Data analysis for Machine Learning

Liu Da
da.liu@ericsson.com
BCSS Learning Service

DISCLAIMER

This is a Learning Product which contains simplifications. Therefore, it must not be considered as a specification of the system.

The content is subject to revision without notice due to ongoing progress in methodology, design and manufacturing. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this Learning Product as some technical reference.

The Learning Product documentation is not intended to replace the technical documentation that was delivered with your system. Always refer to that technical documentation during operation and maintenance.

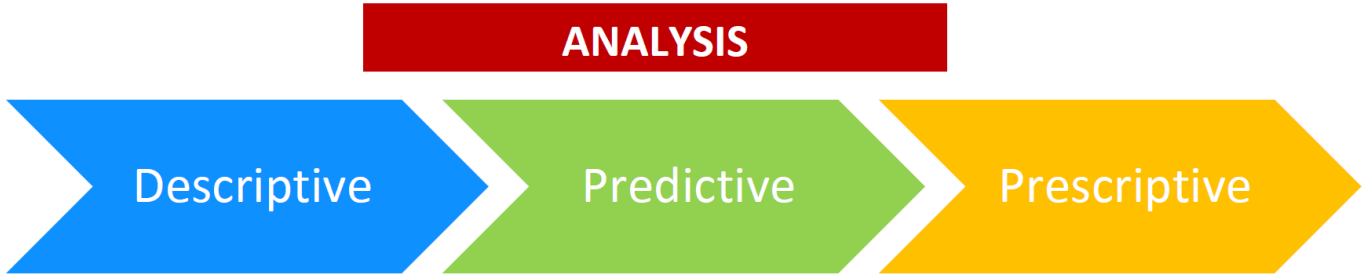
This Learning Product was produced by Ericsson AB.

Table of Content

Data analysis overview

Importance of Business Analytics (Analysis)

- Business Analytics(Analysis) refers to the skills, techniques and practices of continuous iterative exploration and investigation of history business performance to gain insights and drive business planning. Business Analytics focuses on developing new insights and understanding of business performance based on data and statistical methods.
- Business Analysis makes extensive use of analytical modeling and numerical analysis, including explanatory and predictive modeling, and fact-based management to drive decision making.



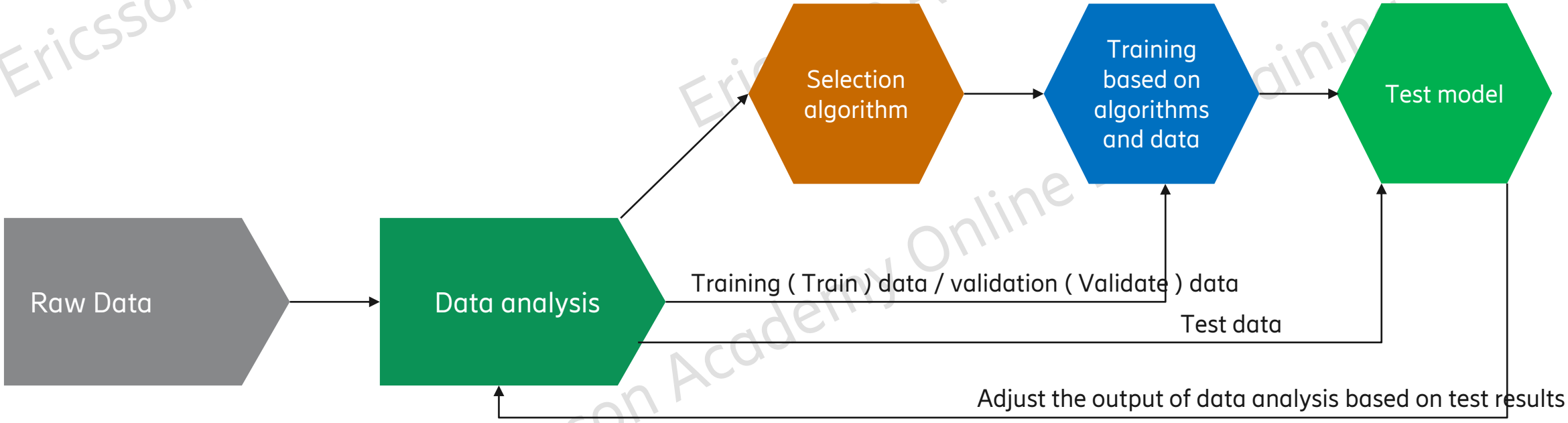
What happened? What is happening?	What will happen?	What to do? How to do it?
• Metrics • Dashboards • Data warehouse • Web analytics	• Prediction models • Statistical/Machine Learning • Text mining • A/B testing	• Optimization • Simulation • Modelling • Expert systems

Data-based analysis

- Descriptive analysis: Helps describe or summarize quantitative data by presenting statistical data . The purpose is to answer: " **What happened?** " and further to dig out "the reasons why similar results occurred." (Diagnostic analysis)
  - Average sales data, median height
  - Distribution of voice call duration
  - CPU , average memory ratio
- Predictive analysis : Based on the analysis of existing data, using statistical analysis and other means to discover available patterns to provide support for future decisions. The goal is to answer: " **What will happen?** "
  - Sales record for a certain month in the future
  - Determine the processing path of TR (Ericsson's actual internal use case)
  - Location planning of communication base stations
- Prescriptive analysis: Based on other types of analysis results to form recommendations to guide next steps. The goal is to answer: " **What should be done next?** "
  - How to boost sales in a given month
  - Expert system

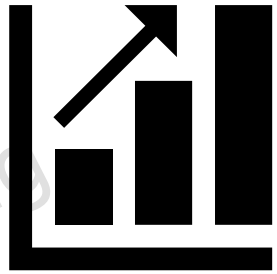
Machine learning data processing process

- The implementation of machine learning is to train a selected algorithm based on existing data and adjust the model parameters (output results) by verifying the quality of the training.
- The model obtained after training is tested through test data to determine the quality of the model.
- Training, validation, and test data should all come from the resulting dataset after analyzing the original raw data



Machine Learning and Data Analysis

Understand the relationships between data attributes and variables	Filter appropriate data sets and supplement missing data	Normalize the data to make it more suitable for machine learning models
Integrate more data sources to make data richer	Discover anomalies and errors in advance	Reconstruct new data items based on existing data to meet the needs of machine learning models



Machine Learning

Data Analysis

Understand the attributes and classification of data

Quantitative data

Discrete data

- Numerical data with a certain degree of accuracy, usually integers
  - number of objects
- Can apply most statistical analysis tools and theories

Continuous data

- Data that can theoretically be represented with any degree of accuracy
  - height, weight
  - object length
  - temperature
- Can apply most statistical analysis tools and theories

Qualitative data

Nominal data , Categorical data

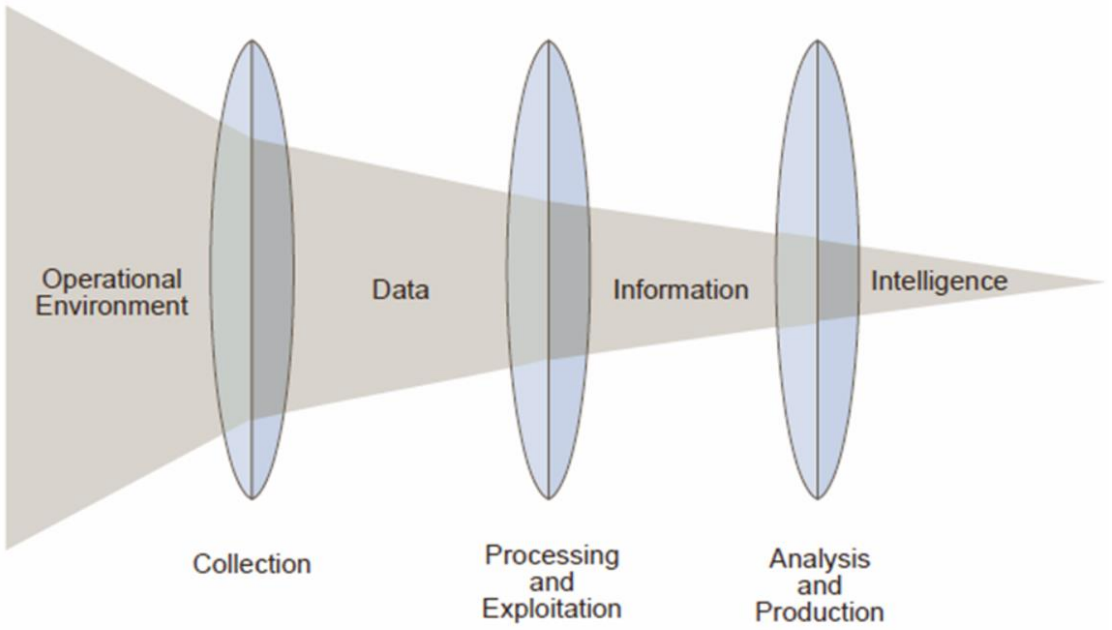
- Identifies data with text type data, which can be used for classification or labeling, and does not have sequential attributes.
  - gender
  - Country of Citizenship
  - color
- Mostly provide classification support for quantitative data analysis without directly analyzing it.

Ordinal data

- Data with sequential attributes, which can be text or numbers
  - Competition ranking
  - Customer experience score
- Can't do math on it

Data analysis overview

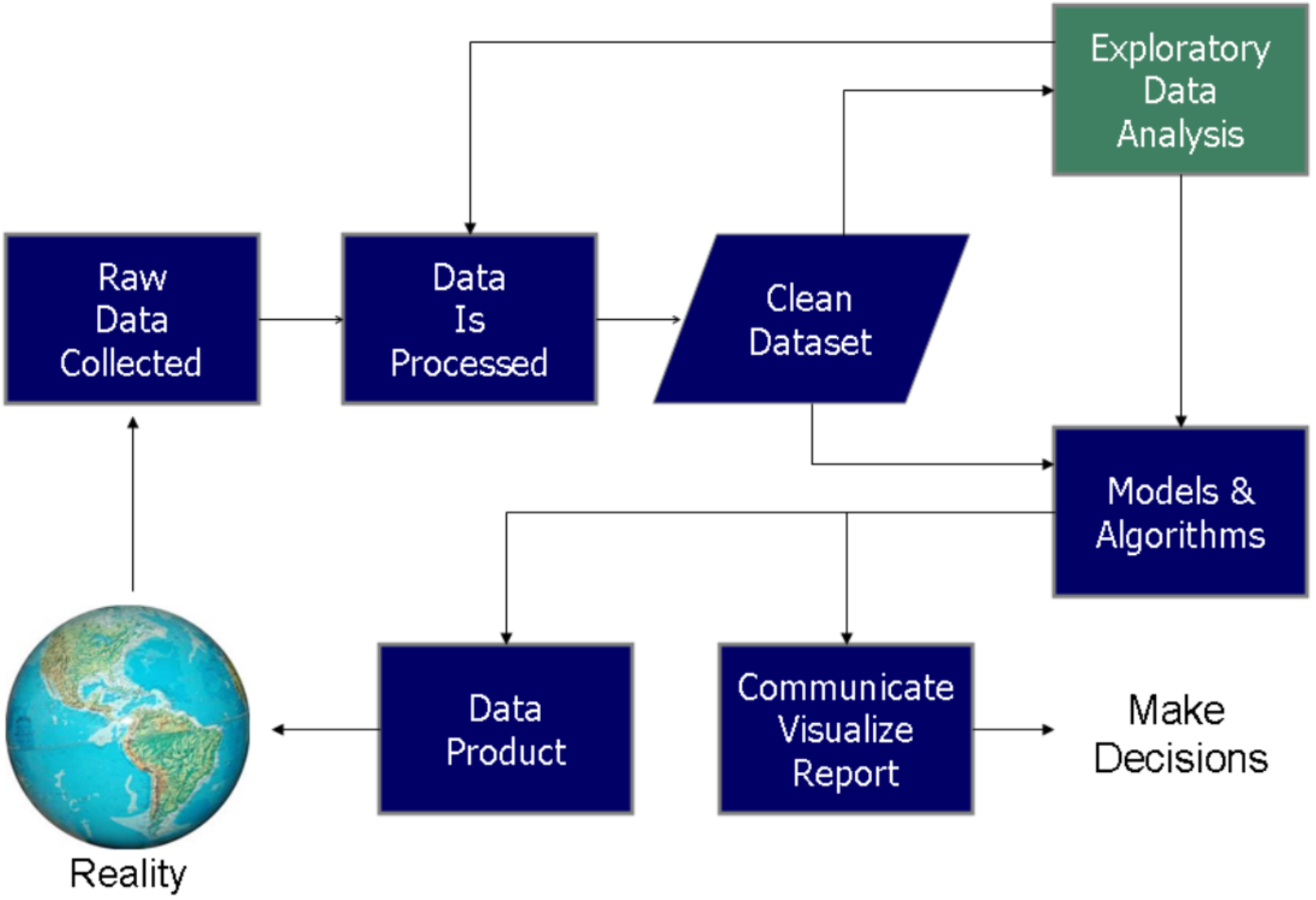
- Data analysis is the process of examining, cleaning, transforming, and modeling data with the goal of discovering useful information, drawing conclusions, and supporting decision making.
- Data analysis is the process of taking raw data and converting it into useful information for user decision-making. Collect and analyze data to answer questions, test hypotheses or refute theories.
- Statistician John Tukey 's definition of data analysis:
  - ***Procedures for analyzing data, techniques for interpreting the results of such procedures, ways of planning the gathering of data to make its analysis easier , more precise or more accurate , and all the machinery and results of (mathematical) statistics which apply to analyzing data.***



Typical process of data analysis

Basic steps of the process:

- Raw data collection
- Data cleaning and processing
- Apply specific analysis methods, such as: EDA , etc.
- Generate statistical analysis models, such as machine learning
- Communicate with demand parties based on visual results or data products
- New data may be generated and returned to the data output source



Important tasks in the data analysis process (1/2)

- Data requirements (business purpose)
  - Data as input to the analysis is necessary and needs to be specified according to the needs of the person directing the analysis or the customer who will use the finished analysis.
  - Analyzing sales data, analyzing demographic data, predicting part failure rates, etc. have their own specific needs, thereby interpreting them as demand input for subsequent steps.
- Raw data collection
  - Based on needs, data information is collected (and saved) from various possible sources, which may come from: enterprise information systems, external collection systems (IoT), the Internet or social networks, survey access data, traditional documents, and binary information (audio , videos and pictures)
  - The data collected should be as large and rich as possible (from a statistical analysis perspective), and different collection methods and storage solutions should be used for different attributes and types of data, such as files, databases, etc. The storage system provides as rich and efficient interfaces as possible so that the next processing and analysis work can proceed smoothly.
- Data (pre)processing
  - The collected data needs to be structured or organized before analysis .
  - Data without an obvious structure requires cooperation with SME for structural transformation and big data platforms for centralization and automation.

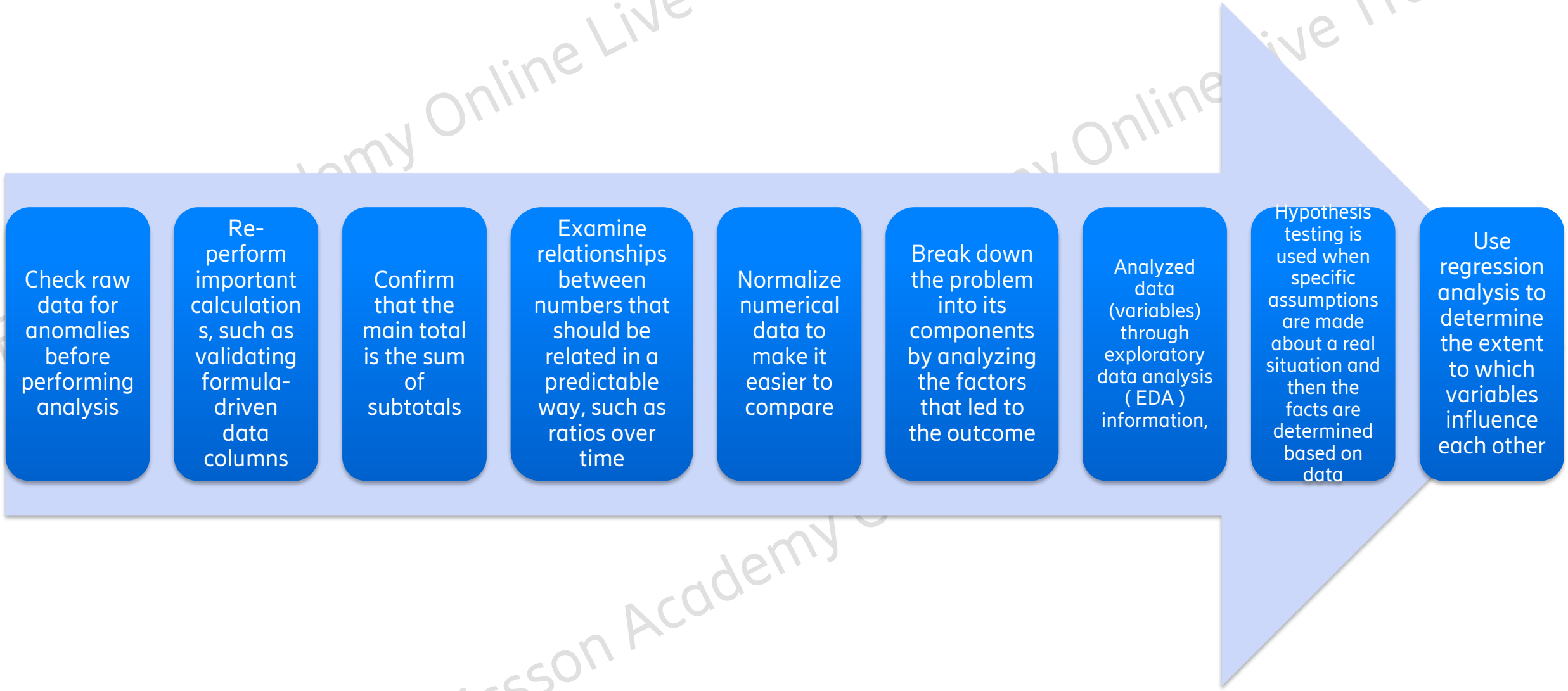
Important tasks in the data analysis process (2/2)

- **Data cleaning:**
  - After preprocessing, the data may be incomplete, contain duplicates, or contain errors. Data cleaning is the process of preventing and correcting these errors.
  - Common tasks include record matching, identifying data inaccuracies, overall quality of existing data, deduplication, and column splitting.
  - Such data issues can also be identified through various analytical techniques. For example : outlier detection methods for quantitative data can be used to remove data with a higher probability of input errors; text data spell checkers can be used to reduce the number of incorrectly entered words.
- **Exploratory data analysis (EDA) Exploratory data analysis:**
  - Exploratory data analysis (EDA) is used to analyze and investigate data sets and summarize their main characteristics, often using data visualization methods. It helps determine how to most effectively process data sources to get the answers you need, making it easier for data scientists to spot patterns, identify anomalies, test guesses, or validate hypotheses.
  - EDA is primarily used to see which data can reveal insights beyond conventional modeling or hypothesis testing tasks, helping to better understand data set variables and the relationships between them. It can also help determine whether the statistical methods you are considering for data analysis are appropriate.
  - After EDA and gained insights, you can use its capabilities for more complex data analysis or modeling, including machine learning.

Data analysis tools

- Open source tools:
  - **Pandas** is a software library for data manipulation and analysis written for the Python programming language. In particular, it provides data structures and operations for manipulating numerical tables and time series.
  - **SciPy** is a free, open-source Python library for scientific and technical computing, containing tools for optimization, linear algebra, integration, interpolation, special functions, FFTs , signal and image processing, ODE solvers, and others common in science and engineering. Task module.
  - **R** is a statistical computing and graphical programming language used by data miners, bioinformaticians, and statisticians for data analysis and developing statistical software.
  - **Apache Spark** is a data processing engine designed primarily for big data analysis, often referred to as large-scale data analysis. This is a very flexible analytics option as it can run on various platforms such as Hadoop , Apache Mesos or Kubernetes . Additionally, it is known for being developer-friendly and extremely fast thanks to its in-memory data engine.
- Commercial tools:
  - SAS BA , Splunk , Power BI , Excel

Data Analysis Best Practices



Common tasks in data analysis

Task	describe	Example
Get attribute information of a dataset	Get specific attributes from a given dataset	• Fuel consumption information for various vehicles • the length of a movie
filter	Find data that meets the requirements based on the given attribute value	• Students • VMs
Calculate derived data values	Calculate their aggregate values based on existing data sets	• The average height of male students in a grade • A division's operating income last quarter
find extreme values	Find the extreme value (certain range) of a certain attribute data in the data set	• Devices that consume the most power • The longest query statement
sort	Sort the data set	• Sort students based on exam ranking • Sorting flow table information based on forwarding priority
Decision scope	Given a set of data and an attribute of interest, find the range of values of the attribute in the data in the set	• The height range of male students in a grade • The operating income range of a department within 3 years
Data distribution characteristics	Describe the distribution of quantitative data in a set of data sets	• Age distribution of users signing up for a certain package • Distribution of power consumption of servers in a certain computer room
Anomaly found	Based on a certain relationship or expectation, discover abnormal items in the data set	• CPU and memory usage • Exception information in performance data
Cluster	Given a set of data, find data items with similar attribute values (not a specific attribute)	• Shoppers with similar buying behavior • VMs
Correlate	Given two attributes in the data set, determine the closeness of the relationship between them	• Is there a relationship between server power consumption and memory usage? • What is the relationship between package price and customer withdrawal?

Python - based data analysis

Popular Python data analysis tools (libraries)

- **Numpy** : Python programming language library that adds support for large multidimensional arrays and matrices, as well as a large collection of advanced mathematical functions to operate on these arrays.

- **Pandas**: Pandas is a software library for data manipulation and analysis. In particular, it provides data structures and operations for working with numerical tables and time series.

- **IPython** : IPython (Interactive Python) is a command shell for interactive computing in multiple programming languages . Originally developed for the Python programming language, it provides introspection , rich media, shell syntax, and tab completion. and historical records.

- **Jupyter Notebook** : Jupyter Notebook (formerly IPython Notebook) is a web -based interactive computing environment for creating, executing, and visualizing Jupyter notebooks . It supports execution environments (aka kernels) for dozens of languages. By default, Jupyter Notebook comes with IPython Kernel.

- **scikit-learn** is a Python machine learning library with a variety of classification, regression and clustering algorithms, including support vector machines, random forests, gradient boosting, k-means and DBSCAN, designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy .

Python overview

- High-level, interpreted, general-purpose programming language
- The first version was released in 1991 and was regarded as an improved version of LISP .
- Can run on almost any operating system
- Language Features:
  - readability
  - Concise syntax
  - Dynamic typing and garbage collection
  - Various programming paradigms :
    - object-oriented
    - imperative
    - Functional
  - Huge and extensive standard library

Install and configure Python

- Download the correct version from the official website: https://www.python.org/downloads/
- Most Linux and Mac OS come with the Python 2.X version pre-installed, and the Python 3.X version can be installed at the same time . Configure the default version by linking the Python pointed to by PATH to the version you want to use.
- Python for most Unix- like systems will only include part of the standard library, and some libraries (packages) need to be downloaded separately.

```
ubuntu@ubuntu-admin-node:~$ python -V
Python 2.7.12
ubuntu@ubuntu-admin-node:~$ which python
/usr/bin/python
ubuntu@ubuntu-admin-node:~$ ls -l /usr/bin/python
lrwxrwxrwx 1 root root 9 Nov 24  2017 /usr/bin/python -> python2.7
ubuntu@ubuntu-admin-node:~$ which python3
/usr/bin/python3
```

```
ubuntu@ubuntu-admin-node:~$ sudo ln -s /usr/bin/python3 /usr/bin/python
ubuntu@ubuntu-admin-node:~$ python -V
Python 3.5.2
```

Install and configure Python (Windows)

- Download the correct version
- Configure it into the PATH environment variable
- The Windows version generally contains all objects of the Python standard library and some other components.

Python Standard Library

- Python has a powerful standard library
- The core of Python language only contains common types and functions such as numbers, strings, lists, dictionaries, and files.
- The Python standard library provides additional capabilities such as system management, network communication, text processing, database interface, graphics system,
  - Text processing, including text formatting, regular expression matching, text difference calculation and merging, Unicode support, binary data processing and other capabilities
  - File processing, including file operations, creating temporary files, file compression and archiving, operating configuration files, etc.
  - Operating system functions, including thread and process support, IO reuse, date and time processing, calling system functions, logging , etc.
  - Network communication, including network sockets, SSL encrypted communication, asynchronous network communication and other capabilities
  - Network protocol, supports HTTP , FTP , SMTP , POP , IMAP , NNTP , XMLRPC and other network protocols, and provides a framework for writing network servers
  - W3C format support, including HTML , SGML , XML processing.
  - Other capabilities, including internationalization support, mathematical operations, HASH , Tkinter , etc.

Install Python packages

- Pip is Python 's package installer . You can use pip from the Python Package Index (Python Package Index PyPI) and other index installation packages.
- pip itself is also installed as a package and is generally included in the Python installation file.
- Make sure to use the correct version and latest pip program:
  - pip can be installed independently using package management software : sudo apt install pip
  - python3 -m pip install --user --upgrade pip
- Can be installed online via the Python package index
  - Official: https://pypi.org/
  - You can build your own index warehouse
  - pip install SomePackage
  - pip install SomePackage ==1.0.4 #Install the specified version
  - pip install ' SomePackage >=1.0.4' #Specify the minimum version
- Can be installed locally
  - pip install SomePackage-1.0-py2.py3-none-any.whl

Python Official Package Index — pypi.org (PyPI)

- PyPI is a software repository for the Python programming language.
- PyPI helps you find and install software developed and shared by other developers.
- Package authors use PyPI to distribute their software.
- Provde useful information of the package:
  - Instructions for use
  - Historic version
  - Source code address information
  - Author information
- Registered users can upload packages developed by themselves

Python command line tool (interpreter)

- After successfully installed Python, you can start the Python interpreter. It is the interpreter that translates Python code into executable code.
- In some Unix or Linux environments, multiple versions of Python may be installed . Confirm the version information through *python -V*
- The operation of the interpreter is very similar to that of Unix or Linux shell . It supports functions such as editing code interactively, querying history records, etc. At the same time, files can be provided to it in the form of parameters for execution.
- When some modules support running as a script, you can also use the –m module [arg] method to execute the module's functions.
- Python interpreter can add corresponding parameters and set environment variables
- Environment variables:
  - PYTHONPATH : Increase the path to search for modules
  - PYTHONSTARTUP : Defines the file to be executed before entering interactive mode
  - PYTHONDEBUG : Start debugging mode

```
python [-bBdEhiIOqsSuvVWx?] [-c command | -m module-name | script | - ] [args]
```

Python coding style

- Python developers need to follow corresponding coding styles to make Python code more readable
- PEP 8 (Style Guide for Python Code) is the code style definition currently followed by the
- Main principles:
  - 4 spaces indent, no tabs
  - Use code wrapping, and each line of code should not exceed 79 characters.
  - Use blank lines to separate functions and classes, and large blocks of code within functions
  - Include comments where possible and place them alongside the code
  - Use docstrings
  - Use spaces around operators and after commas
  - Name your classes and functions consistently ; convention is to use CamelCase for classes,
  - Use lower_case_with_underscores for functions and methods.
  - Always use self as the first parameter of a method
  - Use UTF-8 encoding by default
  - Don't use non- ASCII characters
- An excellent editor or IDE can optimize the code format

```
# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

Python data type: sequence

- Sequence types include three types: list , tuple , range
- The sequence number of the sequence type starts from 0
- If the sequence is operated based on i with a negative sequence number , then the corresponding sequence number is: len (s)+i .
- Sequences are divided into volate(mutable) and non-volatile (immutable)
  - Volatile: list , mostly used to save homogeneous types of data
    - Create list : [],[a,b,c],[x for x in iterable] list()
    - The data of mutable objects can be modified
  - Non-volatile: tuple , mostly used to save different types data
    - Create tuple : () , (a,b,c) , tuple()
    - The data of non-volatile objects cannot be modified

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from `s` (same as `del s[:]`)
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i] == x`
`s.reverse()`	reverses the items of *s* in place

Operation	
`x in s`	
`x not in s`	
`s + t`	
`s * n` or `n * s`	
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

List type and its methods

- List is a volatile sequence type, generally used to store the same type of data.
- Multiple ways
  - [] : Create an empty list
  - [a, b, c, d] : Add members of the list between square brackets , separated by commas
  - [x for x in iterable] : Created based on iterating of other objects
  - list() or list(iterable) : list type constructor (constructor)
- List method:
  - sort : Sort the members in the list . It will not return a new list , but change the current object.
- List is mutable:
  - Assignment: list[index_number] = value
  - Batch assignment: list[2:5] = ['C','D','E']
  - Clear the list : list[:] = []
  - List nesting (multidimensional array): list = [[' a','b','c '],[' d','e','f ']]

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

Tuple type and its methods

- Tuple is a non-volatile sequence type, mostly used to store different types of data.
- Multiple ways
  - () : Create an empty tuple
  - a, or (a,) : Create a single-member tuple
  - a,b,c or (a,b,c) : separate the members in
  - tuple() or tuple(iterable) : created using a builder
- Tuples can contain mutable members:
  t = ([1,2,3],[4,5,6])
- packing and unpacking

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>> a,b,c = t
>>> a
12345
>>> b
54321
>>> c
'hello!'
```

From "string" to tuple

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Range types and methods

- range is a non-volatile sequence type, mostly used to iterate specific numbers in for loops
- The range is created by the builder. The parameters of the builder need to be integers. The default step is 1.
  - range(start, stop[, step])
- The advantage of range over list or tuple is that it takes up very little memory and only saves the values of start, stop and step.

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Typical operation methods for sequence type data

- Enumerate :
  - The enumerate built-in method can iterate sequence data and obtain the index (index) and corresponding value at the same time:

  for i, value in enumerate(sequence_object):

  # i is index of each element and value is the value of the element.
- Sorted:
  - sorted can sort the sequence and then return a new sequence
- Zip:
  - Iterate multiple sequences in parallel, producing a new tuple that includes all iterated sequence

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

dictionary (dict) type and its methods

- dict is a mapping data type, which can implement data structures in the form of ***key : value*** . The key cannot be repeated
- Add key:value data in {} to create a dict , or use dict builder to create
- Main operations:
  - dict [key] = value: assignment
  - key in dict : judge dict Do you have a key ?
  - dict.clear () : clear dict
  - dict.get (key): Get value based on key
  - dict.keys () : returns dict key list
  - dict.pop (key) : Get value based on key and delete
  - dict.popitem (): Obtain (key, value) based on LIFO

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Control flow in

— while :

```
# Fibonacci series:
# the sum of two elements defines the next
a, b = 0, 1
while a < 10:
    print(a)
    a, b = b, a+b
```

— if...else.. :

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
```

Control flow in

— for : In addition to iterate based on numbers, it can also be based on any sequence data (list , string)

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestra
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Control flow in

- When looping over a dict, you can use the items() method to retrieve both the key and the corresponding value
- When looping through a sequence, you can use the enumerate() function to retrieve both the position index and the corresponding value simultaneously
- To loop through two or more sequences simultaneously, the entries in the sequences can be paired with the zip() function
- To loop a sequence in reverse direction, first reference the forward sequence and then call the

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)
```

```
for i, v in enumerate(['tic', 'tac', 'toe']):
    print(i, v)
```

```
for i in reversed(range(1, 10, 2)):
    print(i)
```

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}?  It is {1}.'.format(q, a))
```

File open / close

- Open: open(filename, mode, encoding=None)
  - filename : the file name to open (including path)
  - Open mode:

Mode	Description
r	Read-only mode
w	Write-only mode; creates a new file (erasing the data for any file with the same name)
x	Write-only mode; creates a new file, but fails if the file path already exists
a	Append to existing file (create the file if it does not already exist)
r+	Read and write
b	Add to mode for binary files (i.e., 'rb' or 'wb')
t	Text mode for files (automatically decoding bytes to Unicode). This is the default if not specified. Add t to other modes to use this (i.e., 'rt' or 'xt')

  - Obtain a file object, and subsequently read and write the content based on it
- Close: opened_file_object.close ()
- It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

Read and write file contents

- Prerequisite: The file object has been successfully created.
- File object methods:
  - read(size): Read the contents of the file in bytes as text (string)
  - readline(s): Read the contents of the file in line units
  - write(str): Write the contents of str to the file
  - writelines(strings): Write a series of string contents to the file
  - flush: Write the contents of the I/O cache to disk (file)
  - seek(pos): Position the file reading and writing position to the location defined by pos (in bytes)
  - tell: get the current reading and writing position

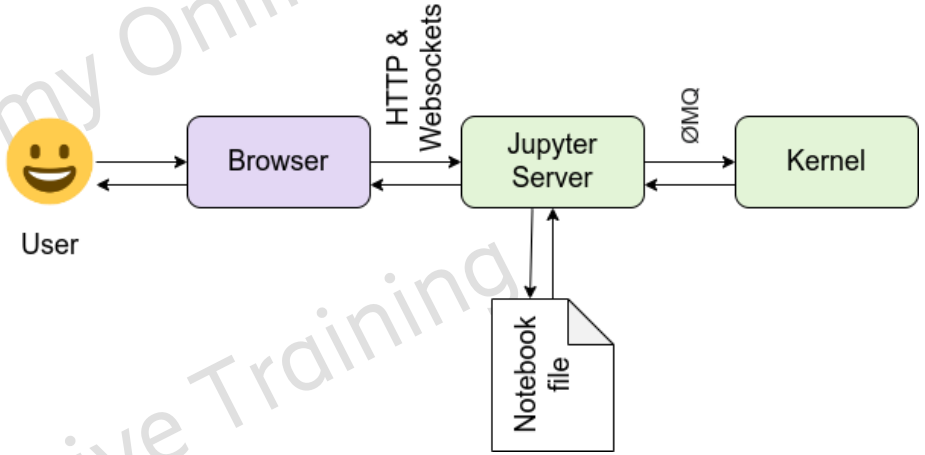
```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

IPython and Jupyter Notebook overview

IPython (Interactive Python) is a command shell for interactive computing in many programming languages . Originally developed for the Python programming language, it provides introspection, rich media, shell syntax, tab completion, and history .

Main components:

- Powerful interactive Python shell
- The kernel can run Python code in Jupyter notesbook



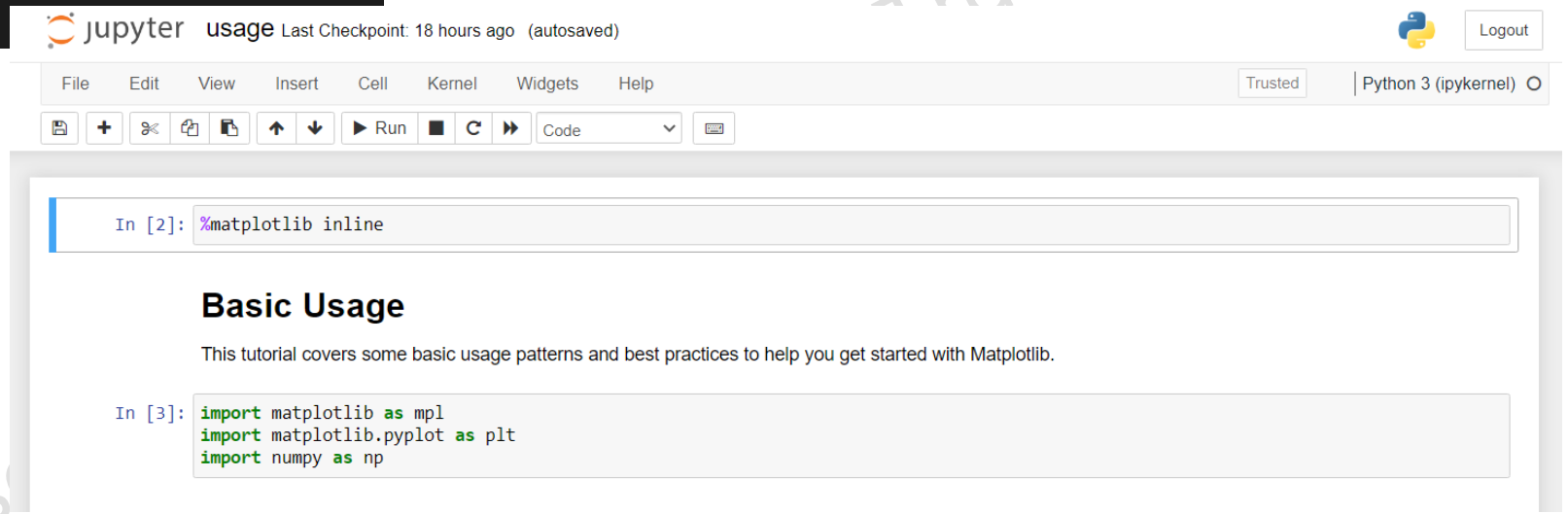
Jupyter Notebook , Server and Kernel :

- Jupyter Notebook and its flexible interface extend notebooks from code to visualization, multimedia, collaboration, and more. In addition to running your code, it also stores the code and output along with markdown comments in an editable document called a notebook. When you save it, it is sent from your browser to the Jupyter server, which saves it on disk as a JSON file with an .ipynb extension.
- The Jupyter server is a communication hub. The browser, the notebook files on disk, and the kernel cannot talk to each other directly. They communicate through a Jupyter server. It is the Jupyter server, not the kernel, that is responsible for saving and loading notebooks, so even if you don't have a kernel for the language, you can edit notebooks - you just won't be able to run the code. The kernel knows nothing about notebook documents: it simply receives code units sent to be executed when the user runs them.

Run IPython

- Install IPython based on pip, and then run ipython to achieve CLI access
- Or access the ready Jupyter Notebook environment directly through the browser

```
$ ipython
Python 3.6.9 (default, Jun 29 2022, 11:45:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print("hello world")
hello world

In [2]:
```



Basic concepts of Jupyter Notebook

Jupyter notebook provides a web -based application suitable for recording the entire computing process: developing, documenting and executing code, and communicating results. Jupyter notebook has two components:

- Web application: A browser-based tool for interactively authoring documents that combine explanatory text, mathematics, calculations, and their rich media output.
  - Edit code in the browser with automatic syntax highlighting, indentation, and tab completion / introspection .
  - The ability to execute code from the browser, with calculation results attached to the code that generated them.
  - Display calculation results using rich media representations such as HTML , LaTeX , PNG , SVG , etc. For example, graphics rendered by the matplotlib library can be included inline .
  - Rich text can be edited in the browser using the Markdown markup language that can provide comments for the code , not just plain text.
  - easily include mathematical symbols in markdown cells using LaTeX and powered by MathJax Local rendering.
- Notebook document: A representation of all visible content in a web application, including computational input and output, explanatory text, mathematics, images, and rich media representations of objects.
  - Notebook documents contain the input and output of an interactive session, as well as additional text that accompanies the code but is not used for execution. In this way, notebook files serve as a complete computational record of a session, interweaving executable code with explanatory text, rich representations of mathematics, and result objects.

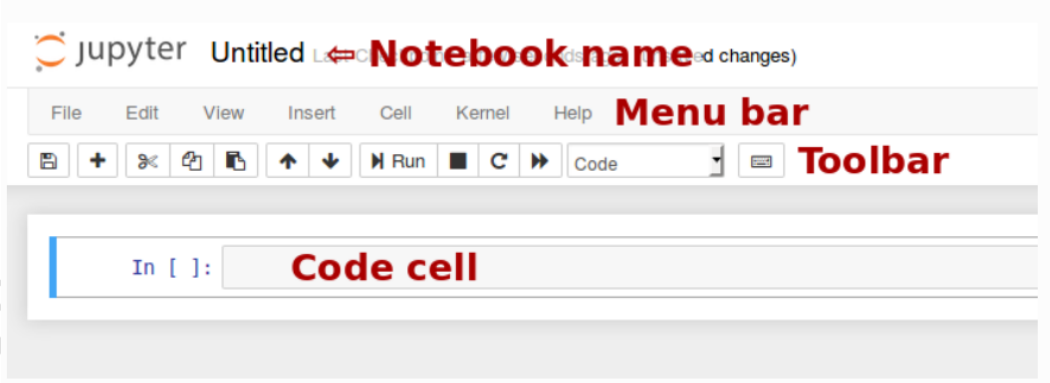
Create Jupyter Notebook

— Based on the Jupyter Notebook web user interface, you can easily create a new Notebook document through the browser:

- File->New

Jupyter Notebook user interface

When you create a new Notebook document, you will see the Notebook name, a menu bar, a toolbar, and an empty code cell .



- Notebook name: The name that appears at the top of the page, next to the Jupyter logo, reflects the name of the .ipynb file. Clicking on the notebook name brings up a dialog box where you can rename it.
- Menu Bar: The menu bar provides different options for operating Notebook's features.
- Toolbar: The toolbar provides a quick way to perform the most common actions in your notebook by clicking an icon.
- Code cells: The default cell type; continue reading for an explanation of cells.

Jupyter Notebook file

- Notebook consists of a series of cells (Cell).
- A cell is a multiline text input field whose contents can be executed by using Shift-Enter or clicking the " play " button in the toolbar .
- The execution behavior of a unit is determined by the type of unit. There are three types of cells: code cells, Markdown cells, and raw (Raw) cells. Each cell starts out as a code cell, but its type can be changed by using the drop-down menu on the toolbar (originally "Code") or the keyboard shortcut.
- Cell status:
  - Command Mode (Command Mode): Press ESC to enter, the cell turns blue, and you can select cells up and down through the keyboard.
  - Edit Mode : Press the Enter key on the selected cell to enter, the cell turns green, and the content of the cell can be edited.
- Code cells: Code cells allow you to edit and write new code, with full syntax highlighting and tab completion. The programming language you use depends on the kernel, and the default kernel (IPython) runs Python code. When a code unit is executed, the code it contains is sent to the kernel associated with the notebook . The results returned from this calculation are then displayed in the notebook as the cell's output.
- Markdown cells: You can record the calculation process in a narrative way, using rich text to alternately display descriptive text and code. In IPython , this is achieved by marking up the text using the Markdown language. The corresponding cells are called Markdown cells.
- Raw cells: Raw cells provide a place where output can be written directly. Notebook executes the content in the original

Markdown syntax

Markdown is a lightweight markup language that can be used to add formatting elements to plain text documents; it is a quick and easy way to take notes, create content for websites, and produce printable documents.

Basic syntax:

- Heading: Use different numbers of # to represent different levels of titles (add a blank line before and after)

heading level 1 -> <h1>heading level 1</h1>

heading level 2 -> <h2>heading level 1</h2>

- Paragraph : To create a paragraph , use blank lines to separate one or more lines of text
- Line breaks: To create a line break or new line (< br >) , end a line with two or more spaces and type return .
- Bold: To bold text, add two asterisks(**) or underlines before or after a word or phrase. To bold the middle of a word for emphasis, add two asterisks without spaces around the letters.
- Italic: To italicize text, add an asterisk or underline before or after a word or phrase. To italicize the middle part of a word for emphasis, add an asterisk without spaces around the letters.
- List: To create an ordered list, add line items with a number followed by a period. When creating an unordered list, add a dash (-) , asterisk (*) , or plus sign (+) in front of the line items .

Markdown example

```
# Heading 1
## Heading 2
### Heading 3
```

```
I really like using Markdown.

I think I'll use it to format all of my documents from now on.
```

```
**Bold text**
*Italic text*
```

```
1. First item
2. Second item
3. Third item
4. Fourth item
```

```
- First item
- Second item
- Third item
- Fourth item
```

```
At the command prompt, type `nano`.
```

```
    def code_block_mardown
        pass
```

Heading 1

Heading 2

Heading 3

I really like using Markdown.

I think I'll use it to format all of my documents from now on.

Bold text *Italic text*

1. First item
2. Second item
3. Third item
4. Fourth item

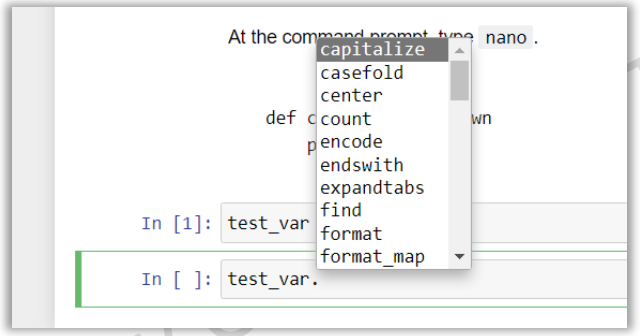
- First item
- Second item
- Third item
- Fourth item

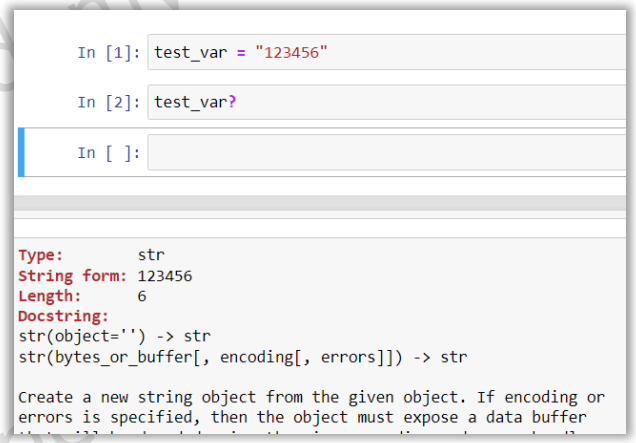
At the command prompt, type `nano`.

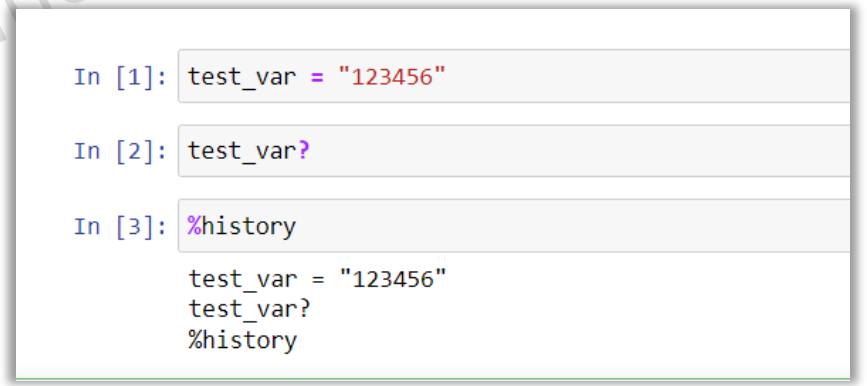
```
def code_block_mardown
    pass
```

IPython tips

- Tab completion:
  - Tab completion, especially for properties, is a convenient way to get the structure of any object you're working with. Just type object_name .<TAB> to view the properties of an object. In addition to Python objects and keywords, tab completion also works for file names and directory names.
- Inspect object information (introspection)
  - Type object_name**?** Various details about any object will be printed, including the docstring, function definition lines (for calling arguments), and constructor details for the class. To get specific information about an object, you can use the magic methods %pdoc , %pdef , %psource and %pfile
- History record
  - IPython Stores the commands you enter and the results they produce. You can use the up and down arrow keys to easily navigate through previous commands, or access your history in a more sophisticated way.
  - The input and output history is kept in variables named In and Out , recorded by prompt numbers, e.g. in [4] .
  - You can use the %history magic method to examine past input and output. Input history from previous sessions is saved in a database, IPython Can be configured to save output history.
- System command call:
  - To run any command in the system shell , just prepend it with ! . You can capture the output into a Python variable (List) , for example: files = !ls.







IPython Magic command

- IPython has a set of predefined "magic methods" that you can call like commands.
- There are two kind of Magic command, one is line-oriented (Line magic) and the other is cell-oriented (Cell magic).
- Line magics are prefixed with a % character and work much like OS command line calls: they take the rest of the line as arguments, where the arguments are passed without parentheses or quotes. Lines magic can return results and can be used as values for assignment operations.
- Cell magics are prefixed with a %% , and they are functions that take as arguments not only the rest of the line, but also the contents of the following lines.

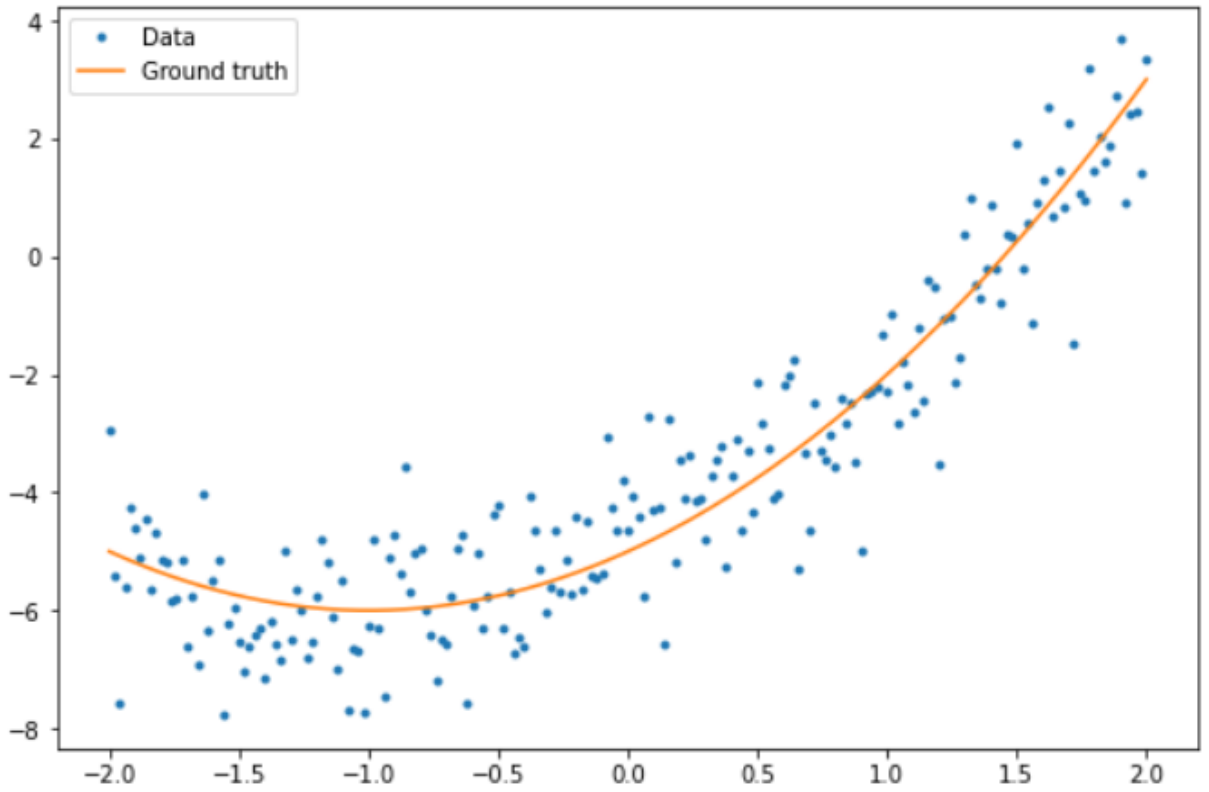
```
In [1]: %timeit range(1000)
100000 loops, best of 3: 7.76 us per loop

In [2]: %%timeit x = range(10000)
...: max(x)
...:
1000 loops, best of 3: 223 us per loop
```

- Commonly used methods:
  - Functions that work with code: %run , %edit , %save , %macro , %recall , etc.
  - that affect the shell : %colors , %xmode , %automagic
  - Other functions such as %reset , %timeit , %%writefile , %load , or %paste .

Integrate

- Jupyter Notebook or IPython It can realize multimedia output, such as graphics, audio and video. The commonly used integration method is integration with Matplotlib to realize data visualization function.
- The integration method is %matplotlib magic method. In Jupyter Notebook , the general writing method is: %matplotlib inline
- IPython Support more other drawing methods, such as: qt , wx , qtk,etc.

```
plt.plot(x.numpy(), y.numpy(), '.', label='Data')
plt.plot(x, f(x),  label='Ground truth')
plt.legend();
```



Numpy fundamental

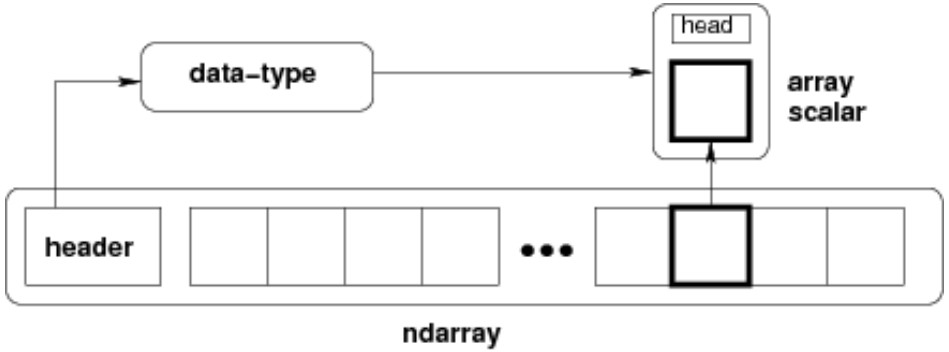
NumPy overview

- NumPy is the basic package for scientific computing in Python . It is a Python library that provides **multidimensional array objects,** various derived objects (such as masked arrays and matrices) and various routines for fast operations on arrays , including mathematics, logic, shape operations, sorting, selection, I/O , discrete Fourier transform, basic linear algebra, basic statistical operations, stochastic simulation, and more.
- At the heart of the NumPy package are **ndarray** objects, which are n-dimensional arrays that can store **homogeneous data** and support many operations that can be run in a high-performance manner.
- The advantages of NumPy as a basic library for scientific computing:
  - NumPy arrays have a fixed size when created, unlike Python lists (which can grow dynamically). Change ndarray will create a new array and delete the original array.
  - The elements in a NumPy array all need **to be of the same data type** and therefore the same size in memory.
  - NumPy arrays facilitate performing advanced mathematical operations and other types of operations on large amounts of data.
  - Calculation operations are implemented in the underlying C language
  - Vectorized operations abstract complex calculation logic (no need for for iterating) , very fast than native python code solution

ndarray – Numpy multidimensional array object

- NumPy provides an N-dimensional array type ndarray, which describes a collection of "items" of the same type. These items can be indexed using N integers.
- The items of all ndarrays are homogeneous: each item occupies the same size block of memory, and all blocks are described in exactly the same way. The description of each item in the array is specified by a separate data-type object. In addition to basic types (integers, floating point numbers, etc.), data type objects can also represent data structures.
- ndarray Object properties:
  - ndarray.ndim : The number of axes (dimensions) of the array.
  - ndarray.shape : The size of the array in each dimension. For a matrix with n rows and m columns, the shape will be (n,m) .
  - ndarray.size : The total number of array elements. This is equal to the product of the shape elements.
  - ndarray.dtype : An object describing the type of elements in the array.
  - ndarray.itemsize : Size of each element in the array (in bytes).

  It is equivalent to ndarray.dtype.itemsize .



Create ndarray

There are several ways to achieve the creation of ndarry:

- Based on the numpy.array method, pass in data of list or tuple type:
  - np.array ([2, 3, 4]) , np.array ([1.2, 3.5, 5.1])
  - np.array ([(1.5, 2, 3), (4, 5, 6)])
  - np.array ([[1, 2], [3, 4]], dtype =complex)
  - ~~np.array (1,2,3,4)~~
- numpy.zero and numpy.one method can create an array of all 0 or 1.
  - numpy.zero (100) numpy.one((10,10))
- numpy.empty: Create an array with all members uninitialized
  - numpy.empty ((1,2,3))
- numpy.arange: Similar to the built-in range method, returns an ndarray object
  - numpy.arange (0,10,1) # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a ones array of the same shape and dtype
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and dtype with all values set to the indicated "fill value" `full_like` takes another array and produces a filled array of the same shape and dtype
`eye, identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Print ndarray

- When printing an array, NumPy displays it similar to a nested list, but with the following layout:
  - the last axis is printed from left to right,
  - the second-to-last is printed from top to bottom,
  - the rest are also printed from top to bottom, with each slice separated from the next by an empty line.
- If the array is too large to print, NumPy will automatically skip the center part of the array and only print the data at the ends.

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4, 3)          # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2, 3, 4)       # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

```
>>> print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100, 100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

ndarray basic operations (1/2)

- The biggest advantage of Numpy's ndarray in terms of calculation is that it avoids the for-loop to implement arithmetic calculations for each member, so most of the arithmetic operations performed on ndarray are the same operations for each element (elementwise).
- Most arithmetic operations on ndarray do not change the original array object but produce new array objects.

```
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
>>> c = a - b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10 * np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854],
>>> a < 35
array([ True,  True, False, False])
```

```
>>> rg = np.random.default_rng(1)  # create instance of default random number generator
>>> a = np.ones((2, 3), dtype=int)
>>> b = rg.random((2, 3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b  # b is not automatically converted to integer type
Traceback (most recent call last):
    ...
numpy.core._exceptions._UFuncOutputCastingError: Cannot cast ufunc 'add' output from dtype
```

ndarray basic operations (2/2)

- When operating on arrays of different types, the type of the resulting array corresponds to the more precise type (a behavior called upcasting) .
- Many unary operations, such as calculating the sum of all elements in an array, are implemented as instance methods of the ndarray class.
- By default, these unary operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0, pi, 3)
>>> b.dtype.name
'float64'
>>> c = a + b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c * 1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

```
>>> a = rg.random((2, 3))
>>> a
array([[0.82770259, 0.40919914, 0.549593
       [0.02755911, 0.75351311, 0.538143
>>> a.sum()
3.1057109529998157
>>> a.min()
0.027559113243068367
>>> a.max()
0.8277025938204418
```

```
>>> b = np.arange(12).reshape(3, 4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)      # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)      # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)  # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Indexing, slicing, and iterating ndarray (1/2)

— Indexing, slicing, and iterating a one-dimensional ndarray are similar to Python's sequence object operations:operations:

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> # equivalent to a[0:6:2] = 1000;
>>> # from start to position 6, exclusive, set every 2nd element to 1000
>>> a[:6:2] = 1000
>>> a
array([1000,    1, 1000,   27, 1000,  125,  216,  343,  512,  729])
>>> a[::-1]  # reversed a
array([ 729,  512,  343,  216,  125, 1000,   27, 1000,    1, 1000])
>>> for i in a:
...     print(i**(1 / 3.))
...
9.999999999999998
1.0
```

ndarray Indexing, slicing, and iterating (2/2)

- Axis of a multidimensional array can have its own index, and specific members can be accessed in the same way as nested sequences.
- When fewer indices are provided than the number of axes, the missing indices are considered full slices.

```
b = np.arange(0,20).reshape(4,5)
b[-1] # the last row. Equivalent to b[-1, :]
```

- A slice is a view of the original array and modifications to it affect the original array.
- Iterating a multidimensional array is done relative to the first axis.
- However, if you want to perform an operation

on each element in the array, you can use the flat attribute,

which is an iterator over all elements of the array:

```
In [15]: for row in m_array:
             print(row)

[0 1 2]
[3 4 5]
[6 7 8]
[ 9 10 11]

In [16]: for row in m_array.flat:
             print(row)

0
1
2
3
4
```

```
In [6]: m_array.shape
Out[6]: (4, 3)

In [7]: m_array
Out[7]: array([[ 0,  1,  2],
               [ 3,  4,  5],
               [ 6,  7,  8],
               [ 9, 10, 11]])

In [8]: m_array[0,1]
Out[8]: 1

In [9]: m_array[1,1]
Out[9]: 4

In [10]: m_array[0:]
Out[10]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])

In [11]: m_array[0:3,1]
Out[11]: array([1, 4, 7])

In [12]: m_array[1,0:]
Out[12]: array([3, 4, 5])

In [13]: m_array[0:2,0:2]
Out[13]: array([[0, 1],
                [3, 4]])

In [14]: m_array[-1]
Out[14]: array([ 9, 10, 11])
```

Boolean array based indexing

- With Boolean indexing, we explicitly choose which items in the array we want and which we don't.
- For boolean indexing, the most natural approach one can think of is to use a boolean array with the same shape as the original array:
- The second way of using boolean indexing is more similar to integer indexing; for each dimension of the array we are given a 1D boolean array to select the slice we want
- The ~ operator is useful when you want to invert a selection condition
- For example: a[~b] # The element whose value a is not greater than 4

```
>>> a = np.arange(12).reshape(3, 4)
>>> b = a > 4
>>> b  # `b` is a boolean with `a`'s shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> a[b]  # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

```
>>> a = np.arange(12).reshape(3, 4)
>>> b1 = np.array([False, True, True])         # first dim selection
>>> b2 = np.array([True, False, True, False])  # second dim selection
>>>
>>> a[b1, :]                                   # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                                      # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[:, b2]                                   # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1, b2]                                  # a weird thing to do
array([ 4, 10])
```

Change ndarray shape

— The shape of the ndarray can be modified as needed, and most operations create new ndarray objects.

```
>>> a = np.floor(10 * rg.random((3, 4)))
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.shape
(3, 4)
```

```
>>> a.ravel()  # returns the array, flattened
array([3., 7., 3., 4., 1., 4., 2., 2., 7., 2., 4., 9.])
>>> a.reshape(6, 2)  # returns the array with a modified shape
array([[3., 7.],
       [3., 4.],
       [1., 4.],
       [2., 2.],
       [7., 2.],
       [4., 9.]])
>>> a.T  # returns the array, transposed
array([[3., 1., 7.],
       [7., 4., 2.],
       [3., 2., 4.],
       [4., 2., 9.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

Matrix transpose

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Mirror-reverse all elements of A around a 45-degree ray from the lower right starting from the element in row 1 and column 1 to obtain the transpose of A

— The resize operation has the same function as reshape , but it changes the original array without generating a new array.

Increase array dimensions

- You can use np.newaxis and np.expand_dims to increase the dimensions (**not rows or columns**) of an existing array .
- Using np.newaxis will increase the dimension of the array by one dimension. This means that a 1D array will become a 2D array, a 2D array will become a 3D array, and so on.
- You can also do this by using np.expand_dims method extends the array by inserting a new axis at the specified axis.
- The transposition (transpose) of a one-dimensional array can be implemented , a very common trick in Pandas operation

```
a = np.array([1, 2, 3, 4, 5, 6])
a2 = a[np.newaxis, :]
a3 = a[:,np.newaxis]
a2.shape, a3.shape
```

```
((1, 6), (6, 1))
```

```
a4 = a.reshape(2,3)[:,np.newaxis]
a4,a4.shape
```

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

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

```
a5 = np.expand_dims(a,axis=0)
a6 = np.expand_dims(a,axis=1)
a5.shape, a6.shape
```

```
((1, 6), (6, 1))
```

Stack ndarray

- The vstack operation implements vertical stacking, and the hstack operation implements horizontal stacking
- The column_stack method can combine multiple 1-dimensional arrays as columns into a 2-dimensional array

```
>>> a = np.floor(10 * rg.random((2, 2)))
>>> a
array([[9., 7.],
       [5., 2.]])
>>> b = np.floor(10 * rg.random((2, 2)))
>>> b
array([[1., 9.],
       [5., 1.]])
>>> np.vstack((a, b))
array([[9., 7.],
       [5., 2.],
       [1., 9.],
       [5., 1.]])
>>> np.hstack((a, b))
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
```

```
In [38]: a_arrary=np.array([1,2,3,4])

In [40]: b_arrary=np.array(["a","b","c","d"])

In [43]: np.hstack((a_arrary,b_arrary))
Out[43]: array(['1', '2', '3', '4', 'a', 'b', 'c', 'd'], dtype='<U11')

In [45]: np.column_stack((a_arrary,b_arrary))
Out[45]: array([['1', 'a'],
                ['2', 'b'],
                ['3', 'c'],
                ['4', 'd']], dtype='<U11')

In [46]: c_arrary=np.array(["one","two","three","four"])

In [47]: np.column_stack((a_arrary,b_arrary,c_arrary))
Out[47]: array([['1', 'a', 'one'],
                ['2', 'b', 'two'],
                ['3', 'c', 'three'],
                ['4', 'd', 'four']], dtype='<U11')
```

Split ndarray

- split function supports even splitting of arrays, while array_split supports uneven splitting
- hsplit splits the array based on horizontal (column), vsplit splits the array based on vertical (row)

```
>>> x = np.arange(9.0)
>>> np.split(x, 3)
[array([0.,  1.,  2.]), array([3.,  4.,  5.]), array([6.,  7.,  8.])]
```

```
>>> x = np.arange(8.0)
>>> np.split(x, [3, 5, 6, 10])
[array([0.,  1.,  2.]),
 array([3.,  4.]),
 array([5.]),
 array([6.,  7.]),
 array([], dtype=float64)]
```

```
>>> x = np.arange(8.0)
>>> np.array_split(x, 3)
[array([0.,  1.,  2.]), array([3.,  4.,  5.]), array([6.,  7.])]
```

```
>>> x = np.arange(9)
>>> np.array_split(x, 4)
[array([0, 1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]
```

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
       [ 4.,  5.],
       [ 8.,  9.],
       [12., 13.]]),
 array([[ 2.,  3.],
       [ 6.,  7.],
       [10., 11.],
       [14., 15.]])]
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
       [ 4.,  5.,  6.],
       [ 8.,  9., 10.],
       [12., 13., 14.]]),
 array([[ 3.],
       [ 7.],
       [11.],
       [15.]]),
 array([], shape=(4, 0), dtype=float64)]
```

Add and delete members in ndarray

- insert function inserts a new value along the given axis before the given index position. If axis is not specified, the array will be "flattened" first.
- append function adds a new value to the end of the array
- delete function deletes values along the specified axis and returns a new array (commonly used instead with Boolean selection)

```
>>> arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> np.delete(arr, 1, 0)
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
```

```
>>> np.delete(arr, np.s_[::2], 1)
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
>>> np.delete(arr, [1,3,5], None)
array([ 1,  3,  5,  7,  8,  9, 10, 11, 12])
```

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([1, 5, 1, ..., 2, 3, 3])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
```

```
>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
array([1, 2, 3, ..., 7, 8, 9])
```

When *axis* is specified, *values* must have the correct shape.

```
>>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)
Traceback (most recent call last):
    ...
ValueError: all the input arrays must have same number of dimensions, but
the array at index 0 has 2 dimension(s) and the array at index 1 has 1
dimension(s)
```

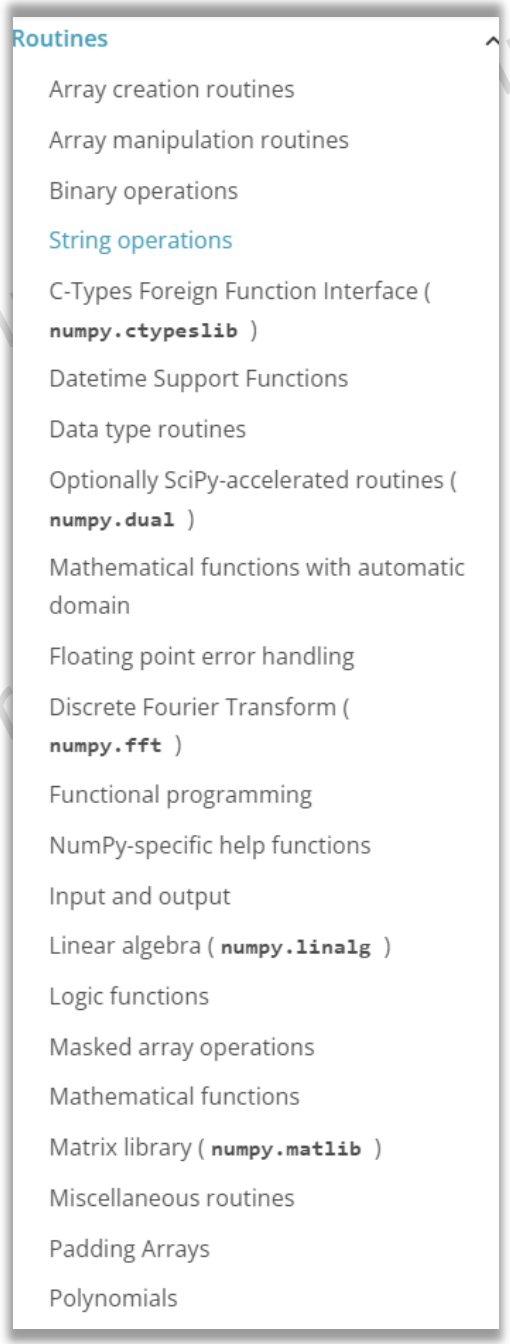
Numpy ufunc

- A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a “vectorized” wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.
- There are currently more than 60 universal functions defined in numpy on one or more types, covering a wide variety of operations. Some of these ufuncs are called automatically on arrays when the relevant infix notation is used (e.g., add(a, b) is called internally when a + b is written and a or b is an ndarray).

arithmetic operations	Trigonometric functions	Bitwise operations	comparison operation	floating point operations
• add • subtract • multiply • divide • absolute • log • mod • square • sqrt	• sin • cos • tan	• bitwise_and • bitwise_or • invert • left_shift • right_shift	• greater • greater_equal • less • less_equal • not_equal • equal	• isfinite • isinf • isnan • isnat • fabs • signbit • floor • ceil • trunc

Numpy Routine

- Numpy Most of the native functions provided are packaged as Routine , syntax: numpy.ROUTINE_NAME .
- Commonly used routines :
  - ndarray Create: empty , ones , zeros , full
  - ndarray operate:
    - Transform ndarray Shape: reshap , ravel
    - Change dimensions: broadcast
    - Change type: asarray , asmatrix
    - Stacking and splitting ndarray
    - for ndaaray Add and remove elements
  - Date and time processing: datatime_data
  - Data type processing: dtype , finfo
  - Mathematical calculation methods: basic arithmetic, trigonometric functions, logarithms



Numpy Broadcasting in (1/2)

- The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes.
- General Broadcasting Rules(2 arrays):
  - Compares the last dimension of two arrays (from right to left) if:
    - they are equal or
    - One of them is 1
- The input arrays do not need to have the same dimensions. The resulting array will have the same dimensions as the input array with the largest dimension.

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

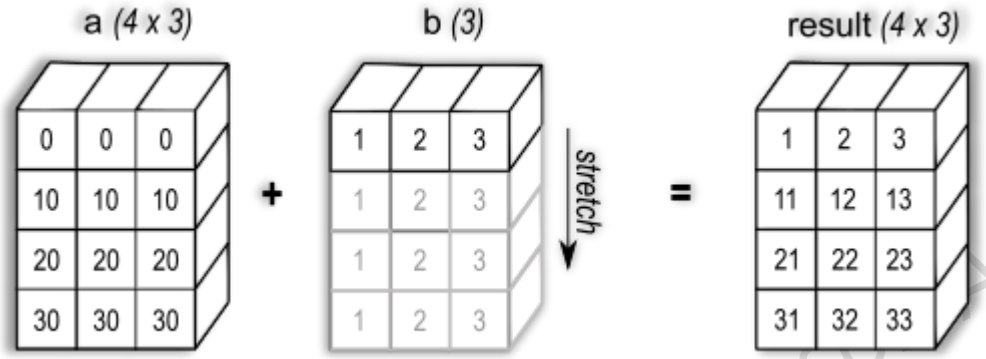
```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

Numpy Broadcasting in (2/2)

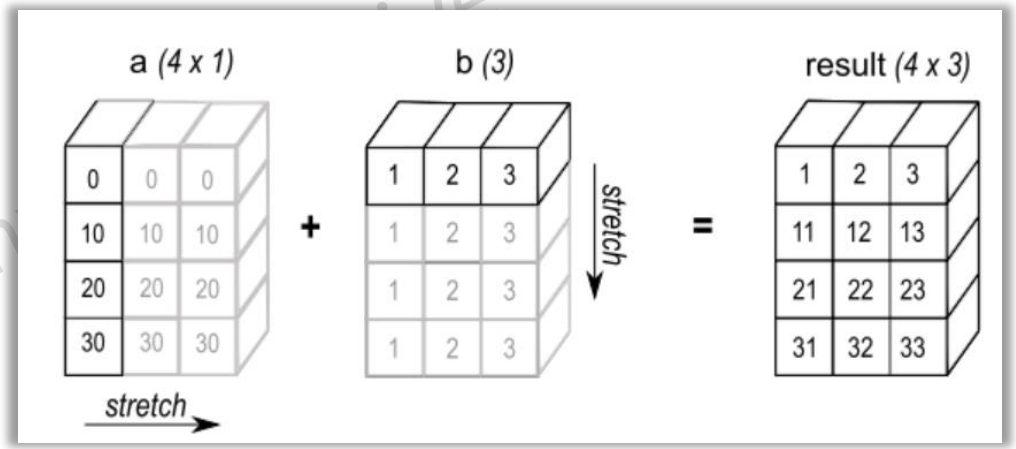
— 2- dimensional array + 1- dimensional array:

```
>>> a = np.array([[ 0.0,  0.0,  0.0],
...               [10.0, 10.0, 10.0],
...               [20.0, 20.0, 20.0],
...               [30.0, 30.0, 30.0]])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a + b
array([[ 1.,   2.,   3.],
       [11.,  12.,  13.],
       [21.,  22.,  23.],
       [31.,  32.,  33.]])
```



— Outproduct :

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[ 1.,   2.,   3.],
       [11.,  12.,  13.],
       [21.,  22.,  23.],
       [31.,  32.,  33.]])
```



Generate pseudo-random numbers

- The use of random number generation is an important part of the configuration and evaluation of many numerical and machine learning algorithms.
- Numpy The latest (v1.17.0 and later) random number generation method uses BitGenerator to create sequences, and use Generator to use these sequences to sample from different statistical distributions to generate random numbers.
- But currently, many examples and books still use the old version of the generator, using numpy.random.RandomState to access its methods (actually written as numpy.random)

```
>>> np.random.random_sample()
0.47108547995356098 # random
>>> type(np.random.random_sample())
<class 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54
```

- Commonly used methods:
  - random_sample (size) : Generate a random array with a shape defined by size , with a value of [0.0, 1.0)
  - rand(d0,d1,d2...) : Generate random floating point numbers according to the input dimensions, with a value of [0.0, 1.0)
  - randint (low,high,size) : Generate a random integer with a shape defined by size , with a value of [low, high)
  - Generate random numbers from a probability distribution:
    - normal(size) : Generate random floating point numbers with a shape defined by
    - Similarly, it also supports common probability distributions such as Poisson distribution and Bernoulli distribution.
  - shuffle command can re-sort the array randomly (changing the original array), and the permutation will generate a newly sorted array based on the original array (without changing the original array)

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Using Pandas achieve data analysis tasks

Pandas overview

- Pandas is a Python library that provides fast, flexible and expressive data structures designed to make working with "relational" or "tagged" data simple and intuitive.
- It aims to be the essential high-level building blocks for practical, real-world data analysis in Python . The library is highly optimized for performance, with critical code paths written in Cython Or written in C.
- Before using Panda , you need to import it: import pandas as pd
- Main features:
  - Vectorization enables many built-in methods for fast data manipulation
  - DataFrame for variable data manipulation Objects and Series objects
  - Tools for reading and writing data between in-memory data structures and different file formats
  - Integrated data alignment and missingness handling
  - Reshape and rotate data sets
  - Tag-based slicing, flexible indexing, and subsetting of large datasets
  - Insertion and deletion based on data structure columns
  - Group By engine that allows typical split-apply-combine operations on data sets
  - Provide data filtering function

Pandas data structure

- Pandas provides three types of data structures for data processing:
  - Series : one-dimensional **homogeneous** array-like data structure
  - DataFrame : **two- dimensional heterogeneous** array data structure (tabular data, with rows and columns)
  - Index : An object that records data structure axis information. It can have: name, label members, etc.
- The best way to understand Pandas data structures is as flexible containers for low-dimensional data. For example, DataFrame is a container for Series, and Series is a container for scalars, and objects are inserted and deleted from these containers in a dictionary-like manner.

```
df = pd.DataFrame(
    {
        "Name": [
            "Braund, Mr. Owen Harris",
            "Allen, Mr. William Henry",
            "Bonnell, Miss. Elizabeth",
        ],
        "Age": [22, 35, 58],
        "Sex": ["male", "male", "female"],
    }
)
```

Index	Name	Age	Sex
0	Braund , Mr. Owen Harris	twenty two	male
1	Allen, Mr. William Henry	35	male
2	Bonnell , Miss. Elizabeth	58	female

Column

Series

View the properties of a Pandas data structure object

- Pandas objects have many properties and methods for accessing their metadata and basic information:
  - shape property : gives the axis dimension of the object, consistent with ndarray
  - Axis label (Index or subclass object) property
    - Series : index
      - DataFrame : index (rows) and columns
  - info method: Print information about the data structure object, including index data type and columns, non-null values and memory usage.
  - Series array attribute and to_array method can return the raw information of the data object, excluding axis (Axis) labels
  - The to_numpy method can return the ndarray format of the original information of the data object and control the output data type

```
exam_csv.shape
```

```
(1000, 8)
```

```
exam_csv.index
```

```
RangeIndex(start=0, stop=1000, step=1)
```

```
exam_csv.columns
```

```
Index(['gender', 'race/ethnicity', 'parental level of education', 'lunch',
       'test preparation course', 'math score', 'reading score',
       'writing score'],
      dtype='object')
```

```
exam_csv.gender.index
```

```
RangeIndex(start=0, stop=1000, step=1)
```

```
exam_csv.info()
```

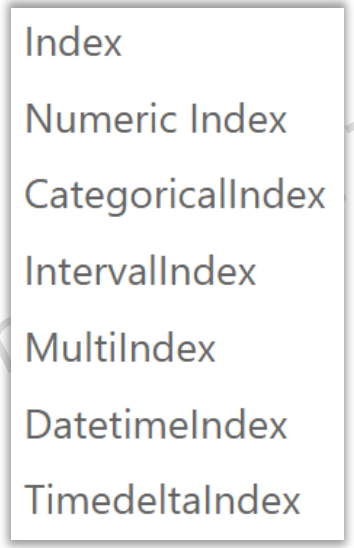
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column                       Non-Null Count  Dtype 
---  ------                       --------------  ----- 
 0   gender                       1000 non-null   object
 1   race/ethnicity               1000 non-null   object
 2   parental level of education  1000 non-null   object
 3   lunch                        1000 non-null   object
 4   test preparation course      1000 non-null   object
 5   math score                   1000 non-null   int64 
 6   reading score                1000 non-null   int64 
 7   writing score                1000 non-null   int64 
dtypes: int64(3), object(5)
memory usage: 62.6+ KB
```

```
exam_csv.gender.array
```

```
<PandasArray>
[  'male', 'female',   'male',   'male',   'male', 'female', 'female',
   'male',   'male',   'male',
 ...
   'male', 'female',   'male', 'female',   'male',   'male',   'male',
 'female', 'female',   'male']
Length: 1000, dtype: object
```

Index object

- index in pandas is an **immutable sequence** used for indexing and alignment , also used to store axis labels for all pandas objects .
- The axis label information in pandas objects is used in several ways:
  - Identify data (metadata) which is important for analysis, visualization, and interactive console displays.
  - Implement automatic and explicit data alignment.
  - Allows intuitively getting and setting subsets of a dataset.
- There are many types that can be used as indexes in Pandas , the most commonly used is pandas.Index (Class)
- The member values saved in the index cannot be modified.
- Index object can store duplicate data (try to avoid it)
- The reindex method of the data object can create a new index object based on new data and attributes.




```
df
```

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

```
new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
             'Chrome']
df.reindex(new_index)
```

	http_status	response_time
Safari	404.0	0.07
Iceweasel	NaN	NaN
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Chrome	200.0	0.02

Series data structure

- Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.
- pandas.Series (data,index =index) is commonly used to create Series
  - data is ndarray : pd.Series (np.random.randn (5), index=["a", "b", "c", "d", "e"])
  - data is dictionary : pd.Series ({"b":1,"a":0,"c":2})
  - data is a scalar and assigned to all members; the index object must be provided: pd.Series (5.0, index=["a", "b", "c", "d", "e"])
- All members in Series have only one type (dtype)
  - All types inherited from Numpy
  - Types for Pandas and third-party extensions: ExtensionDtype
- Series has a name attribute that can be specified when created, or automatically set to the Column's label when selected from a DataFrame

```
In [3]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [4]: s
Out[4]:
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64

In [5]: s.index
Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))
Out[6]:
0   -0.173215
1    0.119209
2   -1.044236
3   -0.861849
4   -2.104569
dtype: float64
```

```
s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])
s.info()
```

```
<class 'pandas.core.series.Series'>
Index: 5 entries, a to e
Series name: None
Non-Null Count  Dtype
--------------  -----
5 non-null      float64
dtypes: float64(1)
memory usage: 80.0+ bytes
```

Series basic operations

- Implement a slice similar to ndarray. Slices are also valid for index objects, and Numpy's ufunc is also valid:

  s[0] s[:3] s[s > s.median ()] s[[4, 3, 1]] np.exp(s)

- The implementation is the same as dictionary, accessing value based on key, where key is the tag value of index:

  s["a"] "e" in s

- Vector operations based on the index tag allow Series to be iterated like a ndarray and aligned based on index
- If the label is not found in one Series or the other , the result will be marked as NaN

```
In [29]: s + s
Out[29]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64
```

```
In [32]: s[1:] + s[:-1]
Out[32]:
a         NaN
b   -0.565727
c   -3.018117
d   -2.271265
e         NaN
dtype: float64
```

DataFrame data structure

- DataFrame is a two-dimensional labeled data structure that can have different types of columns.
- You can think of it as a spreadsheet or a SQL table, or a dictionary of Series objects. It is the most commonly used pandas object. Like Series , DataFrame accepts many different types of input.
  - Dict of 1D ndarrays, lists, dicts, or Series
  - 2-D numpy.ndarray
  - Structured or record ndarray
  - A Series
  - Another DataFrame
- When creating a DataFrame, you are allowed to provide row label (index label) or column label (column label) objects, but the provider must ensure the correspondence between data and labels
- If no axis labels are provided during creation, they will be created automatically based on the entered data.
- If the input data is a labeled Series nested in a dictionary, then the row index labels provided when creating the DataFrame do not match the data labels, then the corresponding data will be deleted (Right sample)

```
In [38]: d = {
   ....:     "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
   ....:     "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
   ....: }
   ....:

In [39]: df = pd.DataFrame(d)

In [40]: df
Out[40]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [41]: pd.DataFrame(d, index=["d", "b", "a"])
Out[41]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [42]: pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
Out[42]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

```
In [43]: df.index
Out[43]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [44]: df.columns
Out[44]: Index(['one', 'two'], dtype='object')
```

Create DataFrame from various ways

- The creation of a DataFrame can be based on a variety of data formats as sources and a variety of methods (DataFrame methods):
  - a dictionary-form Series

```
d = {
    "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
    "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
}
```

  - a dictionary forms or list (tuple)

```
d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}
```

  - Based on structured array data

```
data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])

data[:] = [(1, 2.0, "Hello"), (2, 3.0, "World")]
```

  - a list form of dictionaries

```
data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]
```

- Series to_frame() can used to convert Series to DataFrame with same shape.

File-based data reading and writing

- The pandas I/O API is a set of top-level reader functions accessed like pandas.read_csv(), which typically returns a pandas object. The corresponding writer function is an object method accessed like DataFrame.to_csv().
- Reading from file:
  - pd.read_csv ("FILE_NAME") #Return DataFrame object
  - pd.read_excel ("FILE_NAME", **sheet_name = "SHEET_NAME"**)
  - pd.read_csv ('examples/ex6.csv', **nrows =5**) #Read 5 rows of data
  - chunker = pd.read_csv ('examples/ex6.csv', **chunksize =1000**) #Divide the read file into 1000 parts, and then iterate and read
- Write to file:
  - DataFrame.to_execel ("FILE_NAME", **sheet_name = " SHEET_NAME", index =False**) #Ignore the index tag information
  - data.to_csv (sys.stdout , **sep ='|'**) #Set data separator
  - data.to_csv (sys.stdout , na_rep ='NULL') #Fill specific values for missing data

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	to_orc
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

Data indexing and selection

- Axis label information (index) in the pandas object allows intuitively getting and setting subsets of the dataset.
- Main implementation methods:

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Select rows by boolean vector	`df[bool_vec]`	DataFrame

- Order based on index selection:

Object Type	Indexers
Series	`s.loc[indexer]`
DataFrame	`df.loc[row_indexer,column_indexer]`

```
In [13]: df_dict_series.loc["a","one"]
Out[13]: 1.0
```

```
In [4]: df_dict_series
```

Out[4]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

```
In [5]: df_dict_series.loc["a"]
```

```
Out[5]: one    1.0
        two    1.0
        Name: a, dtype: float64
```

```
In [6]: df_dict_series.loc["a":"c"]
```

Out[6]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0

```
In [8]: df_dict_series.loc["a":]
```

Out[8]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

```
In [12]: df_dict_series.iloc[0]
```

```
Out[12]: one    1.0
         two    1.0
         Name: a, dtype: float64
```

Perform data operations based on [] (column)

- DataFrame Operations:
  - df [" column_name "] #Read the entire column based on the column label
  - df ["three"] = df ["one"] * df ["two"] #Create a new column based on existing column data
  - df ["flag"] = df ["one"] > 2 #Create a new boolean column based on existing column data
  - df [:3] #Read data based on rows
  - df.assign () #Based on the original DataFrame Add new columns and return a new DataFrame
  - df.insert (column_index,column_label,ndarry) #In column_index Insert new column at position instead of at the end
  - del df [" column_name "] #Delete columns based on column labels
  - df.drop (" column_name",axis =1) #Delete columns based on column labels and return the modified DataFrame
  - df.pop (" column_name ") #Delete columns based on column labels and return the corresponding columns
- Series Operations
  - s[:5] #The first five elements based on the index position
  - s[::2] #The radix position element based on the index position, the first, the third ...
  - s[::-1] #Reverse reading based on index order

Select data from DataFrame based on row index (1/2)

- Use the .loc attribute to manipulate data based on index tags (also applicable to Series)
  - df.loc [row_indexer,column_indexer]: The first parameter in [] is the row index label, and the second parameter is the column index label.
  - df.loc ['a'] #The row with row label a and all its columns, the return type is Series
  - df.loc [['a', 'b', 'd'], :] #The row labels are a, b, c row and all its columns
  - df.loc ['d':, 'A':'C'] #All rows starting from row label d and their A , B , C columns
  - df.loc [:, df.loc ['a'] > 0] #Select columns based on Boolean array
- When using .loc with slicing, if both start and end labels are present in the index, return the elements in between (inclusive)

```
In [45]: df1
Out[45]:
          A         B         C         D
a  0.132003 -0.827317 -0.076467 -1.187678
b  1.130127 -1.436737 -1.413681  1.607920
c  1.024180  0.569605  0.875906 -2.211372
d  0.974466 -2.006747 -0.410001 -0.078638
e  0.545952 -1.219217 -1.226825  0.769804
f -1.281247 -0.727707 -0.121306 -0.097883

In [46]: df1.loc[['a', 'b', 'd'], :]
Out[46]:
          A         B         C         D
a  0.132003 -0.827317 -0.076467 -1.187678
b  1.130127 -1.436737 -1.413681  1.607920
d  0.974466 -2.006747 -0.410001 -0.078638
```

```
In [47]: df1.loc['d':, 'A':'C']
Out[47]:
          A         B         C
d  0.974466 -2.006747 -0.410001
e  0.545952 -1.219217 -1.226825
f -1.281247 -0.727707 -0.121306
```

```
In [48]: df1.loc['a']
Out[48]:
A    0.132003
B   -0.827317
C   -0.076467
D   -1.187678
Name: a, dtype: float64
```

Select data from DataFrame based on row index (2/2)

- Position-based integer indexing starts at 0 . When slicing, include the starting boundary and exclude the ending boundary. [1:3] includes 1,2 but not 3 .
- Use the .iloc attribute to manipulate data based on index tags (also applicable to Series)
  - df. i loc [row_indexer,column_indexer]: The first parameter in [] is the row index position value, and the second parameter is the column index position value.
  - df.iloc [3] : The fourth row and all its columns, the return type is Series
  - df.iloc [:3] : the first to third rows and all their columns
  - df.iloc [1:5, 2:4] : columns two to five and columns three to four
  - df.iloc [[1, 3, 5], [1, 3]] : The second, fourth, and sixth rows and their second and fourth columns

```
In [68]: df1
Out[68]:
           0         2         4         6
0   0.149748 -0.732339  0.687738  0.176444
2   0.403310 -0.154951  0.301624 -2.179861
4  -1.369849 -0.954208  1.462696 -1.743161
6  -0.826591 -0.345352  1.314232  0.690579
8   0.995761  2.396780  0.014871  3.357427
10 -0.317441 -1.236269  0.896171 -0.487602
```

```
In [71]: df1.iloc[[1, 3, 5], [1, 3]]
Out[71]:
           2         6
2  -0.154951 -2.179861
6  -0.345352  0.690579
10 -1.236269 -0.487602
```

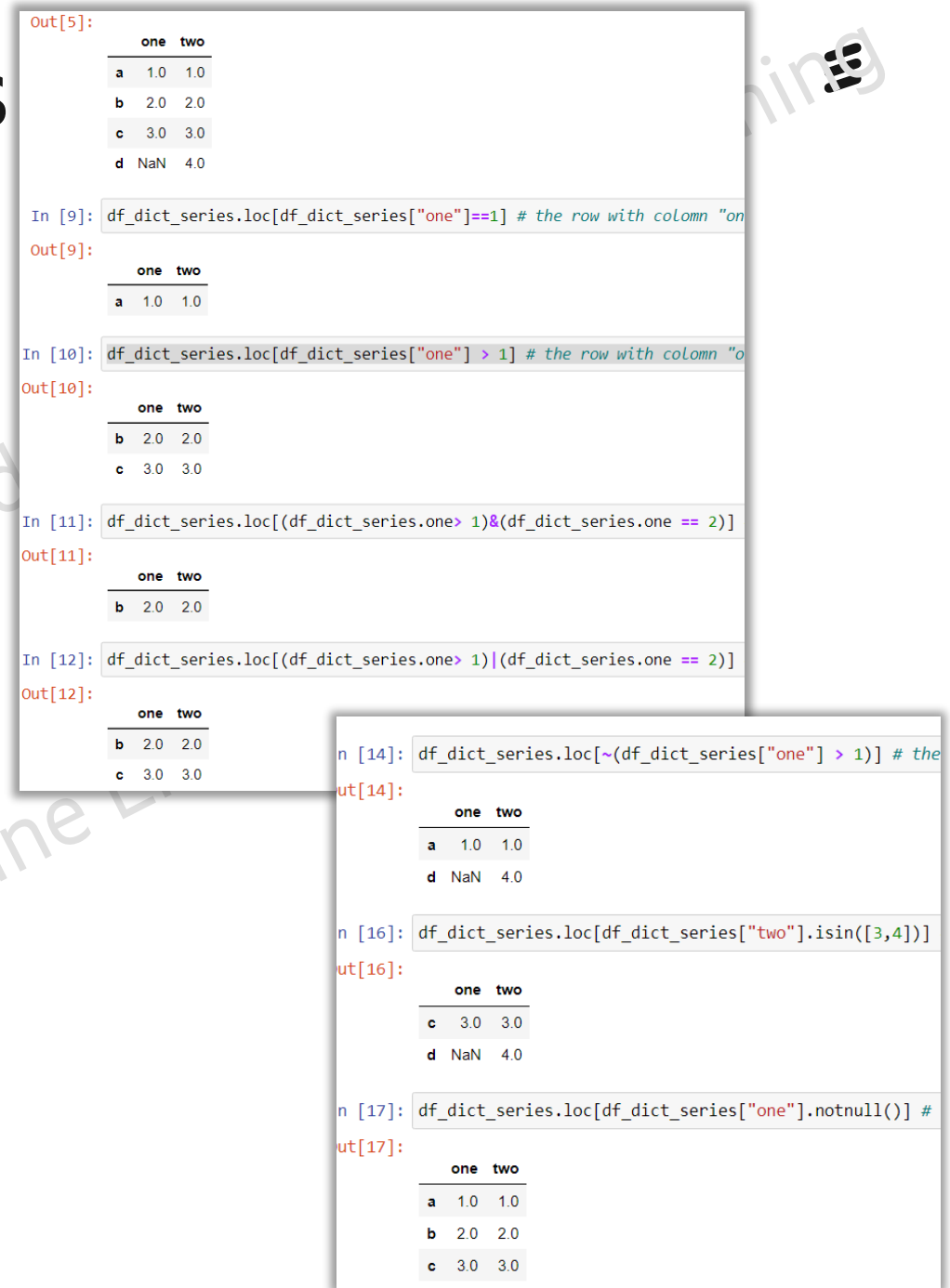
```
In [69]: df1.iloc[:3]
Out[69]:
          0         2         4         6
0  0.149748 -0.732339  0.687738  0.176444
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161
```

```
In [70]: df1.iloc[1:5, 2:4]
Out[70]:
          4         6
2  0.301624 -2.179861
4  1.462696 -1.743161
6  1.314232  0.690579
8  0.014871  3.357427
```

Data selection based on conditions

— In addition to the standard DataFrame- based Or Series row and column information for data selection. There are also some methods to select the required data based on **logical conditions.**

- df.loc [df [column] == value] # Based on the value of a certain column as a conditional judgment, select the entire row of data
- df.loc [(df [column1] == value)&(df [column2] > value)] #Multiple condition combination, similar to AND
- df.loc [(df [column1] == value)|(df [column2] > value)] #Multiple condition combination, similar to OR
- df.loc [~(df [column] == value)] #Conditional inverse selection
- df.loc [(df.column.isin ([list of values])]#isin method determines whether to match the list value
- df.loc [(df.column.notnull ()]#notnull method selects rows whose columns are non-null values



```
Out[5]:
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

```
In [9]: df_dict_series.loc[df_dict_series["one"]==1] # the row with colomn "on
Out[9]:
```

	one	two
a	1.0	1.0

```
In [10]: df_dict_series.loc[df_dict_series["one"] > 1] # the row with colomn "o
Out[10]:
```

	one	two
b	2.0	2.0
c	3.0	3.0

```
In [11]: df_dict_series.loc[(df_dict_series.one> 1)&(df_dict_series.one == 2)]
Out[11]:
```

	one	two
b	2.0	2.0

```
In [12]: df_dict_series.loc[(df_dict_series.one> 1)|(df_dict_series.one == 2)]
Out[12]:
```

	one	two
b	2.0	2.0
c	3.0	3.0

```
n [14]: df_dict_series.loc[~(df_dict_series["one"] > 1)] # the
ut[14]:
```

	one	two
a	1.0	1.0
d	NaN	4.0

```
n [16]: df_dict_series.loc[df_dict_series["two"].isin([3,4])]
ut[16]:
```

	one	two
c	3.0	3.0
d	NaN	4.0

```
n [17]: df_dict_series.loc[df_dict_series["one"].notnull()] #
ut[17]:
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0

Query data based on "query" method

- DataFrame Objects have a query() method that allows the use of expressions for data selection.
- query accepts an expression and uses column names as variables to define selection conditions:
  - df [df.A > df.B] can be written as df.query ('A > B')
  - If the defined variable (such as A or B above) does not have a corresponding column name, it will continue to be searched in the Index of the row .
  - You can also directly use the keyword index to replace the Index number of the row.
  - In addition to common comparison operations, in and not in are also supported (the usage is similar to == and !=)
  - If the column name is complex (for example, it contains spaces), you need to use `` to enclose it.
- query's performance is better than that of query by column.

```
df[(df['a'] > df['b']) & (df['b'] < df['c'])]
```

	a	b	c
3	8	6	7
4	2	0	1
5	4	3	5

```
df.query('(a > b) & (b < c)')
```

	a	b	c
3	8	6	7
4	2	0	1
5	4	3	5

```
df.query('(a > b) and (b < c)')
```

	a	b	c
3	8	6	7
4	2	0	1
5	4	3	5

```
In [225]: df.query('index < b < c')
Out[225]:
   b  c
2  5  6
```

```
df = pd.DataFrame({'a': list('aabbccddeeff'), 'b': list('aaaabbbbcccc'),
                   'c': np.random.randint(5, size=12),
                   'd': np.random.randint(9, size=12)})
```

```
df[df['a'].isin(df['b'])]
```

	a	b	c	d
0	a	a	4	7
1	a	a	1	5
2	b	a	1	7
3	b	a	1	5
4	c	b	4	1
5	c	b	4	7

```
df.query('a in b')
```

	a	b	c	d
0	a	a	4	7
1	a	a	1	5
2	b	a	1	7
3	b	a	1	5
4	c	b	4	1
5	c	b	4	7

"Take a look "of the dataset

— When you got a dataset, you should have a very quick view of it with several (instance) functions

- df.sample(n):
  - Randomly select fraction of n rows from dataset

```
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0
```

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

- df.nlargest(n, columns), df.nsmallest(n, columns)
  - Return the first n rows ordered by columns in descending(ascending) order.

```
>>> df
           population      GDP alpha-2
Italy        59000000  1937894      IT
France       65000000  2583560      FR
Malta          434000    12011      MT
Maldives       434000     4520      MV
Brunei         434000    12128      BN
Iceland        337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

```
>>> df.nlargest(3, 'population')
          population      GDP alpha-2
France    65000000  2583560      FR
Italy     59000000  1937894      IT
Malta       434000    12011      MT
```

- df.head(n), df.tail(n)
  - Return firt(last) n rows.

Binary operations between data objects

- Binary operations (addition, subtraction, multiplication, division, comparison, etc.) between data objects can be implemented in two ways:
  - Symbol mode:
    - dataframe1 + dataframe2; dataframe1 + series1; dataframe1 > dataframe2
  - Based on a specific method: Each binary operation corresponds to at least one method. The difference from the symbolic method is that parameters can be set in the method implementation:
    - axis : Specifies to operate on rows (default, 0) or columns (1)
    - fill_value : fill in values for missing values (methods related to comparisons (lt, eq, etc.) do not have this parameter)

`DataFrame.add`
Add DataFrames.
`DataFrame.sub`
Subtract DataFrames.
`DataFrame.mul`
Multiply DataFrames.
`DataFrame.div`
Divide DataFrames (float division).
`DataFrame.truediv`
Divide DataFrames (float division).
`DataFrame.floordiv`
Divide DataFrames (integer division).
`DataFrame.mod`
Calculate modulo (remainder after division).
`DataFrame.pow`
Calculate exponential power.

`DataFrame.eq`
Compare DataFrames for equality elementwise.
`DataFrame.ne`
Compare DataFrames for inequality elementwise.
`DataFrame.le`
Compare DataFrames for less than inequality or equality elementwise.
`DataFrame.lt`
Compare DataFrames for strictly less than inequality elementwise.
`DataFrame.ge`
Compare DataFrames for greater than inequality or equality elementwise.
`DataFrame.gt`
Compare DataFrames for strictly greater than inequality elementwise.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

Descriptive statistics

- It is impossible to fully understand the properties of the data by just looking at the data items saved in the data object. Additional information needs to be expressed through other statistical-related information.
- Most related operations will achieve data aggregation (from high dimension to low dimension). By default, processing is based on rows. You can set the axis=1 parameter to achieve column-based processing. skipna Options can control whether to ignore invalid values, the default is True .
- The most convenient method is describe to quickly view multiple commonly used

```
In [97]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=["a", "b", "c", "d", "e"])

In [98]: frame.iloc[::2] = np.nan

In [99]: frame.describe()
Out[99]:
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean     0.033387    0.030045   -0.043719   -0.051686    0.005979
std      1.017152    0.978743    1.025270    1.015988    1.006695
min     -3.000951   -2.637901   -3.303099
25%     -0.647623   -0.576449   -0.712369
50%      0.047578   -0.021499   -0.023888
75%      0.729907    0.775880    0.618896
max      2.740139    2.752332    3.004229
```

```
In [101]: s = pd.Series(["a", "a", "b", "b", "a", "a", np.nan, "c", "d", "a"])

In [102]: s.describe()
Out[102]:
count     9
unique    4
top       a
freq      5
dtype: object
```

count	Number of non-NA observations
sum	Sum of values
mean	Mean of values
mad	Mean absolute deviation
median	Arithmetic median of values
min	Minimum
max	Maximum
mode	Mode
abs	Absolute Value
prod	Product of values
std	Bessel-corrected sample standard deviation
var	Unbiased variance
sem	Standard error of the mean
skew	Sample skewness (3rd moment)
kurt	Sample kurtosis (4th moment)

Commonly used statistical information methods

- **count; value_counts ; nunique** : Statistics of the number of non- NA values based on the column ; statistics of the number of unique values based on the column; returns the number of unique values of the corresponding dimension
- **min, max** : Calculate the minimum and maximum values
- **argmin , argmax** : Calculate the index position (integer) of the minimum or maximum value respectively.
- **idxmin , idxmax** : Calculate the index label that obtain the minimum or maximum value respectively.
- **quantile** : Calculate the sample quantile in the range of 0 to 1
- **sum** : Sum of values
- **mean** : Mean of values
- **median** : Arithmetic median (50% quantile)
- **var** : Sample variance of values, squared value of std
- **std** : Sample standard deviation of the values
- **cumsum**: Cumulative sum of values
- **pct_change**: Calculate percentage change

Handling missing data

- There are several ways to deal with missing data in the data:
  - Determine missing data information:
    - **isna()**: Returns a data object of the same shape, True represents a null value, otherwise False ;
    - **notna()** and **notnull()** is counter-method
  - Delete missing data information:
    - dropna : You can delete rows or columns containing missing data; you can also specify the deletion judgment conditions (how is any or all , thresh specifies the data); you can also specify the columns for deletion judgment; and whether to perform it in the source data object (inplace =True)
  - Fill in missing data:
    - fillna : implements filling missing data with specified values or filling based on different methods (ffill , bfill) and directions; you can also fill different rows or columns with different values (value); at the same time, you can specify the number of fills (limit Set the first few)

```
     A    B    C    D
0  NaN  2.0  NaN  0.0
1  3.0  4.0  NaN  1.0
2  NaN  NaN  NaN  NaN
3  NaN  3.0  NaN  4.0
```

```
>>> df.fillna(method="ffill")
     A    B    C    D
0  NaN  2.0  NaN  0.0
1  3.0  4.0  NaN  1.0
2  3.0  4.0  NaN  1.0
3  3.0  3.0  NaN  4.0
```

```
>>> values = {"A": 0, "B": 1, "C": 2, "D": 3}
>>> df.fillna(value=values)
     A    B    C    D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  2.0  1.0
2  0.0  1.0  2.0  3.0
3  0.0  3.0  2.0  4.0
```

Replace data (1/2)

— In addition to processing missing data, it is sometimes necessary to replace certain data (alternative ways to handle missing data). The commonly used methods are:

- replace : replace ***to_replace*** with ***value*** The value given in to_replace Can be: str, regex, list, dict , Series, int, float
  - Supports limit , inplace , method and other parameters
  - The regex parameter can control whether to use regular expressions to implement replacement.

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s.replace(1, 5)
0    5
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

Replace data (2/2)

— where method: replace the element whose condition (cond) is False with the new value (other). The opposite method is mask, which is used similarly.

- ***cond*** is a conditional judgment parameter, which can be a judgment expression, such as s>10 , or a Boolean value sequence.
- ***other*** is the value that will be replaced if the judgment result is False. NaN if not provided

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```
>>> s = pd.Series(range(5))
>>> t = pd.Series([True, False])
>>> s.where(t, 99)
0     0
1    99
2    99
3    99
4    99
dtype: int64
>>> s.mask(t, 99)
0    99
1     1
2    99
3    99
4    99
dtype: int64
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  2  3
2  4  5
3  6  7
4  8  9
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

Combine data objects – concat()

— You can use pd.concat() to concatenate pandas objects along a particular axis

```
df_con_col = pd.concat([df,df],ignore_index=True)
df_con_row = pd.concat([df,df],axis=1)
df_con_col
```

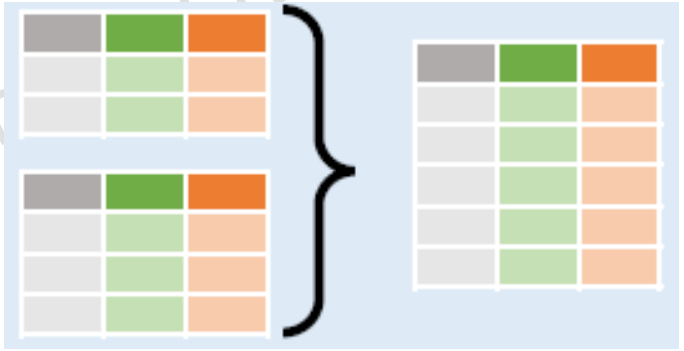
✓ 0.0s

	Name	Age	Sex
0	Braund, Mr. Owen Harris	22	male
1	Allen, Mr. William Henry	35	male
2	Bonnell, Miss. Elizabeth	58	female
3	Braund, Mr. Owen Harris	22	male
4	Allen, Mr. William Henry	35	male
5	Bonnell, Miss. Elizabeth	58	female

```
df_con_row
```

✓ 0.0s

	Name	Age	Sex	Name	Age	Sex
0	Braund, Mr. Owen Harris	22	male	Braund, Mr. Owen Harris	22	male
1	Allen, Mr. William Henry	35	male	Allen, Mr. William Henry	35	male
2	Bonnell, Miss. Elizabeth	58	female	Bonnell, Miss. Elizabeth	58	female



— join parameter can be inner and outer.

Sorting data

- Pandas supports three kinds of sorting:
  - sorting by index label,
  - sorting by column value
  - a combination of both.
- By index label: The Series.sort_index() and DataFrame.sort_index() methods are used to sort pandas objects by index level.
- The Series.sort_values () method is used to sort a Series by value .
- The DataFrame.sort_values () method is used to sort DataFrame by column or row values.
- The optional ***by*** parameter of DataFrame.sort_values () can be used to specify one or more columns to use to determine the sort order.
- Parameter "ascending = True" is default.

```
In [302]: unsorted_df
Out[302]:
      three       two       one
a       NaN -1.152244  0.562973
d -0.252916 -0.109597       NaN
c  1.273388 -0.167123  0.640382
b -0.098217  0.009797 -1.299504

# DataFrame
In [303]: unsorted_df.sort_index()
Out[303]:
      three       two       one
a       NaN -1.152244  0.562973
b -0.098217  0.009797 -1.299504
c  1.273388 -0.167123  0.640382
d -0.252916 -0.109597       NaN
```

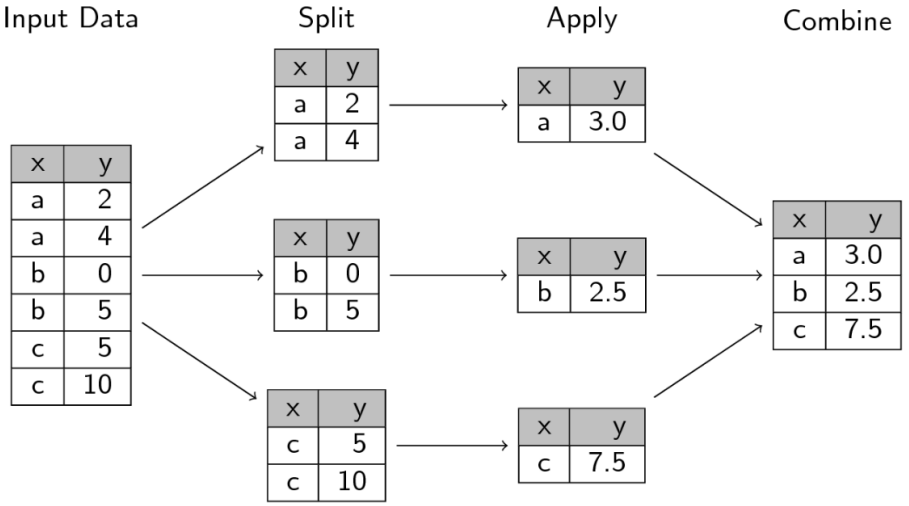
```
In [311]: df1 = pd.DataFrame(
   .....:     {"one": [2, 1, 1, 1], "two": [1, 3, 2, 4], "three": [5, 4, 3, 2]}
   .....: )
   .....:

In [312]: df1.sort_values(by="two")
Out[312]:
   one  two  three
0    2    1      5
2    1    2      3
1    1    3      4
3    1    4      2
```

```
In [313]: df1[["one", "two", "three"]].sort_values(by=["one", "two"])
Out[313]:
   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

Data grouping processing (Group by)

— Data grouping generally includes the following three steps of processing tasks that are often connected in series (split-apply-combine):

- Split : Divide data into groups based on certain conditions
- Apply : Apply certain methods independently to each data group, usually:
  - Aggregation : Calculate the summary statistics of each group: such as the average value of each group, the number of each group
  - Transformation : Perform some group-specific calculations and return an index-like object, such as standardizing data or filling missing data based on this group of data
  - Filtration : Determine whether to retain a specific data group based on the calculated value of the group data
- Combine : combine the results into a new data structure



Split data into groups

— Using DataFrame groupby method for grouping data

- The supplied string (list) specifies the columns or rows on which to group. Generally, classification is based on columns (values in columns). Column values are generally categorical data (gender, level, nationality, discrete data), which is not suitable for columns whose values are continuous data (height, temperature, etc.)

	A	B	C	D
1	bar	one	-1.162737	-0.073989
3	bar	three	0.821402	1.324895
5	bar	two	-0.320695	-0.762205
0	foo	one	0.180786	-1.258844
2	foo	two	-0.315659	-0.283271
4	foo	two	0.464645	1.165404
6	foo	one	-0.908225	-0.274201
7	foo	three	-0.238296	1.027909

```
df_group.groupby("A").count()
```

A	B	C	D
bar	3	3	3
foo	5	5	5

```
df_group.groupby(["A","B"]).count()
```

A	B	C	D
bar	one	1	1
	three	1	1
	two	1	1
foo	one	2	2
	three	1	1
	two	2	2

Groupby common parameters and attributes

- The first list type parameter is a name based on a column or row. You can use by=[LIST_OF_NAME] to have the same effect.
- By default, if there are missing characters in missing rows or columns, they will be automatically deleted. You can set dropna =False to retain the missing characters as the grouping parameter.
- By default, the grouping key will be used as the Index of the result . You can set as_index =False to make it a new column with the column name unchanged (consistent with the original data format)
- The ***groups*** property of the object returned by groupby can provide grouping result information.
- Grouping results can be iterated based on the ***groups*** attribute information.
- The ***get_group*** method of the groupby return object can obtain specific group data from the grouping results.

```
# In order to allow NaN in keys, set ``dropna`` to False
In [31]: df_dropna.groupby(by=["b"], dropna=False).sum()
Out[31]:
      a  c
b
1.0   2  3
2.0   2  5
NaN   1  4
```

```
df_group.groupby(["A"]).groups
```

```
{'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}
```

```
df_group.groupby(["A"]).get_group("bar")
```

	A	B	C	D
1	bar	one	-1.162737	-0.073989
3	bar	three	0.821402	1.324895
5	bar	two	-0.320695	-0.762205

Aggregate grouped data

- The grouped data can be aggregated using aggregation methods to aggregate multiple pieces of data in each group into one value (count, total, average, etc.)
- When as_index=False is not set during grouping, the aggregation result will use the grouping name as the new index value; otherwise, the new index will be sorted by 0...N – 1
- As a general aggregation method, the aggregate (agg is its alias) method can introduce other methods that can implement aggregation operations as input to implement customized aggregation operations.

	a	b	c
0	1	2.0	3
1	1	NaN	4
2	2	1.0	3
3	1	2.0	2

```
df_list_na.groupby(["a"]).sum()
```

a	b	c
1	4.0	9
2	1.0	3

```
In [68]: grouped = df.groupby("A")

In [69]: grouped[["C", "D"]].aggregate(np.sum)
Out[69]:
            C         D
A
bar  0.392940  1.732707
foo -1.796421  2.824590

In [70]: grouped = df.groupby(["A", "B"])

In [71]: grouped.aggregate(np.sum)
Out[71]:
                  C         D
A   B
bar one    0.254161  1.511763
    three  0.215897 -0.990582
    two   -0.077118  1.211526
foo one   -0.983776  1.614581
    three -0.862495  0.024580
    two    0.049851  1.185429
```

Function	Description
`mean()`	Compute mean of groups
`sum()`	Compute sum of group values
`size()`	Compute group sizes
`count()`	Compute count of group
`std()`	Standard deviation of groups
`var()`	Compute variance of groups
`sem()`	Standard error of the mean of groups
`describe()`	Generates descriptive statistics
`first()`	Compute first of group values
`last()`	Compute last of group values
`nth()`	Take nth value, or a subset if n is a list
`min()`	Compute min of group values
`max()`	Compute max of group values

Apply multiple aggregate functions at once

- Provides the aggregate(agg) method with a set of aggregate functions that can be applied to grouped data simultaneously
- If you want to apply different aggregate functions to different columns in grouped data, you need to provide agg with a mapping definition in dict form.
- The column names of the output results can be modified through the concatenated rename method.

```
In [82]: grouped = df.groupby("A")

In [83]: grouped["C"].agg([np.sum, np.mean, np.std])
Out[83]:
            sum      mean       std
A
bar    0.392940  0.130980  0.181231
foo   -1.796421 -0.359284  0.912265
```

```
In [85]: (
   ....:     grouped["C"]
   ....:     .agg([np.sum, np.mean, np.std])
   ....:     .rename(columns={"sum": "foo", "mean": "bar", "std": "baz"})
   ....: )
   ....:
Out[85]:
            foo       bar       baz
A
bar    0.392940  0.130980  0.181231
foo   -1.796421 -0.359284  0.912265
```

```
In [95]: grouped.agg({"C": np.sum, "D": lambda x: np.std(x, ddof=1)})
Out[95]:
            C         D
A
bar    0.392940  1.366330
foo   -1.796421  0.884785
```

```
In [96]: grouped.agg({"C": "sum", "D": "std"})
Out[96]:
                C         D
A
bar    0.392940  1.366330
foo   -1.796421  0.884785
```

Transform the grouped data (Transformation)

- Different from the purpose of the aggregation operation (multiple pieces of data become one piece of data), the conversion does not change the size of the original data, but changes the value of the data, based on the grouping result as a unit, such as: replacing the value with the average value of the group.
- Method to realize:
  - apply (func) method: Apply a specific method to the grouped data (in groups), and then combine the results into a new DataFrame .
  - transform(func) method: faster than ***apply*** when performing built-in aggregation operations, but supports func There are corresponding restrictions.

```
df = pd.DataFrame({'A': 'a a b'.split(),
                   'B': [1,2,3],
                   'C': [4,6,5]})
g1 = df.groupby('A', group_keys=False)
g2 = df.groupby('A', group_keys=True)
g1[['B', 'C']].apply(lambda x: x / x.sum())
```

	B	C
0	0.333333	0.4
1	0.666667	0.6
2	1.000000	1.0

```
g2[['B', 'C']].apply(lambda x: x / x.sum())
```

		B	C
A			
a	0	0.333333	0.4
	1	0.666667	0.6
b	2	1.000000	1.0

Filter grouped data (Filtration)

- The filtering operation is to remove grouping elements that do not meet the conditions and then return a new DataFrame object
- Method to realize:
  - filter(func) method: If the elements in the group do not satisfy func specified boolean criteria, they are filtered.
  - func passed to filter Take the entire grouped data as input, perform conditional judgment, and output True or False
  - The dropna =False parameter can mark the filtered group value as NaN in the output results.
  - If the data in the group contains multiple columns, you need to specify the columns to which the judgment condition applies.

```
dff = pd.DataFrame({"A": np.arange(8), "B": list("aabbbbcc")})
```

```
dff.groupby("B").filter(lambda x: len(x) > 2)
```

	A	B
2	2	b
3	3	b
4	4	b
5	5	b

```
dff.groupby("B").filter(lambda x: len(x) > 2, dropna=False)
```

	A	B
0	NaN	NaN
1	NaN	NaN
2	2.0	b
3	3.0	b
4	4.0	b
5	5.0	b
6	NaN	NaN
7	NaN	NaN

```
dff["C"] = np.arange(8) # Add a new column
dff.groupby("B").filter(lambda x: len(x["C"]) > 2)
```

	A	B	C
2	2	b	2
3	3	b	3
4	4	b	4
5	5	b	5

Time series data in Pandas

- Any value observed or measured at multiple points in time can form a time series data, which is also widely used in the field of data analysis.
- Time series data in pandas includes
  - Timestamp : represents a specific point in time, similar to datetime.dateime of the standard library .
  - Time span (Period): represents a time and the frequency (freq) and number of repetitions
  - Time delta (TimeDelta): represents a fixed length of time, such as: 1 month, 1 day, etc.
  - Time offset (Offset): similar to time delta, but can control the length of time in a more fine-grained manner and can be used as a unit of frequency
- Time series data is mostly used as the index of pandas data objects, and of course it also supports operations as data values.

```
In [19]: pd.Series(range(3), index=pd.date_range("2000", freq="D", periods=3))
Out[19]:
2000-01-01    0
2000-01-02    1
2000-01-03    2
Freq: D, dtype: int64
```

```
In [20]: pd.Series(pd.date_range("2000", freq="D", periods=3))
Out[20]:
0   2000-01-01
1   2000-01-02
2   2000-01-03
dtype: datetime64[ns]
```

Date and time in Python standard library

- The datetime module and the classes provided in the Python standard library are the basis for pandas to implement time series data and require import.
- Basic type:
  - date object represents the date in the calendar (year, month, day)
  - The time object represents the (local) time of day, independent of any specific date, and can be adjusted through the tzinfo object.
  - A datetime object is an object that contains all the information from date objects and time objects.
  - A timedelta object represents a duration, that is, the difference between two dates or times.
  - tzinfo is the basic type for storing time zone information

```
from datetime import datetime,timedelta
now = datetime.now()
now.year,now.month,now.day,now.hour,now.minute
```

```
(2023, 2, 7, 9, 36)
```

```
now + timedelta(days=12) # Offset 12 days
```

```
datetime.datetime(2023, 2, 19, 9, 36, 40, 42378)
```

```
now + timedelta(hours=12) # Offset 12 hours
```

```
datetime.datetime(2023, 2, 7, 21, 36, 40, 42378)
```

```
old_time=datetime(2022,12,31)
```

```
now-old_time
```

```
datetime.timedelta(days=38, seconds=34600, microseconds=42378)
```

```
str(old_time)
```

```
'2022-12-31 00:00:00'
```

```
old_time.strftime('%Y-%m-%d')
```

```
'2022-12-31'
```

Creation and transformation of time series data

- Creation:
  - Timestamp:

```
In [28]: pd.Timestamp(datetime.datetime(2012, 5, 1))
Out[28]: Timestamp('2012-05-01 00:00:00')

In [29]: pd.Timestamp("2012-05-01")
Out[29]: Timestamp('2012-05-01 00:00:00')

In [30]: pd.Timestamp(2012, 5, 1)
Out[30]: Timestamp('2012-05-01 00:00:00')
```

  - Time span:

```
In [31]: pd.Period("2011-01")
Out[31]: Period('2011-01', 'M')

In [32]: pd.Period("2012-05", freq="D")
Out[32]: Period('2012-05-01', 'D')
```

  - Time delta:

```
pd.Timedelta("1 days")
```

```
Timedelta('1 days 02:00:00')
```

```
pd.Timedelta("1 days 2 hours")
```

```
Timedelta('1 days 02:00:00')
```

```
pd.Timedelta(days=1, seconds=1)
```

```
Timedelta('1 days 00:00:01')
```

```
pd.Timedelta(timedelta(days=1, seconds=1))
```

```
Timedelta('1 days 00:00:01')
```

- Conversion:
  - Timestamp:
    - Mainly based on pandas.to_datetime nethods to implement conversion of multiple input formats and options

```
pd.to_datetime(pd.Series(["Jul 31, 2009", "2010-01-10", None])) # return a series
```

```
0   2009-07-31
1   2010-01-10
2          NaT
dtype: datetime64[ns]
```

```
pd.to_datetime(["2005/11/23", "2010.12.31"]) # return a DatetimeIndex
```

```
DatetimeIndex(['2005-11-23', '2010-12-31'], dtype='datetime64[ns]', freq=None)
```

```
pd.to_datetime("2010/11/12", format="%Y/%m/%d") # return a Timestamp with format argument
```

```
Timestamp('2010-11-12 00:00:00')
```

```
pd.to_datetime([1, 2, 3], unit="D", origin=pd.Timestamp("1960-01-01")) # based on origin and unit arguments
```

```
DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype='datetime64[ns]', freq=None)
```

Index based on time series data - DatetimeIndex

- To generate an index with timestamp you can use DatetimeIndex or Index constructor and pass in a list of datetime objects
- A more efficient way is to use date_range() and bdate_range() to define based on parameters such as start and end time, frequency, span (period) number, etc.

```
In [82]: pd.date_range(start, end, freq="BM")
Out[82]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-30'],
              dtype='datetime64[ns]', freq='BM')

In [83]: pd.date_range(start, end, freq="W")
Out[83]:
DatetimeIndex(['2011-01-02', '2011-01-09', '2011-01-16', '2011-01-23',
               '2011-01-30', '2011-02-06', '2011-02-13', '2011-02-20',
```

```
In [84]: pd.bdate_range(end=end, periods=20)
Out[84]:
DatetimeIndex(['2011-12-05', '2011-12-06', '2011-12-07', '2011-12-08',
               '2011-12-09', '2011-12-12', '2011-12-13', '2011-12-14',
               '2011-12-15', '2011-12-16', '2011-12-19', '2011-12-20',
               '2011-12-21', '2011-12-22', '2011-12-23', '2011-12-26',
               '2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30'],
              dtype='datetime64[ns]', freq='B')

In [85]: pd.bdate_range(start=start, periods=20)
Out[85]:
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
```

Access data based on time series data index

- Data access is based on a string that can be parsed into date and time information. The string can be the complete timestamp information or partially include
  - 1/31/2011
  - 10/31/2011:12/31/2011
  - 2011
  - 2011-6
  - 2013-1:2013-2
  - 2013-1:2013-2-28
  - 2013-1:2013-2-28 00:00:00
- The truncate method defines the **unselected area** by setting the before and after parameters :

```
In [136]: rng2 = pd.date_range("2011-01-01", "2012-01-01", freq="W")

In [137]: ts2 = pd.Series(np.random.randn(len(rng2)), index=rng2)

In [138]: ts2.truncate(before="2011-11", after="2011-12")
Out[138]:
2011-11-06    0.437823
2011-11-13   -0.293083
2011-11-20   -0.059881
2011-11-27    1.252450
Freq: W-SUN, dtype: float64
```

Use case for time series data

— Create datetime column from string type data

```
In [5]: air_quality.head()
Out[5]:
    city country                  datetime location parameter  value   unit
0  Paris      FR  2019-06-21 00:00:00+00:00  FR04014       no2   20.0  µg/m³
1  Paris      FR  2019-06-20 23:00:00+00:00  FR04014       no2   21.8  µg/m³
2  Paris      FR  2019-06-20 22:00:00+00:00  FR04014       no2   26.5  µg/m³
3  Paris      FR  2019-06-20 21:00:00+00:00  FR04014       no2   24.9  µg/m³
4  Paris      FR  2019-06-20 20:00:00+00:00  FR04014       no2   21.4  µg/m³
```

```
In [7]: air_quality["datetime"] = pd.to_datetime(air_quality["datetime"])

In [8]: air_quality["datetime"]
Out[8]:
0      2019-06-21 00:00:00+00:00
1      2019-06-20 23:00:00+00:00
2      2019-06-20 22:00:00+00:00
3      2019-06-20 21:00:00+00:00
4      2019-06-20 20:00:00+00:00
```

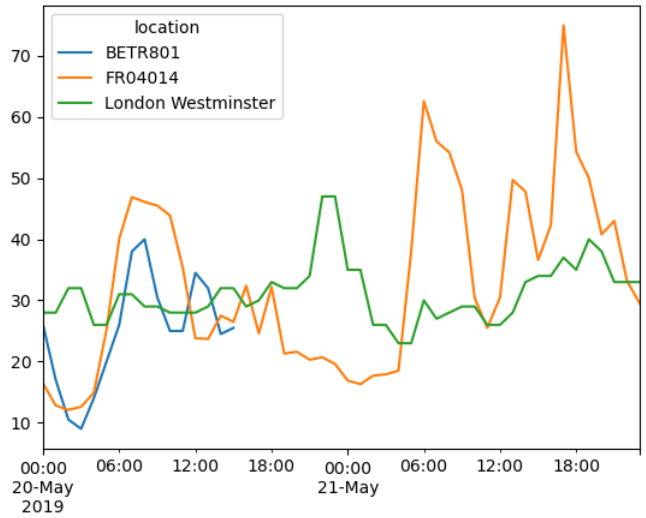
— Work with datetime type data: Using pandas.Timestamp for datetimes enables you to calculate with date information and make them comparable.

```
In [9]: air_quality["datetime"].min(), air_quality["datetime"].max()
Out[9]:
(Timestamp('2019-05-07 01:00:00+0000', tz='UTC'),
 Timestamp('2019-06-21 00:00:00+0000', tz='UTC'))
```

— Set date&time column as index

```
In [18]: no_2 = air_quality.pivot(index="datetime", columns="location", values="value")

In [19]: no_2.head()
Out[19]:
location                  BETR801  FR04014  London Westminster
datetime
2019-05-07 01:00:00+00:00     50.5     25.0                23.0
2019-05-07 02:00:00+00:00     45.0     27.7                19.0
2019-05-07 03:00:00+00:00      NaN     50.4                19.0
2019-05-07 04:00:00+00:00      NaN     61.9                16.0
2019-05-07 05:00:00+00:00      NaN     72.4                 NaN
```

```
In [21]: no_2["2019-05-20":"2019-05-21"].plot();
```



Resample a time series to another frequency

- A very powerful method on time series data with a datetime index, is the ability to resample() time series to another frequency (e.g., converting secondly data into 5-minutely data)
- The resample() method is similar to a groupby operation:
  - it provides a time-based grouping, by using a string (e.g. M, 5H,...) that defines the target frequency
  - it requires an aggregation function such as mean, max,...

```
In [18]: no_2 = air_quality.pivot(index="datetime", columns="location", values="value")

In [19]: no_2.head()
Out[19]:
location                  BETR801  FR04014  London Westminster
datetime
2019-05-07 01:00:00+00:00     50.5     25.0                23.0
2019-05-07 02:00:00+00:00     45.0     27.7                19.0
2019-05-07 03:00:00+00:00      NaN     50.4                19.0
2019-05-07 04:00:00+00:00      NaN     61.9                16.0
2019-05-07 05:00:00+00:00      NaN     72.4                 NaN
```

```
In [22]: monthly_max = no_2.resample("ME").max()

In [23]: monthly_max
Out[23]:
location                  BETR801  FR04014  London Westminster
datetime
2019-05-31 00:00:00+00:00     74.5     97.0                97.0
2019-06-30 00:00:00+00:00     52.5     84.7                52.0
```

Python - based data visualization

Data visualization overview

- Data and information visualization is an interdisciplinary field that deals with the graphical representation of data and information. This is a particularly effective way of communicating when there is a lot of data or information (such as a time series).. It makes complex data easier to access, understand, and use, as well as simplify it.
- To convey information clearly and effectively, data visualization uses statistical graphs, charts, information graphics, and other tools. Numerical data can be encoded using points, lines, or bars to visually convey quantitative information.
- Commonly used graphics:
  - Line graph
  - Scatter plot
  - Bar chart
  - Pie chart
  - Histogram

Python data visualization tools

- Matplotlib is the most widely used visualization library. It provides fine control over plotting, making it a versatile software package with a variety of graphics types and configuration options
- seaborn is a visualization library that makes Matplotlib plotting practical. It abstracts away the complexity of Matplotlib and provides intuitive syntax and renderable results out of the box.
- Bokeh is a visualization library that provides a structured way to create charts and supports server-side rendering of interactive visualizations in web applications.
- Altair is a visualization library that provides a unique declarative syntax for interactive plot creation. It relies on the VEGA-LITE syntax specification and allows you to compose diagrams from graphical units and combine them in a modular way.
- Plotly is an open source data visualization library that is a great tool for building interactive, business-focused visualizations and dashboards.

Matplotlib overview

- matplotlib is a plotting library for the Python language and its numerical computing library NumPy . It provides an object-oriented API for embedding drawing into applications

```
import matplotlib.pyplot as plt
```

- Can be used to create static, animated and interactive visualizations in Python, include:
  - Create high-quality graphical output
  - Create interactive graphics that can be zoomed, panned, and updated
  - Customize visual style and layout
  - Export to multiple file formats
  - Embed Jupyter Notebook and graphical user interface
  - Use a rich set of third-party packages built on
- Data analysis tasks are often achieved based on data visualization.
  - Understanding of data
  - Check the execution effect

Matplotlib basic concepts

— Matplotlib plots data on **Figures**. Each Figure can contain one or more **axes**, and an area where the position of the plot point can be determined based on x-y coordinates.

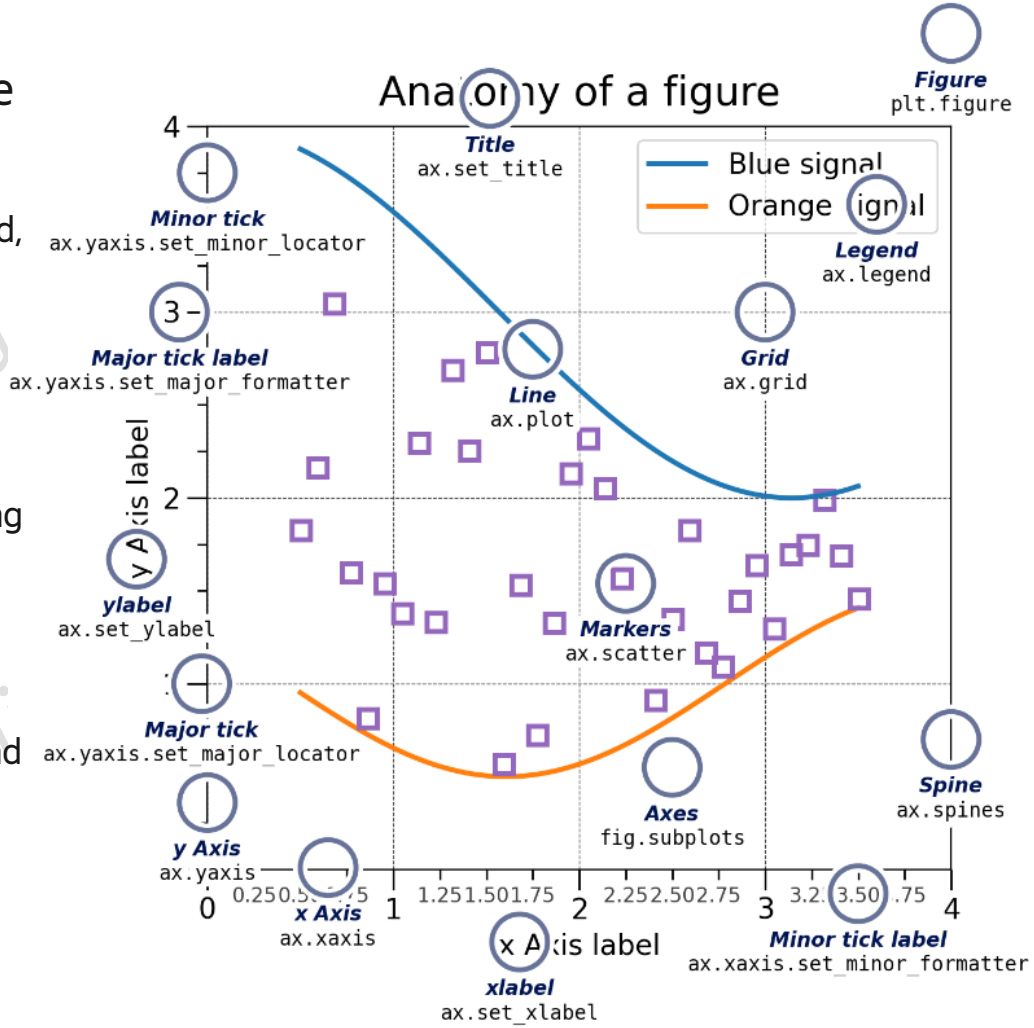
- Figure : A Figure contains its Axes and a set of special " Artist " objects (title, legend, color bar, etc.).

  fig = plt.figure () # an empty figure with no Axes

  fig, ax = plt.subplots () # a figure with a single Axes

  fig, axs = plt.subplots (2, 2) # a figure with a 2x2 grid of Axes

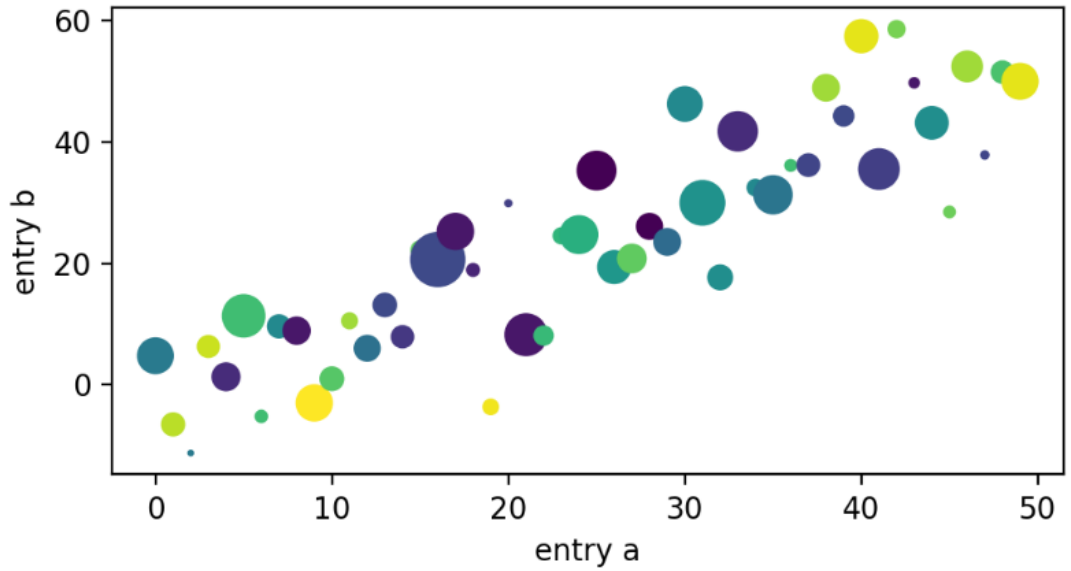
- Axes : An Axes is an Artist attached to a Figure , which contains an area for plotting data, and usually includes two (or three in the case of 3D) Axis objects, which provide scales and scale labels for the data in the axes. information
  - Axes object provides the main drawing methods, such as the plot method
- Axis : Generate ticks (marks on the axis) and tick labels (strings marking ticks), and set their scale and limits. The position of the tick is determined by the Locator object, and the tick label string is formatted by the Formatter . The right combination of Locator and Formatter provides great control over tick positions and labels.
- Artist : Basically, everything visible on the Figure is an Artist (even Figure , Axes , and Axis objects).



Plot data supported by Matplotlib

- Plotting method expecting numpy.array or numpy.ma.masked_array as input, or object (list , tuple)can be passed to numpy.asarray
- Generally, drawing methods require the input of data for different Axis respectively . Some methods provide ***data*** parameters to directly input multi-dimensional data to it, and then reference it through the corresponding string.

```
np.random.seed(19680801)  # seed the random number generator.
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.scatter('a', 'b', c='c', s='d', data=data)
ax.set_xlabel('entry a')
ax.set_ylabel('entry b');
```



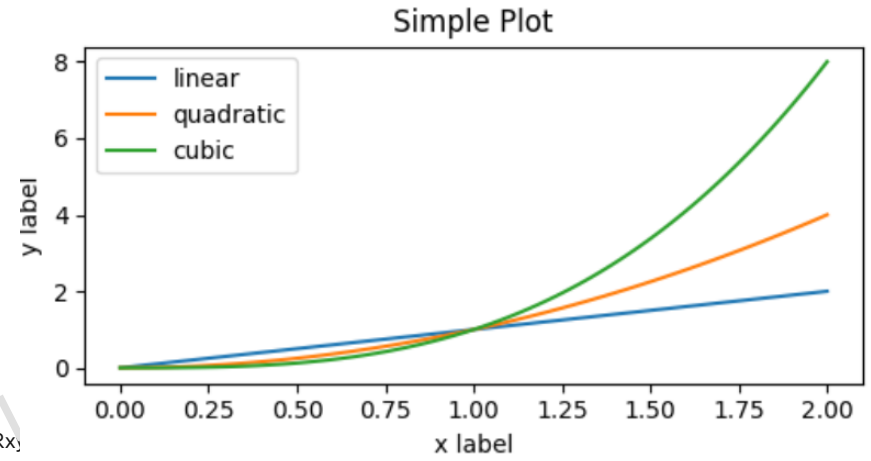
Two styles of plotting

— Drawing based on the Matplotlib Python API supports two writing methods:

- Explicitly create Figures and Axes, then call their methods ("Object-Oriented (OO) style"). (Left)
- Rely on ***pyplot*** to implicitly create and manage Figures and Axes and use pyplot functions for plotting. (Right)

```python
x = np.linspace(0, 2, 100)  # Sample data.

# Note that even in the OO-style, we use `.pyplot.figure` to create the Figure.
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.plot(x, x, label='linear')  # Plot some data on the axes.
ax.plot(x, x**2, label='quadratic')  # Plot more data on the axes...
ax.plot(x, x**3, label='cubic')  # ... and some more.
ax.set_xlabel('x label')  # Add an x-label to the axes.
ax.set_ylabel('y label')  # Add a y-label to the axes.
ax.set_title("Simple Plot")  # Add a title to the axes.
ax.legend();  # Add a legend.
```

```python
x = np.linspace(0, 2, 100)  # Sample data.

plt.figure(figsize=(5, 2.7), layout='constrained')
plt.plot(x, x, label='linear')  # Plot some data on the (implicit) axes.
plt.plot(x, x**2, label='quadratic')  # etc.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend();
```

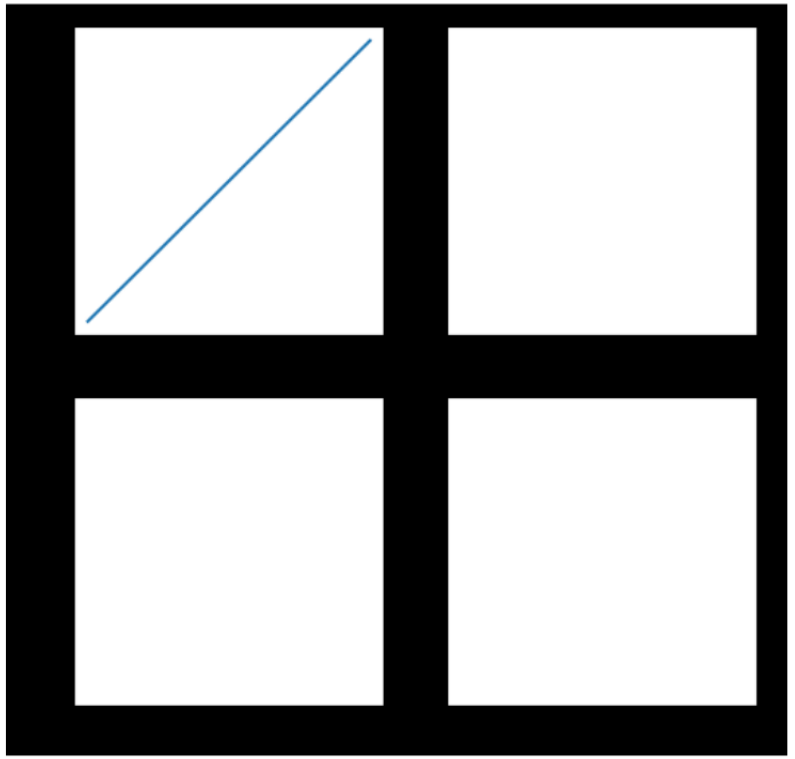


Basic tasks: Create Figures and Axes

- Using pyplot (plt) subplots method can create Figure and Axes at the same time and perform initial settings:
  - nrows, ncols set the subplot grid, you can create multiple subplots in a Figure, and then select based on the returned Axes object, first row and then column.
  - Set the size of Figure by figsize = (x,y) , in feet, default: 6.4*4.8
  - layout : control layout, ***constrained , tight*** , default is None
  - facecolor and edgecolor : control the color of the background and border
  - *The drawn picture must be output using the plt.show() method
- Using pyplot (plt) methods:
  - figure create Figure
  - subplot specifies the corresponding subplot (Axes): 2, 2, 1 or 221
  - plot and other methods for drawing

```
plt.figure(figsize=(6,6),facecolor="black",edgecolor="red")
plt.subplot(2,2,1)
plt.plot(x,y)
plt.subplot(2,2,2)
plt.subplot(2,2,3)
plt.subplot(2,2,4)
```

```
fig,ax= plt.subplots(nrows=2,ncols=2,figsize=(6,6),facecolor="black",edgecolor="red")
x = range(100)
y = range(100)
ax[0][0].plot(x,y)
plt.show()
```

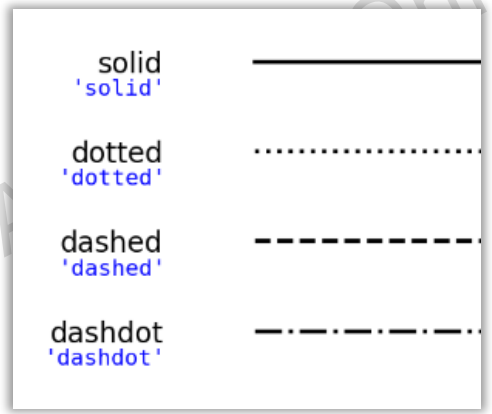


Set drawing style

- Most drawing methods support setting the Artist object to adjust the drawing style, such as: color, line width, line style, etc.
- Color settings:
  - Supported formats
    - Single character shorthand notation or string for some basic colors.
    - RGB : (0.1, 0.2, 0.5) or #0f0f0f'
    - X11/CSS4
    - xkcd
  - There are some drawing methods that support setting multiple colors. For example, scatter has facecolor and edgecolor that can be set.
  - A color's alpha value specifies its transparency, where 0 means fully transparent and 1 means fully opaque.

```
data.plot.bar(ax=axes[0], color='k', alpha=0.7)
```

- Line weight, line style and marker shape:
  - linestyle: solid, dotted, dashed, dashdot, etc.
  - linewidth: set line width
  - marker: set marker shape

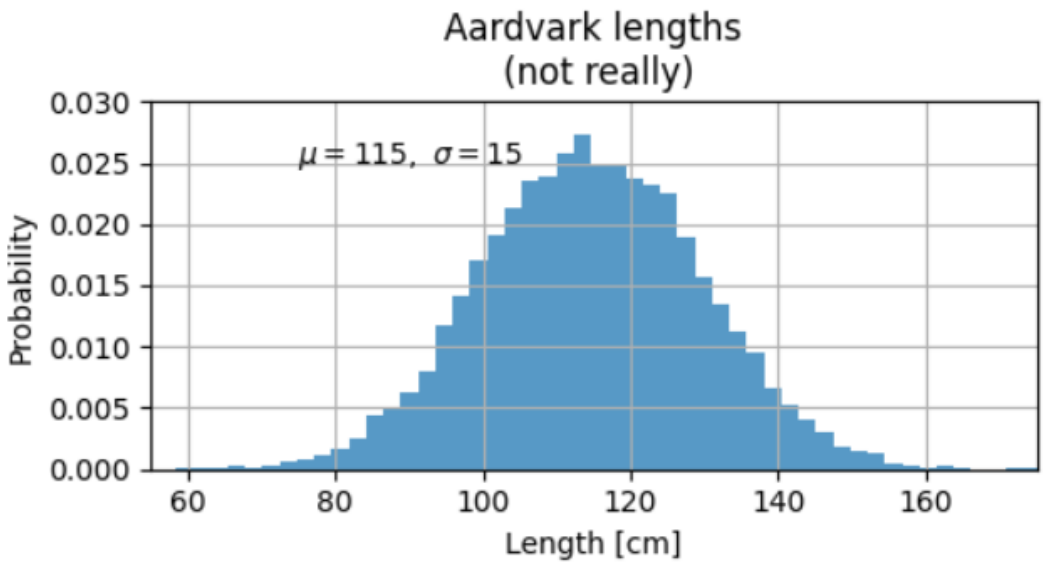


marker	symbol	description
"."	•	point
","	·	pixel
"o"	●	circle
"v"	▼	triangle_down
"^"	▲	triangle_up
"<"	◀	triangle_left
">"	▶	triangle_right

Set plot labels

- The set_xlabel and set_ylabel methods of the Axes object are used to add text at a specified location.
- You can also use the text method to add text directly to the drawing with position.

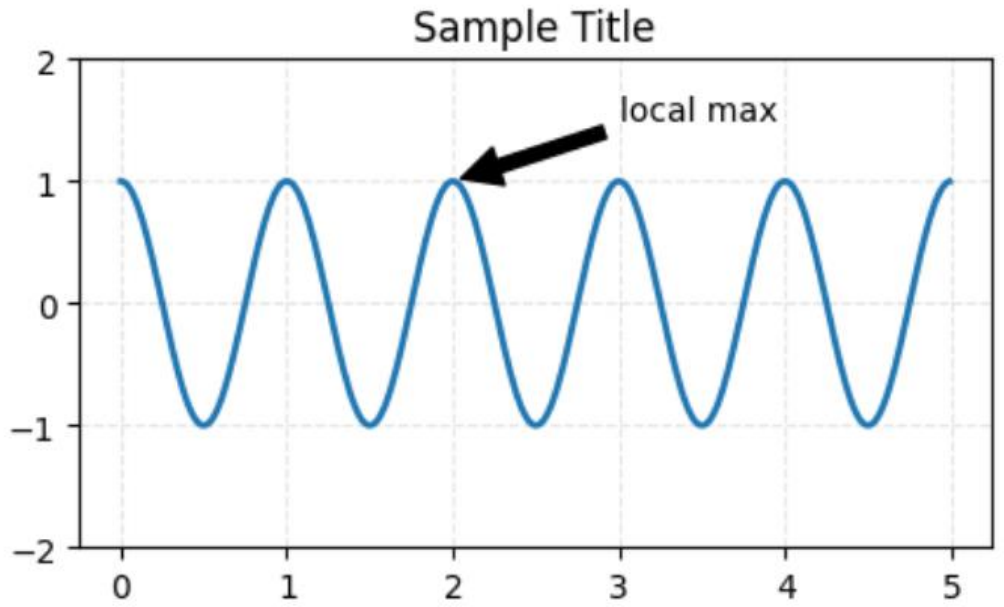
```
mu, sigma = 115, 15
x = mu + sigma * np.random.randn(10000)
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
# the histogram of the data
n, bins, patches = ax.hist(x, 50, density=True, facecolor='C0', alpha=0.75)

ax.set_xlabel('Length [cm]')
ax.set_ylabel('Probability')
ax.set_title('Aardvark lengths\n (not really)')
ax.text(75, .025, r'$\mu=115,\ \sigma=15$')
ax.axis([55, 175, 0, 0.03])
ax.grid(True)
```



Set notes and titles for drawings

- The annotate method of Axes can add annotation information by connecting the arrow pointing to xy to a piece of text at xytext.
- The set_title method of Axes can add a title to the image.
- The grid method of Axes can control the grid display style of the background

```
fig, ax = plt.subplots(figsize=(5, 2.7))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05))
ax.set_title("Sample Title")
ax.grid(True, alpha=0.3, linestyle="--")
ax.set_ylim(-2, 2);
```



Set plot legend

— By adding a legend, you can identify lines or markers on the graph through the legend method of Axes :

- When legend is given without any additional arguments, the elements to be added to the legend are automatically determined. The element information comes from the label information of the Artist object

```
ax.plot([1, 2, 3], label='Inline label')
ax.legend()
```

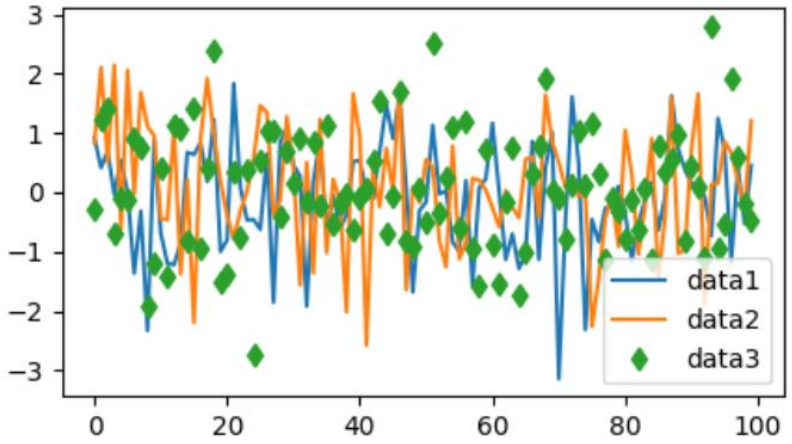
- To fully control which Artists have legend entries, you can pass a list of Artists individually , followed by an iterable of legend labels:

```
ax.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

- Similar to the previous method, except that the legend method only sets the Artist list, and the legend label comes from the Artist 's label setting:

```
line1, = ax.plot([1, 2, 3], label='label1')
line2, = ax.plot([1, 2, 3], label='label2')
ax.legend(handles=[line1, line2])
```

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(np.arange(len(data1)), data1, label='data1')
ax.plot(np.arange(len(data2)), data2, label='data2')
ax.plot(np.arange(len(data3)), data3, 'd', label='data3')
ax.legend();
```

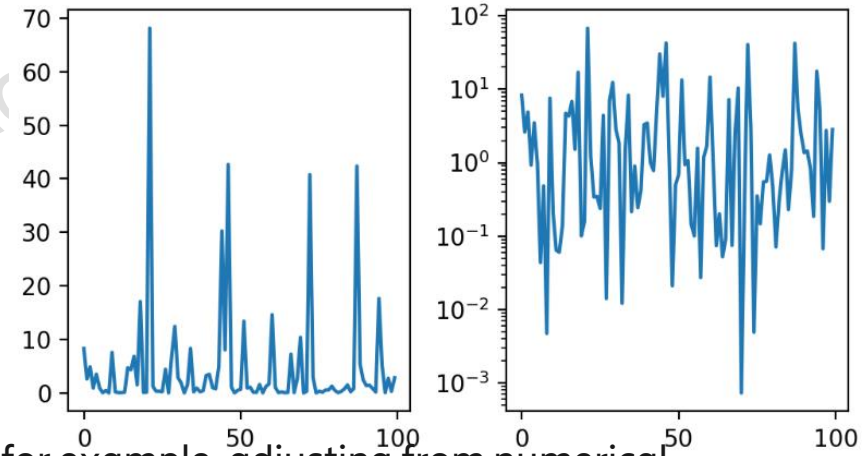


Axis scale

— Axes generally have multiple Axis objects to represent different dimensions (2D or 3D images), and each Axis has methods to control its scale settings.

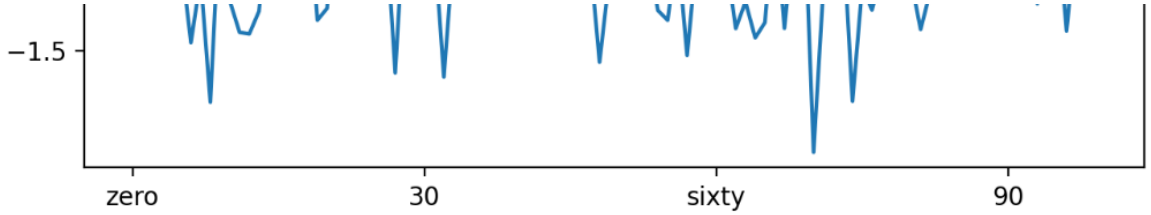
- sex_xscale, sex_yscale are used to set the precision settings of the scale. The default is linear (liner) precision, but sometimes it may be adjusted to exponential precision (log-scale) due to data values.

```
fig, axs = plt.subplots(1, 2, figsize=(5, 2.7), layout='constrained')
xdata = np.arange(len(data1))  # make an ordinal for this
data = 10**data1
axs[0].plot(xdata, data)

axs[1].set_yscale('log')
axs[1].plot(xdata, data)
```



- sex_xticks, sex_yticks are used to set the display value of the scale, for example, adjusting from numerical to text:

axs [1] .set_xticks (np.arange (0, 100, 30), ['zero', '30', 'sixty', '90'])

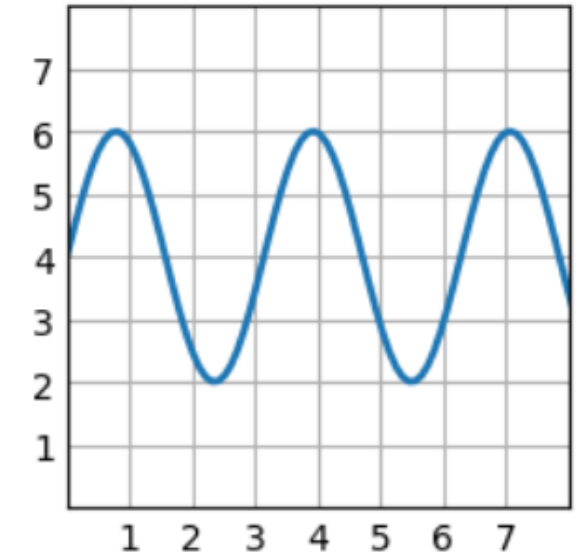


Typical data visualization graphics – Line graph

- Plot: The most basic type of plot, suitable for showing the changing trend of one variable (x) with another variable (y)
- Suitable for continuous data
- Parameters:

```
plot(x, y)        # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

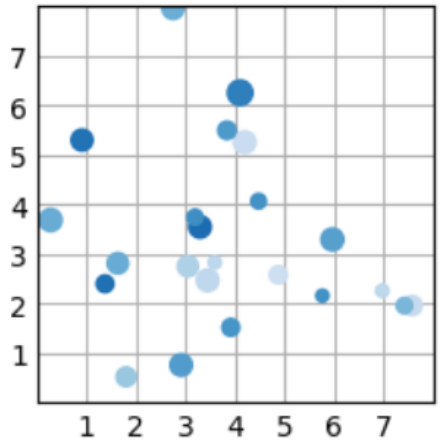
```
plt.style.use('_mpl-gallery')

# make data
x = np.linspace(0, 10, 100)
y = 4 + 2 * np.sin(2 * x)

# plot
fig, ax = plt.subplots()

ax.plot(x, y, linewidth=2.0)
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.show()
```



Typical data visualization graphics – scatter plot

- scatter : Scatter plots use data points to plot two measures anywhere along the scale.
- Scatter plots are useful for exploring the relationships between different sets of data.
- Commonly used parameters:
  - The first two parameters are x and y data
    - s controls the size of the marker point
    - c controls the color of the marker point

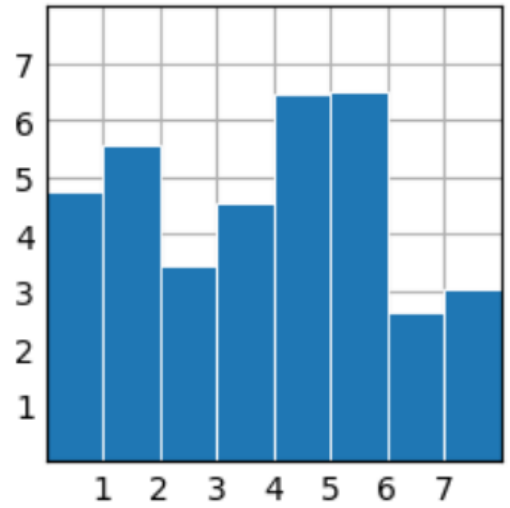
```
plt.style.use('_mpl-gallery')

# make the data
np.random.seed(3)
x = 4 + np.random.normal(0, 2, 24)
y = 4 + np.random.normal(0, 2, len(x))
# size and color:
sizes = np.random.uniform(15, 80, len(x))
colors = np.random.uniform(15, 80, len(x))

# plot
fig, ax = plt.subplots()

ax.scatter(x, y, s=sizes, c=colors, vmin=0, vmax=100)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```



Typical data visualization graphics – bar chart

- bar or barh: Bar charts use vertical data markers to compare individual values.
- Bar charts are useful for comparing discrete data or showing trends over time.
- Commonly used parameters:
  - The first two parameters, x and y (height) data (barh no need to adjust the order)
    - width : bar width, default 0.8

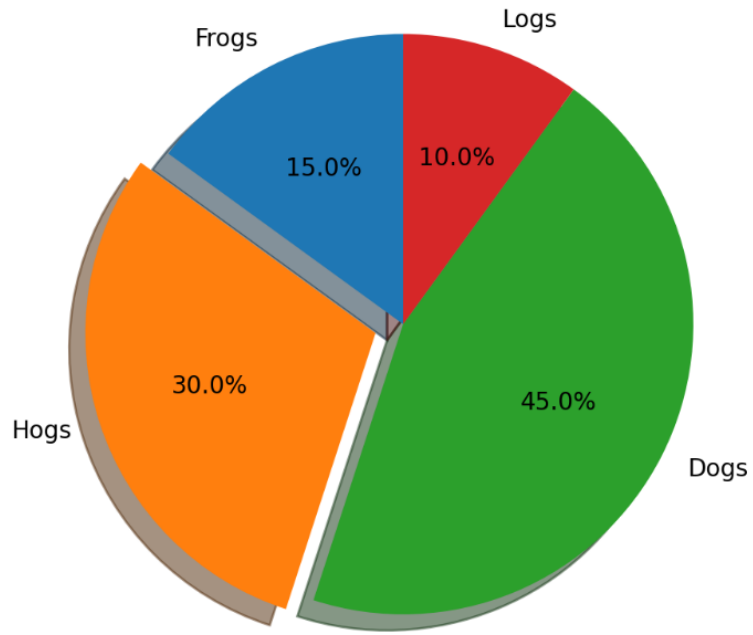
```
# make data:
np.random.seed(3)
x = 0.5 + np.arange(8)
y = np.random.uniform(2, 7, len(x))

# plot
fig, ax = plt.subplots()

ax.bar(x, y, width=1, edgecolor="white", linewidth=0.7)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```



Typical data visualization graphics – pie chart

- Pie charts are useful for highlighting proportions.
- Pie charts use sectors of a circle to show the relationship of parts to a whole.
- Parameters:
  - The first parameter is a one-dimensional data, which is the core data of the pie chart.
  - labels : Set the data label list
  - radius : Radius length, default is 1
  - autopct : The description text in each piece of pie generally displays actual data in a manner similar to %.1f , and displays hundred percent data in a manner similar to
  - explode : Set the highlighted pie slice list
  - startangle : Set the angle at which the first pie is displayed. The default is the positive direction of X axis.

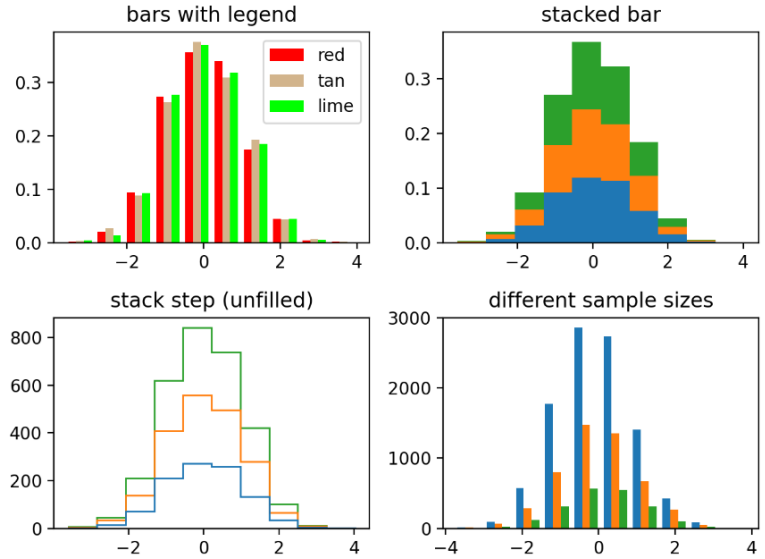
```python
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```



Typical data visualization graphics – histogram (hist)

- hist: Histogram is a graph used to represent **the frequency distribution of several data points of a variable** .
- Histograms typically classify data into various "bins" or "range groups" and count the number of data points that belong to each bin.
- Mainly used for analysis: frequency distribution, data symmetry, changes over time and other data attributes.
- This method uses numpy.histogram to bin the data in x, counts the number of values in each bin, and then draws the distribution.
- Parameters:
  - The first parameter is array data, one or more dimensions
  - bins : The number of bins or the value range definition: [10,20,30,40]
  - color : Set the color of data in different dimensions
  - label : Set labels for data of different dimensions, used for legends
  - stack : Whether multidimensional data is displayed in stacks
  - histyle : display style, the default is bar , can be set to step , displayed in the form of step lines

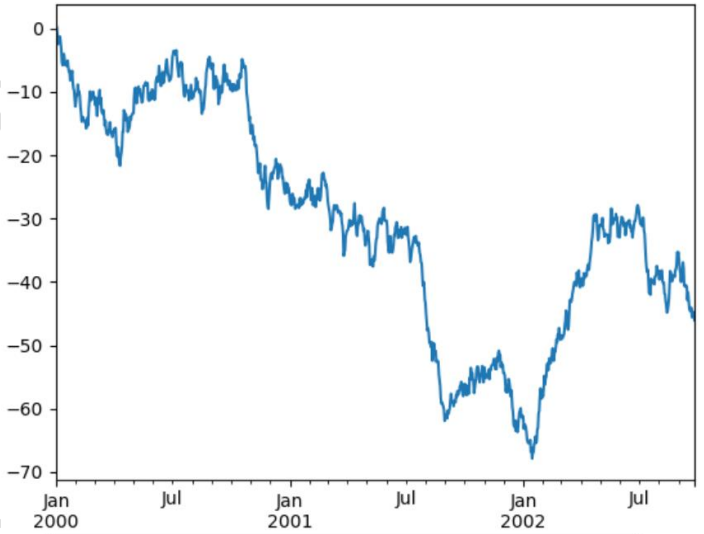
```
n_bins = 10
x = np.random.randn(1000, 3)

fig, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)

colors = ['red', 'tan', 'lime']
ax0.hist(x, n_bins, density=True, histtype='bar', color=colors, label=colors)
ax0.legend(prop={'size': 10})
ax0.set_title('bars with legend')

ax1.hist(x, n_bins, density=True, histtype='bar', stacked=True)
ax1.set_title('stacked bar')

ax2.hist(x, n_bins, histtype='step', stacked=True, fill=False)
ax2.set_title('stack step (unfilled)')

# Make a multiple-histogram of data-sets with different length.
x_multi = [np.random.randn(n) for n in [10000, 5000, 2000]]
ax3.hist(x_multi, n_bins, histtype='bar')
ax3.set_title('different sample sizes')
```



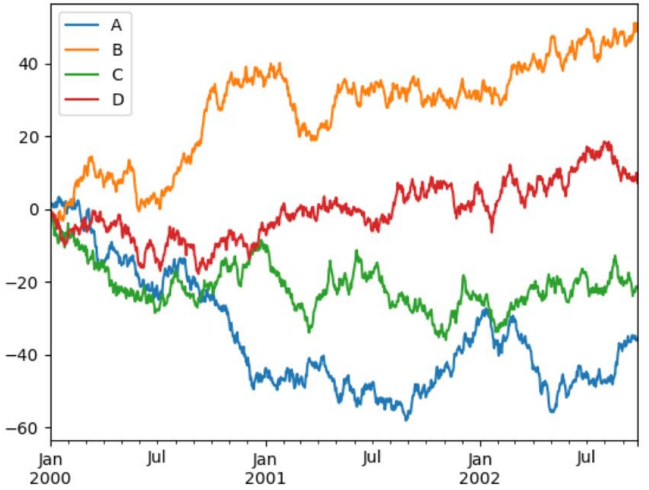
Drawing methods provided by pandas

- Pandas ' data objects (Series and DataFrame) have built-in support for data visualization, the default is: matplotlib and there are many other pandas- compatible libraries to choose from.
- The syntax of the plotting method is basically compatible with matplotlib :

```
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))

ts = ts.cumsum()

ts.plot();
```




```
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))
ts = ts.cumsum()
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list("ABCD"))
df = df.cumsum()
df.plot()
plt.show()
```

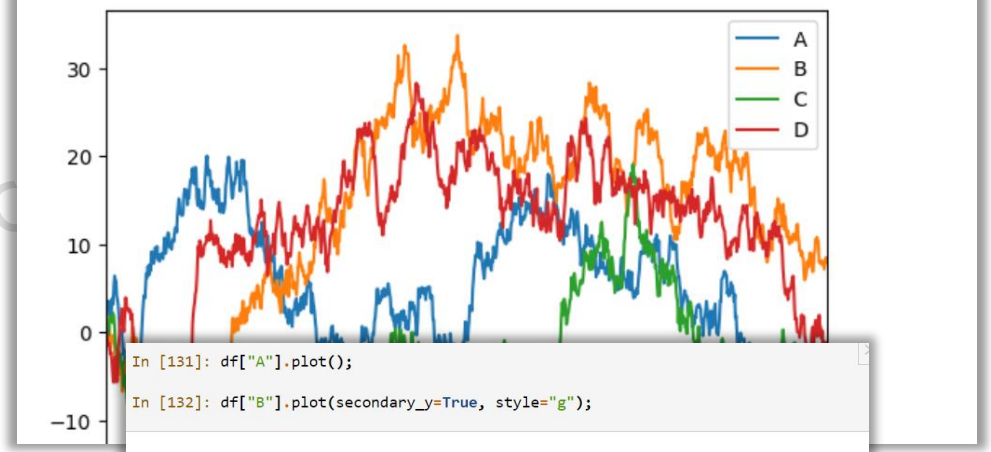


DataFrame.plot commonly used parameters

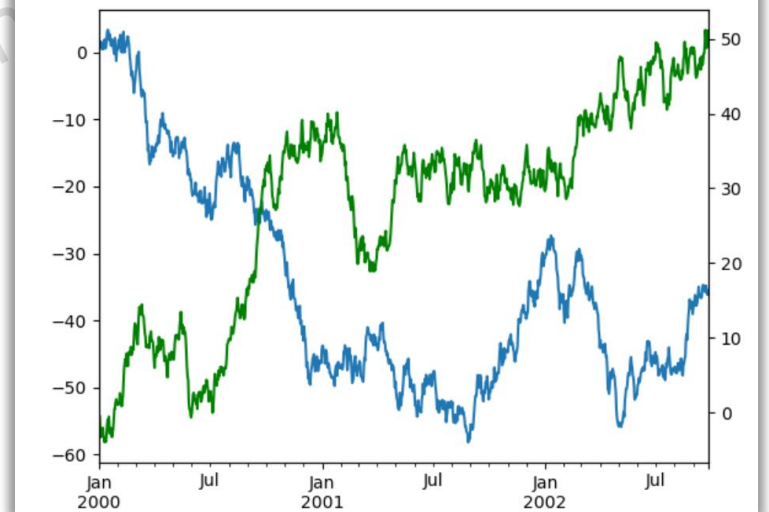
- Specify the type of graph by providing the ***kind*** parameter to the plot method; or directly use the sub-method corresponding to the graph to create the corresponding graph: ***DataFrame.plot.<kind>***
- Currently supported types: area, barh, density hist, line, scatter, bar ,box,hexbin,kde,pie.
- Graphics created based on DataFrame data with multiple columns will be plotted based on Index as the x-axis and column values as the y-axis.
- The ***stack*** parameter can control whether graphics corresponding to multiple columns are stacked (default False)
- The legend will be automatically embedded and can be turned off using legend=False
- You can set the xlabel and ylabel parameters to provide custom labels for the plot for the x and y axes. By default, pandas will select the index name for the xlabel and leave it blank for the ylabel.
- Additional columns of axes can be displayed on the right Y axis secondary_y=True

```
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))
ts = ts.cumsum()
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list("ABCD"))
df = df.cumsum()
df.plot()
```

```
<AxesSubplot: >
```



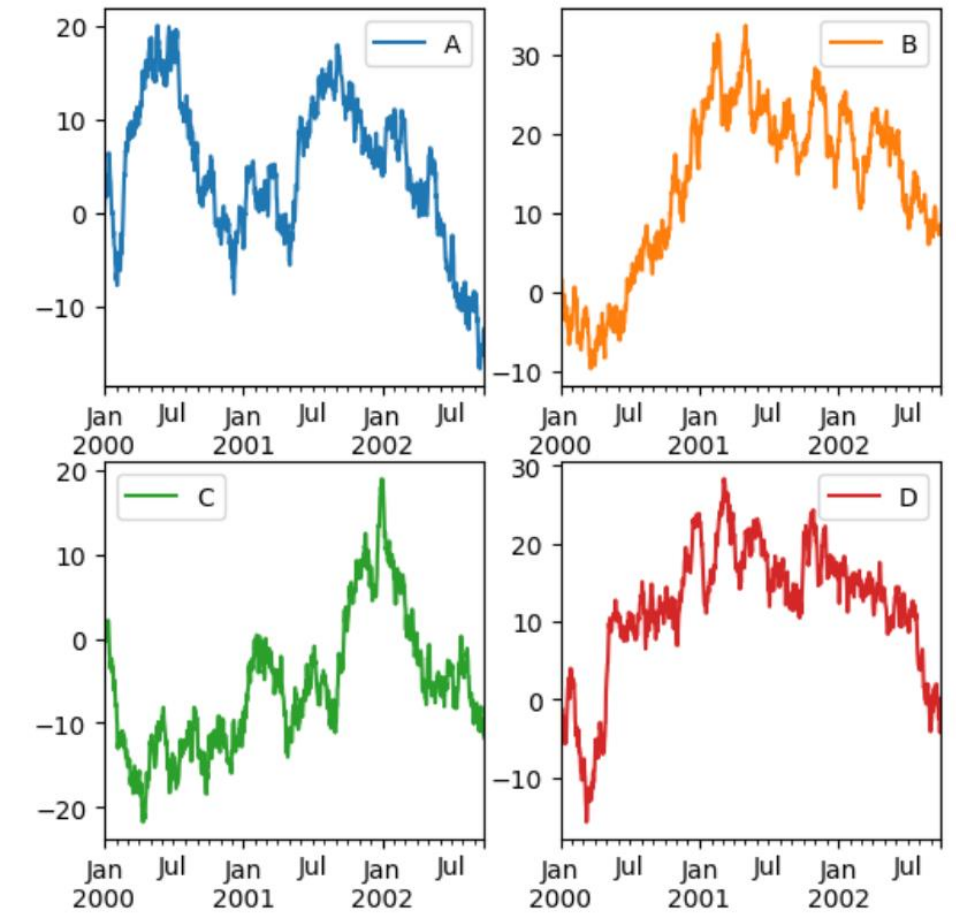

```
In [131]: df["A"].plot();

In [132]: df["B"].plot(secondary_y=True, style="g");
```



subplot display and control

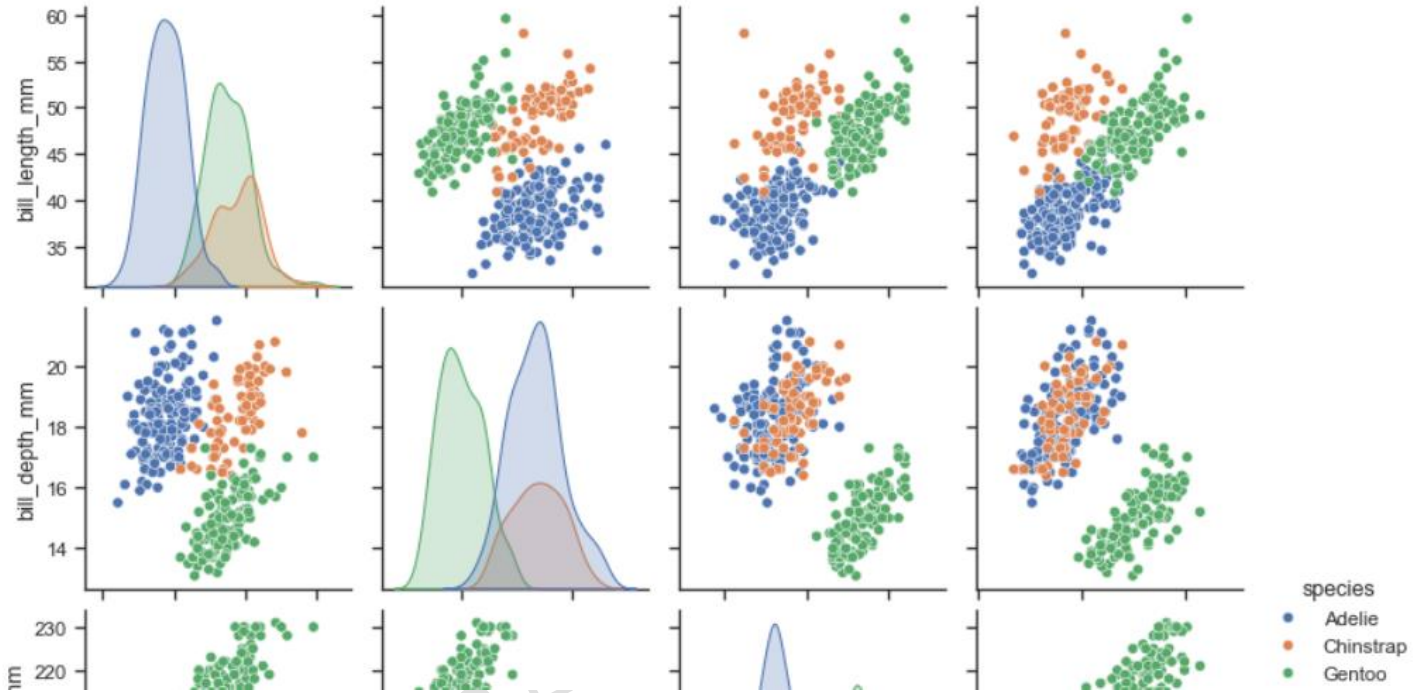
- Each Series in the DataFrame can be plotted on a different axis using the subplots keyword
- The layout of a subgraph can be specified by the layout keyword. It can accept settings in the form of (rows, columns), and can also set a certain dimension value to -1 to automatically calculate

```
df.plot(subplots=True, layout=(2, -1), figsize=(6, 6),sharex=False)
plt.show()
```



Seaborn - Statistical data visualization tool

- Seaborn is a library for making statistical graphics in Python. It builds on top of matplotlib and integrates closely with pandas data structures.
- Seaborn helps you explore and understand your data. Its plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots.
- Its dataset-oriented, declarative API lets you focus on what the different elements of your plots mean, rather than on the details of how to draw them.

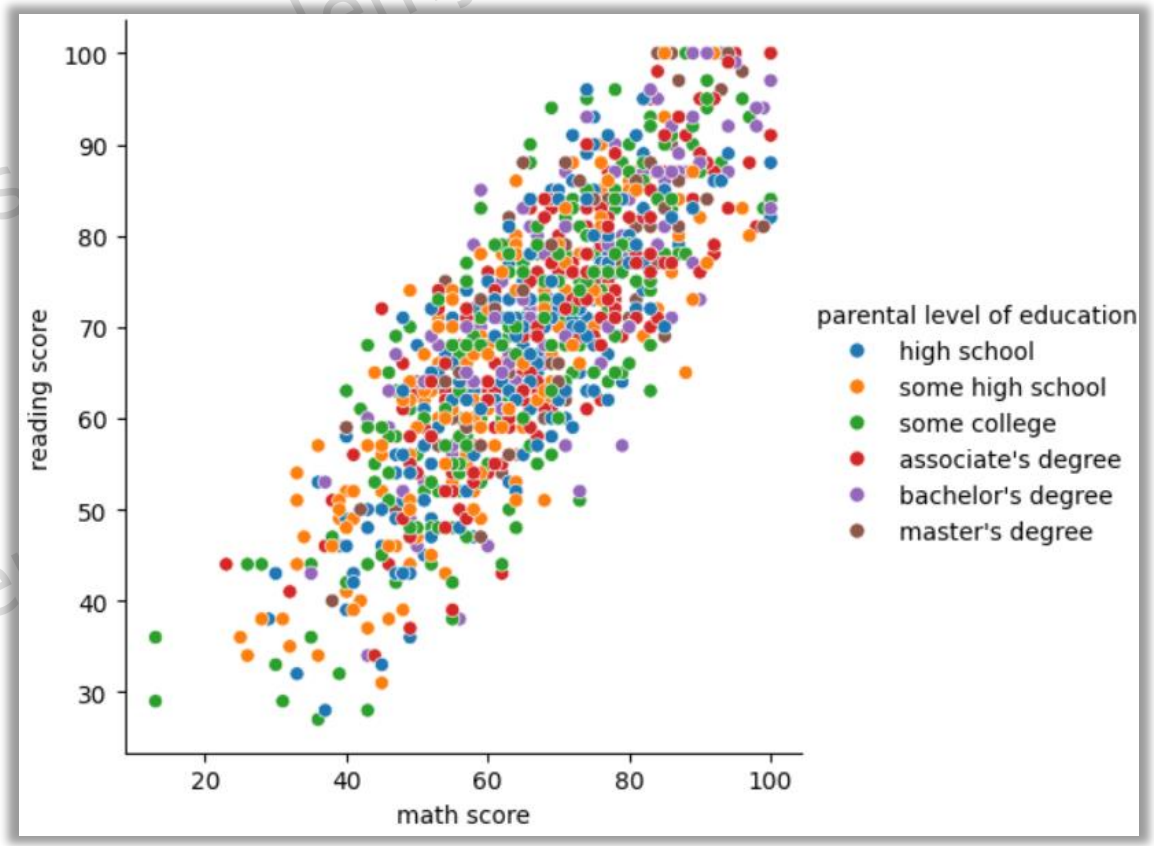


seaborn

relplot - Visualize different statistical relationships

- Figure-level interface for drawing relational plots onto a FacetGrid.
- This function provides access to several different axes-level functions that show the relationship between two variables with semantic mappings of subsets. The kind parameter selects the underlying axes-level function to use:
  - scatterplot() (with kind="scatter"; the default)
  - lineplot() (with kind="line")
- The relationship between x and y can be shown for different subsets of the data using the hue, size, and style parameters. These parameters control what visual semantics are used to identify the different subsets.

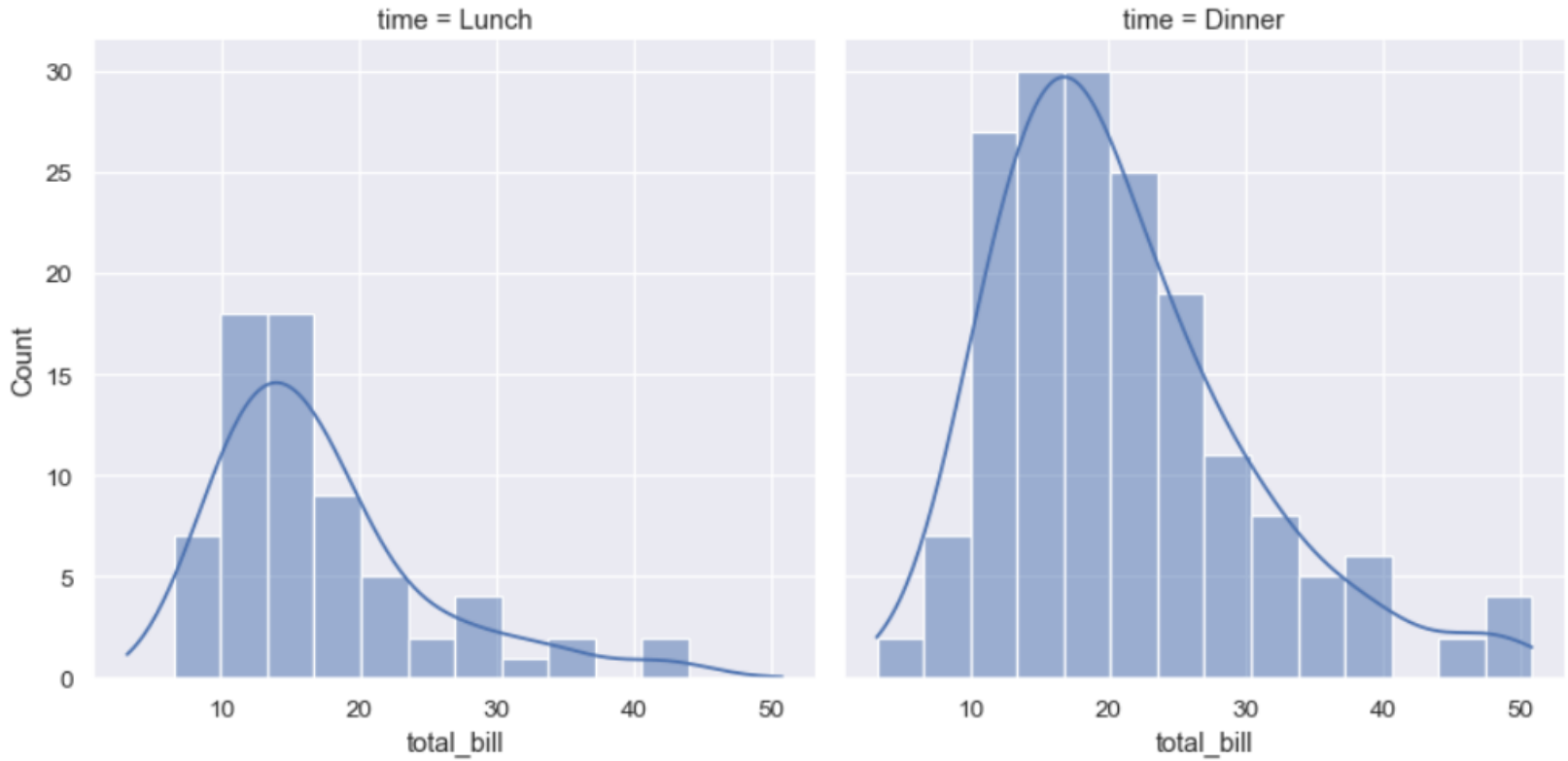
```
df_exam_data = pd.read_csv("exams.csv")
sns.relplot(data=df_exam_data,x="math
score",y="reading score",hue="parental level of
education")
```



displot - distribution of variables

— The seaborn function displot() supports several approaches to visualizing distributions. These include classic techniques like histograms and computationally-intensive approaches like kernel density estimation

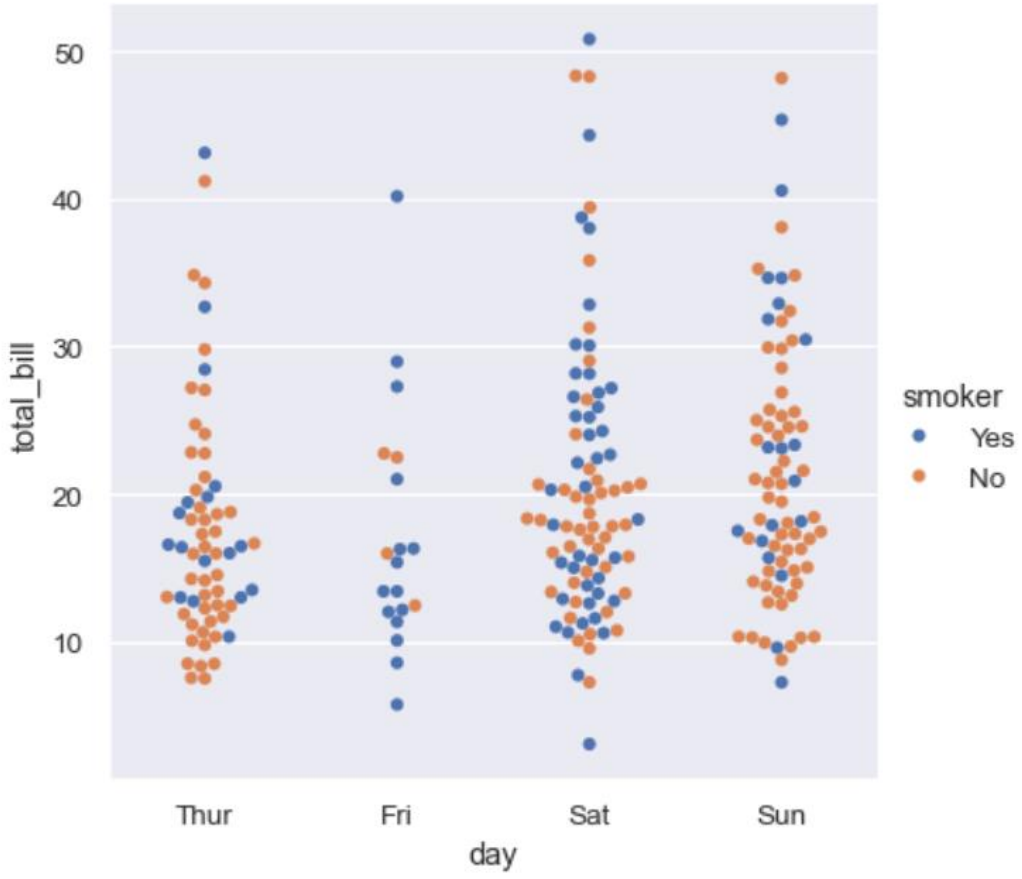
```
sns.displot(data=tips, x="total_bill", col="time", kde=True)
```



catplot - Plots for categorical data (1/2)

- Several specialized plot types in seaborn are oriented towards visualizing categorical data. They can be accessed through catplot().
- These plots offer different levels of granularity. At the finest level, you may wish to see every observation by drawing a “swarm” plot: a scatter plot that adjusts the positions of the points along the categorical axis so that they don’t overlap:

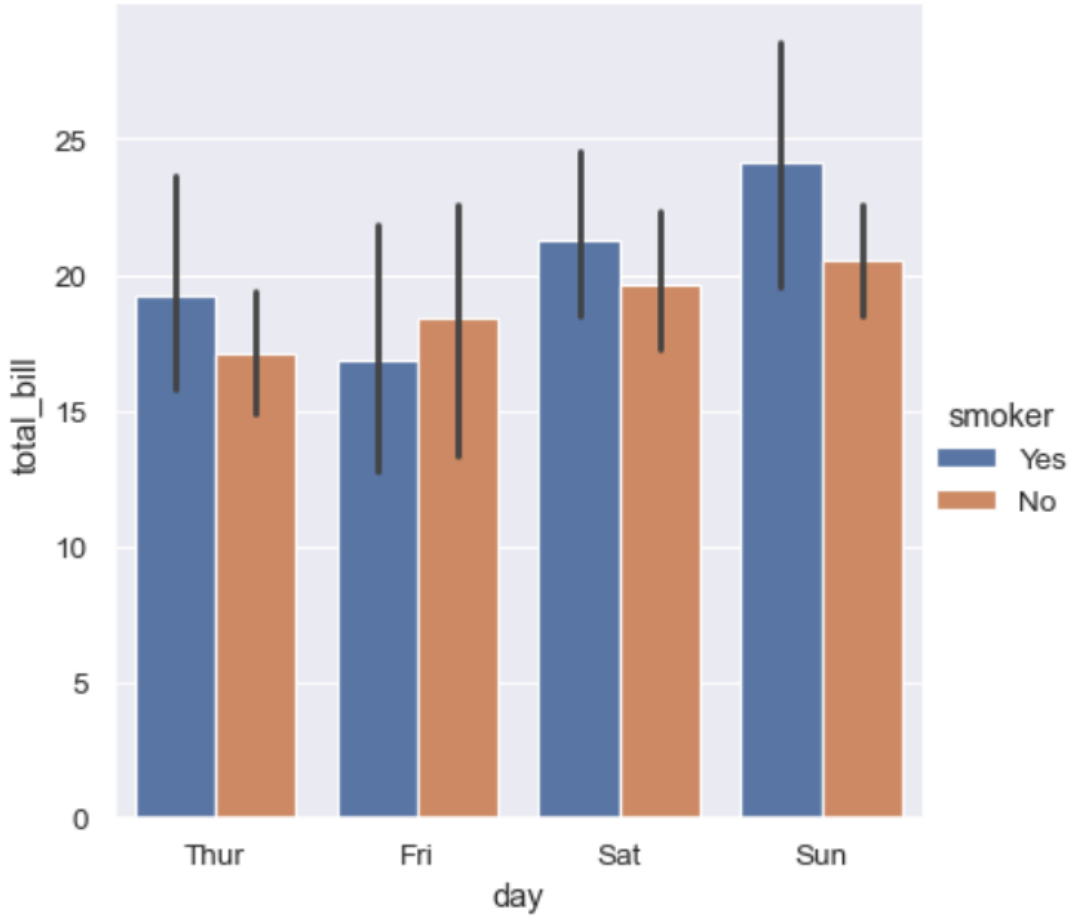
```
sns.catplot(data=tips, kind="swarm", x="day", y="total_bill", hue="smoker")
```



catplot - Plots for categorical data (2/2)

By default, Seaborn represents the mean of the data as the height of the bar and represents the dispersion of the data with a small grey line that crosses though the top of the bar. The top and bottom of that line represent the 95% confidence interval.

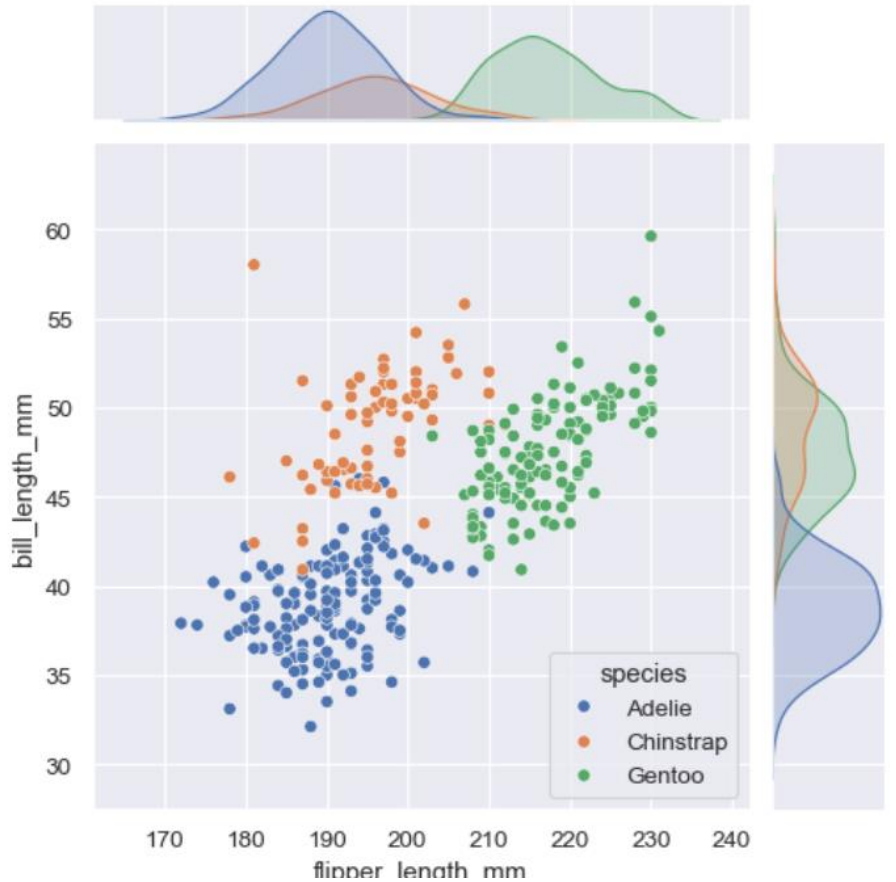
```
sns.catplot(data=tips, kind="bar", x="day", y="total_bill", hue="smoker")
```



Multivariate views on complex datasets(1/2)

Some seaborn functions combine multiple kinds of plots to quickly give informative summaries of a dataset. One, jointplot(), focuses on a single relationship. It plots the joint distribution between two variables along with each variable's marginal distribution:

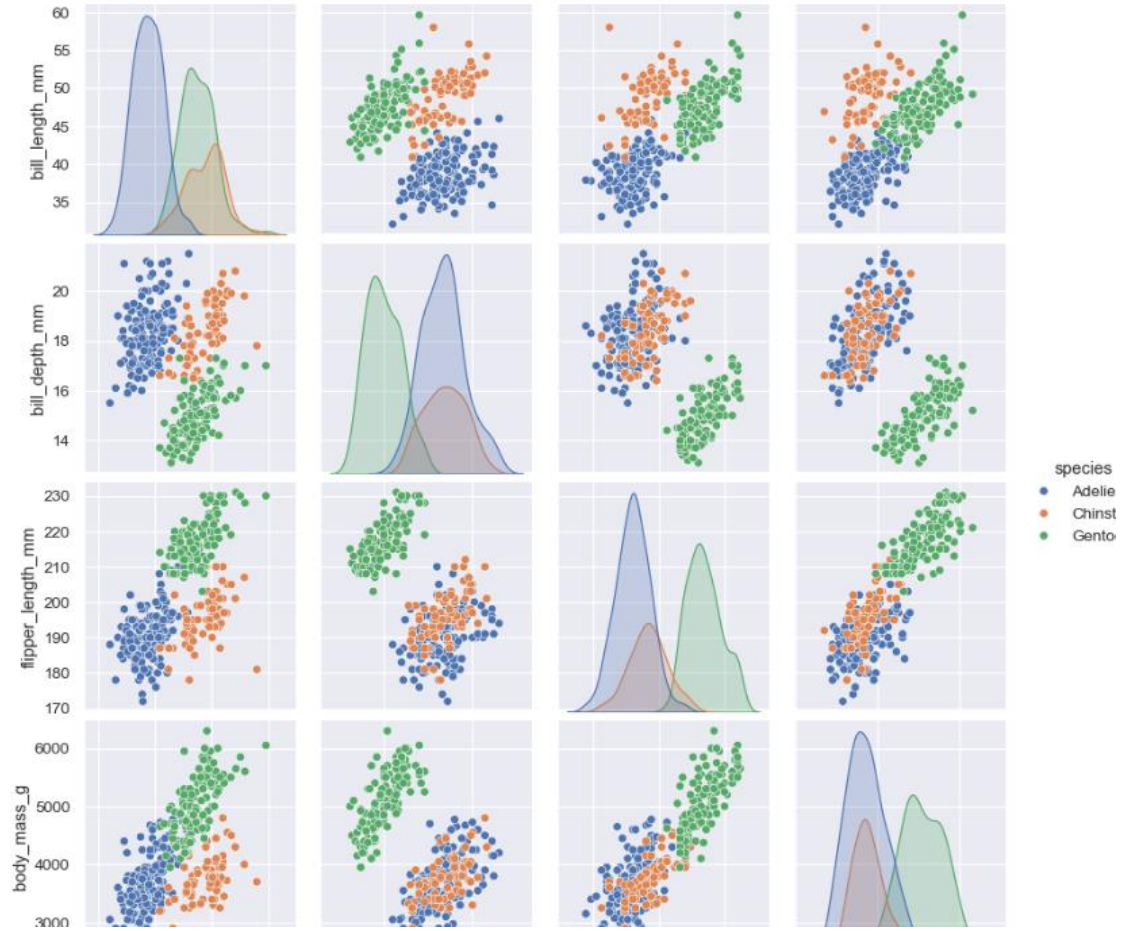
```
penguins = sns.load_dataset("penguins")
sns.jointplot(data=penguins, x="flipper_length_mm", y="bill_length_mm", hue="species")
```



Multivariate views on complex datasets(2/2)

pairplot(), takes a broader view: it shows joint and marginal distributions for all pairwise relationships and for each variable, respectively

```
sns.pairplot(data=penguins, hue="species")
```



Typical data analysis tasks

Typical data analysis tasks

- Read data
- View and judge data information
  - Is the data information reasonable?
  - Is there any missing data?
  - Is there duplicate data?
- Data cleaning
  - Complete or delete missing data
  - Remove duplicate data
  - Convert unreasonable data formats or values
  - Abnormal data found
- View statistical analysis information of data
  - Distribution or grouping status
  - Correlation information

Read data

— Most of the data to be processed will appear in file formats (CSV , Excel,etc). Since the forms of data in the real world are various, it is recommended to check the original data file when actually performing the read operation to determine whether the following special operations are required (CSV as an example):

- Set column separator: sep =','
- Specify which columns to read: usecols =[]
- Do not use the first row as column names: header=None
- Specify the column names after reading: names=[]
- Set a column as an index: index_col ='COLUMN_NAME'
- Skip certain rows: skip_rows =[]
- What values to use to fill missing data: na_values ={}
- Parse column values into datetime format: parse_dates =[]

```
In [12]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

	A	B	C
aaa	-0.264438	-1.026059	-0.619500
bbb	0.927272	0.302904	-0.032399
ccc	-0.264273	-0.386314	-0.217601
ddd	-0.871858	-0.348382	1.100491

```
In [23]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

View and judge data information of each column

- You can quickly view the overall and column information in the data through some methods.
  - info() : Get the name of the column, non-missing values, and the type of column value
    - If there are missing values in a column, you need to decide how to deal with them
    - Whether non-numeric types need to be converted depends on the type of grouping processing: categorical type, date and time type
  - describe() obtains information related to data statistical analysis and distribution:
    - Numerical data: commonly used statistical analysis values
    - Non-numeric data: Statistics based on categorical data
    - Ignore missing data for statistics
  - Numerical nominal data and ordinal data require special attention, because the mathematical calculations that support statistical analysis have no meaning for them and you can choose to delete or convert them.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 9 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   gender                       1000 non-null   object
 1   race/ethnicity               1000 non-null   object
 2   parental level of education  1000 non-null   object
 3   lunch                        1000 non-null   object
 4   test preparation course      1000 non-null   object
 5   math score                   1000 non-null   int64
 6   reading score                1000 non-null   int64
 7   writing score                1000 non-null   int64
 8   math_100                     14 non-null     float64
dtypes: float64(1), int64(3), object(5)
memory usage: 70.4+ KB
```

```
df_students_exam_copy.describe()
```

	math score	reading score	writing score	math_100
count	1000.000000	1000.000000	1000.000000	14.0
mean	66.396000	69.002000	67.738000	100.0
std	15.402871	14.737272	15.600985	0.0
min	13.000000	27.000000	23.000000	100.0
25%	56.000000	60.000000		
50%	66.500000	70.000000		
75%	77.000000	79.000000		
max	100.000000	100.000000		

```
df_students_exam_copy.describe(include=['object'])
```

	gender	race/ethnicity	parental level of education	lunch	test preparation course
count	1000	1000	1000	1000	1000
unique	2	5	6	2	2
top	male	group C	some college	standard	none
freq	517	323	222	652	665

Process missing data

- In the data cleaning stage, it is necessary to study the missing data to determine the possible reasons for its absence and whether its absence will bring bias to the analysis results , then to determine how to deal with missing data.
  - If the proportion of missing data in rows or columns is too high, the meaning of the data will be lost.

```
(df_students_exam_copy['math_100'].notnull().sum()/df_students_exam_copy.shape[0])*100
```

```
1.4000000000000001
```

  - If you want to fill in missing values, you need to consider the selection of values: mean, median, 0, etc., based on the comparison of statistical analysis data before and after filling.
- The dropna method will delete all rows containing missing data by default.
  - how='all' only deletes rows where all columns are missing
  - axis=1 delete by column

Process duplicate data

- Duplicate data will also be encountered in data analysis. It may be that the entire row of data is repeated, or the data in certain columns is repeated (especially if uniqueness is required), which will cause some problems in data analysis.
- Processing logic:
  - Find duplicates based on the duplicated method: rows or certain columns (subset=[])
    - Get the number of duplicates by using sum() for result
  - Deleting duplicate data based on the drop_duplicates method: rows or certain columns (subset=[])
  - common ***keep*** parameter can be set to retain duplicates: the first one (default value); the last one (last): or none (false)
- Think about the conditions for judging duplicate data:
  - Repeat throughout the line
  - Single column repetition (the basic form of categorical data)
  - Repeating multiple columns: numeric type, categorical type and combination, for example: three exams have the same scores, personal information is almost the same
    - How to display query results so that all duplicate rows are displayed?

String data processing

- The data will contain a large amount of string data, and their format will not be consist and uniform, making analysis tasks very difficult, such as grouping based on categorical data.
  - ***abc , ABC, Abc -> abc #Different cases cause differences in data recognition***
- First make object type converted to StringDtype type
- String objects exist in: data values, row or column indexes, are processed in the same way, the basic method is based on the method provided by
  - Case conversion: lower, upper
  - Get the length: len
  - Intercept spaces: strip, lstrip, rstrip
  - Intercept prefix and suffix characters: removeprefix and removesuffix
  - Replace: replace, you can implement complex replacement based on regular expressions
  - Split: split, which can split the contents of Series into list or DataFrame (expand=True)
  - Splicing: cat, which can realize the merging of Series content

More general data conversion methods

- The previous methods for processing strings are only for string data and only support specific methods.
- If you want to use a customized method (function) to convert or map the data, you can choose:
  - Transform function of Series : For Series or DataFrame columns for processing and can also support inputting multiple methods at the same time

```
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

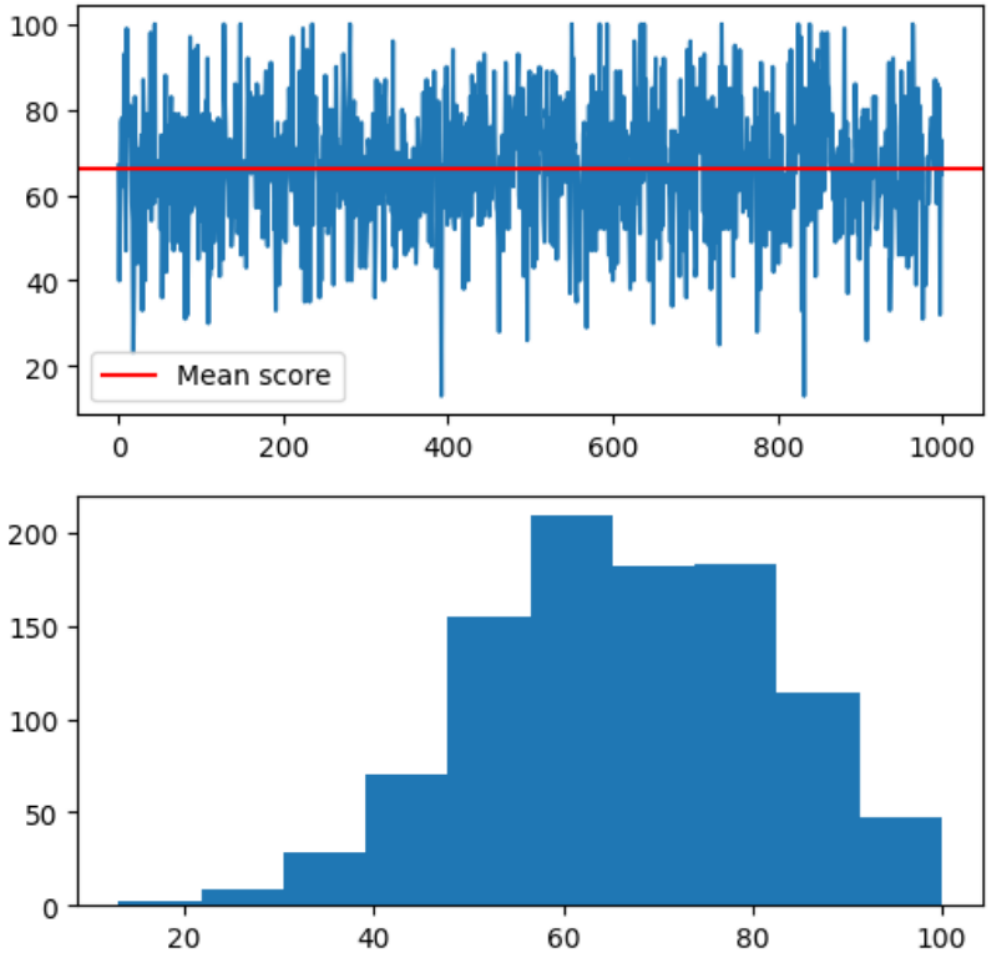
  - DataFrame's applymap (map) function: Similar to Series transform , for DataFrame all elements in are processed in sequence.

    df * df == (df. **applymap** (lambda x: x*x))

    df. **transform** (lambda x: x+x.mean ()) == (df.mean () + df)

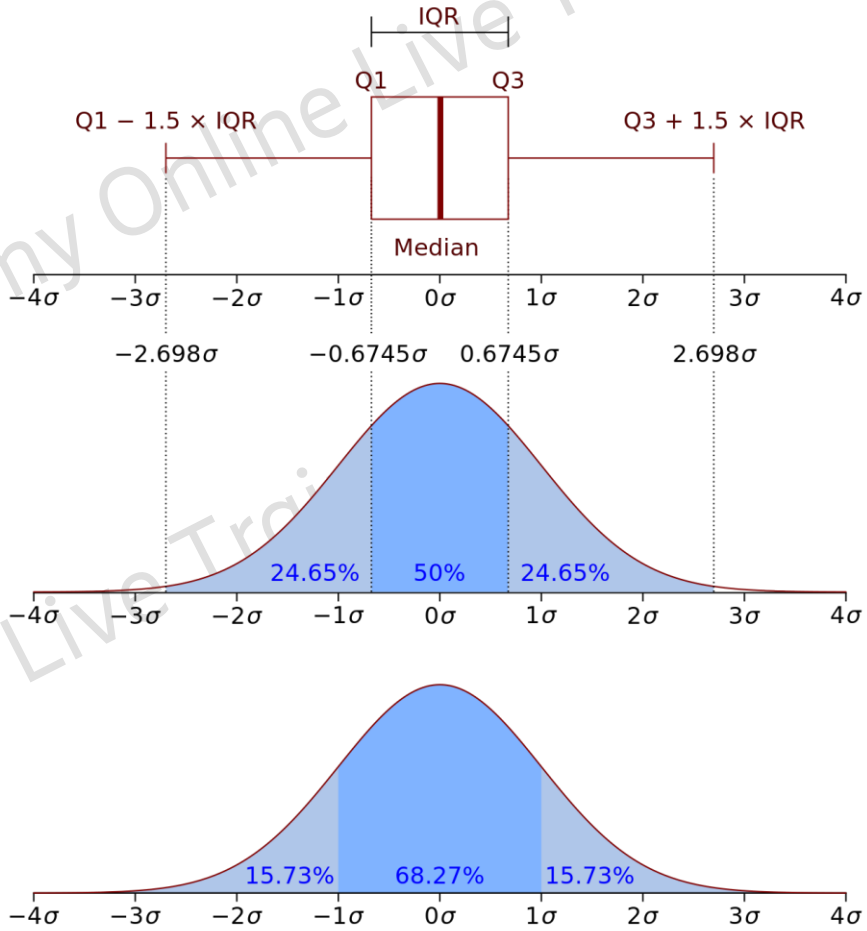
Get the distribution of numerical data

- Data analysis or machine learning will process and calculate a large amount of numerical data, and it is very important to fully understand their information.
- The statistical analysis information and additional parameters provided by describe are the most commonly used means
- View data information graphically:
  - Quantity or probability distribution
  - Changes with index or time
  - Correlation between two sets of variables
  - Use mixed charts to compare and view relevant data information



Discover and handle abnormal data

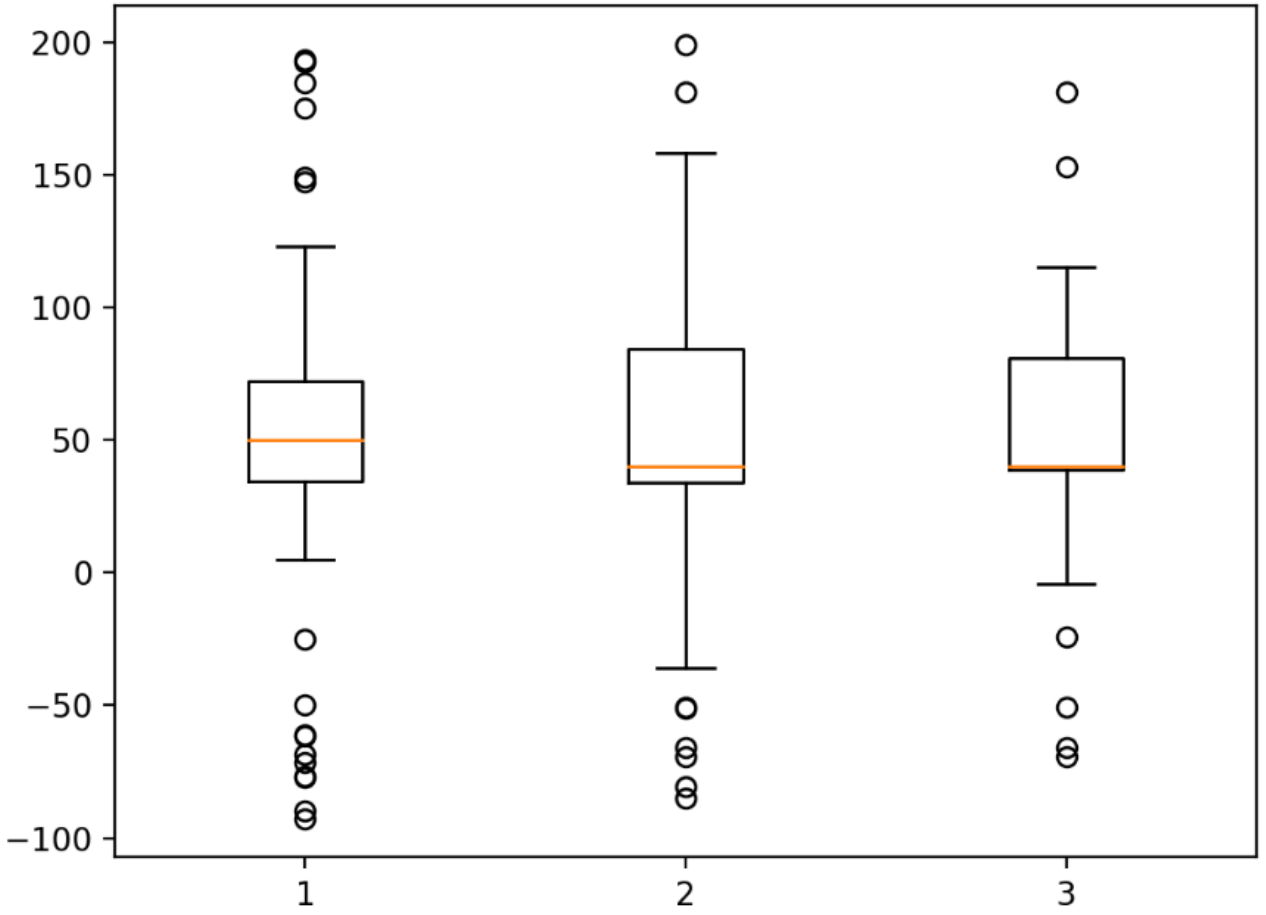
- In addition to missing data and duplicate data, in order to improve efficiency in subsequent processing, you can first find some abnormal data values (numeric type) and process them.
- Discovery method
  - Histogram
  - Box plot (boxplot)
- Judgment basis for common anomalies
  - Standard deviation or z-score: Measured based on the standard deviation distance between a value and the mean, suitable for normal or Gaussian distributed data
  - Interquartile range IQR (interquartile range): data between Q3-Q1
- Processing method:
  - Delete
  - Winsorize (Winsorize)
  - Logarithmic (log) transformation



Boxplot

- A boxplot is a standardized method of summarizing a data set based on five numbers:
  - the minimum value
  - the maximum value
  - the sample median(Q2)
  - the first (Q1)
  - third (Q3) quartile.
- boxplot in matplotlib , replace the maximum and minimum values with Q3+1.5*IQR and Q1-1.5*IQR
- The length of the cabinet is Q3-Q1 (IQR)
- Values outside the above range are automatically identified as outliers

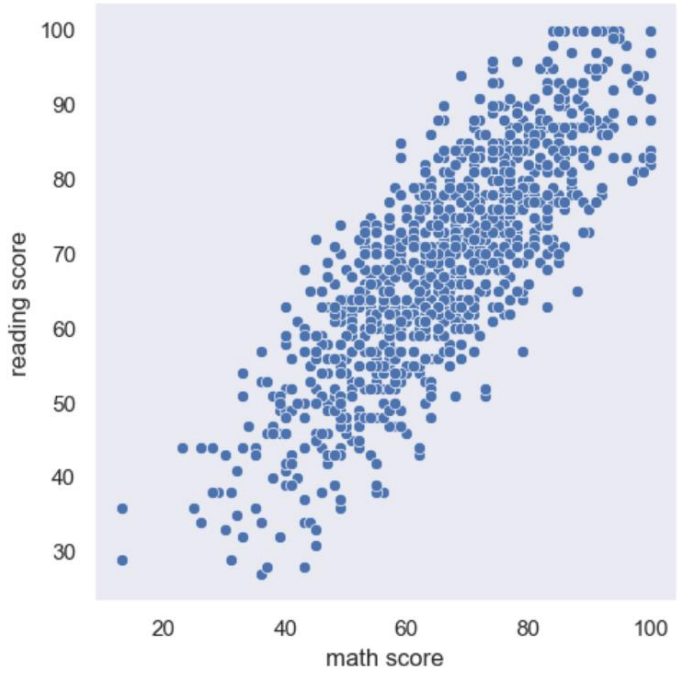
** IQR: Interquartile range*



See correlations between multiple variables

- In statistics, correlation or dependence is any statistical relationship between two random variables or bivariate data, **but is not a causal relationship (因果関係)**. It usually refers to the degree to which a pair of variables are linearly related. The value will be between 1 and -1 , indicating positive to negative correlation.
- corr function in pandas can be used to calculate correlations:
  - The corr method of Series implements the calculation of correlation between it
  - DataFrame's corr calculates the correlation between all numeric data in it
- Understand correlation graphically:

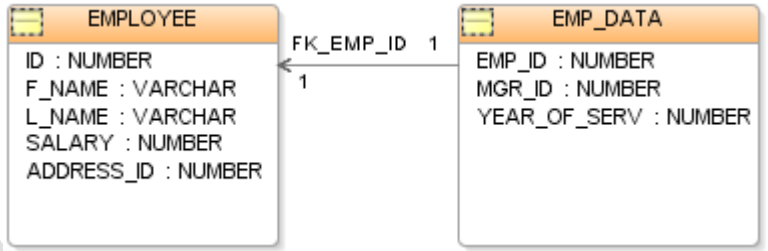
```
>>> def histogram_intersection(a, b):
...     v = np.minimum(a, b).sum().round(decimals=1)
...     return v
>>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                   columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
      dogs  cats
dogs   1.0   0.3
cats   0.3   1.0
```

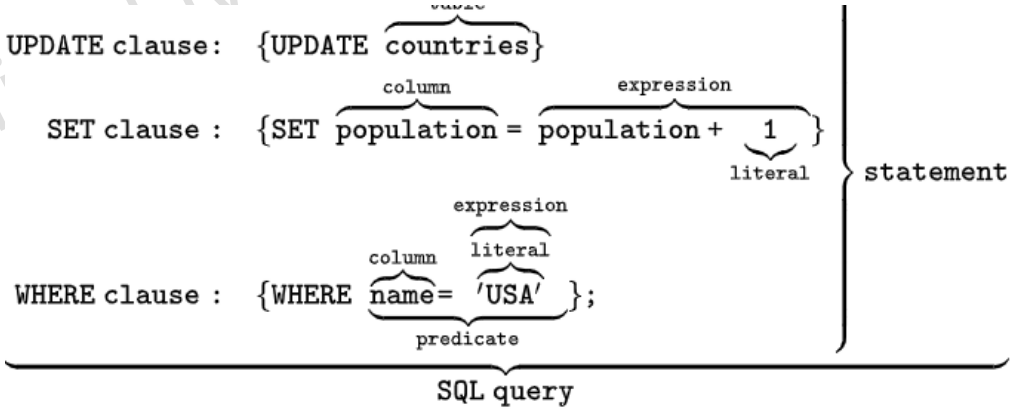


Use SQL to implement data analysis tasks

Relational databases and SQL

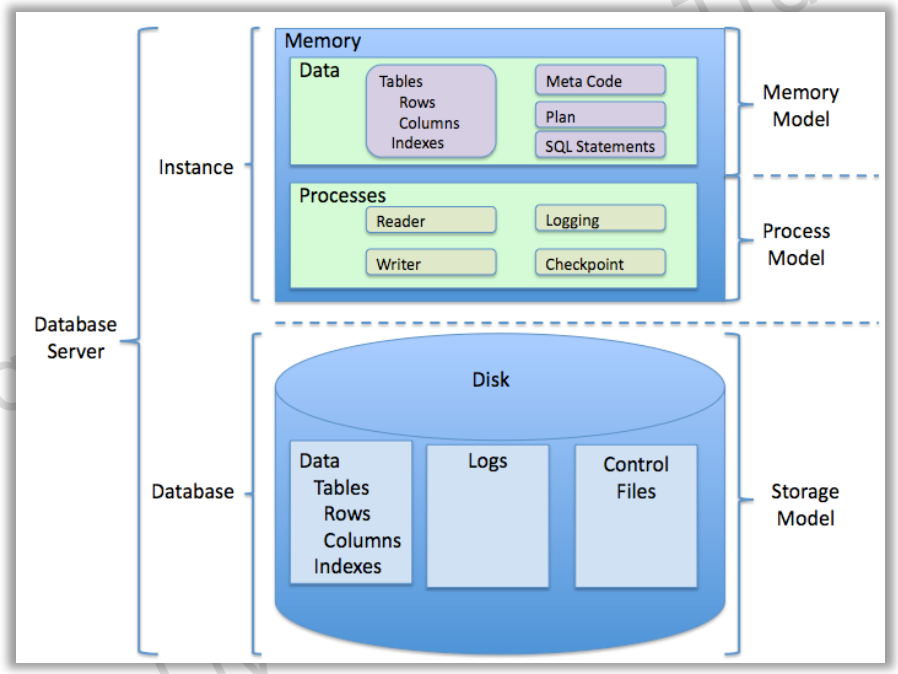
- Relational database is a database created based on the relational model. Various entities in the real world and various connections between entities are represented by relational models.
- The standard data query language SQL is a language based on relational databases. This language performs retrieval and operation of data in relational databases.
- SQL (Structured Query Language) language is a general-purpose, highly functional relational database language that can implement data insertion, query, update and deletion, database schema creation and modification, and data access control.
- SQL statements:
  - Data Definition Language (DDL): CREATE, ALTER, DROP
  - **Data Manipulation Language (DML): SELECT, DELETE, UPDATE, INSERT**
  - Data Control Language : GRANT , DENY
  - Transaction control language: BEGIN, COMMIT, ROLL BACK

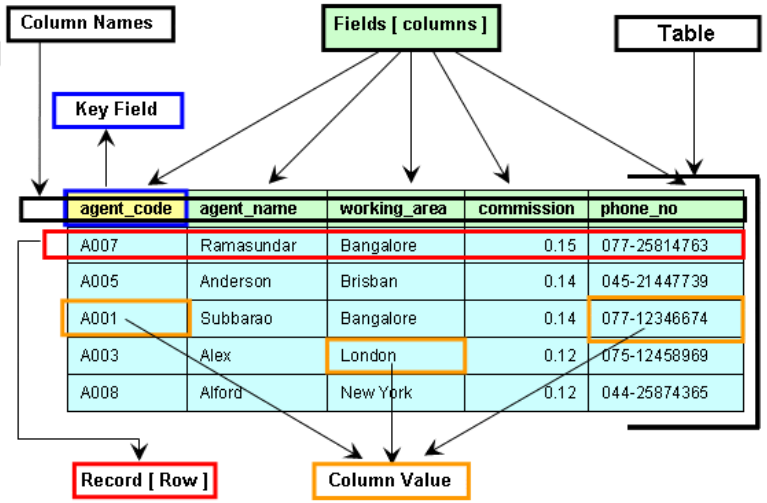




Overview of basic objects of relational data sets

- Relational database management system (RDBMS): Software that implements relational databases and provides management interfaces, such as MySQL .
- Database instance: A running RDBMS , which can be understood as part of its software or program process.
- (Logical) database: A database instance can create and manage multiple logical databases, which should have their own data, logs and other objects.
  - Table: It is a collection of data organized in the form of rows and columns. A database can include one or more tables (Relation)
  - A row represents a complete data record (in a single table), composed of one record for each column
  - Columns represent separate data for each record, with independent data types and attributes: primary key, foreign key, index, uniqueness, etc.





Create database table

- Most relational database systems need to create an independent logical database (database) first, and then create a database table (table) in it, so most data table operations need to be performed under a specified database context.
- Syntax for creating a table:

CREATE TABLE { table_name } (

{column_name_1} {data_type_1} {column_constraint_1},

...

{ column_name_last } { data_type_last } { column_constraint_last },);

  - Table Name
  - Column names and types
    - numerical value
    - character
  - Constraints for each column (optional)
    - NOT NULL
    - Uniqueness (UNIQUE)
    - PRIMARY KEY

```
CREATE TABLE countries (
key INT PRIMARY KEY,
name text UNIQUE,
founding_year INT,
capital text
);
```

Modify and delete database tables

- Modification of the database table is an operation that will affect the current data. Similar operations must be performed with **great caution and after investigation and data backup** .
- Add and delete columns:
  - After adding a new column, new data needs to be added manually

    ALTER TABLE { table_name }

    ADD COLUMN { column_name } { data_type };

  - Delete column syntax:

    ALTER TABLE { table_name }

    DROP COLUMN { column_name };
- Delete database table
  - Syntax:

    DROP TABLE { table_name };
  - Some database systems will prevent you from deleting tables that are referenced by foreign keys. You need to delete the references first and then delete the table.

Update data in table

- Insert new data
  - The new data entered needs to meet the format and quantity requirements of each column in the table

```
INSERT INTO { table_name } ({column_1], {column_2}, ...{ column_last }) \
VALUES ({column_value_1}, {column_value_2}, ... { column_value_last });
```

- Update existing data
  - Updating data requires providing the updated conditions (where) and the corresponding columns and their new values.

```
UPDATE { table_name } \
SET {column_1} = {column_value_1}, \
{column_2} = {column_value_2}, \
WHERE  {conditional};
```

- Delete existing data rows
  - Deleting data rows requires providing deletion conditions (where), otherwise all data rows will be deleted (the table still exists)

```
DELETE FROM { table_name } \
WHERE {conditional};
```

Query data in table

— For data analysis based on SQL , the most important task is to rely on ***SELECT*** to implement data query, which can include:

- Operation: The first part of the query describes what to do, ***SELECT*** and the following column names represent the tasks and goals of this operation.
- Data: That is one or more table names after the ***FROM*** keyword, indicating which data to filter, select, and calculate.
- Conditions: Use ***WHERE*** to represent the conditions that the data needs to meet to implement filtering.
- Grouping: Use the key created by the ***GROUP BY*** clause to group the data and then the aggregation method calculates the corresponding value, such as: sum, average, etc.
- Post-processing: Use ***ORDER BY*** and ***LIMIT*** to sort the query results, limit the quantity, and other subsequent operations.

```
-- Fetch customers in the state of Florida in alphabetical order

SELECT email
FROM customers
WHERE state='FL'
ORDER BY email

-- Pull all the first names, last names, and email for ZoomZoom cu

SELECT first_name, last_name, email
FROM customers
WHERE city='New York City'
and state='NY'
ORDER BY last_name, first_name

-- Fetch all customers that have a phone number ordered by the dat

SELECT *
FROM customers
WHERE phone IS NOT NULL
ORDER BY date_added
```

Basic query with SELECT (1/2)

- The most basic query method is to specify the query table name and column name to query. Use * instead of the column name to query all the data in the table:
  - ***SELECT * FROM products;***
  - ***SELECT product_id,model FROM products;***
- Use WHERE to set query conditions:
  - For numerical data, you can use comparison to set conditions

SELECT model FROM products WHERE year>2014;

  - Use IN/NOT IN can be used to set whether the data belongs to / does not belong to certain values.

SELECT model FROM products WHERE year IN (2014, 2016, 2019);

  - set whether the data value is NULL through IS NULL/IS NOT NULL

SELECT * FROM products WHERE production_end_date IS NULL;

  - Logical combinations of multiple conditions

SELECT model FROM products WHERE year=2014 OR product_type ='automobile';

SELECT * FROM products WHERE year>2014 AND year<2016 OR product_type ='scooter';

Basic query with SELECT (2/2)

— Use ORDER BY to sort data results based on a certain column or several columns

- The default sorting is ascending order (ASC), DESC is descending order; if there are multiple columns, you can set the sorting method separately.

SELECT model FROM products ORDER BY production_start_date DESC;

SELECT * FROM products ORDER BY year DESC, base_msrp ASC;

- Instead of using the column name, you can also use the sequence number of the column (starting from 1):

SELECT model FROM products ORDER BY product_id ;

SELECT model FROM products ORDER BY 1 ;

— Use LIMIT to limit the number of output results:

SELECT model FROM products ORDER BY product_id LIMIT 5;

Analyze data based on aggregation methods

- In addition to focusing on individual data, analysis tasks will also focus on the general information of the data (mainly numerical data) and further discover patterns and rules.
- SQL will implement the above tasks through aggregation methods. Common aggregation methods are:
  - COUNT(columnX) : Statistics of columnX the number of non-empty values in the column. If columnX is * , the number of rows in the table is counted.
  - MIN/MAX (columnX) : get columnX column min / max value
  - SUM (columnX) : calculate columnX sum of all values in column
  - AVG(columnX) : calculate columnX mean of all values in a column
  - STDDEV (columnX) : calculate columnX standard deviation of all values in a column
- The aggregation method is more often applied to some data in the table, based on:
  - WHERE set conditions

  SELECT COUNT(*) FROM customers WHERE state='CA';
  - Grouping implemented by GROUP BY

Group and aggregate data

- The aggregation method is suitable for numeric data, while the categorical data is suitable for grouping data as needed.
- SQL 's GROUP BY groups based on the defined information, then applies the aggregation data and finally outputs the aggregated results of each group.

  SELECT {KEY}, {AGGFUNC(column1)} FROM {table1} GROUP BY {KEY}

  - {KEY} is a column used to create a group or a function applied to a column. The {KEY} after SELECT is not required.

```
pd.read_sql(text("select gender,count(*) from exam group by gender;"),conn)
```

	gender	count
0	female	483
1	male	517

- After GROUP BY, you can continue to use ORDER BY to sort the output.

```
pd.read_sql(text("SELECT state, COUNT(*) FROM customers GROUP BY state ORDER BY count DESC"),conn)
```

	state	count
0	None	10934
1	CA	10076
2	TX	9730
3	FL	7496
4	NY	4790
5	OH	3312
6	VA	3110
7	DC	2894
8	PA	2838

Use HAVING to add filter conditions for GROUP BY

- WHERE can be followed by the definition of the column value to implement the filtering function, but it cannot be followed by the aggregation method, for example:

  ~~... WHERE COUNT > 1000~~

- Similar needs can be achieved through

```
uery_text="SELECT state, gender, COUNT(*) AS count FROM customers GROUP BY state, gender HAVING  COUNT(*) > 1000 ORDER BY count;"
d.read_sql(text(query_text),conn)
```

	state	gender	count
0	CO	F	1032
1	AZ	F	1032
2	CO	M	1052
3	NC	F	1068
4	NC	M	1072
5	IL	F	1084

Using JOIN for multi-table query

- The characteristic of relational databases is to establish relationships between data through the tables themselves and between tables, so joint queries between multiple tables are also a frequently used method for data analysis.
- A possible scenario is to use the query results in one table as query conditions for querying another table:
  - All books published by women authors in 2020: book table, author table
  - 5G package contract users from this city : user table, contract record table
- JOIN types:
  - (INNER) JOIN: Returns records with matching values in both tables
  - LEFT (OUTER) JOIN: Returns all records in the left table and matching records in the right table
  - RIGHT (OUTER) JOIN: Returns all records in the right table and matching records in the left table
  - FULL (OUTER) JOIN: Returns all records when there is a match in the left or right table

INNER JOIN

- INNER JOIN joins rows from different tables together based on defined conditions.
- In many cases, the condition is rows with equal column values in both tables.

  SELECT {columns}

  FROM {table1}

  INNER JOIN {table2} ON {table1}.{common_key_1}={table2}.{common_key_2}

- The columns used for comparison in the two tables generally have established foreign key relationships and are unique, otherwise a large amount of duplicate data will be generated.
- Generally, filtering conditions are further set based on WHERE.
- The output columns can be specified for display by ***table_name.column_name***.

```
query_text="""SELECT s.*
FROM salespeople as s
INNER JOIN dealerships
ON s.dealership_id = dealerships.dealership_id
WHERE dealerships.state = 'CA'
ORDER BY 1"""
pd.read_sql(text(query_text),conn)
```

	salesperson_id	dealership_id	title	first_name	last_name	suffix	username	gender	hire_date	termi
0	23	2	None	Beauregard	Peschke	None	bpeschkem	Male	2018-09-12	
1	23	2	None	Beauregard	Peschke	None	bpeschkem	Male	2018-09-12	
2	23	2	None	Beauregard	Peschke	None	bpeschkem	Male	2018-09-12	
3	23	2	None	Beauregard	Peschke	None	bpeschkem	Male	2018-09-12	
4	51	5	None	Lanette	Gerriessen	None	lgerriessen1e	Female	2018-06-24	
...	...	...	...	...	...	...	...	...	...	
135	290	5	None	Ibby	Ryrie	None	iryrie81	Female	2016-01-13	
136	298	5	None	Cary	MacAllester	None	cmacallester89	Male	2018-04-16	
137	298	5	None	Cary	MacAllester	None	cmacallester89	Male	2018-04-16	
138	298	5	None	Cary	MacAllester	None	cmacallester89	Male	2018-04-16	
139	298	5	None	Cary	MacAllester	None	cmacallester89	Male	2018-04-16	

OUTER JOIN

- INNER JOIN only returns rows that satisfy both tables based on conditions, while OUTER JOIN returns all rows from at least one table.
- LEFT OUTER JOIN will return all the rows of the left table (the first table), and return the content of the right table that meets the conditions
- Use the keyword LEFT (OUTER) JOIN to define
- RIGHT OUTER JOIN is defined similarly to LEFT , except that it will return all the rows of the right table (the second table) and the content of the left table that meets the conditions.
- RIGHT OUTER JOIN will display the contents of both tables, and those that meet the conditions will be displayed again. Used with caution, a very large result set may be output.

```
query_text="""SELECT *
FROM customers c
LEFT OUTER JOIN emails e ON c.customer_id = e.customer_id
WHERE e.customer_id IS NULL
ORDER BY c.customer_id
LIMIT 1000;"""
pd.read_sql(text(query_text),conn)
```

	customer_id	title	first_name	last_name	suffix	email	gender	ip_address	phone	street_address	...	date_added	email_id
0	27	None	Anson	Fellibrand	None	afellibrandq@topsy.com	M	64.80.85.50	203-107-9111	65 Shelley Road	...	2019-04-07	None
1	27	None	Anson	Fellibrand	None	afellibrandq@topsy.com	M	64.80.85.50	203-107-9111	65 Shelley Road	...	2019-04-07	None
2	32	None	Hamnet	Purselowe	None	hpurselowev@oaic.gov.au	M	225.215.209.222	239-462-	5 Johnson Way	...	2019-02-07	None

```
query_text="""SELECT *
FROM emails e
RIGHT OUTER JOIN customers c ON e.customer_id=c.customer_id
ORDER BY c.customer_id
LIMIT 1000;"""
pd.read_sql(text(query_text),conn)
```

	email_id	customer_id	email_subject	opened	clicked	bounced	sent_date	opened_date	clicked_date	customer_id	...	gender	ip_address	phone	str
0	323983.0	1.0	Save the Planet with some Holiday Savings.	f	f	f	2018-11-23 15:00:00	NaT	NaT	1	...	F	98.36.172.246	None	
1	323983.0	1.0	Save the Planet with some Holiday Savings.	f	f	f	2018-11-23 15:00:00	NaT	NaT	1	...	F	98.36.172.246	None	
2	370722.0	1.0	A New Year, And Some New EVs	f	f	f	2019-01-07 15:00:00	NaT	NaT	1	...	F	98.36.172.246	None	
3	282584.0	1.0	Black Friday. Green Cars.	t	f	f	2017-11-24 15:00:00	2017-11-26 01:12:32	NaT	1	...	F	98.36.172.246	None	
			Save the												

Implement JOIN using subquery

- The target of the previous JOIN is a certain table as a whole. If the requirement is to perform JOIN with a part of the data of a certain table, it can be achieved by using a subquery.
- The definition of a subquery is to define the SELECT query statement within a pair of brackets, and then give it an alias. The JOIN operation can treat the query results corresponding to this alias as an independent table.

```
SELECT *
FROM salespeople
INNER JOIN (
SELECT * FROM dealerships
WHERE dealerships.state = 'CA'
) d
ON d.dealership_id = salespeople.dealership_id
ORDER BY 1
```

```
SELECT *
FROM salespeople
WHERE dealership_id IN (
SELECT dealership_id FROM dealerships
WHERE dealerships.state = 'CA'
)
ORDER BY 1
```

#When the subquery output has only one column, you can use IN instead of JOIN.

Using UNION for multi-table query

- The JOIN operation implements horizontal splicing between two tables, and the output results include the columns in the two tables. UNION implements vertical table splicing and query operations.
- Assume that the requirement is to query the data of the same column from two tables separately, but you want the output results to be combined together, then you can use UNION to assemble the two subquery statements together.
- UNION requires that the subqueries within it have columns with the same name and the same column data type.
- By default, UNION does not return all rows of a subquery and removes any duplicate rows from the output. If you want to keep duplicate rows, you can use UNION ALL.

```
query_text="""(
SELECT street_address, city, state, postal_code
FROM customers
WHERE street_address IS NOT NULL
)
UNION
(
SELECT street_address, city, state, postal_code
FROM dealerships
WHERE street_address IS NOT NULL
)
ORDER BY 1;"""
pd.read_sql(text(query_text),conn)
```

	street_address	city	state	postal_code
0	00003 Continental Crossing	Suffolk	VA	23436
1	00003 Sullivan Road	Des Moines	IA	50981
2	00006 Birchwood Plaza	Lakeland	FL	33805
3	00006 Roth Plaza	Fort Smith	AR	72916
4	00006 Vidon Place	Dallas	TX	75358
...	...	...	...	...
44548	9 Westport Pass	Grand Rapids	MI	49510
44549	9 Westport Point	Biloxi	MS	39534
44550	9 Westridge Drive	Seattle	WA	98148
44551	9 Westridge Trail	Orlando	FL	32808
44552	9 West Street	Virginia Beach	VA	23454

44553 rows × 4 columns

Convert query results (1/2)

- The query results may not necessarily meet the final requirements. You can use the methods provided by the database system to convert the results:
  - CASE WHEN : Allows the query to map various values in the column to other values, creating a **new column** to hold the converted results:

CASE WHEN condition1 THEN value1

WHEN condition2 THEN value2

WHEN conditionX THEN valueX

ELSE else_value END

```
SELECT *,
CASE WHEN postal_code ='33111' THEN 'Elite Customer'
CASE WHEN postal_code ='33124' THEN 'Premium Customer'
ELSE 'Standard Customer' END
AS customer_type # the new column stored the converted data
FROM customers;
```

Convert query results (2/2)

- COALESCE : Convert NULL in the result to the specified value

```
SELECT first_name,last_name ,
COALESCE (phone, 'NO PHONE') as phone
FROM customers
```

- NULLIF : Contrary to COALESCE logic, sets the specified value to NULL
- DISTINCT and DISTINCT ON : only display non-duplicate (unique) data

```
SELECT DISTINCT year FROM products  #Only display non-duplicate year column data
SELECT DISTINCT ON (year) * FROM products;
```

- Data format conversion:

  Syntax: column_name ::datatype

```
SELECT product_id,model , year::TEXT, product_type ,
FROM products;
```

Course summary

- Data analysis overview
- Python - based data analysis
- Numpy Base
- Implement data analysis using Pandas
- Python - based data visualization
- Typical data analysis tasks
- Use SQL to implement data analysis tasks

ericsson.com/training