     row order: data.to_json(sys.stdout, orient='records')
::{"a": {"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}
```

JSON is also an approach developed to deal with sparse data, which stores relevant data only & omits the rest.

E. XML & HTML (Web Scraping)

HTML is essentially a variation of plaintext, surrounded by <element> *tags* to tell browser on how to render the pages.

Using Pandas method, it attempts to parse all tabular data within <table> tags.
(Otherwise, in native Python, the standard lib. may incl.— lxml, Beautiful Soup, and html5lib)

```
tables = pd.read_html('local_path/HTML'/file_obj, flavor='bs4')
df = tables[0]
```

```
<table >
  <tr>
    <td>Pizza Place</td>
    <td>Orders</td>
    <td>Slices </td>
  </tr>
  <tr>
    <td>Domino Pizza</td>
    <td>10</td>
    <td>100</td>
  </tr>
  <tr>
    <td>Little Caesars</td>
    <td>12</td>
    <td >144 </td>
  </tr>
</table>
```

Pizza Place	Orders	Slices
Domino's Pizza	10	100
Little Caesars	12	144

Note: <tr> defines table row ; <td> table cell

XML is another document-style formatted data packets commonly from a dynamic web display.
(*ie*. hierarchical, nested data in tags arranged as series of records than of standard doc. form)

```
from lxml import objectify   #alternately, use ElementTree lib.

with open('examples/mta_perf/Performance_MNR.xml') as fid:
    parsed = objectify.parse(fid)
root = parsed.getroot()   #parse & retrieve the root node

data = []
skip_fields = ['PARENT_SEQ','INDICATOR_SEQ','DESIRED_CHANGE','DECIMAL_PLACES']

#'root.INDICATOR' returns a generator yielding each <INDICATOR>'s element
for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():   #iterate thru each element & its child tag
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
perf = pd.DataFrame(data)
```

Similarly, the attributes & text content for a node/element can be access by—

```
root  ;  root.get('key')  ;  root.text
```

By Pandas method, the process is simplified into single line expr:

```
perf2 = pd.read_xml('examples/mta_perf/Performance_MNR.xml')
```

While `lxml` is comparatively faster, Pandas has other std libs. for better handling of a malformed HTML/XML files.

F. Interacting with Databases

Common relational database system,

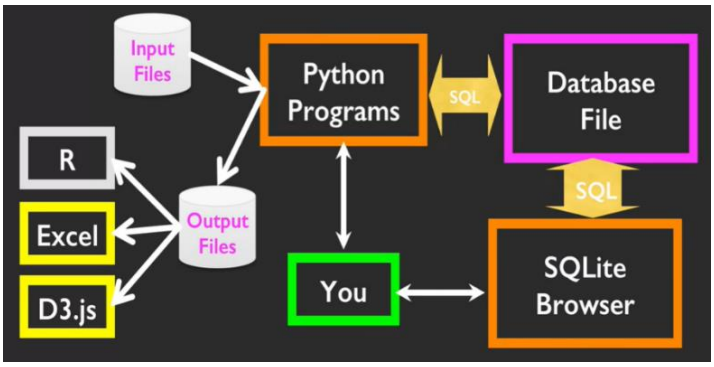
- Oracle – Large, commercial, enterprise-scale, very tweakable
- MySql – Simpler but very fast and scalable (commercial open source)
- SqlServer – Very nice, from Microsoft
- Many other smaller projects incl.— HSQL, SQLite, PostgreSQL, etc.

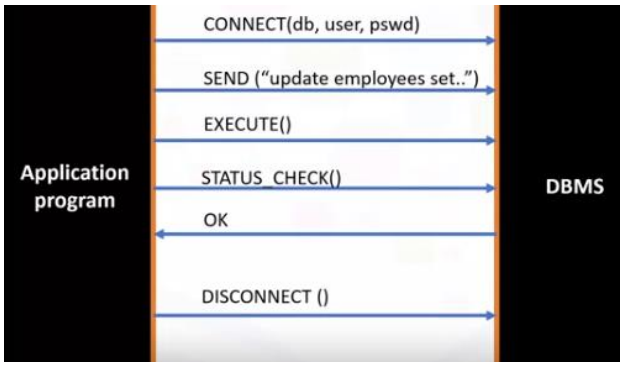
The hard part about SQL is not syntax writing, but what questions do you want to ask of the data.

Data analysts usually use SQL to interact with databases and inspect dataset on interesting subpopulations of the data they have in mind.

(—*eg.* A query to extract middle-aged men living in Northeast for examining churn rate could compose as:

```
SELECT * FROM CUSTOMERS WHERE AGE > 45 and SEX='M' and DOMICILE = 'NE' )
```





*DB-API is the driver used by lib. (eg. sqla) for different database systems to connect to Python program.

Internally, Python program uses sqlite3 DB-API to access the database:

```
import sqlite3   standard DB-API- IBM DB2: Ibm_db/ibm_db_sa ; PostgreSQL: psycopg2 ; MongoDB: PyMongo

conn = sqlite3.connect('music.sqlite')      # import database file
cur = conn.cursor()      # access file as handle

# Primary SQL instructions incl.: # create table # retrieve # insert # update # delete {CRUD}
cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)', ('Thunder', 20))
conn.commit()   # commit / rollback SQL statements as of ACID control

cur.execute('SELECT `title`, plays FROM Tracks')   *double quote/ back-tick for mixed-case & sp. chr name
rows = cur.fetchall()   # return result in rows of tuple
data = pd.DataFrame(rows, columns=[x[0] for x in cur.description])

conn.close()
```

Cursor's methods conduct / retrieve row(s) of command to the database—

execute, [2] executemany, [3] executescript, [4] fetchone, [5] fetchmany(*n*), [6] fetchall, close.

Note: Managing views & ability to save results as new tables are useful when original dataset changes continuously and you only interested in specific set of data for further analysis.

#1 Use '?' or %(*name*)s – as pyformat style placeholder for tuple / dict: **(value,) ; {name: value}**

#2 Cursor '`description`' provides details of each column, but only column label for SQLite.
:: (('title', None, None, None, None, None, None),
('play', None, None, None, None, None, None),

#3 The commit (*ie*. queue to disk/cloud) should be conducted at optimal frequency to preserve the efficiency of writing process while retaining minimal data in queue that may be lost.

SQLAlchemy has many dialects of relational databases, and abstracts away the need of separate DB-API implementations which the DB-API is readily installed.

```
import sqlalchemy as sqla
engine = sqla.create_engine('sqlite:///mydata.sqlite')
# Retrieve query result into a DataFrame:
df = pd.read_sql(sql_query, engine)
::              a           b     c  d
  0       Atlanta     Georgia  1.25  6
  1  Tallahassee     Florida  2.60  3
  2    Sacremento  California  1.70  5
```

[# As SQLite is a file-based database (connects to local files), the Database URL is simply DBAPI & filepath]

(For external connection, first we'll need to retrieve the credentials from database)

Note: - When working on a Notebook, connection may be set up directly (**%sql ibm_db_sa:// …**), which employs sqlalchemy on backend.:: !pip install-ed ipython-sql & %load_ext sql

{! a. To reference Python variables inside an SQL statement, use- WHERE name = :Pyvar / {var} ;

b. Otherwise, assigning single / multiple SQL lines to a Python variable at–

:: a = %sql SELECT * FROM t \
LIMIT 10

or,

%%sql result << SELECT product, price FROM Book
ORDER BY price DESC
LIMIT 10;

c. To complete for analysis-ready, SQL result set is converted into more versatile DataFrame structure–

:: dataframe = result.DataFrame( ) }

When the dataset to be analyzed is available as .CSV file from an internet source,
if SQL is demanded by the application, it first needs to be read & stored in the database.

```
conn = sqlite3.connect('RealWordData.db')
%sql sqlite:///RealWorldData.db

df = pd.read_csv('filepath.csv')   # load the downloaded CSV file by
df.to_sql('table_name', engine/sqlite3.connection, if_exists='replace'
          index=False, method='multi')

%sql SELECT * FROM table_name LIMIT 5   # verify the table creation
```

SQL database utility is preferred in the case of —

- when application requires interactive & periodic updates within a large dataset,
- the data is too large to fit in a dict. and requires repetitive retrieval, or
- to sustain a long-running process, enabling stop and restart w/o losing previous data.
  (eg. scraping web API for data using Python & SQL)

Oftentimes, we would like to inspect the metadata about a schema to get an intuition before working on the data.

DB2: **syscat.tables** ; SQL Server: **information_schema.tables** ;
Oracle: **all_tables / user_tables** ; SQLite: **sqlite_master**

(1) Retrieve a list of tables & their properties by querying the system catalogue at—

DB2
```
SELECT * FROM syscat.tables   /   SELECT tabschema, tabname, create_time
                                  FROM syscat.tables
                                  WHERE tabschema = 'LCT12330'
```
-schema / ibm_username

(2) To obtain all column names for a table or, specific column properties at—

DB2
```
SELECT * FROM syscat.columns   /   SELECT distinct(name), coltype, length
WEHRE tabname = 'Table_name'       FROM sysibm.syscolumns
                                   WHERE tabname = 'Table_name'
```

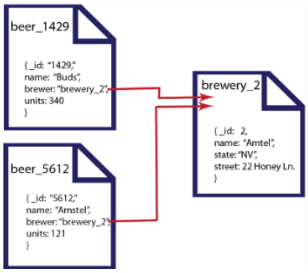
SQLite
```
SELECT DISTINCT name          /    SELECT distinct(name), type
FROM PRAGMA_TABLE_INFO('Tbl')      FROM PRAGMA_TABLE_INFO('Tbl')
```

Note: For MySQL, they can be retrieved simply by – SHOW COLUMNS FROM db_name.table_name

Non-relational databases & *search-based data models* are typicaly for low latency application development and searching of log [unstructured data] files as—

(1) Key-value store

- ✓ data is serialized as collection of key-value pairs where both can be anything from simple int/string to JSON document.
- ✘ Not fit for:
  - query on specific data value
  - need relationship btw records & multiple unique keys



(2) Document-store

- ✓ serialize each record and its associate data into a single document, enabling flexible indexing, powerful ad hoc queries & analytics over collections of documents.
- ✘ Not fit for:
  - running complex search query
  - performing multi-operation transactions

(Preferred for eCommerce / CRM platform, and medical records storage)

(3) Column-store

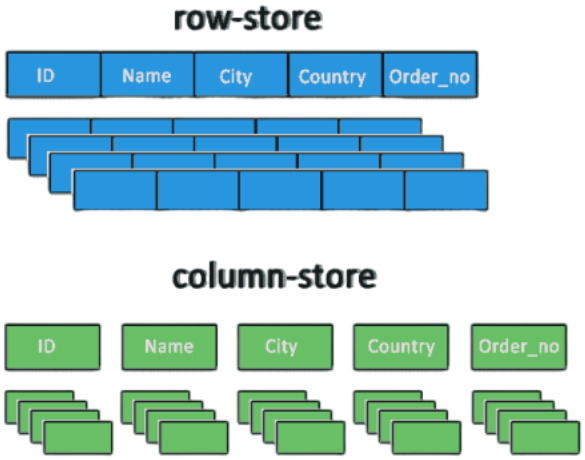
- ✓ serializing data into columns w/o enforcing primary keys & indexes to join tables, and optimized for row-intensive query at few columns, and large data repositories.
  (where it is aligned w/ continuous disk entry, allowing easier : faster access & search)
- ✘ Not fit for:
  - running complex search & changing patterns query frequently
  - frequent updates & deletes, as a record is distributed across multiple columns

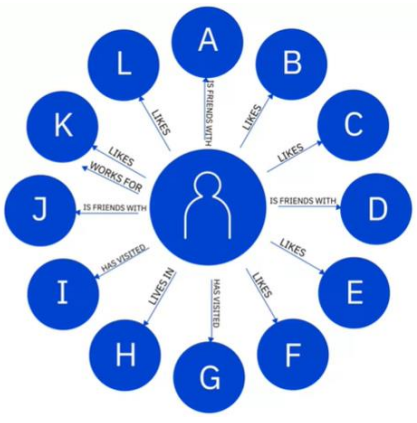
(Preferred for heavy write requests, time-series & IoT data)

(4) Graph-store

- ✓ uses a graphical model : nodes to represent & serialize data
  (which is useful for visualizing and analyzing connections btw records)
- ✘ Not fit for:
  - processing high volume of transactions

(Preferred for product recommendations, Fraud detection & Access management)





* Column-store is efficient at storing large volumes of data w/ missing or repeated values by allowing a surrogate value along with a lookup to the full value & compression.

COMBINING DATASETS:

V. Concat & Append

- extend rows / columns along an axis of Pandas structures based on the relational Set manner.

```
pd.concat(dfs, axis=0, join='outer', ignore_index=False, keys=None, levels=None,
          names=None, verify_integerity=False, sort=False, copy=True)
```

By default, concatenation takes place in row-wise with few useful options:

(A) Catching the repeats as an error.

—raise an exception at duplicate indexes from datasets.

```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as err:
    print('ValueError':, err)
::Value Error: Indexes have overlapping values: [0, 1]
```

(B) Ignoring the index.

—adopt / ignore previously defined indexes on tables that may be duplicated.

(C) Adding multi-index keys.

—dict. keys {if a mapping of `dfs` is passed} or, new labels to construct hierarchical index at the outermost level

```
print(x); print(y); pd.concat([x, y], ignore_index=True, keys=['x', 'y'])
::   A   B      A   B         A   B
  0 A0  B0   0 A2  B2   x  0 A0  B0
  1 A1  B1   1 A3  B3      1 A1  B1
                       y  2 A2  B2
                          3 A3  B3
```

Subsequently, methods from 'Hierarchical Indexing', can be applied to transform the data.

(D) Concatenate with set operations.

—standard database joins instead of union (join='outer').

* Utilize logic, common sense & experience to understand how different datasets can be compatibly integrated to provide the common interest / goal.

Note: .append( ) is very inefficient as it creates new index & data buffer for every inclusion.

—For multiple append( ) operations, build a list of DataFrames and pass them to concat( )

```
eg. nutrients = [ ];          for rec in db:
                                  fnuts = pd.DataFrame(rec['nutrients'])
                                  fnuts['id'] = rec['id']
                                  nutrients.append(fnuts)
    nutrients = pd.concat(nutrients, ignore_index=True)
```

VI. Merge & Join

- database-style that combines relational objects based on common entities btw the two.

The behavior encompasses ⊂ relational algebra (*ie.* set operations, relational processes, agg functions) as the conceptual foundation of operations in most databases.

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
```

Type of joins: (*one_to_one, many_to_one/one_to_many, many_to_many*)

[# the type may be specified in parameter as checker— validate='one_to_one'/ '1:1']

The method will raise an error if validation fails,
eg. MergeError: Merge keys are not unique in either left or right dataset; not a one-to-one merge

1_ **One-to-one:** join by intersecting column's unique *keys* on both datasets.

```
df3 = pd.merge(df1, df2)
::   employee       group  hire_date
0       Bob   Accounting       2008   [# Merge discards the original index,
1      Jake  Engineering       2012      except specified merge on index option]
2      Lisa  Engineering       2004
3       Sue           HR       2014
```

2_ **Many-to-one:** join w/ 1- duplicate values & 1-unique identifier on left/right datasets.

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
pd.merge(df3, df4)
::     employee       group   hire_date  supervisor
       0     Bob   Accounting       2008       Carly
       1    Jake  Engineering       2012       Guido
       2    Lisa  Engineering       2004       Guido
       3     Sue           HR       2014       Steve
```

3_ **Many-to-many:** values of the column in left & right dataset can have duplicates.

```
df5 = pd.DataFrame({'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR', 'HR'],
                    'skills': ['spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
pd.merge(df1, df5)
::        employee       group        skills
          0     Bob   Accounting  spreadsheets
          1    Jake  Engineering        coding
          2    Jake  Engineering         linux
          3    Lisa  Engineering        coding
          4    Lisa  Engineering         linux
          5    Sue           HR  spreadsheets
          6    Sue           HR  organization
```

Alternatively, the structured data can be joined by SQL-relational practice w/ how={'left', 'right', 'outer', 'inner', 'cross'}

```
pd.merge(left_obj, right_obj, how='inner', on=None, left_on=None,
         right_on=None, left_index=False, right_index=False, sort=False,
         suffixes=('_x', '_y'), copy=True, indicator=False, validate=None)
```

The options provide various ways to handle non-perfectly match column's values as keys:

(A) Explicitly specify the column

—identify the column name / list of column names to be used as logical keys of join.

(B) Join datasets on unique keys from different column labels.

```
df_opt1 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                        'salary': [70000, 80000, 120000, 90000]})
pd.merge(df1, df_opt1, left_on='employee', right_on='name')
  ::  employee        group  name  salary
    0      Bob   Accounting   Bob   70000
    1     Jake  Engineering  Jake   80000
    2     Lisa  Engineering  Lisa  120000
    3      Sue           HR   Sue   90000
```

[# The redundant column:'name' can be dropped using .drop('name', axis=1) subsequently]

(C) Merge on indexes.

—specify left_index and/or right_index flags as placement of the new index object.

```
df_opt2a = df1.set_index('employee'); df_opt2b = df2.set_index('employee')
pd.merge(df_opt2a, df_opt2b, left_index=True, right_index=True)
  ::              group  hire_date
    employee
    Lisa     Engineering       2004
    Bob       Accounting       2008
    Jake     Engineering       2012
    Sue              HR       2014
```

(D) Specifying a relational / mathematical set manner. { *SQL- Join* }

—{how='inner'}: join at intersection of keys on both tables

—{how='outer'}: union of the columns' keys, and fills missing values as NAs

—{how='left'/'right'}: join at all keys of left/right column & intersect keys

—{how='cross'}: join at the product of all combination in keys from both tables

```
pd.merge(df_opt3a, df_opt3b, how='left')
  ::    name   food         name  drink          name   food  drink
    0  Peter   fish    0    Mary   wine  =>  0  Peter   fish    NaN
    1   Paul  beans    1  Joseph   beer      1   Paul  beans    NaN
    2   Mary  bread                         2   Mary  bread   wine
```

(E) Overlapping column names.

—automatically add suffix _x / _y or custom suffix to make append columns unique

```
pd.merge(df_opt4a, df_opt4b, on='name', suffixes=['_L', '_R'])
  ::   name  rank       name  rank        name  rank_L  rank_R
    0   Bob     1    0   Bob     3      0   Bob       1       3
    1  Jake     2    1  Jake     1  =>  1  Jake       2       1
    2  Lisa     3    2  Lisa     4      2  Lisa       3       4
    3   Sue     4    3   Sue     2      3   Sue       4       2
```

FEATURE ENGINEERING

The common transformation to modify existing features in a way that improves accuracy in a modeling context after the decision is made on meaningful features **:-**

[1] Normalization / Standardization: bring features onto the same scale / range w±wo accounting for statistical variance.
[2] Synthetization: create new feature(s) from the older ones.
[3] Discretization: replace feature's values by an equally sized or count bins.
[4] Encoding: construct dummy var or linear representation of factors.

I. Feature Scaling

(Standardization) or, *Z-score Normalization*

```
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] – X[:, 0].mean( )) / X[:, 0].std( ))
X_std[:, 1] = (X[:, 1] – X[:, 1].mean( )) / X[:, 1].std( ))
```

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler( ).set_output(transform='pandas')    # scale at Std ND
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

Statistical standardization that centers features data to **μ**=0, **σ**=1 for—

1. Interpretability of coefficients
2. Ability to rank the relative coefficient by importance at post-shrinkage coefficient of estimates
3. No noeed for intercept

to facilitate *effective* & *equal weight magnitude* of optimization towards {local/global} convergence at uniform learning rate across features.

(Min-max Normalization)

$$X_{normalized} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler( ) .set_output(transform='pandas')
X_train_norm = mms.fit_transform(X_train)    # bringing features onto the same scale [0, 1]
X_test_norm = mms.transform(X_test)
```

* We can always try scaling the predictor variables in modeling which may help besides optimization.

Note: For any optimization or distance-based solving technique, feature scaling is encouraged.
(*if target variable of training set is normalized, the test set should be adjusted similarly*)

II. Encoding categorical data

To get categorical data into shape for modeling & ML as well as they work with numerical data,

	color	size	age	price	targetlabel
0	green	M	kid	10.1	class1
1	red	L	adult	13.5	class2
2	blue	XL	kid	15.3	class1
…					…

* Generally, the approach depends on whether categorical var is equal or hierarchical weight in relation to the target variable (# For Multinomial classification, it is dummy encoded & performed by One-vs-All strategy).

(A) Mapping / Encoding ordinal features.

```
from sklearn.preprocessing import OrdinalEncoder, LabelEncoder, OneHotEncoder
# Ordered & Reverse weight mapping: (however, make sure the case labels are consistent)
size_mapping = {'XL': 3, 'L': 2, 'M': 1}
df['size'] = df['size'].map(size_mapping)
inv_mapping = {v, k for k, v in size_mapping.items()}
```

[# Sometimes, 0-value is undesired in arithmetic operation due to its indeterminate weight / sparse property.]

Similarly, it can be transformed by pre-defined Scikit-learn method, or merely dS.cat.codes:

```
ord = OrdinalEncoder()  # classes btw {0, n-1}
cat_x = ord.fit_transform(df[['size', 'age']].values)  ;  ord.inverse_transform(cat_x)
```

For target labels {y}, LabelEncoder( ) which incorporates 1-D array should be used:

```
1- class_le = LabelEncoder( )  #not necessarily equally spaced
2- class_mapping = {label:idx for idx, label in
                    enumerate(np.unique(df['targetlabel']))}
```

Value	Description
1	Cabin
2	Substandard
5	Fair
10	Very good
12	Luxury
13	Mansion

(B) Encoding nominal features.

```
ohe = OneHotEncoder(categorical_features=[0], drop='first')
ohe.fit_transform(df.values).toarray() # translate sparse matrix to dense array,
                                       or use sparse=False
```

```
df = pd.get_dummies(df, drop_first=True, columns=['color', …])
::    price  size  color_green  color_red
  0   10.1    1        1          0
  1   13.5    2        0          1   # w/o unintended equidistant relationship or ranking
  2   15.3    3        0          0
```

Note: 1. 'pd.get_dummies' method can conveniently implement on DataFrame, but only convert string / category datatype cols- use df[cols_to_str].astype(str).

2. Multicollinearity in original features won't badly affect tree regr, clustering & nearest-neighbours (*X* solved by matrix inversion) which information should be retained but consider using PCA.

3. However, it is always advisable to remove redundant var created in dummy var (1st ref level), of where the weight in the normalized distance btw 2-points will be doubled.

Besides the focus here, there are several different ways to encode, known as *contrast* encoding system.

III. Extracting useful information from data

Sometimes, new features may be produced using domain knowledge from existing ones to deliver more predictive power.

① For example, if the bank computer were known to have glitch that caused many credit card transaction mistakenly declined in the past month, it would be reasonable to suspect that the incident is likely to have an impact on churn at:

$$\text{OctUseRatio} = \frac{\text{no. of Oct. trx}}{\text{avg no. of monthly trx}}$$

[*eg.* high ratio of impacted customers by declined trns are more likely to churn.]

② Or, it can be as simple as creating a feature that weighs each customer at:

$$\text{Loyalty} = \frac{\text{Tenure}}{\text{Age}}$$

[*eg.* people who have been customers for a greater proportion of their lives might be less likely to churn.]

DATA CLEANING & MANIPULATIONS

During the manipulation of data, recognizing the {dtype, unique values, range} of a variable is helpful on what : how it can be operated, analyzed, or aggregated with other variables.

I. Handling missing data & outliers —

The general stretegies to indicate missing data in a relational table incl.:

(A) Masking: Separate Boolean array, or appropriation of 1-bit in data representation from the 64-bit OS that globally indicates missing values.

(B) Sentinel: Replace/Allocate rare value from range of numerical values.
(eg. global convention—NaN, rare IEEE floating-point value/ bit pattern)

Pandas exploits sentinels on— { existing Python null [1] floating values: *NaN* ; [2] object: *None* }

i. `None`: Pythonic missing data

- infers Python object, whereby any operations on the data will done at Python level, conforming to its overheads {metadata}.
(ie. Cannot be used in favor of any NumPy/Pandas structure, but vectorized string ops)

```
vals1 = np.array([1, None, 3, 4])
::array([1, None, 3, 4], dtype='object')
```

[*The aggregation function btw* `int` & `None` *would be undefined* (#error)]

ii. `NaN`: Numerical missing data

- special IEEE floating-point value recognized by all system that supports fast operation.

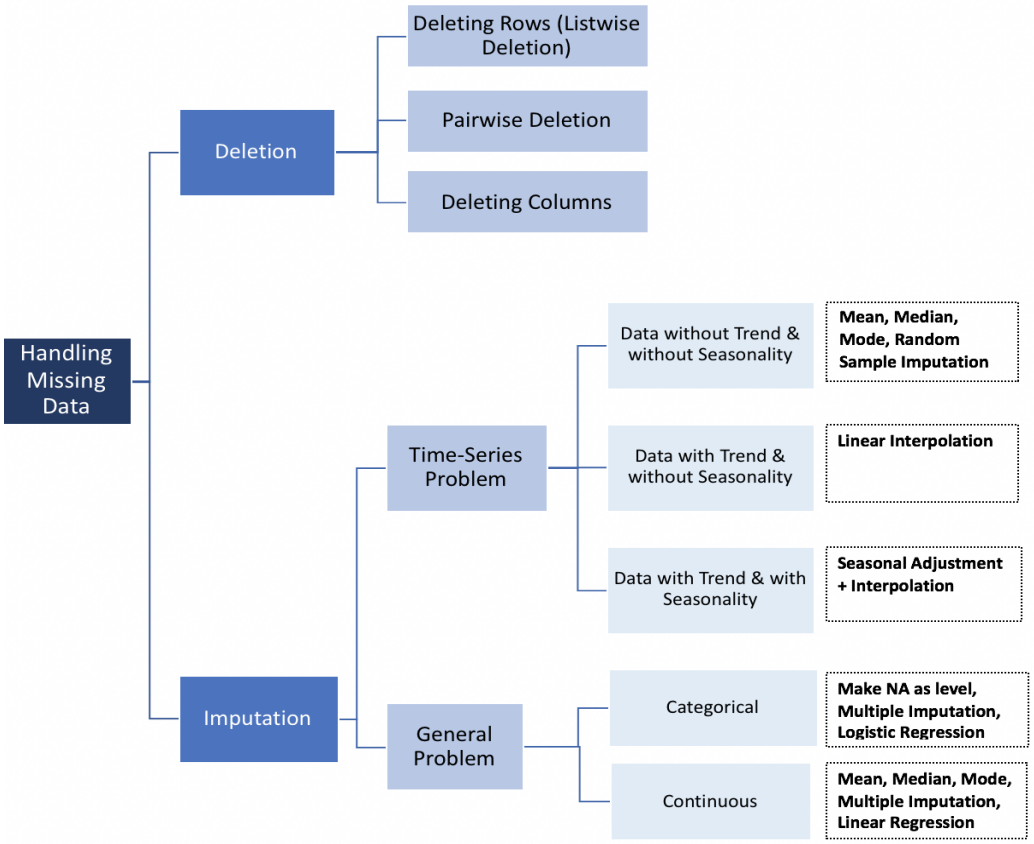
Note: Any arithmetic operation with `NaN` will be result in another `NaN`, eg. data-virus

Pandas handle 'NaN' and None nearly interchangeably, converting btw them where appropriate.
eg. For type of non-available sentinel value, Pandas automatically type-casts (fit others)

```
pd.Series([1, np.nan, None])
::0  1.0
  1  NaN
  2  NaN
dtype: float64
```

Typeclass	Conversion of NA	NA sentinel value
`float`	No change	`np.nan`
`object`	No change	`None/np.nan/pd.NA`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None/np.nan/pd.NA`
`datetime`	No change	`NaT/pd.NaT`

Note: Pandas also has 'pd.NA' that is an alias of NA.



How to deal with missing data?

- Check with the source to fill in on large missing data
- Impute / Delete based on its variant
- Create a separate category until the underlying flaw / bias in collection process is identified

@Depending on the impact to project analysis, one can choose to discuss the inadequency to progress through with the client or depict the exclusion by note.

1. Missing Completely at Random (MCAR)

—the missing values are randomly distributed across all groups of the target being studied.

(eg. *People of all income groups generally do not like to reveal their age, or non-associative missing*)

! Chi2-test to determine if the missing is by chance in each group, of which the expected values are to follow a uniform random distribution ($P = total\_freq_{missing} * 1/n_{groups}$) -binning first for numerical features

2. Missing at Random (MAR)

—the missing values can be predicted, or forecast from other variables, but not its own.

(eg. *The missing in* 'num-of-doors' *is related to* 'body-style' *of a car, where it can be imputed by the mode of freq as- 84% sedan is 4 doors*)

3. Missing Not at Random (MNAR)

—the missing values have a pattern that is dependent on, which simply removing it will induce bias on the resulting data.

(i) Hypothetical target value: *People with high salaries generally do not want to reveal their incomes.*

(ii) Other variable's value: *Age 25-35 do not want to reveal their incomes in a salary survey.*

I. Consider `does that change the data story` & `what are the ethical implications` when determining the strategy.

II. Remove columns (1) & (2) at ≥50% missing ; rows at unresolved entries in target or small random group at <20% total.

[Fill & Deletion]

Null values operation: `# Verify NA-related values as Boolean mask / Scalar`

- isin( ), isnull( ), notnull( ), isnull( ).any( ), notnull( ).all( )

`# Remove / Fill NA values`

- dropna( ), fillna( ), interpolate( )*equiv. to MATLAB' `interp1()`'*

(* Documenting where the missing data and how it was replaced is important, as it may impact the downstream interpretation and use of the data.)

Analyze the system prior to cleaning. *—Understand its root cause to control / rectify it.*
Missing data is not always ill-condition that needed to be fixed right away, but can be an important sign to underlying design flaws or biases in the data collection process.

```
# Complex selection logic
mask = ((df['c1']=='v1') & (df['c2'].isin([...]))
        ) | ((df['c1']=='v2') & (df['c2']==s1))
df[mask]
```

create 1-D Series as row indexer than 2-D entries selection

```
# Remove based on missing criteria:
df.dropna(thresh=2, subset=[1,2], axis=0)
::      0    1  2   # equiv. to,
  1  2.0  3.0  5   # df[df.isnull( ).any(axis=1)]
```

```
# Find missing dates in daily records by comparing to a date series:
full_date_range = pd.date_range(start='2018-01-01', end='2018-12-31')
full_date_range.difference(df['date'])
```

```
# 1-D interpolation method of value
df.interpolate(method='linear')
```

(Refer. piecewise interpolation, LinearRegression)

```
# Fill entries: Sequential forward/back
df.fillna(method='ffill', limit=2)
df.fillna({'B': 0, 'C': df['C'].mean()})
```

[# The choice of value to fill NA often involves making assumptions about the typical value]

! Libraries like statsmodels & scikit-learn generally cannot be fed with missing values.
(*to identify columns / rows with missing value, use*— `df.isnull().sum(); df[df.isnull().any(axis=1)]`)

🔀 Outside of EDA, ML & regression modeling have more complex variations on imputing values for missing / outlier:

```
from sklearn.impute import SimpleImputer

# mean / median / most_frequent method
imr = SimpleImputer(missing_values='NaN', strategy='mean', axis=0)
imputed_data = imr.fit_transform(df.values)   # data transformation
```

* No. of features in transform & fit data array needs to be identical

Types of Outlier:

I. **Global,** entries that are completely different from the overall data values & have no association with any other outliers. (eg. sets of human height - {1.7, 1.9, 1.6, 7.9, 1.8})

II. **Contextual,** normal entries under standalone feature / condition but become anomalies when associated with others, most commonly in time-series data. (eg. huge spike in movie sales after a decade)

III. **Collective,** a group of abnormal points that follow a similar pattern and are isolated from the rest of the group. (eg. occupied parking lot after store hour)

⋮ Effective strategy is devised based on the type in a numerical var via boxplot & histogram visualization.

How to deal with outliers?

- Delete them: If outliers are certain to be mistakes, typos, or errors and the dataset will be used for modeling / ML, provided that rectifying it is not an easy option either.
- Reassign them: If the dataset is small and/or the data will be used for modeling / ML, changing to values that fit within the general distribution of data is more desirable.
- Leave them: For a dataset that is planned to conduct solely on EDA / Analysis, or preparing for a model that is resistant to outliers, keeping it is a viable option.

[# Either decision that is chosen should take into account the nature of the outlying data & assumptions of the building ML model based on project plan.]

* For big data problems, outliers are generally not a problem in a fitting model.
Instead, they are central to anomaly detection, where outliers carry the critical diagnostic pattern.

```
#1 Capping feature based on intuition & statistics:
tenth_percentile = np.percentile(df['num_of_strikes'], 10)
ninetieth_percentile = np.percentile(df['num_of_strikes'], 90)

df['number_of_strikes'] = df['number_of_strikes'].apply(lambda x: (
    tenth_percentile if x < tenth_percentile
    else ninetieth_percentile if x > ninetieth_percentile
    else x))

#2 Reassign outliers so that distribution based solely on non-outlier values:
df['col'] = np.where(df['col'] < lower_limit, median, df['col'])
```

Ex. When reviewing two years sales data at a retail business, an abnormal one month sales of a typically popular item are down dramatically.

① *It may be assumed as typo in the reported data, but as a diligent data professional, questions are raised and discovered - top salepeople were on leave & new product advertised.*

② *Leaving them & using this information, strategy can devised ahead of product launch to prevent a future drop in sales numbers by hiring additional sales member or limitng requested employee leave.*

Typically, global outlier should always be eliminated to warrant an accurate predictive model.

II. Handling duplicates & invalid entries—

'.duplicated()' returns a Boolean Series by checking against previously existed rows/ values.

```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                     'k2': [1, 1, 2, 3, 3, 4, 4]})
data.duplicated()  ;  data.drop_duplicates(keep='first', ignore_index=True)
 ::0  False            ::    k1  k2
   1  False              0  one   1
   2  False              1  two   1
   3  False              2  one   2
   4  False              3  two   3
   5  False              4  one   3
   6   True              5  two   4   # specify subset=['col'] for certain criteria if needed
   dtype: bool
```

[# Useful against duplicate index in import data, *ie.* complicate indexing by returning: Series > scalar.]

Data Mapping

```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'pastrami'
                     'corned beef', 'bacon','pastrami', 'honey ham', 'nova lox'],
                     'ounces': [4, 3, 12, 6, 0, 8, 3, 5, np.nan]})

# Organize a collection of desired key-value mapping
meat_to_animal = {'bacon': 'pig', 'pulled pork': 'pig', 'pastrami': 'cow',
                  'corned beef': 'cow', 'honey ham': 'pig'}
data['animal'] = data['food'].map(meat_to_animal)
::          food  ounces  animal
  0        bacon     4.0     pig
  1  pulled pork     3.0     pig
  2        bacon    12.0     pig
  3     pastrami     6.0     cow       [OR configure- lambda x: meat_to_animal[x] ]
  4  corned beef     0.0     cow
  5        bacon     8.0     pig
  6     pastrami     3.0     cow
  7    honey ham     5.0     pig
  8     nova lox     NaN     NaN

# To replace whole matching value in the entries:
data['ounces'].replace([6.0, 7.5], [0, np.nan])OR  ({6.0: 0, 7.5: np.nan})
```

* For vectorized operation on string, use str.replace to alter whole / part of the matching chrs pattern.

General Notes:

(Trade-offs of different conventions)

- Separate mask array allocates a new additional Boolean array (besides: True/False), in both storage and computation.
- Whereby, sentinel value reduces the range of valid values that can be represented, and may require extra (non-optimized) logic in CPU and GPU arithmetic.

There is no best universal solution, the different convention is adopted based on preference

" *eg.* ***R*** ~ uses reserved bit pattern within each datatype as sentinel value;
SciDB ~ uses an extra byte attached to every cell to indicate a NA state. "

Referring to Pandas object, the approach may not be perfect but practical for many users.

ie. *R contains 4-basic datatypes, NumPy supports 14-basic for integer types alone*
—accounting to available precisions, signedness, and endianness of the encoding.

i. Deriving a new Boolean array will elevate unnecessary overheads in storage, computation, and code maintenance.
ii. Sacrificing a bit as a mask, will significantly reduce range of values in smaller datatype (eg. 8-bit integer).
iii. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types.

III. Handling noisy data — {inaccurate values}

[Binning]

☑ Sort & categorize numerical values in bins, whereby smoothing can be applied at individual bin mean/median/boundary to distinguish potential outliers or error values.
(*eg.* data = [2, 3, 3, 2.5, 9], the values can be smoothen &/ replaced by a median value)

Note: If the values are clearly not mistakes, it should be taken into account in dataset as a whole.

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
bins = [18, 25, 35, 60, 100]

# Categorize values into equal or custom width bins: pd.cut / pd.qcut
cats = pd.cut(ages, bins, right=True, labels=None) # `label` to specify list of bin names
::[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (35, 60], (25, 35]]
  Length: 12
  Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]

pd.value_counts(cats)  # Categorical dtype
::(18, 25]   5
  (35, 60]   3
  (25, 35]   3
  (60, 100]  1
```

Alternately, *Regression* / *Clustering* approach may be used for smoothing noisy data based on specific feature(s), and *Rolling Time Windows* for timeseries data.

IV. Handling Categorical Type

- Pandas' special datatype for holding data that can be represented by integer-based (code) categorical encoding. (Refer. Databases, PythonStructuralProgramming)

The approach is similar to how data systems manage fast overhead reference with lesser scanning {chr store & compare}, yielding an improve in computational performance.

(***eg.*** in a data model, relational tables utilize the unique integer values as key to store the primary records for ease of reference btw each tables)

```
fruit_cate = df.astype({"fruit": 'category'})   #dtype converter
::0   apple
  1  orange
  2   apple
  3   apple
  4   apple
  5  orange
  Name: fruit, dtype: category
  Categories (2, object): [apple, orange]
```

[# The repeated values are more efficiently stored & processed in categorical type.]

```
fruit_cate.cat.codes   # retrieve its int representation
::array([0, 1, 0, 0, 0, 1], dtype=int8)
fruit_cate.cat.categories
::Index(['apple', 'orange'], dtype=object)
```

—It may seem counterintuitive to convert to a categorical type in order to then transform it to numerical data, but the process splits data into useful classes instead of anonymous strings.

```
# Categorical object Constructor:
month_ord = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
cate = pd.Categorical(df['month'], categories=month_ord, ordered=True)

type(cate)
::pandas.Series / pandas.core.categorical.Categorical {depending on input}
```

* To instate an unordered categorical at a later stage, dS.cat.as_ordered( ) can be used.

To alter the representation of values for categorical encoded data,

```
categories = ['foo', 'bar', 'baz'];    codes = [0, 1, 2, 0, 0, 1]
cate_2 = pd.Categorical.from_codes(codes, categories, ordered=True)
::[foo, bar, baz, foo, foo, bar]
  Categories (3, object): [foo < bar < baz]
```

Note: The categorical array can consist of any immutable value types, not only string (as shown).

Bucketing, sorts & classifies a continuous var into categorical intervals of its own—

```
draws = np.random.randn(1000)
bins = pd.qcut(draws, 4, labels=['T0', 'T1', 'T2', 'T3'])
::[T1, T2, T1, T3, …, T2, T1, T0, T2, T3]   # ordered cat into equal quantile buckets
  Length: 1000
  Categories (4, object): [T0 < T1 < T2 < T3]
bins = pd.Series(bins, name='quartile')
results = (pd.Series(draws)   # GroupBy operation is significantly faster w/ categoricals
          .groupby(bins)
          .agg(['count', 'min', 'max'])
          .reset_index())
::    quartile  count       min       max
  0         T0    250  -2.949343  -0.685484
  1         T1    250  -0.683066  -0.010115
  2         T2    250  -0.010032   0.628894
  3         T#    250   0.634238   3.927528
```

Categorical Methods

(1) When the actual set is known, '`set_categories`' method can be used for revising the categories as:

```
actual_categories = ['a', 'b', 'c', 'd', 'e']
cate_s2 = cate_s.cat.set_categories(actual_categories)
cate_s.value_counts()    ;    cate_s2.value_counts()
::d  2                         :: d  2
  c  2                            c  2
  b  2                            b  2
  a  2                            a  2
  dtype: int64                    e  0
                                  dtype: int64
```

[# The data may appear unchanged, but will be reflected in operations that use them]

(2) Whereby, in subset to a larger Series, '`remove_unused_categories`' method removes the omitted rank for needless computation:

```
cate_s3 = cate_s[cat_s.isin(['a', 'b'])]
cat_s3.cat.remove_unused_categories()
::0  a                                  :: 0  a
  1  b                  =>                 1  b
  4  a                                     4  a
  5  b                                     5  b
  dtype: category                         dtype: int64
  Categories (4, object): [a, b, c, d]  Categories (2, object): [a, b]
```

List of other methods—

[*'add_categories', 'as_unordered', 'remove_categories', 'rename_categories', reorder_categories*.]

VECTORIZED STRING OPERATIONS — {Fixing irregular context / format}

- The efficient retrieval & handling of each Pandas object's value -index, Series, DataFrame containing str with a string operation via Pandas UFunc {*element-wise manner*}.

Vectorized operation dismissed manifestation in the needlessly recursive actions,
—*ie*. omitted the overhead tag verification & function dispatch on each individual element.

```
data = ['peter', 'Paul', None, 'Mary', gUIDO']
[s.capitalize() for s in data]
:: AttributeError
[# cannot operate on 'NoneType' ]
```

```
names = pd.Series(data)
names.str.capitalize()
::0  Peter
  1  Paul
  2  None
  3  Mary
  4  Guido
  dtype: object
```

[# size & shape of DataFrame will not affect the efficiency of operation, and should not be a concern]

*NumPy is generally structured for numerical computations, and does not provide simple access for string datatype & missing value (*where it is not readily identified as its own type for separate processes*).

> For Pandas' type specific attributes & methods,
> the {**string, categorical, datetime**} object is to be preceded by— {`.str`, `.cat`, `.dt`}
> [*eg*. pd.Series.dt.month_name( )] to specify for the respective class.

Mapping btw Pandas & function in 're' module

Method	Description	Method	Description
`match()`	returning Boolean of each element	`contains()`	re.search() on each element
`extract()`	returning matched groups as strings	`count()`	count occurances of pattern
`findall()`	re.findall() on each element	`split()`	str.split(), but accepts regexps
`replace()`	replace occurances with others	`rsplit()`	str.rsplit(), but accepts regexps

Note: split is often combined w/ strip to trim whitespaces off the substrings.

Pandas string miscellaneous methods

Method	Description	Method	Description
`get()`	Index each element	`repeat()`	Repeat values
`slice()`	Slice each element	`normalize()`	Return Unicode of string
`slice_replace()`	Replace slice with passed value	`pad()`	Add whitespace to L/R/Both
`cat()`	Concatenate strings	`wrap()`	Split long strings into lines /w lengthless than given width
`get_dummies()`	Extract dummy variables as DataFrame	`join()`	Join strings in each element with passed separator

List of Pandas str methods that mirror Python string methods:

len()	lower()	translate()	islower()
ljust	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()
center()	rfind()	isalnum()	isdecimal()
zfill()	index()	isalpha()	split()
strip()	rindex()	isdigit()	rsplit()
rstrip()	capitalize()	isspace()	partition()
lstrip()	swapcase()	istitle()	rpartition()

I. Indexing & Slicing vectorized item.

`df.str[0]`
::returning the nth chr of each value

`df.str[0:3]`
::slicing the text from 0th to nth chrs of each value.

```
# To retrieve indexer for specific string value(s) :
df.index.get_indexer(['John', 'Terry', 'Eric'])
::array([0, 1, 2])
```

For method chaining {pipe}: `monte.str.split().str.get(-1)` / `monte.str.split().str[-1]`

II. Indicator variables.

—useful for splitting code indicator column (here: A: USA, B: UK, C: Amateur, D: Wealthy) separated by a delimiter into dummy variables.

```
(1) full_monte = pd.DataFrame({'name': monte,
                               'info': ['B|C|D', 'B|D', 'A|C', 'B|D',
                                        'B|C', 'B|C|D']})
dummies = full_monte['info'].str.get_dummies(sep='|')
full_monte_dummies = full_monte['name'].join(dummies)
::    info       name                    A  B  C  D      name
  0  B|C|D  Graham Chapman          0  0  1  1  1  Graham Chapman
  1    B|D     John Cleese          1  0  1  0  1     John Cleese
  2    A|C   Terry Gilliam   =>     2  1  0  1  0   Terry Gilliam
  3    B|D       Eric Idle          3  0  1  0  1       Eric Idle
  4    B|C     Terry Jones          4  0  1  1  0     Terry Jones
  5  B|C|D   Michael Palin          5  0  1  1  1   Michael Palin
```

```
(2) values = np.random.rand(10)
::array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645])
bins = [0, 0.2, 0.4 0.6, 0.8, 1.]
pd.get_dummies(pd.cut(values, bins))
::  (0.0, 0.2] (0.2, 0.4] (0.4, 0.6], (0.6, 0.8] (0.8, 1.0]
  0          0          0          0           0          1
  1          0          1          0           0          0
  2          1          0          0           0          0
  3          0          1          0           0          0
  4          0          0          1           0          0
  5          0          0          1           0          0
  6          0          0          0           0          1
```

[# Single column's values to construct unique dummy variables]

NOTE: Method chaining is the sequential method call w/ 'self' from left to right, *ie. instance1.method1 + instance2.method2*

" The re module functions fall into 3-categories: *pattern matching*, *substitution* and *splitting*. "

(Refer. Regular Expressions (pg.213), Python for Data Analysis book)

Example.

Cleaning text /w invalid string values involving mistype— unpredictable & inconsistent pattern.

```
df = pd.DataFrame({'Data: ['1987_M_US_1', '1990?_M_UK_1', '1985_F_I T_2']})
```

(Text Handling: Str)

```
import regex as re

# Split long string tag into separate categories:
df['Data'].str.split(pat='_', expand=True)   #string/regexp to split on
df.columns=['Year', 'Sex', 'Country', 'No Child']

# Fixing flaw on each categorical string value:
df['Year'].str.contains('\?')
df['Country'].str.strip()   #remove whitespaces from both ends
df['Year'].str.replace(r'(?P<year>\d{4})\?', lambda m: m.group('year'))
```

(Extract Date from Str)

ID	Description	Value
0	Made payment on 04/11/2019	2000
1	Meeting with clients (07/06/2014)	0
2	Christmas party will take place on 20/12/2018 ...	1400
3	Valentine day is on 14/02/2019 this year	140
4	Easterns was in 21/04/2018 this year	740
5	17/06/2019 was a hot day in Lithuania	20
6	My birthday is on 28/05/2019, not quite long ago	175

```
data = pd.read_excel('data_with_dates.xlsx').set_index('ID')
data['Date'] = None

# Define pattern of the value, dd/mm/yyyy
date_pattern = r'([0-9]{2}\/[0-9]{2}\/[0-9]{4})'
# Identify the loc of date in data value
for row in range(0, len(data)):
    date = re.search(date_pattern, data.iat[row, 0]).group()
data.iat[row, 2] = date

# Re-arrange columns in DataFrame:
data = data[['Data', 'Description', 'Value']]
```

Recommend: Create regex obj w/ re.compile in same expr to many strings to save CPU cycles.

HIERARCHICAL INDEXING (MULTI-INDEX)

- to represent 2-D data within 1-D Series **OR** `ndim` ≥3 data in DataFrame.
(each extra level in multi-index represents an extra dimension of group/feature data)

use— `pd.MultiIndex.from_arrays() /_tuples` or, `pd.MultiIndex.from_product()`

```
(1) pd.MultiIndex.from_array([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
(2) pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
(3) pd.MultiIndex.from_product([['a', 'b'], [1, 2]])   #cartesian product
```

```
dS_ind = [('California', 2000), ('California', 2010), ('New York', 2000),
          ('New York', 2010), ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253656, 18976457, 19378102, 20851820, 25145561]

# Multi-index object constructor:
index = pd.MultiIndex.from_tuples(dS_ind, names=['state', 'year'])
pop = pd.Series(populations, index=index)
```

```
::state       year
  California  2000  33871648
              2010  37253956
  New York    2000  18976457
              2010  19378102
  Texas       2000  20851820
              2010  25145561
```

(# Conventional DataFrame)

year	2000	2010
states		
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

[# `pop.unstack()/stack()` to transform long / wide data for easier comparison or create charts.]

This allows independent index selection, instead of simultaneously matching multiple index
eg. `pop[:, 2010]` —access all data with 2nd index of 2010.

♠ Otherwise, these objects can be passed to index argument when creating Series/DataFrame or `reindex()`, which is convenient for explicit specification individually:

```
ind = pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```
#fancy indexing to coordinate different level in labels.

where, levels: lists of name in each level; labels: lists of integer organization to the named level.

Multi-index constructor in DataFrame:

```
(1) df = pd.DataFrame(np.random.rand(4, 2),
                      index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

(2) data = {('California', 2000): 33871648, ('California', 2010): 37253956,
            ('Texas', 2000): 20851820, ('Texas', 2010): 25145561,
            ('New York', 2000): 18976457, ('New York', 2010): 19378102}
```

In DataFrame, (row & col) are symmetrical (*ie*. multiple levels are applicable for both headings)
All the ufuncs (index preservation & alignment) also work with hierarchical indexes.

```
# Hierarchical indices & columns:
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type'])
# mock some data:
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37
# create DataFrame:
health_data = pd.DataFrame(data, index=index, columns=columns)
```

::subject		Bob		Guido		Sue	
type		HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

(# Useful for records w/ multiple labeled measurements across multiple times of a subject)

(Multi-Index): Indexing & Slicing

1. Series–

Total indexing `pop['California', 2000]` OR `pop.loc['California', 2000]`

Partial indexing `pop['California']` OR `pop.iloc[0]`

```
::33871648
```

```
::year
  2000  33871648
  2010  37253956
```

```
# Conventional Indexing
pop[:, 2000]

# List Indexing
pop[['California', 'Texas']]

# Boolean selection
pop[pop > 22000000]
```

[# sorted MultiIndex is required for partial slicing, see. Rearranging Multi-Indexes]

However, creating a slice within multi-index of lists/tuples often leads to syntax error.

eg. `health_data.loc[(:, 1), (:, 'HR')]` # Pandas does not support coordinated pair, only list of int values

For multiple level selection/slicing, use— `pd.IndexSlice` method:

2. DataFrame–

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

Rearranging Multi-Indexes

- ways to control the rearrangement of data btw hierarchical index and column labels.

```
index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
data = pd.Series(np.random.rand(6), index=index)
# Specify index's names attribute:
data.index.names = ['char', 'int']
::char  int
  a     1    0.003001
        2    0.164974
  c     1    0.741650      (#Not lexicographically sorted data)
        2    0.569264
  b     1    0.001693
        2    0.526226
```

ie. `data['a':'b']` will raise KeyError :'Key length (1) was greater than MultiIndex lexsort depth (0)'

(A) To fulfill partial slices & similar operations that require *sorted* level,

```
Series.sort_index() or, DataFrame.sort_index()
:: ::char  int
  a     1    0.003001
        2    0.164974
  c     1    0.741650      (#lexicographically sorted data)
        2    0.569264
  b     1    0.001693
        2    0.526226
dtype: float64
```

(B) Convert/Revert dataset from a stacked multi-index to simple 2-D representation, *vice-versa*

(*optionally specifying the level of transformation)

```
pop.unstack().stack(level=-1, dropna=True)
::state        year
  California   2000    33871648
               2010    37253956
  New York     2000    18976457
               2010    19378102
  Texas        2000    20851820
               2010    25145561
```

(C) Set/Reset the level(s) of index to a DataFrame, *{default: moving all levels to columns}*

```
pop.reset_index(name='population')
::        state  year  population
  0  California  2000    33871648
  1  California  2010    37253956
  2    New York  2000    18976457
  3    New York  2010    19378102
  4       Texas  2000    20851820
  5       Texas  2010    25145561
```

```
pop_flat.set_index(['state', 'year'])
::                    population
  state       year
  California  2000  33871648
              2010  37253956
  New York    2000  18976457
              2010  19378102
  Texas       2000  20851820
              2010  25145561
```

[# Optionally, specify the name of the data for column label]

*Data Aggregations on Multi-Indexes: `dF.mean(axis=1, level='year')` can be computed.

AGGREGATION & GROUPING

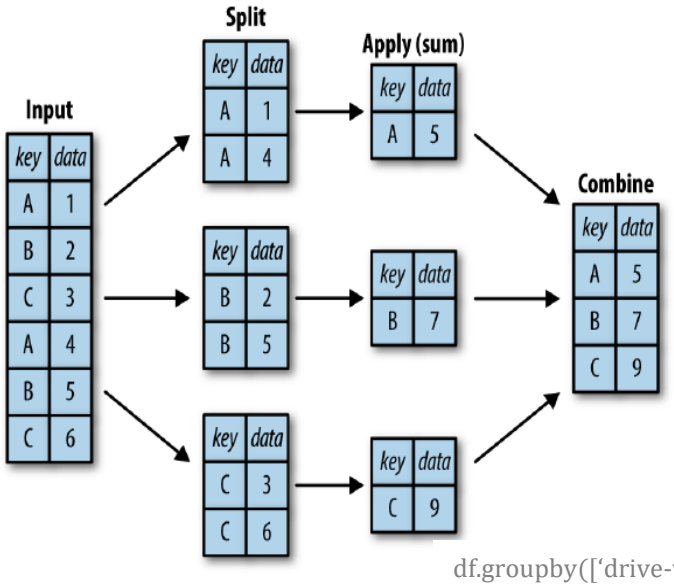
Pandas' efficient statistical methods are defaulted to excl. NA during computation.

Aggregates merely return summarized values in reduced structure:
(Refer. DataWrangling with Pandas cheat sheet for more available functions)

```
df = pd.DataFrame({'A': rng.random(5), 'B': rng.random(5)})
df.mean() / df.mean(axis=1)
::          A         B                                    0  0.088290
  0  0.155995  0.020584          A  0.477888     OR       1  0.513997
  1  0.058084  0.969910     =>   B  0.443420              2  0.849309
  2  0.866176  0.832243          dtype: float64          3  0.406767
  3  0.601115  0.212339                                  4  0.444949
  4  0.708073  0.181825                                  dtype: float64
```

[# To summarize on subsets of data, properties need to individually organzined in a DataFrame, so that the *GroupBy* method can be applied efficiently alongside with the aggregate functions]

GroupBy method— aggregates / function computation in a single pass for the grouped data.



	drive-wheels	body-style	price
0	4wd	convertible	20239.229524
1	4wd	sedan	12647.333333
2	4wd	wagon	9095.750000
3	fwd	convertible	11595.000000
4	fwd	hardtop	8249.000000
5	fwd	hatchback	8396.387755
6	fwd	sedan	9811.800000
7	fwd	wagon	9997.333333
8	rwd	convertible	23949.600000
9	rwd	hardtop	24202.714286
10	rwd	hatchback	14337.777778
11	rwd	sedan	21711.833333

df.groupby(['drive-wheels', 'body-style'], as_index=False).mean( )

(1) **Split**— separate data into groups based on criteria in list of Primary, Secondary, … keys.

(2) **Apply**— assign each group to an identical / different aggregate, function, transformation, w/wo filtering criterion for groups as well as row data.

(3) **Combine**— examine & return a sensibly assembled *DataFrameGroupBy* object.

" A string key may refer to column/index level name, returning `ValueError` for coincidence. "

Simply treat it as a collection of DataFrame (if complete):

```
df.groupby(by='col')
::<pandas.core.groupby.DataFrameGroupBy object at 0x117272160>
```

(*a flexible abstraction of GroupBy's DataFrame structure— merely splitted in n-rows for each group*)

[1] .size( ) – method to get a Series of group sizes for each key

[2] .get_group('key')['select'] – added to retrieve particular group (more on Pandas cheat sheet)

The basic GroupBy functionality incl.:

I. Column Indexing: —Support indexing similar to DataFrame, which returns the specified column(s) of the DataFrameGroupBy object.

```
(1){Reduced to SeriesGroupBy object}
df.groupby('method')['obital_period']
::<pandas.core.groupby.SeriesGroupBy object at 0x117272da0>

(2){Retained DataFrameGroupBy object}
planets.groupby('method')[['obital_period']]   # follow the structure of the indexer
::single column DataFrame
```

II. Iteration over Groups: —Support direct iteration over the groups, returning each group as a Series or DataFrame.

```
for (key, df_group) in df.groupby('key'):
    print('{0:30s} shape={1}'.format(key, df_group.shape))
::Astrometry                     shape=(2, 6)
  Eclipse Timing Variations      shape=(9, 6)
  Imaging                        shape=(38, 6)
  Microlensing                   shape=(23, 6)
  Orbital Brightness Modulation  shape=(3, 6)
  Pulsar Timing                  shape=(5, 6)
  Pulsation Timing Variations    shape=(1, 6)
```

III. Dispatch methods: —Any delegated method that is not part of GroupBy object will be passed to the underlying DS Class. (*ie.* group's df.method()).

```
planets.groupby('method')['year'].describe().unstack()
```

::the describe() method of DataFrame performs a set of aggregations to each individual group, and the results are then combined within GroupBy.

Specifying the split key(s)

Besides the column(s) values, the records can be summarized differently by the grouping keys:

(a) list/nd-array— an equal length of individually defined group for each record.

(b) dict./Series— a mapping of indexes to the group keys.

(c) Python function— a callable, similar to mapping, that will input indexes & ouput the groups.

(a)
```
L = [0, 1, 0, 1, 2, 0]
df.groupby(L).sum()
```
::
	data1	data2
0	7	17
1	4	3
2	4	7

(b)
```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant, 'C': 'consonant'}
df2.groupby(mapping).sum()
```
::
	data1	data2
consonant	12	19
vowel	3	8

(c)
```
df.groupby(str.lower).mean()
```
::
	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

(d)
```
df.groupby([str.lower, mapping]).mean()
```
::
		data1	data2
a	vowel	1.5	4.0
b	consonant	2.5	3.5
c	consonant	3.5	6.0

see. (df example below)

Note: A list of the above things can also be specified as a combination of mapping on multi index analogous to (d).

GroupBy's methods:

- to efficiently implement useful operations on each group instead of aggregation only before they are combined.

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6), 'data2': [5, 0, 3, 3, 7, 9]
```

(A) **Aggregation—** allows the flexibility of one or more aggregates over the specified axis.

```
df.groupby('key').agg(['min', np.median, max]) / {'data1': 'min', 'data2': 'max'}
```

```
::     data1             data2
       min median max     min median max
key
A        0    1.5   3       3    4.0   5
B        1    2.5   4       0    3.5   7
C        2    3.5   5       3    6.0   9
```

*Otherwise, a labeled agg can be specified as – name = pd.NamedAgg(column='data1', aggfunc='min').

(B) **Filtering—** dropping data based on group properties.

```
def filter_func(x):
    return x['data2'].std() > 4
```

(i)

```
::key    data1     data2
   A   2.12132  1.414214
   B   2.12132  4.949747
   C   2.12132  4.242641
```

(i) `df.groupby('key').std()`

(ii) `df.groupby('key').filter(filter_func)`

(ii)

```
   key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

(C) **Transformation—** return some transformed version of the full data.

```
df.groupby('key').transform(lambda x: x - x.mean())
```

```
::    data1  data2
  0   -1.5    1.0
  1   -1.5   -3.5
  2   -1.5   -3.0        [# Center the data by group-wise mean]
  3   -1.5   -1.0
  4   -1.5    3.5
  5   -1.5    3.0
```

(D) **Apply—** apply a custom function to group-wise's DataFrame.

```
def norm_by_data2(x):
    # x is a DataFrame of each group
    x['data1'] /= x['data2'].sum()
    return x
```

```
df.groupby('key').apply(norm_by_data2)
```

```
::   key     data1  data2
  0    A  0.000000      5  (#0/8)
  1    B  0.142857      0  (#1/7)
  2    C  0.166667      3  (#2/12)
  3    A  0.375000      3  (#3/8)
  4    B  0.571429      7  (#4/7)
  5    C  0.416667      9  (#5/12)
```

[# The custom function should take a DataFrame and return Pandas object / scalar value]

- depending on the return variable, combine operation will be tailored to Series/DataFrame

(ie. **Series** ~ index: value pairs, specialization of Python dict.;
DataFrame ~ arrange into structure following the order of the originally defined indexes.)

PIVOT TABLES

- takes column-wise data as bins / classes for grouping along the column & another on row.

To look at mean survival by both 'sex' and 'class' features:

(*1-D* Standard GroupBy routine)

```
titanic.groupby(['sex', 'class'])['survived'].mean().unstack()
```

> # group *by* class & gender, which the *select* survival is *apply* w/ mean aggregate, *combine* the resulting groups, and then *unstack* the hierarchical index to reveal the distinct rows.

(*n-D* Pivot Table)

```
titanic.pivot_table('survived', index='sex', columns='class')
```

> # pivot table is used to provide a quick, clutter-free views at specific parts of the dataset (ie. sort, reorganize, group, count, total, or average summarization of a row to col fields)

```
::class       First    Second     Third
  sex
  female  0.968085  0.921053  0.500000
  male    0.368852  0.157407  0.135447
```

[# The call signature & input parameters for constructing a pivot_table is handy, akin to DataFrame]

Using PivotTable for analysis of—

- Find out how much revenue was generated each year
- Find the average revenue per movie
- Check findings for some possible trends / unsual events

(Here, to seek possible explanation for why the avg_revenue is comparatively low in 2015; if the hypothesis of more movies that earn less than $10M in revenue were released is true)

Release Date - Y	SUM of Box Office Revenue ($)	AVERAGE of Box Office Revenue ($)	COUNT of Box Office Revenue ($)
2012	$18,078,040,000.00	$170,547,547.17	106
2013	$13,672,800,000.00	$160,856,470.59	85
2014	$20,013,420,000.00	$168,180,000.00	119
2015	$13,521,310,000.00	$109,042,822.58	124
2016	$11,921,900,000.00	$161,106,756.76	74
Grand Total	**$77,207,470,000.00**	**$151,983,208.66**	**508**

(Conditional filter- boolean indexing movies w/ revenue less than $10M so we can figure out how many inadequate movies there were in each yr)

Box Office Revenue ($) (Multiple Items)

Release Date - Year	SUM < $10M	AVERAGE < $10M	COUNT < $10M	Percent of total movies
⊞2012	$67,940,000.00	$6,794,000.00	10	9.43%
⊞2013	$26,400,000.00	$3,300,000.00	8	9.41%
⊞2014	$46,920,000.00	$4,265,454.55	11	9.24%
⊞2015	$93,040,000.00	$4,652,000.00	20	16.13%
⊞2016	$41,100,000.00	$5,137,500.00	8	10.81%
Grand Total	**$275,400,000.00**	**$4,831,578.95**	**57**	11.22%

(*Besides, a customized pct of total movies may also be added to further verify the average*)

```
df.pivot_table(values=None, index=None, columns=None, aggfunc='mean',
               fill_value=None, margins=False, dropna=True,
               margins_name='All',observed=False, sort=True)
```

(A) Aggregation.

- string representation: 'sum', 'count', 'mean', 'min', 'max', etc.
- aggregation function: np.sum(), min(), sum(), etc.
- dict. mapping individual columns to any above.

(B) Computing totals along each index & column group.

```
titanic.pivot_table(['survived', 'fare'], index='sex', columns='class',
                    aggfunc={'survived': 'sum', 'fare': 'mean'},
                    margins=True, margins_name='Total')
```

::	fare				survived			
class	First	Second	Third	total	First	Second	Third	total
sex								
female	106.125798	21.970121	16.118810	44.979818	91	70	72	233
male	67.226127	19.741782	12.661633	25.523893	45	17	47	09
total	84.154687	20.662183	13.675550	32.204208	136	87	119	342

[# At the intersection of each row & column, an aggregated summary: sum, count, avg is calculated.]

*When specifying mapping for aggfunc, 1st — 'values' input parameter can be omitted as is determined automatically, unless other options needed to gain access to the keys. (here: margin)
(#otherwise, keyerror 'pclass')

```
# For Multi-level PivotTable:
age = pd.cut(titanic['age'], [0, 18, 80])
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

::fare		[0, 14.454]			(14.454, 512.329]		
class		First	Second	Third	First	Second	Third
sex	age						
female	(0, 18]	NaN	1.000000	0.714286	0.909091	1.000000	0.318182
	(18, 80]	NaN	0.880000	0.444444	0.972973	0.914286	0.391304
male	(0, 18]	NaN	0.000000	0.260870	0.800000	0.818182	0.178571
	(18, 80]	0.0	0.098039	0.125000	0.391304	0.030303	0.192308

Just as in GroupBy, the grouping supports multiple levels (here: 4-dimensional aggregations)

Note: For creating category of bin intervals, (see. binning, Data Cleaning)
(1) **pd.cut()** —bin the value into separate quantitative scale.
(2) **pd.qcut()** —divide data into buckets w/ equal no. of records on specified quantiles.

WORKING WITH TIME-SERIES DATA

- **Timestamp**: refers to particular moments in time (eg. July 4th, 2015, 7:00 a.m.)
- **Period**: refers to uniform length of time over a range (eg. daily, quarterly, or every 12H)
- **Time delta/ interval**: refers to spans of time / difference in reference timestamp(s) (eg. useful for experiment or elapsed time)

*Besides the primary datetime format classes, Pandas has `DateOffset` that is handy for adjusting to calender events/ business date.

I. Native Python: datetime & dateutil modules

```
from datetime import datetime / dateutil.parser import parse

# Datetime object constructor:
datetime(2015, 7, 4)          or,      parse('Jan 31, 1997 10:45 PM')
::datetime.datetime(2015, 7, 4, 0, 0)  ::datetime.datetime(1997, 1, 31, 22, 45)
```

It supports flexible & easy syntax with diverse methods for operations, but weak in large scale processes.

[Refer: Python for Data Analysis (book), pg.318 for more Python calender-related functionality]

II. Vectorized Datetime operation: NumPy's datetime64

```
np.array(['2015-07-04', datetime(2015, 7, 5), dtype=np.datetime64)
::array(['2015-07-04', '2015-07-05'], dtype='datetime64[D]')
```

NumPy array date is represented very compactly, but demands a specific input format by the 'dtype'.

Unit code for `datetime64` & `timedelta64` dtype.

Code	Meaning	Timespan (relative)
Y	Year	± 9.2e18 years
M	Month	± 7.6e17 years
W	Week	± 1.7e17 years
D	Day	± 2.5e16 years
h	Hour	± 1.0e15 years
m	Minute	± 1.7e13 years
s	Second	± 2.9e12 years
ms	Millisecond	± 2.9e9 years
us	Microsecond	± 2.9e6 years
ns	Nanosecond	± 292 years
ps	Picosecond	± 106 days
fs	Femtosecond	± 2.6 hours
as	Attosecond	± 9.2 seconds

*In the datatype attribute, the range of encodable time is limited to *64-bit* precision.

(*ie*. 2^64 of unit time {ns}, the trade-off btw *time resolution* & *accountable value range* in memory)

Note: 'datetime64[ns]' is useful default which encodes range of modern dates w/ optimal fine precision.

strftime & strptime format code

(Refer. Python Datetime for detailed application)

Option	Description	Example
%a	Weekday as locale's abbreviated name	Sun, Mon, …, Sat
%A	Weekday as locale's full name	Sunday, Monday, …, Saturday
%w	Weekday as decimal no. {*zero-based index*}	0, 1, …, 6
%d /-d	Day of the month with zero-padded decimal	01, 02, …, 31 / 1, 2, …, 31
%b	Month as locale's abbreviated name	Jan, Feb, …, Dec
%B	Month as locale's full name	January, February, …, December
%m /-m	Month as zero-padded decimal	01, 02, …, 12 / 1, 2. …, 12
%y	Year w/o century as zero-padded decimal	00, 01, …, 99
%Y	Year with century as decimal no.	0001, 0002, …, 2013, 2014, …, 9999
%H /-H	Hour (24-hr clock) as zero-padded decimal	00, 01, …, 23 / 0, 1, …, 23
%I /-I	Hour (12-hr clock) as zero-padded decimal	01, 02, …, 12 / 0, 1, …, 12
%p	Locale's equivalent of AM / PM	AM, PM
%M /-M	Minutes as zero-padded decimal	00, 01, …, 59 / 0, 1, …, 59
%S /-S	Second as zero-padded decimal	00, 01, …, 59 / 0, 1, …, 59
%f	Microsecond as zero-padded decimal to 6-digits	000000, 000001, …, 999999
%z	UTC offset in form ±HHMM[SS[.ffffff]] (naïve object: empty string)	(empty), +0000, -0400, +1030, +063415, -030712.345216
%Z	Time zone name (naïve object: empty string)	(empty), EDT, JST, WET, UTC, GMT
%j /-j	Day of the year as zero-padded decimal	001, 002, …, 366 / 1, 2, …, 366
%U /-U	Week number of the year as zero-padded decimal (Sunday as 1st day ; All dates in new year preceeding the first Sunday are considered to be in week 0)	00, 01, …, 53 / 0, 1, …, 53 (±1: before / after the Sunday week count)
%W /-W	Week number of the year as zero-padded decimal (Similar with %U, but starts on Monday)	00, 01, …, 53 / 0, 1, …, 53
%c	Locale's appropriate date & time representation	Tue Aug 16 21:30:00 1988
%x	Locale's appropriate date representation	08/16/1988 (en_US); 16.08.1988 (de_DE)
%X	Locale's appropriate time representation	21:30:00 (en_US); 21:30:00 (de_DE)
%%	A literal '%' character	%

Note: (1) %a, %A, %b, %B, %c, %p, %x, %X as locale-specific; (-) UNIX / (#) Window as platform-specific.

(2) ISO doesn't start on 1st day of the year (Gregorian), and need to have full 7-days in the 1st & last wk.

ISO 8601 format: a leap week calendar system representation (2020-07-10T15:00:00.000000000)

Option	Description	Example
%G	ISO 8601 year with century	0001, 0002, …, 2013, 2014, …, 9999
%u	ISO 8601 weekday as decimal no.{*one-based index*}	1, 2, …, 7
%V	ISO 8601 week as decimal no. (Monday as 1st day ; Week 01:: week containing Jan 4)	01, 02, …, 53

Note: A unix timestamp of total sec passed since Jan 1st, 1970 UTC is used to compare & work with time across multiple timezones.

∴ All timezone are defaulted to localize time at system's clock by `tz_localize()`

Pandas' Time-series constructor/ converter:

(1) `pd.Timestamp()`/`pd.Period()`/`pd.Timedelta()`
— to initialize single 'Timestamp', 'Period', or 'Timedelta' datetime's instance.

(2) `pd.to_datetime()`/`pd.Timestamp.to_period()`/`pd.to_timedelta()`
— to convert date of recognizable string format / attr columns into Pandas ts- datatype.

['freq' is inferred from previous type- pd.date_range('2000-01-01', periods=3, freq='M') or, specified]

Timestamps.

Pandas *ts*-class is built upon the flexibility of native Python, and efficient storage & vectorized capabilities of a NumPy datetime64 type.

```
import datetime

# The constructor supports many input formats & structures for an arg:
dt = pd.to_datetime([datetime(2015,7,2), '2015/7/3', '4th July 2015',
                     '07-07-2015', '20150708', 'July 10, 2015'])   string of any ISO8601 format
```

:: scalar – Timestamp ; array-like – DatetimeIndex ; Series/DataFrame/dict-like – Series,datetime64

```
# Extract / Convert datetime component following the string format code:
1. df['date'].dt.strftime('%Y-W%V')   /   pd.to_datetime('15/07/04 09:34:00-0700',
::'2016-W31'   # str week no.                              format=r"%y/%m/%d %H:%M:%S%z")
2. df['date'].dt.isocalendar().week     ::Timestamp('2015-07-04 09:37:00-0700')
::31   # int week no.          For date count- pd.to_datetime([1, 2,3], unit='D', origin=pd.Timestamp('1998-01-01'))
```

*Also, date part has incorporated - attrs.: dt.year|hour|... / methods: dt.month_name( ).str.slice(stop=3)

```
# Vectorized datetime operation:   (date1 – date2) / np.timedelta64(1, 'm') or, .dt.total_seconds( )
dates = date + pd.to_timedelta(np.arange(4), 'D')
::DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07'],
                dtype='datetime64[ns]', freq=None)
```

Periods.

As working with date will often require dividing into smaller parts, separating timestamps into periods can be handy in grouping & ordering of the datetime data for analysis.

```
df['quarter'] = df['date'].dt.to_period('Q')
:: 2016-08-05   2016Q3
   2016-12-05   2016Q4     (eg. To inform about likelihood / risk of an event in uniform periods)
   2017-01-05   2017Q1
   ...             ...
```

Period Arithmetic —*objects must have equal period freq*

```
p = pd.Period(2012, freq='A-DEC')     pd.Period('2014', freq='A-DEC') - p
::Period('2012', 'A-DEC')             ::<7 * YearEnds: month=12>

# To retrieve timestamp at 4PM on 2nd-last-business day of the quarter:
period = pd.Period('2012Q4', freq='Q-JAN')
p4pm = (period.asfreq('B', how='end') - 1).asfreq('H', how='start') + 16
::Period('2012-01-30 16:00', 'H')
p4pm.to_timestamp()
::Timestamp('2012-01-30 16:00:00')
```

Note: To create regular sequence of the datetime datatype, pd.{date|period|timedelta}_range may be used.

Period Frequency Conversion

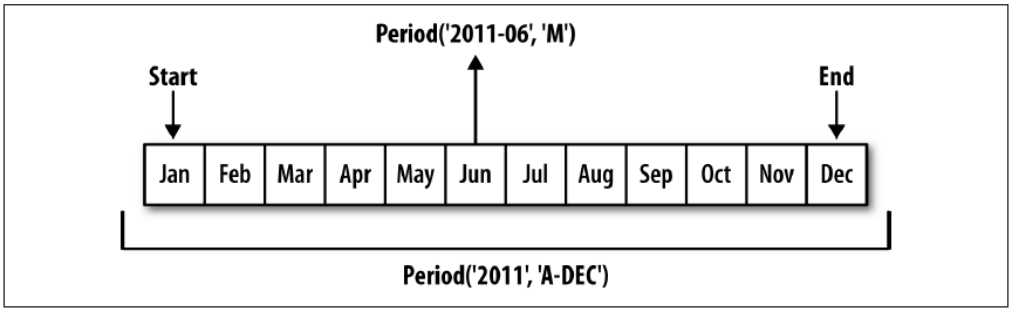
When converting from high to low frequency, Pandas determines the superperiod depending on where the subperiod belongs,

(***eg.*** in A-JUN frequency, the Aug-2007 is actually part of the 2008 period by default 'end')

```
p = pd.Period('Aug-2007', 'M')
p.asfreq('A-JUN')
::Period('2008', 'A-JUN')
```

equiv. to,

```
p = pd.Period('Aug-2007', 'A-JUN')
::Period('2008', 'A-JUN')
```




```
# Changing the freq of PeriodIndex:
rng = pd.period_range('2006', '2009',
                      freq='A-DEC')
pe = pd.Series(np.arange(len(rng)),
               index=rng)
# Default subset taken at the end
pe.asfreq('M', how='end')
  ::2006-12  0
    2007-12  1
    2008-12  2
    2009-12  3
```

```
# Changing the freq of DatetimeIndex:
rng = pd.date_range('01-01-2000',
                    periods=4, freq='D')
ts = pd.Series(np.arange(len(rng)),
               index=rng)
ts.asfreq('12H')
::2000-01-01 00:00:00     0.0
  2000-01-01 12:00:00     NaN
  2000-01-02 00:00:00     1.0
  2000-01-02 12:00:00     NaN
  2000-01-03 00:00:00     2.0
  2000-01-03 12:00:00     NaN
  2000-01-04 00:00:00     3.0
```

[# Unlike Timestamp—a continuous sequence, Period merely alters to a super/subset of the time unit]

Pandas supports 12 possible quarterly frequencies {Q-JAN:Q-DEC} depending on fiscal year end

(***ie.*** Quarterly frequency ending in Jan, 2012Q4 runs from Nov thru Jan)

```
# Initiate a single period instance
quarterpe = pd.Period('2012Q4', freq='Q-JAN')
```

@check by converting to daily frequency

```
quarterpe.asfreq('D', how='start')
::Period('2011-11-01', 'D')
```

;

```
quarterpe.asfreq('D', how='end')
::Period('2012-01-31', 'D')
```

Note: To create regular sequence of the datetime datatype, pd.{date|period|timedelta}_range may be used.

a. Time-series Index Object

To create index object from arrays:

- Timestamps: `'DatetimeIndex'` —Pandas index structure for specific moment.
- Periods: `'PeriodIndex'` —Pandas index structure for non-overlapping timespans.
- Timedeltas: `'TimedeltaIndex'` —Pandas index structure for elapsed time.

```
# Timestamp index object constructor:
schedule_date = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                                  '2015-07-04', '2015-08-04'])
data = pd.Series([0, 1, 2, 3], index=schedule_date)
:: 2014-07-04    0
   2014-08-04    1
   2015-07-04    2
   2015-08-04    3
   dtype: int64
```

```
# useful for date indexing / slicing:
- df.loc['2015'] / df.loc['04/07/2014'] /df.asof(date)
  :: the later always retrieves last row w/o any NaN, and avoid keyerror.
- df ['2014-09':'2015-07'] / df.truncate(after='2015-07')
```

!Important: (Refer. Time-series Basics, Python for Data Analysis)

(1) Because most time-series index is ordered chronologically, interpolating/ extrapolating query of time-index (eg. :'2015-06') is possible.

(2) For duplicate timestamps indexing, multiple values will be selected.
» use `'ts.groupby(level=0)'` to aggregate the data having non-unique timestamp.

At fixed frequency datasets: timespan information spread across multiple columns.

```
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi    cpi
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9  28.98
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7  29.15
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4  29.35
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3  29.37
4  1959.0      1.0  2847.699    1770.5  311.722   462.199   1955.5  29.54
```

```
# Period index object constructor:
index = pd.PeriodIndex(year=data.year, quarter=data.quarter, freq='Q-DEC')
::PeriodIndex(['1959Q1', '1959Q2', '1959Q3' '1959Q4', '1960Q1', '1960Q2',
               ...
               '2007Q1', '2007Q2', '2007Q3' '2007Q4', '2008Q1', '2008Q2'],
               dtype='period[Q-DEC]' length=203)
```

Evidently, the unit specification is in accordance with frequency parameter as:

(A) pd.PeriodIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07'], freq='D')
(B) pd.PeriodIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07'], dtype='period[D]')

A 'TimedeltaIndex' is the temporal difference btw 2-timestamp values:

```
dates - dates[0]OR
::TimedeltaIndex(['0days', '1days', '2days'], dtype='timedelta64[ns]', freq=None)
```

```
pd.TimedeltaIndex(['1 days, 00:00:05', np.timedelta64(2, 'D'),
                   datetime.timedelta(days=2, seconds=2)])
::TimedeltaIndex(['1 days 00:00:05', '2 days 00:00:00', '2 days 00:00:02'],
                 dtype='timedelta64[ns]', freq=None)
```

Any ts index can be converted: 'DatetimeIndex' ⇌ 'PeriodIndex' ⇌ 'TimedeltaIndex' by— `pd.DatetimeIndex.to_period` *or* `pd.PeriodIndex.to_timestamp` *or* temporal difference.

```
rng = pd.date_range(start='1/29/2000', periods=6, freq='D')
ts = pd.Series(np.random.randn(6), index=rng)     ts.to_period('M')
::2000-01-29   0.244175                           2000-01   0.244175
  2000-01-30   0.423331                           2000-01   0.423331
  2000-01-31  -0.654040          =>               2000-01  -0.654040
  2000-02-01   2.089154                           2000-02   2.089154
  2000-02-02  -0.060220                           2000-02  -0.060220
  2000-02-03  -0.167933                           2000-02  -0.167933
```

However, since periods refer to non-overlapping timespans, a timestamp can only belong to a single period for a given frequency.

The 'start' & 'end' parameters define strict boundaries for the generated date ***(zero-based index)***.
» To properly account for the business date / cycle containing the working days in a month, a suitable frequency code should be specified.

Basic frequency unit / Offset aliases & PeriodDtype (Refer. Python for Data Analysis, pg.329)

Code	Description	Code	Description
D	Calender day		****excl. in PeriodDtype**
W	Weekly	**BM**	Business month end
M	Month end	**BQ**	Business quarter end
Q	Quarter end	**BA**	Business year end
A	Year end	**BH**	Business hours
H	Hours	# Default start/end at DEC	
T	Minutes	**MS**	Month start
S	Seconds	**BMS**	Business month start
L	Milliseconds	**QS**	Quarter start
U	Microseconds	**BQS**	Business quarter start
N	Nanoseconds	**AS**	Year start
B	Business day	**BAS**	Business year start

(1) Futhermore, the specific month can be defined on top of a basic frequency unit:

- ✓ `Q-JAN, BQ-FEB, QS-MAR, BQS-DEC`, etc.
- ✓ `A-JAN, BA-FEB, AS-MAR, BAS-DEC`, etc.

(2) To modify the split-point of a weekly frequency:

- ✓ `W-SUN` —same as 'W', `W-MON, W-TUE, W-WED` … {weekly on given day}

(3) To generate a specific day of the week on each month:

- ✓ `WOM-1MON, WOM-2MON, WOM-3FRI` … {WeekOfMonth: second Monday of each month}

For customization of dateoffset, employs `pd.Dateoffset()` *or* `pd.offsets.basefreq()`

```
freq = pd.offsets.CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)
```

b. Fixed frequency DatetimeIndex)

- create regular ranges in time for timestamps, periods, and timedeltas, ***relatedly*** — *np.arange*.

At least 3 parameters must be specified from— {'start', 'end', 'freq', 'periods'}.

a) `pd.date_range`(start, end, periods, freq='D', tz, normalize=False, name, inclusive='both')
- eg. periods=8; freq='6H'; tz='Asia/Hongkong'; inclusive={'both'/'neither'/'left'/'right'}

b) `pd.period_range`(start, end, periods, freq='D', name)
- eg. periods=5, freq=pd.offsets.MonthEnd(3) —intervals offset to correct month end

c) `pd.timedelta_range`(start, end, periods, freq='D', name, closed=None)
- eg. freq='2H30T'; closed={'left'/'right'/None} —if incl. of boundary value, None: 'both'

Time Zone Handling

- Time zones allow different parts of the world to possess an identical clock hour, which offset from global standard (UTC±00:00) to regulate clocks— *so, the sun is overhead at 12 pm everywhere.*

  (*eg*. UTC does not adjust for DST, but a localized- New York is at 4Hr during DST & 5Hr otherwise)

* Daylight saving time (DST) is the practice of adjusting clocks for Nothern & Southern regions so that darkness falls at a later clock time during summer months, and *vice-versa*.

In native Python, time zone information comes from 3rd-party 'pytz' library which exposes the *Olson database*, a thorough compilation of historical data compliance w/ changes at times.

```
import pytz
# Timezone object constructor:
tz = pytz.timezone('America/New_York')
::<DstTZInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Pandas methods accept both timezone as name of location identifier & object (as shown).

ie.
```
rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')
ts = pd.Series(np.random.randn(len(rng)), index=rng)
::2012-03-09 09:30:00  -0.202469
  2012-03-10 09:30:00   0.050718
  2012-03-11 09:30:00   0.639869
  2012-03-12 09:30:00   0.597594
  2012-03-13 09:30:00  -0.797246
  2012-03-14 09:30:00   0.472879
  Freq: D, dtype: float64
```

! By default, time-series in Pandas is timezone naïve – print(ts.index.tz) ::None

```
# Setting naïve to global / localized standard time zone:
ts_utc = ts.tz_localize('UTC') eg. tz='CET', 'EET', or 'US/Eastern'
print(ts_utc.index)
::DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00'],
  dtype='datetime64[ns, UTC]', freq='D')

# Changing the localized time zone by the name of location:
ts_utc.tz_convert('America/New_York')
```

Note: UTC time is the neutral location time of reference while a local time is the offset from UTC based on time zone .

Localizing naïve timestamps also checks for ambiguous/non-existent times on DST transitions

```
::2012-03-09 04:30:00-05:00  -0.202469
  2012-03-10 04:30:00-05:00   0.050718
  2012-03-11 05:30:00-04:00   0.639869
  2012-03-12 05:30:00-04:00   0.597594
  2012-03-13 05:30:00-04:00  -0.797246
  2012-03-14 05:30:00-04:00   0.472879
  Freq: D, dtype: float64
```

When performing time arithmetic, Pandas respects DST transitions where possible.

```
from pandas.tseries.offsets import Hour

# 30-minutes before transitioning to DST
stamp = pd.Timestamp('2012-03-12 01:30', tz='US'/Eastern')
::Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')
stamp + Hour()
::Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')
```

```
# 90 minutes before transitioning out of DST
stamp = pd.Timestamp('2012-11-04 00:30', tz='US'/Eastern')
::Timestamp('2012-11-12 00:30:00-0400', tz='US/Eastern')
stamp + pd.Timedelta(2, 'H')
::Timestamp('2012-11-12 01:30:00-0500', tz='US/Eastern')
```

[# As the result is insufficient for hourly adjustment in (1), the timestamp remains in previous DST]

Note: for arithmetic operations btw 2-different time-zones, the result will be in UTC.

Summary of Pandas' datetime classes:

Concept	Scalar Class	Array Class	pandas Data Type	Primary Creation Method
Date times	`Timestamp`	`DatetimeIndex`	`datetime64[ns]` or `datetime64[ns, tz]`	`to_datetime` or `date_range`
Time deltas	`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	`to_timedelta` or `timedelta_range`
Time spans	`Period`	`PeriodIndex`	`period[freq]`	`Period` or `period_range`
Date offsets	`DateOffset`	`None`	`None`	`DateOffset`

* `pd.offsets` is alias for `pd.tseries.offsets`; *ie.* both denotes the same package.

Time Resampling & Frequency Conversion.

The process of converting a time-series from one frequency to another: **Up/Down- sampling**.

```
DataFrame.resample(freq, closed=None, label=None, origin='start_day',
                   on=None, level=None, offset=None, group_keys=False)
```

Primary parameters–

`freq` - String or DateOffset of desired resample frequency; *eg*. 'ME', '5min', Second(15).
`offset` - An offset timedelta or string added to the origin.
`closed` - In down-sampling, which end of interval is incl. in the calculation; *eg*. 'left', 'right'.
`label` - In down-sampling, which side to adopt as the aggregated result; *eg*. 'left', 'right'.
`level` - For a MultiIndex, the str or int level to use for resampling.
`group_keys` – Whether to incl. the group keys in the result index when using `.apply()`.

* PeriodIndex data is required to be converted to DatetimeIndex before resampling.

Grouping timeseries data based on a specific time intervalization/period similar to `df.groupby`.

```
ts_data.resample('5T')
:: <pandas.core.resample.DatetimeIndexResampler object at 0x00000235A6C0AC70>
```

(1) In down-sampling, the agg. functions/ '`.apply`'/'`.transform`' ought to be used for summarizing data with the criteria:

- which side of each interval to closed
- how to label each aggregated bin, start/end of interval

```
rng = pd.date_range('2000-01-01', periods=100, freq='D')
ts = pd.Series(np.random.randn(len(rng)), index=rng)
ts.resample('M').mean()
```

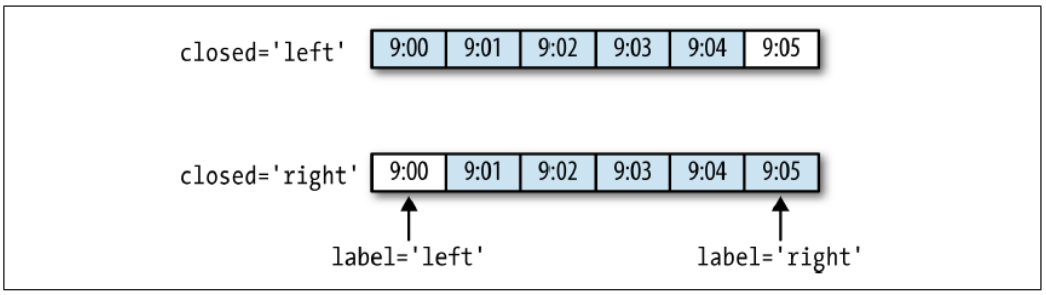
```
::2000-01-01   0.631634              2000-01-31  -0.165893
  2000-01-02  -1.594313              2000-02-29   0.078606
  2000-01-03  -1.519937      =>      2000-03-31   0.223811
  2000-01-04   1.108752              2000-04-30  -0.063643
                 …                   Freq: M, dtype: float64
  2000-04-08  -2.253294
  2000-04-09  -1.772919
  Freq: D, Length: 100, dtype:float64
```

* For results inclusive of the first timestamp value, specify— {**closed='right'**}, {**label='right'**}

(*Alternatively,*) to retrieve the values at fixed interval w/o any aggregation—

```
ts.asfreq('5')   # separation at no. of data
```

Note: *ts*-data should be cleaned of any missing value for the method to work effectively.



Open-High-Low-Close (OHLC) resampling

A popular way of timeseries aggregation in finance, is to compute 4-values for each bucket:

```
rng = pd.date_range('2000-01-01', periods=12, freq='T')
ts_data = pd.Series(np.arange(12), index=rng)
ts.resample('5min').ohlc()
::                     open  high  low  close
  2000-01-01 00:00:00     0     4    0      4
  2000-01-01 00:05:00     5     9    5      9
  2000-01-01 00:10:00    10    11   10     11
```

(2) In up-sampling, as missing value is involved, classical '.fillna( )', '.interpolate( )', or other filling methods may be incorporated to fill the gap in the data.

```
rng = pd.date_range('2000-01-01', periods=2, freq='W-WED')
frame = pd.DataFrame(np.random.standard_normal((2, 2)), index=rng)
frame.resample('D').fillna(method='ffill', limit=2)
::                    0         1                          0         1
  2000-01-05  -0.631634  0.927238      2000-01-05  -0.165893  0.927238
  2000-01-12  -0.493841 -0.155434      2000-01-06  -0.896431  0.927238
                                  =>  2000-01-07  -0.896431  0.927238
                                      2000-01-08        NaN       NaN
[# New date index need not overlap the old one]       2000-01-09        NaN       NaN
                                      2000-01-10        NaN       NaN
                                      2000-01-11        NaN       NaN
                                      2000-01-12   0.493841 -0.155434
```

Since periods refer to timespans, the rule about up/down-sampling are more rigid—

- ✓ In downsampling, the 'freq' must be a *subperiod* of the source frequency.
- ✓ In upsampling, the 'freq' must be a *superperiod* of the source frequency.

(*eg.* Target timespans: Q-MAR only line up with A-MAR, A-JUN, A-SEP, and A-DEC
:: *for annual periods starting* DEC: A-DEC [2000, 2001], *the quarter freq will be* [2000Q4, ..., 2002Q3])

If a DataFrame contains multiple time-series, marked by additional group key column:

```
times = pd.date_range('2017-05-20 00:00', freq='1min', periods=15)
df = pd.DataFrame({"time": times.repeat(3),
                   "key": np.tile(['a', 'b', 'c'], 15),
                   "value": np.arange(15 * 3.)})
```

```
::                   time  key value
0  2017-05-20 00:00:00    a   0.0
1  2017-05-20 00:00:00    b   1.0
2  2017-05-20 00:00:00    c   2.0
3  2017-05-20 00:01:00    a   3.0
4  2017-05-20 00:01:00    b   4.0
5  2017-05-20 00:01:00    c   5.0
   ...                        ...
42 2017-05-20 00:14:00    a  42.0
43 2017-05-20 00:14:00    b  43.0
44 2017-05-20 00:14:00    c  44.0
```

To do resampling by multiple columns' keys— 'key' & 'time_key', and aggregate:

```
# Manipulate datetime's attr. grouping to align with groupby
time_key = pd.Grouper(freq='5min')
resampled = (df.set_index('time')
             .groupby(['key', time_key])
             .sum())
```

```
::                          value
  key time
  a   2017-05-20 00:00:00   30.0
      2017-05-20 00:05:00  105.0
      2017-05-20 00:10:00  180.0
  b   2017-05-20 00:00:00   35.0
      2017-05-20 00:05:00  110.0
      2017-05-20 00:10:00  185.0
  c   2017-05-20 00:00:00   40.0
      2017-05-20 00:05:00  115.0
      2017-05-20 00:10:00  190.0
```

[# time must be the index when using 'pd.Grouper']

[# Ideally, the operation is similar to how 'sort_values' method works, but referring to datetime's attr.]

Time Shifting (Leading / Lagging).

Moving data backward or forward through time, but leaving the index unmodified.

```
ts = pd.Series(np.random.standard_normal(4),
               index=pd.date_range("2000-01-01", periods=4, freq="M"))

ts.shift(1) OR  ts.shift(1, freq='M')
::2000-01-31  -0.066748      2000-01-31        NaN   /   2000-02-29  -0.066748
  2000-02-29   0.838639      2000-02-29  -0.066748       2000-03-31   0.838639
  2000-03-31  -0.117388  =>  2000-03-31   0.838639       2000-04-30  -0.117388
  2000-04-30  -0.517795      2000-04-30  -0.117388       2000-05-31   0.517795
  Freq: M, dtype: float64    Freq: M, dtype: float64    Freq: M, dtype: float64
```

[# A common use is computing consective percent changes in a time-series— `ts / ts.shift(1) - 1`]

> *Alternatively*, statistics can be computed at:
> - `pd.DataFrame.diff(intervals); Series.pct_change()`
>
> 

Relatedly, Pandas custom dateoffsets can be used with index or, `datetime` / `Timestamp` object.
(see. Periods, basic frequency/offsets aliases)

```
now = pd.to_datetime('2011-11-17')
now + 3 * pd.offsets.Day()
::Timestamp('2011-11-20 00:00:00')
```

Note: Because naïve shifts leave the index unmodified, some data is discarded.
[*# Thus, if frequency is known, it can be added to advance the timestamps than simply data interval w/o changing the underlying frequency of data*.]

Rolling Time Window.

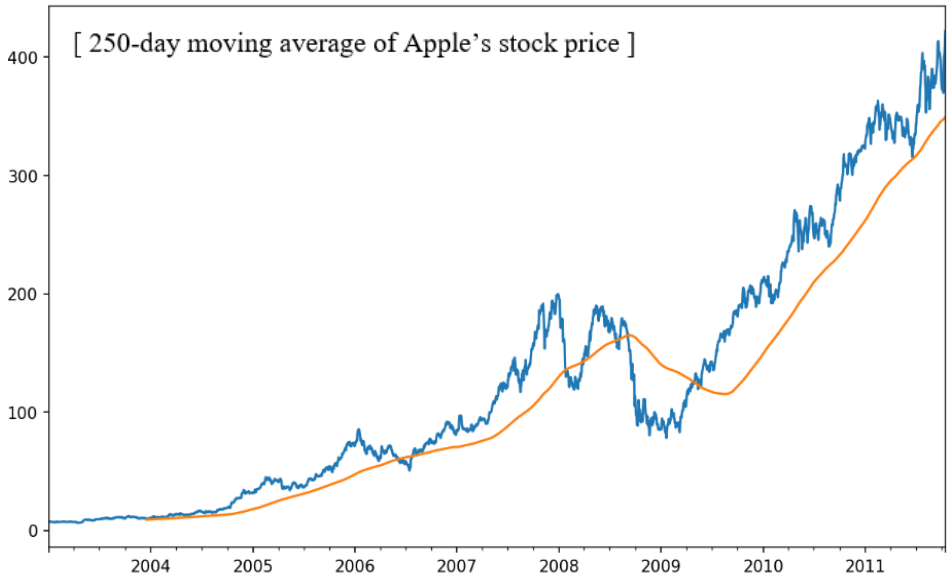
—Smoothing noisy / gappy time-series data using a statistical or differentiating function and over a sliding window w±wo exponentially decaying weight (**win_type=None**) to find patterns .

(A) Instead of aggregating data monthly / yearly at fixed partitions, the 'rolling' method computes results serially for each of the groups with a moving window (timeframe).

```
df = pd.read_csv("examples/stock_px.csv", parse_dates=True, index_col=0)
close_px = df[['AAPL', 'MSFT', 'XOM']]
close_px = close_px.resample('B').ffill()   # dateoffsets can be set internally as 'window'

close_px['AAPL'].plot() ;  close_px['AAPL'].rolling(250)
::<AxesSubplot:>          ::Rolling[window=250,center=False,axis=0,method=single]

close_px['AAPL'].rolling(250).mean().plot()
```



Note: By default, the method requires all values in the window to be non-NA at: **min_periods=None**.

[# For gappy data, set the minimum records for computation, to avoid returning all NA values]

Calling the method on a DataFrame applies the transformation to each column.

```
close_px.rolling('20D').mean()
```

::

	AAPL	MSFT	XOM
2003-01-02	7.400000	21.110000	29.220000
2003-01-03	7.425000	21.125000	29.230000
2003-01-06	7.433333	21.256667	29.473333
2003-01-07	7.432500	21.425000	29.342500
2003-01-08	7.402000	21.402000	29.240000
...	...	...	...
2003-10-10	389.351429	25.602143	72.527857
2003-10-11	388.505000	25.674286	72.835000
2003-10-12	388.531429	25.810000	73.400714
2003-10-13	388.826429	25.961429	73.905000
2003-10-14	391.0.3800	26.048667	74.185333

[2292 rows x 3 columns]

Options:

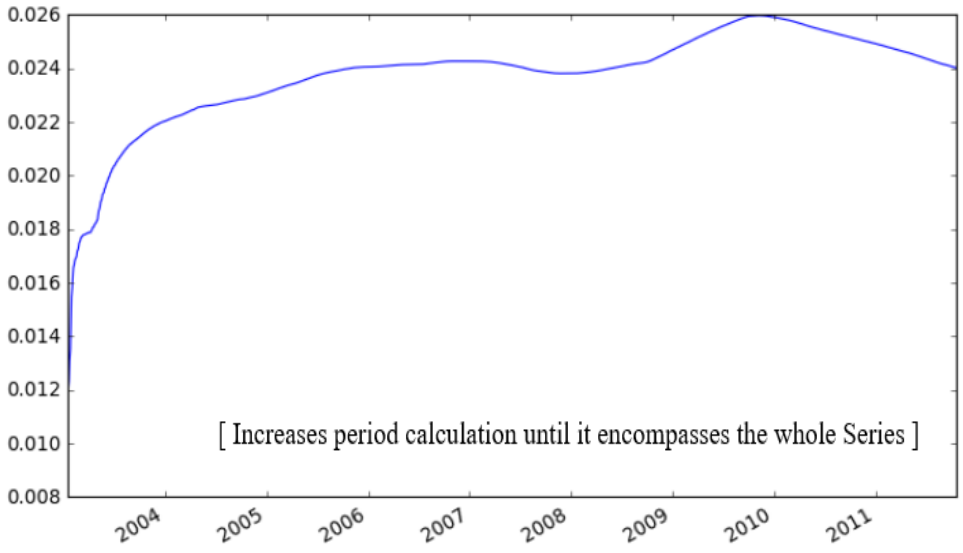
(1) **center**- if the right edge or center wn idx returned as smoothed idx.

(2) **closed**- {*'right'/'left'/'both'/'neither'*} if the 1st / last pt. in the window is excluded from calculations.

NOTE: The result is unaffected by window size/period computation due to default 'min_period=1' for dateoffset.

(B) If an expanding window is desired, 'expanding' method can be used.

```
plt.figure()
std250 = close_px['AAPL'].pct_change().rolling(250, min_periods=10).std()
std250.expanding().mean().plot()
```



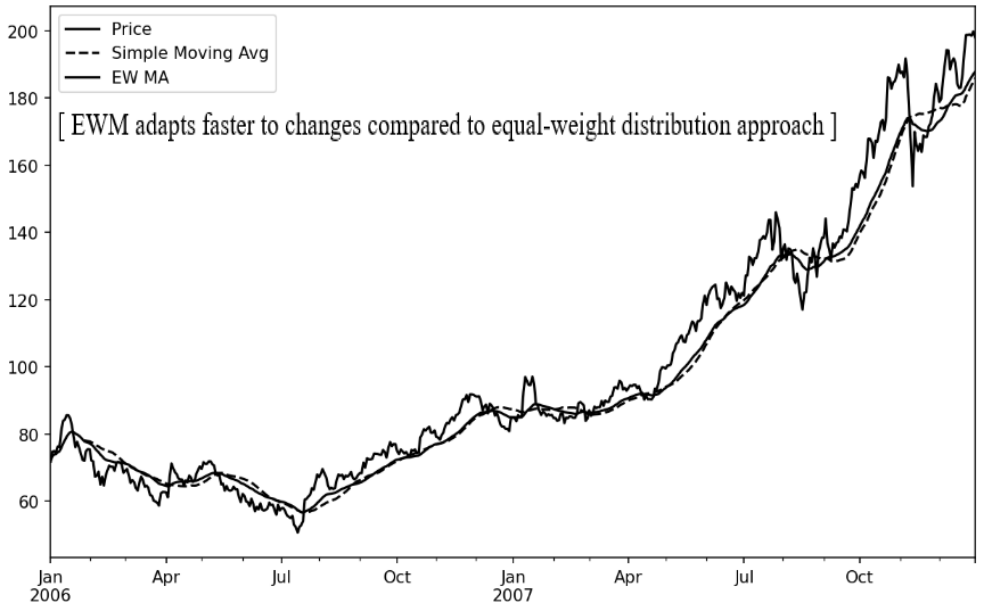
(C) Expanding at constant decay factor, 'ewm'— emphasize more weight to more recent data.

```
aapl_px = close_px['AAPL']['2006':'2007']

# Simple Moving Average:
ma30 = aapl_px.rolling(30, min_periods=20).mean()

# Exponentially Weighted Moving Average:
ewma30 = aapl_px.ewm(span=30).mean()

aapl_px.plot(style='k-', label='Price')
ma30.plot(style='k--', label='Simple Moving Avg')
ewma30.plot(style='k-', label='EW MA')
plt.legend()
```



Due to the emphasis of index object in Pandas time-series generator (eg. pd.date_range), statistical functions, like correlation & covariance, need to operate on 2-time series DataFrame.

```
spx_px = close_px_all["SPX"]
spx_rets = spx_px.pct_change()
returns = close_px.pct_change()

# Binary Moving Window Functions
corr = returns['AAPL'].rolling(125, min_periods=100).corr(spx_rets)
corr.plot()
```

*After '`rolling`', the '`corr`' aggregate function can then compute the correlation synchronous rolling with variable: spx_rets.

Rather, if application of custom function is desired, the '`apply`' method on '`rolling`'/related-methods provides a convenient way to constitute for **User-Defined Moving Window Functions**.

```
from scipy.stats import percentileofscore

def score_at_2percent(x):
    return percentileofscore(x, 0.02)

returns['AAPL'].rolling(250).apply(score_at_2percent).plot()
```

NOTE: Similarly, the method can be applied on multiple columns in a DataFrame (*eg. rolling correlation of many –to- one*).

HIGH-PERFORMANCE PANDAS: EVAL( ) & QUERY( )

- to effectively filter out rows than the 2-steps execution: *Boolean mask* & *indexing*.

MODELING LIBRARIES IN PYTHON

The common workflow for model development as—
Pandas data loading & preparation before switching to modeling library to build the model.
ie. *implementation of analytical techniques w/ model fitting & scoring*

The point of contact btw Pandas & other analysis libs. is usually by NumPy array {Positional},

```
data = pd.DataFrame({'x0': [1, 2, 3, 4, 5], 'x1': [0.01, -0.01, 0.25, -4.1, 0]})
```

```
data.values OR  to_numpy()
::array([[1.,  0.01],
         [2., -0.01],
         [3.,  0.25],
         [4., -4.1 ],
         [5.,  0.  ]])
```

```
df3 = data.copy()
df3['strings'] = ['a', 'b', 'c', 'd', 'e']
df.values
::array([[1, 0.01, 'a'],
         [2, -0.01, 'b'],
         [3, 0.25, 'c'],
         [4, -4.1, 'd'],
         [5, -0.0, 'e']], dtype=object)
```

[# NumPy array of homogeneous data is preferred in numerical function operations as it is easier to operate at minimal attr. constraints.]

Depending on the accepted structure of a function argument, some libs have native support for Pandas and directed this operation internally (eg. Statsmodels / Scikit-learn / Seaborn)

i. Creating Model Descriptions w/ Patsy

- one of the alternative to simple & less error-prone model specification.

Python supports model description w/ R-style formula in 'smf.' api using Patsy as- *string* "mathematical function" syntax, {Wilkinson Notation}

The equivalent internal operation as followed—

```
import patsy
data = pd.DataFrame({'x1': np.arange(1, 6),          # data: DataFrame/Dict. of arrays
                     'x2': [0.01, -0.01, 0.25, -4.1, 0.],
                     'y': [-1.5, 0., 3.6, 1.3, -2.]})
y, X = patsy.dmatrices('y ~ x1 + x2', data)
```

```
::DesignMatrix with shape(5, 1)  ;  DesignMatrix with shape(5, 3)
     y                                Intercept  x0     x1
  -1.5                                1          1   0.01
   0.0                                1          2  -0.01
   3.6                                1          3   0.25
   1.3                                1          4  -4.10
  -2.0                                1          5   0.00
  Terms:                           Terms: 'Intercept' (column 0)
   'y' (column 0)                         'x1' (column 1)
                                          'x2' (column 2)
```

[# To suppress intercept term: add '+ 0' to the formula]

Note: The dmatrix & dmatrices functions are identical, *except* it returns two matrices instead of one.

Patsy 'DesignMatrix' instances— NumPy *ndarrays* w/ metadata, are created based on the eqs, which can be passed as array structure into algorithms like np.linalg.lstsq().

```
coef, resid, _, _ = np.linalg.lstsq(X, y)
coef                    ;     metadata = X.design_info.column_names
::array([[ 0.3129],           ::['Intercept', 'x1', 'x2']
         [-0.0791],
         [-0.2655]])
coef = pd.Series(coef.squeeze(), index=metadata)
::Intercept    0.312910
  x1          -0.079106
  x2          -0.265464
  dtype: float64
```

The '.design_info' attribute contains: *column_name_indexes, terms, term_names, factor_infos, …* which is the metadata object used by Patsy to pass to statistical libraries for further processing.

(Refer: Patsy Documentation, for more transformations & model settings)

NOTE:

A. In multi-variate eqs, the categorical encoded data can be interpreted as a grouped feature at:

```
y, X = patsy.dmatrices('v2 ~ C(keys2)', data)
```

B. When multiple variables are involved in a models, an interaction terms may be assumed at:

```
y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)
```

Variable Transformation

As part of a modeling process, different model settings may be applied to accommodate distributional assumptions. (*eg.* statistical transformation & feature scaling)

```
(1) y, X = patsy.dmatrices('y ~ standardize(x1) + center(x1)', data) #Patsy built-in fn
 # Apply identity function to isolate the term for independent computation:
(2) y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

 # Pass the stateful transformation to new out-of-sample dataset:
 new_X = patsy.build_design_matrices([X.design_info], new_data)
```

Relatedly, any Python function can be mixed into Patsy formula.

```
y, X = patsy.dmatrices('y ~ x1 + np.log(np.abs(x2) + 1)', data)
:: DesignMatrix with shape(5, 3)
     Intercept  x1   np.log(np.abs(x2) + 1)
     1           1                 0.00995
     1           2                 0.00995
     1           3                 0.22314
     1           4                 1.62924
     1           5                 0.00000
     Terms: 'Intercept' (column 0)
            'x1' (column 1)
            'np.log(np.abs(x2) + 1)' (column 2)
```

NOTE: Stateful transformation should be applied w/ caution to ensure uniform distribution statistics in different datasets.

ii. Statsmodels API— {Classical frequentist statistical methods}

- a Python lib. for fitting statistical models, performing statistical tests, and data exploration.

Some commonly used modeling interfaces in statsmodels include:

- Linear models, generalized linear models, robust linear models
- Linear mixed effects models
- Analysis of variance (ANOVA) methods
- Time-series processes & state space models
- Generalized method of moments

Linear models in statsmodels have 2-different interfaces, *ie.* array-based & formula-based which takes Positional/Keyworded structure as conservative or, constructive terms for an eqs.

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

```
(1) model = sm.OLS(y, X)    # use sm.add_constant(X) / df.assign(const=1) to incl. bias col
(2) model = smf.ols('y ~ x1 + x2 + x3', data=data)    # incl. -1 to suppress default
```

(*ie*. depending on NumPy ndarray / Pandas object, the results will be returned as followed— arbitrarily ordered values or indexed dimension table)

NOTE: In formula-based interface, an intercept term will be fitted to Patsy's design matrix by default.

iii. Scikit-learn API— {Bayesian statistical & ML methods}

- contains standard supervised & unsupervised methods, w/ tools for model selection and evaluation, data transformation, data loading, and model persistence.

```
model = linear_model.LogisticRegression()
model.fit(X_train, y_train)
y_predict = model.predict(X_test)

# Cross-validation (CV) by hand:
scores = cross_val_score(model, X_train, y_train, cv=4)
```

@Where, this library is built on NumPy, SciPy and Matplotlib.

Note: The different applications of Scikit-learn API is beyond the scope of this topic and would be best referenced along with statistics. (Refer: Practical Statistics for Data Science)