



Samsung Innovation Campus

| Artificial Intelligence Course

Together for Tomorrow!
Enabling People

Education for Future Generations

Chapter 3.

Exploratory Data Analysis: NumPy Arrays for Optimized Numerical Computation and Pandas

| AI Course

Chapter Description

◆ Chapter objectives

- ✓ Understand the precise use of NumPy and be able to process data efficiently.
- ✓ Learn the basics of NumPy arrays, indexing, and slicing and the various ways of their application.
- ✓ Learn to create and handle series and data frame objects.
- ✓ Learn the appropriate methods of optimal model execution for data preprocessing using the Pandas library to explore and convert data.
- ✓ Be able to find the appropriate analysis method by implementing a data visualization suitable for the data scale.

◆ Chapter contents

- ✓ Unit 1. NumPy Array Data Structure for Optimal Computational Performance
- ✓ Unit 2. Optimal Data Exploration Through Pandas
- ✓ Unit 3. Pandas Data Preprocessing for Optimal Model Execution
- ✓ Unit 4. Data Visualization for Various Data Scales

Unit 1.

NumPy Array Data Structure for Optimal Computational Performance

- | 1.1. NumPy Arrays
- | 1.2. NumPy Array Basics
- | 1.3. NumPy Array Operations
- | 1.4. NumPy Indexing and Slicing
- | 1.5. Array Transposition and Axis Swap

Data Structure

The most important concepts in programming are data types, data structures, and algorithms. Knowing the clear differences between these three concepts is essential for easily dealing with various programming languages and solving many errors.

I Data Type

- ▶ First, a data type, in computer science and programming languages, is a classification that identifies a type of data such as floats, integers, Booleans, characters, and strings while also determining the size of the data. It should be used by applicable data types. The data types differ by programming language, but most of the data type concepts from the C language are inherited and used and have affected other languages.
- ▶ Memory is expensive, but when C was introduced, it was expensive and difficult to store a lot of data. As a result, it was designed to be optimized for storing as little space as necessary, which eventually resulted in a data size problem.
- ▶ For example, in C, integer data types are divided into char(compatible with integers), short, int, and long. Each byte size differs from the other data types.

| Data Type

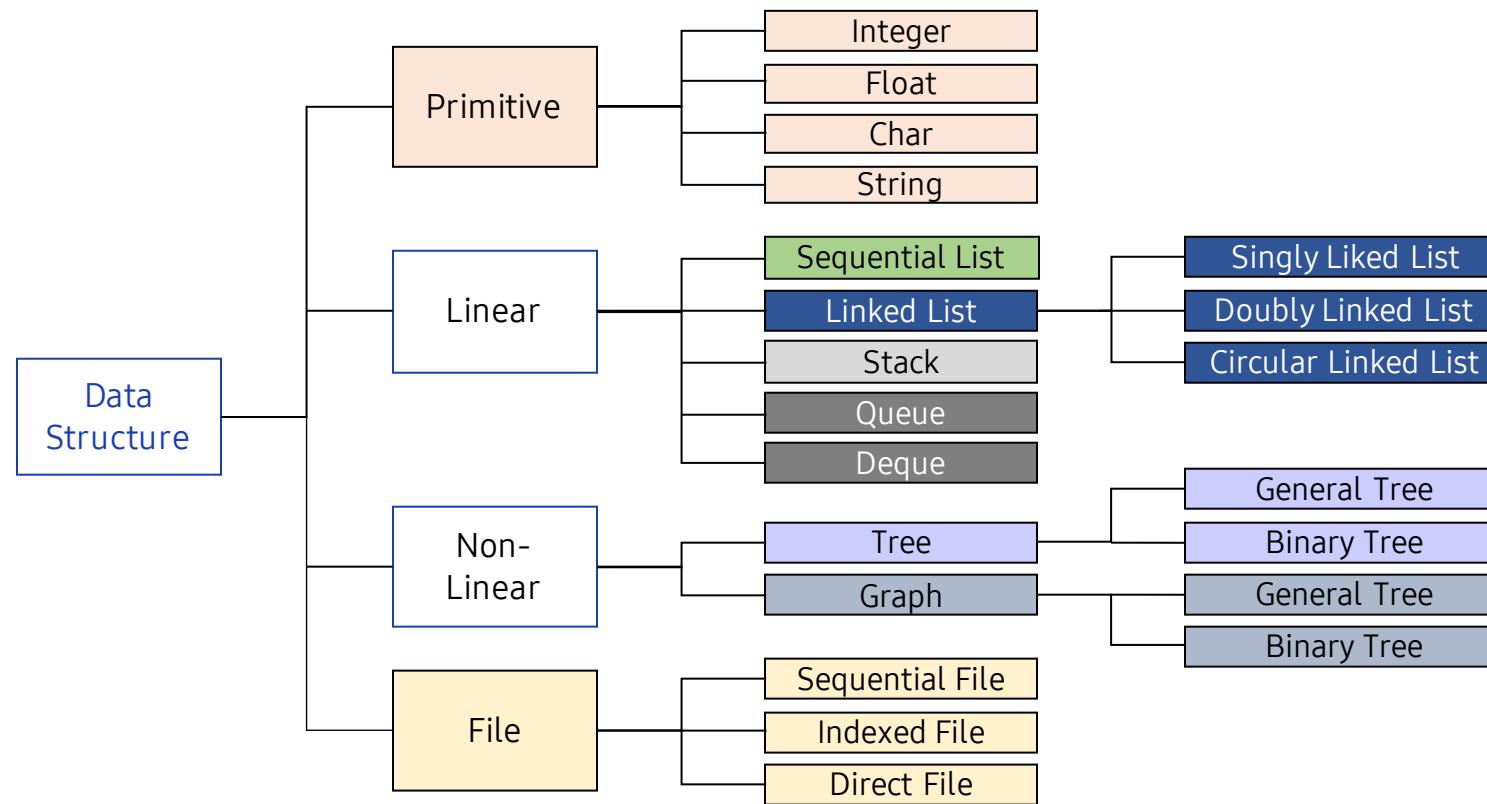
- ▶ Second, data structure in computer science refers to the organization, management, and storage of data that enables efficient access and modification. It relies heavily on the data structure to find data in the fastest way. In other words, finding data is a matter of algorithms (which will be explained later), so a good data structure can be said to be a necessary and sufficient condition for a good algorithm.
- ▶ To be more specific, the data structure refers to a group of data values, a relationship between data, and a function or command applicable to the data. Carefully selected data structures make it possible to use more efficient algorithms.
- ▶ An effectively designed data structure allows operations to be performed with minimal resources, such as execution time or memory capacity.
- ▶ There are several types of data structures, each of which is tailored for each operation and purpose.
- ▶ When designing various programs, it should be the priority to consider and select the most appropriate data structure. This is because when manufacturing a large-scale system, the implementation difficulty and the final product's performance depend heavily on the data structure.
- ▶ Once the data structure is selected, it becomes relatively clear which algorithm needs to be applied. There are times when this order is reversed, and they are when the target operation necessarily requires a particular algorithm, and the given algorithm produces the best performance with the particular data structure. In any case, it is essential to select an appropriate data structure.

I Algorithm

- ▶ Third, an algorithm is a formulation of a set of procedures or methods to solve any solvable problem and refers to a step-by-step procedure for executing a calculation.
- ▶ Algorithms are the most important in the fields of machine learning and deep learning, areas we will cover in the future because it requires work with a lot of data.
- ▶ In the end, as the performance of the algorithm is directly related to the performance of the data structure, it can be said that it is most important to know precisely where to use a certain data structure.

Data Structures in Python

| First, the following are the types of data structures used in computer science:

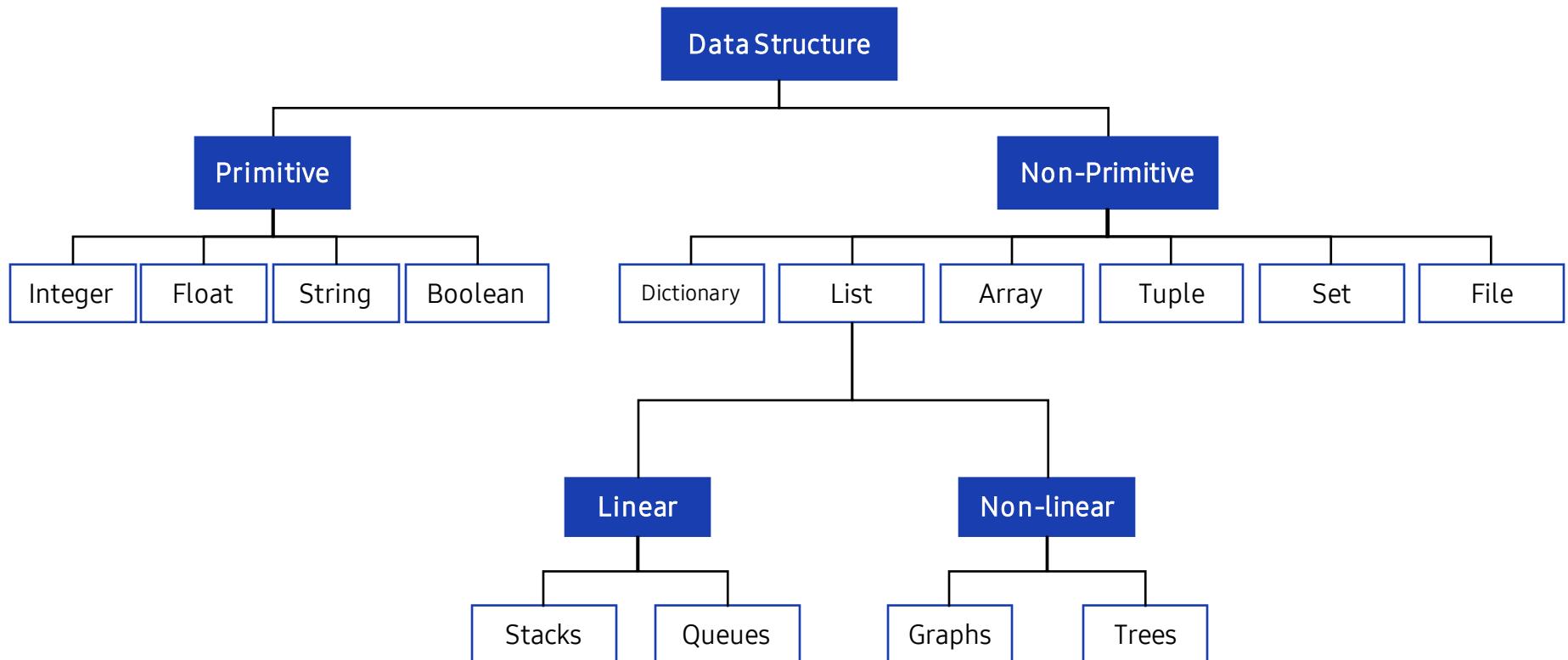


I Types of Data Structures (1/2)

- ▶ However, we will only concern ourselves with the data structure of the Python language here. In general, it can be largely divided into a primitive data structure, a structure for basic types, and a non-primitive data structure, a structure for effectively storing multiple data with basic data types.
- ▶ The most commonly and conveniently used non-primitive data structure in Python is the list. This is because the size of the data is not fixed while storing several different data types (integer, character, etc.)
- ▶ However, the advantage has become inappropriate in the fields of machine learning and deep learning, where more data must be quickly and exclusively processed in numbers.
- ▶ This is because computers must take decimal numbers understood by humans and convert them into binary forms to perform fast operations and use the same numerical data to directly access and calculate each element without repetition.
- ▶ In the end, computers can only perform operations when letters/characters are converted into numbers.

I Types of Data Structures (2/2)

- Compensating for such shortcomings is the array data structure. Python, however, does not directly support the array; rather, the array data structure can be used through the NumPy library.



I Method of Memory Storage Per Data Structure

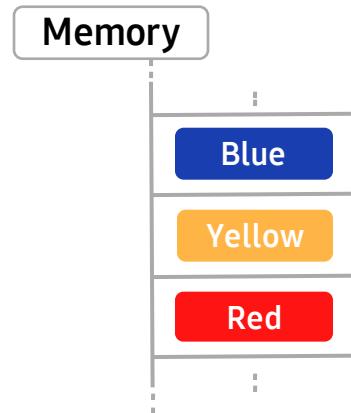
- ▶ To understand why array data structures enable such rapid operations, we first need to know how they are stored in memory for each data structure.
- ▶ Normally, for most languages, an array is referred to as a group of data created by listing data of the same data type and storing them contiguously in memory.
- ▶ Each value in it is called an element of an array. These elements use the number called an index, which always starts at zero, to simply distinguish an array's elements. Most data types can be configured in an arrangement and consist of a one-dimensional arrangement, a two-dimensional arrangement, and a three-dimensional arrangement, depending on the configuration type.
- ▶ Previously, it was explained that the data structure was a concept that focused primarily on effectively storing data. Finding out how the data are stored in the array and the list is the precise way to understand the array's pros and cons and know the reason for its use.

I Storage and Access of Data in an Array (1/2)

- ▶ The following is the process of accessing existing data when storing it in an array and adding and deleting new data. You should compare it with the list that follows.
- ▶ Suppose three pieces of data with strings representing the following colors are stored in an array.



- ▶ Each element may be accessed through each index. An index is a number representing an order. Data is sequentially stored in a contiguous location of memory, as shown in the figure below.



I Storage and Access of Data in an Array (2/2)

- Since the data is stored in a contiguous location, the address of the memory can be accessed with an index, and the data can be randomly selected to access the desired location of the data. Here is a picture where the data approaches the red in the third room (we will express the concept of a variable that mainly stores one value as a room) and approaches the blue room. Random access is possible through the index.

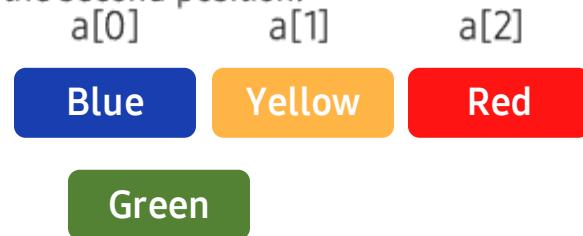


Random Access



I Adding and Deleting Data From the Array (1/4)

- ▶ Another feature of the array is that adding or deleting data to a specific location requires more computation and space in comparison to the list. The calculation here is that it takes more time for the CPU to calculate, and more space in memory is also needed.
- ▶ Consider adding the value “Green” to the second position.



- ▶ First, we need to secure additional space at the end of the arrangement.



I Adding and Deleting Data From the Array (2/4)

- To add data to the second space, the data behind the second space must move to the right one by one.

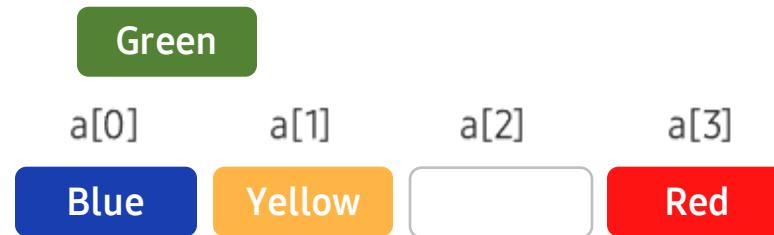


- The “Green” data is then added to the empty space.



I Adding and Deleting Data From the Array (3/4)

- Conversely, when deleting the second element—the “Green” value, remove the element first, then move the value to the left one by one so there is no empty space.



| Adding and Deleting Data From the Array (4/4)

- ▶ It is completed by deleting the last remaining space.



I Saving and Accessing Data From the List (1/4)

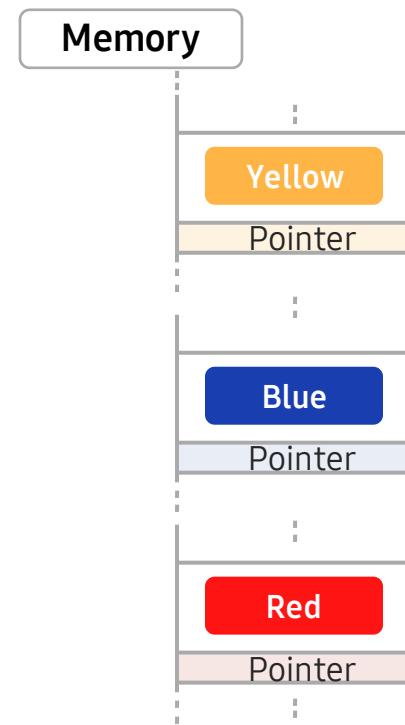
- The following is a method of storing data in the list data structure.



- To understand lists, you must first understand the concept of the pointer. In simple terms, the pointer is an address value that points to a certain value.
- As shown in the figure above, the blue room (the concept of a variable representing one value will be expressed as a room) points to the yellow room. Let's say the blue room has the address of the yellow room's memory location. Then each room can point to the next room.

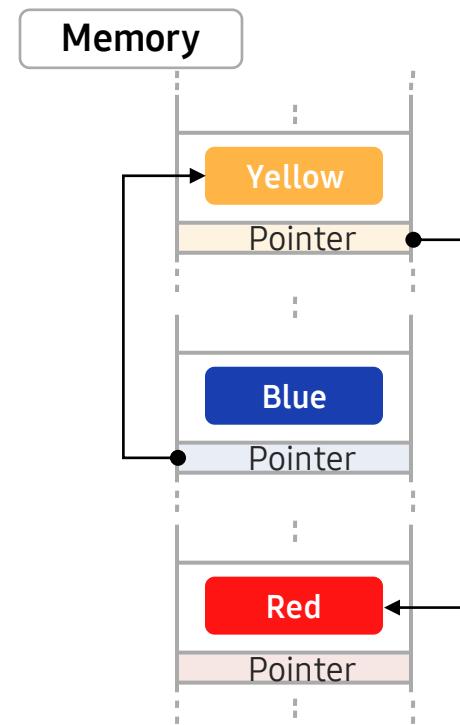
I Saving and Accessing Data From the List (2/4)

- ▶ Lists are not stored sequentially but in separate locations.



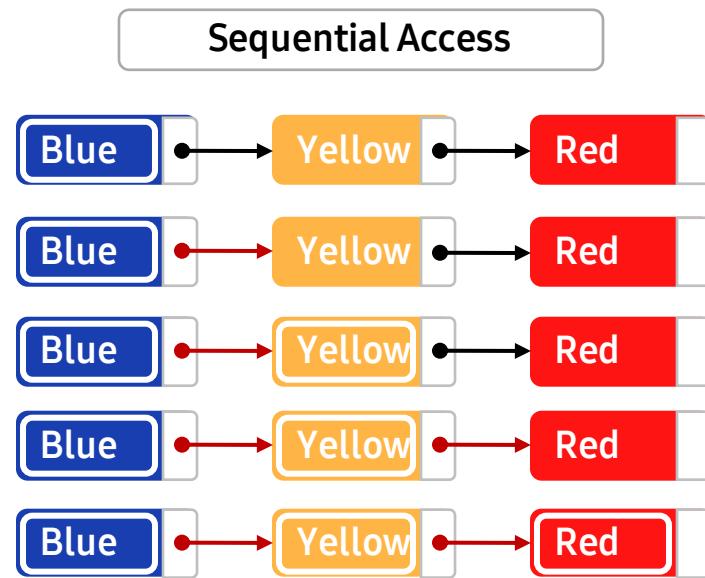
I Saving and Accessing Data From the List (3/4)

- Since the pointer of the “Blue” value refers to the address of the “Yellow” value, and the pointer of the yellow value refers to the “Red” value, the order may be maintained.



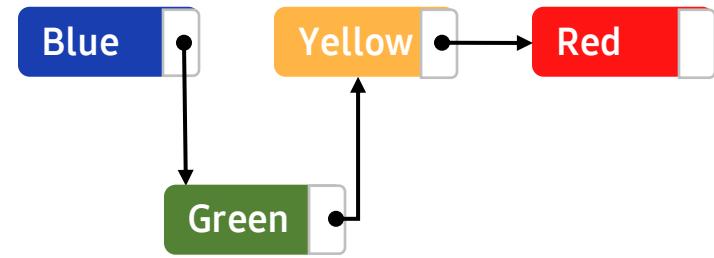
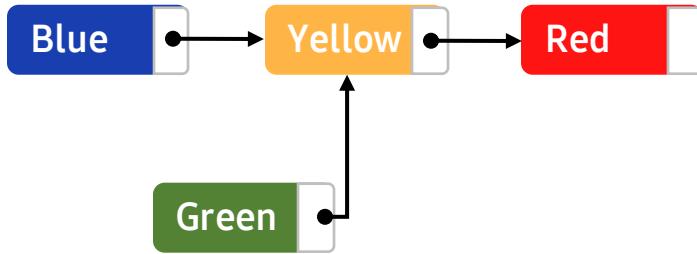
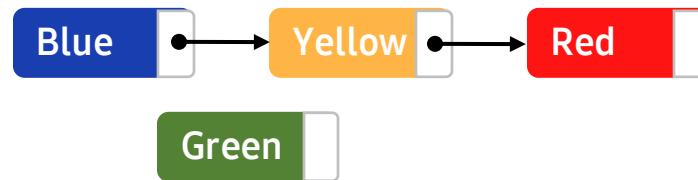
I Saving and Accessing Data From the List (4/4)

- Since the data is stored in non-contiguous addresses, sequential approaches are accessed through the pointer that precedes them.



I Adding and Deleting Data From the List

- Now, let's look at the case of adding the "Green" data.



- As shown in the picture above, using each pointer, the blue points to the green, and the green points to the yellow to add the green.

What is NumPy?

- | NumPy stands for Numerical Python, which is a basic package for Data Science.
 - ▶ It is a Python Library that provides multidimensional array objects, various derived objects (matrices, etc.), and an assortment of routines for fast operation on arrays.
 - ▶ It supports discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulations, etc.
 - ▶ The ndarray object is the core of the NumPy package. It processes n-dimensional arrays of homogenous data types.

| Distinguish the differences between NumPy arrays and standard Python sequences (Lists, Tuples, Dictionaries, Sets).

- ▶ NumPy arrays have a fixed size when generated as opposed to Python Lists (which can grow dynamically).
- ▶ A change in the size of the ndarray creates a new array and deletes the original.
- ▶ The elements of the NumPy array are all homogenous data types; thus, they maintain the same size in memory.
- ▶ Lists, however, allow for arrays of various sizes.
- ▶ NumPy arrays easily calculate advanced mathematical and other types of operations on large numbers of data.
- ▶ Normally, such operations are performed more efficiently and with less code than when performed with Python's built-in sequences.
- ▶ More and more scientific and mathematical Python-based packages are using NumPy arrays. These typically support Python sequence input, but they convert such input to NumPy arrays before processing and often output NumPy arrays.
- ▶ In other words, it is insufficient to rely only on Python's built-in sequences to efficiently use today's scientific/mathematical Python-based software. One also needs to have a knowledge of NumPy array to increase efficiency.

I Why Use NumPy Arrays? (1/3)

- ▶ Previously, we learned that Python does not directly support arrays but allows for data structure arrangement through the NumPy library.
- ▶ The arrangement concepts learned in standard programming languages are not too different in Python.
- ▶ In programming, decreasing loops is the method to increase performance.
- ▶ Loops in large-scale computations require a computation per repetition, causing poor performance.
- ▶ NumPy arrays allow for a wide variety of data processing operations through concise array operations as opposed to loops.
- ▶ The use of array computing to explicitly remove loops is called Vectorization. Mathematical operations for vectorized arrays are typically two to three, if not ten or even a hundred times faster than pure Python operations. Broadcasting, another method that we will learn later, is a very powerful vector operation.

I Why Use NumPy Arrays? (2/3)

- ▶ Since the NumPy array operation uses an internal iteration implemented in C, it is faster than the Python iteration and performs a linear algebraic operation using a vectorized operation.
- ▶ A vectorization operation is one of the linear transformations that transform a matrix into a vertical vector, as shown below.
- ▶ Since it works throughout the vector, it can be used instead of for and while loops.

$$\text{2x2 matrix } A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{ through vectorization become } \text{vec}(A) = \begin{bmatrix} a \\ c \\ b \\ d \end{bmatrix}$$

- ▶ In scikit-learn, the NumPy arrangement is the basic data structure. In other words, the NumPy arrangement should be used as the standard input/output for machine learning.

I Why Use NumPy Arrays? (3/3)

- ▶ In scikit-learn, the NumPy arrangement is the basic data structure. In other words, the NumPy arrangement should be used as the standard input/output for machine learning.
- ▶ Scikit-Learn is a Python machine learning library.
- ▶ Since scikit-learn receives data in the form of a NumPy array as input, all data to be used in the future must be converted into a NumPy array.

Unit 1.

NumPy Array Data Structure for Optimal Computational Performance

- | 1.1. NumPy Arrays
- | **1.2. NumPy Array Basics**
- | 1.3. NumPy Array Operations
- | 1.4. NumPy Indexing and Slicing
- | 1.5. Array Transposition and Axis Swap

How to Import NumPy

I NumPy Library Import

- ▶ For better readability of the code, abbreviate NumPy to np. This is a rule adopted by all those working with codes so that it can be easily understood.
- ▶ In Python, objects generally have properties and methods. The property is another Python object stored inside the object, and the method refers to a function that allows access to the internal data of the object. It can be approached in the form of np.attribute_name.

```
In [1]: import numpy as np
```

```
In [2]: np.__version__
```

```
Out[2]: '1.19.5'
```

Line 1

- As stands for alias, meaning it is abbreviated.

Line 2

- To view the version of NumPy, use the built-in property __version__.

NumPy Array Basics

I NumPy ndarray

- ▶ An n-dimensional array object is called a ndarray. It can be used to process and store large datasets.
- ▶ Fast and flexible arrangements use a similar grammar used for operations between scalar elements. They use mathematical operations for the entire data block.

I First, an arrangement may be made into a sequence (list, tuple, array, set).

```
In [1]: import numpy as np
```

```
In [2]: np.array([1,3,5,7,9])
```

```
Out[2]: array([1, 3, 5, 7, 9])
```

```
In [3]: arr1=np.array([1,3,5,7,9])
```

```
In [4]: type(arr1)
```

```
Out[4]: numpy.ndarray
```

| Here is one more arrangement.

```
In [5]: arr2=np.array([1,3,5,7,9])
```

| Check the address of each array using a function representing the address id.

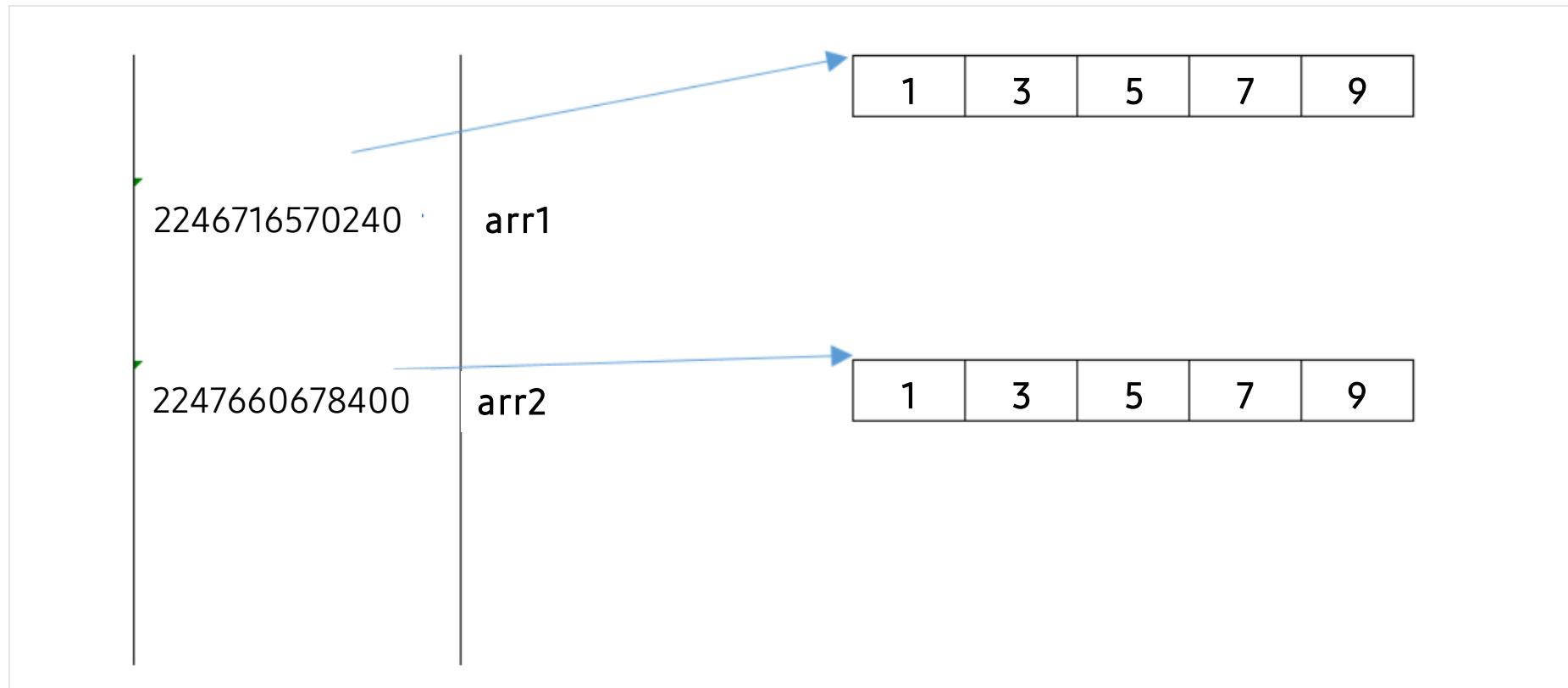
```
In [6]: id(arr1)
```

```
Out[6]: 2246716570240
```

```
In [7]: id(arr2)
```

```
Out[7]: 2247660678400
```

The actual value inside is the same, but the address is confirmed to be different because it is its own object.



- In Python, the `=` symbol is an assignment operator that assigns address values.

| Let's assign the address arr1 to the variable arr3.

```
In [8]: arr3=arr1
```

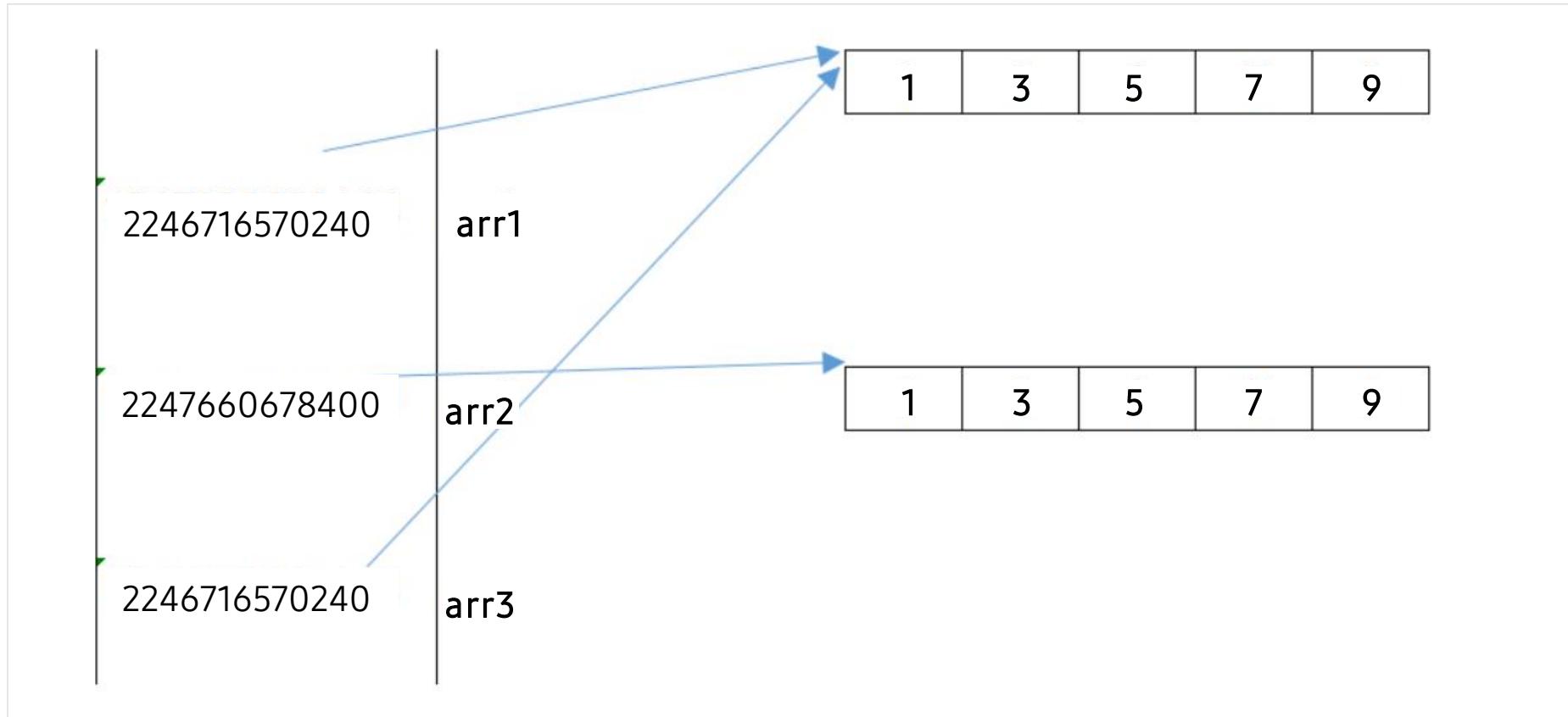
```
In [9]: print(id(arr1))  
print(id(arr3))
```

```
2246716570240  
2246716570240
```

 Line 9

- After assigning the address values, we can see that the address values are the same.

We can see that the address values are the same.



| If you want to copy only the value, you can use np.copy(). See help(np.copy) for more detail.

```
In [10]: arr4=np.copy(arr1)
```

```
In [11]: arr4
```

```
Out[11]: array([1, 3, 5, 7, 9])
```

| Then let's check the address value.

```
In [12]: id(arr4)
```

```
Out[12]: 2247660705472
```

Line 12

- Since only the value was copied, you can see that the address value is different.

I Creating Arrays with Tuples

```
In [13]: type((1,3,5,7,9))
```

```
Out[13]: tuple
```

```
In [14]: np.array((1,3,5,7,9))
```

```
Out[14]: array([1, 3, 5, 7, 9])
```

I Creating Arrays with Dictionaries

```
In [15]: type({'one':1,'two':2,'three':2 })
```

```
Out[15]: dict
```

```
In [16]: np.array({'one':1,'two':2,'three':2 })
```

```
Out[16]: array({'one': 1, 'two': 2, 'three': 2}, dtype=object)
```

| Creating Arrays with Sets

```
In [17]: {1,2,2,2,2,3,3,3,4,4}
```

```
Out[17]: {1, 2, 3, 4}
```

```
In [18]: type({1,2,2,2,2,3,3,3,4,4})
```

```
Out[18]: set
```

```
In [19]: np.array({1,2,2,2,2,3,3,3,4,4})
```

```
Out[19]: array([1, 2, 3, 4], dtype=object)
```

- ▶ Normally, lists are commonly used to create arrays.

I Creating Arrays with arange

- ▶ This is the same concept as the range in standard Python.

```
In [20]: [i for i in range(10)]
```

```
Out[20]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [21]: [i for i in np.arange(10)]
```

```
Out[21]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [22]: np.arange(10)
```

```
Out[22]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [23]: np.arange(1,10)
```

```
Out[23]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [24]: np.arange(1,10, 2)
```

```
Out[24]: array([1, 3, 5, 7, 9])
```

- ▶ For NumPy arrays, use np.arange.

| Use the len() method and the size property for the size of the array.

```
In [25]: len10=np.arange(10)
```

```
In [26]: len(len10)
```

```
Out[26]: 10
```

```
In [27]: len10.size
```

```
Out[27]: 10
```

| The NumPy array deals with each element by converting it to the same data type. (1/3)

- ▶ The important thing is that the NumPy arrays process the same type quickly and effectively, as one of its characteristics.
- ▶ However, although arrays can be made with different types of arrays, each type is converted into the same type.

```
In [28]: np.array([111, 2.3, True])
```

```
Out[28]: array([111., 2.3, 1.])
```

 Line 28

- Different types of arrays can be created, such as integers, floats, and Booleans.

| The NumPy array deals with each element by converting it to the same data type. (2/3)

- ▶ Due to the . in the integer type, the array is converted into a float when integer, float, and Boolean types are together.

```
In [29]: arr5=np.array([111,2.3,True])
```

```
In [30]: type(arr5[0])
```

```
Out[30]: numpy.float64
```

```
In [31]: arr6=np.array([111,2.3,'hi'])
```

Line 30

- The data type of NumPy is returned in NumPy.datatype format.

Line 31

- Different types of arrays are created, such as integers, floats, and strings.

| The NumPy array deals with each element by converting it to the same data type. (3/3)

```
In [32]: arr6[0]
```

```
Out[32]: '111'
```

```
In [33]: type(arr6[0])
```

```
Out[33]: numpy.str_
```

Line 33

- Due to the '111' in the integer type, the array is converted into a string when integer, float, and string types are together.

| The Data Types of NumPy Arrays are as follows:

Data Type	Explanation
int8, int16, int32, int64, int_ uint8, uint16, uint32, uint64	Integer
float16, float32, float64, float_	Floating point
bool_	Boolean
string_, unicode_	String

- ▶ The 8 in int8 is 8 bitss, which is the range of values that this value can represent.

<https://docs.scipy.org/doc/NumPy-1.17.0/reference/arrays.dtypes.html>

I Explanation of 8 bits

```
In [34]: 2**8
```

```
Out[34]: 256
```

- ▶ Since the bit represents two values, 8 bits can become 2 to the power of 8 , representing up to 256 integers. In integers, this value ranges from -128 to 127 . The range is up to 127 , and not 128 , because zero is excluded.
- ▶ If only positive values are used, the space in the range of negative numbers needs to be crossed over to positive numbers. At this time, u , the first letter of the unsigned (meaning positive), is used.
- ▶ Therefore, uint8 ranges from 0 to 255 .

| Data Types of NumPy Arrays

```
In [35]: np.array(['apple', 'banana', 'strawberry'])
```

```
Out[35]: array(['apple', 'banana', 'strawberry'], dtype='<U10')
```

```
In [36]: arr7=np.array(['apple', 'banana', 'strawberry'])
```

```
In [37]: type(arr7[0])
```

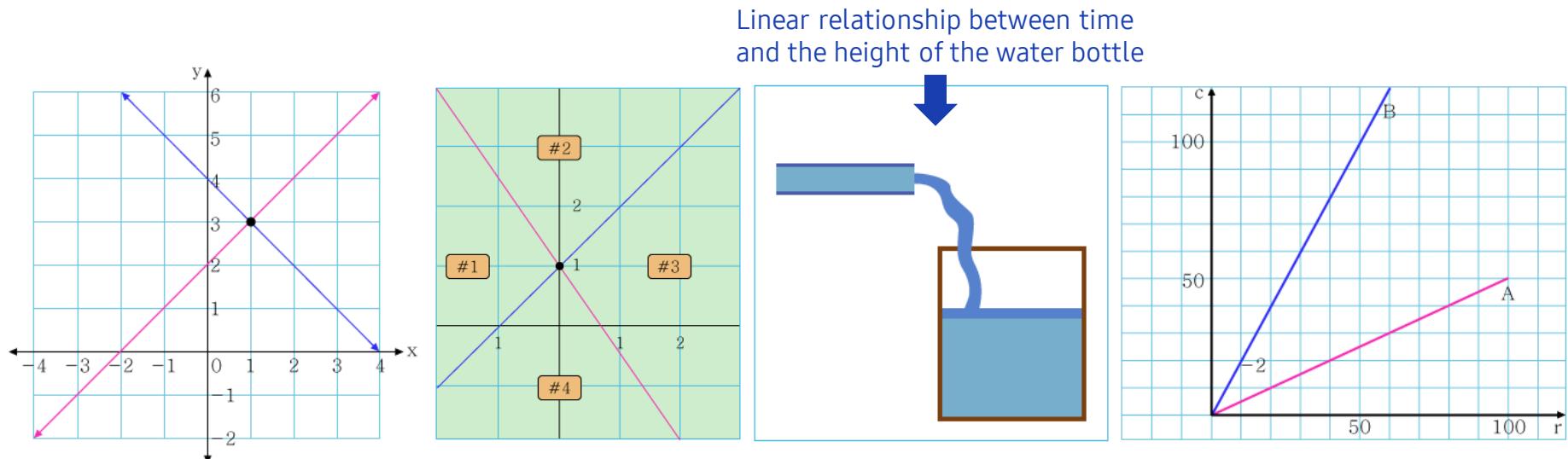
```
Out[37]: numpy.str_
```

Line 35

- The U10, here, represents Unicode, and the < symbol represents 10byte or less.

| Creating Arrays with the linspace Function (1/2) – See help(np.linspace) for more detail.

- ▶ Linear means that it can be expressed in the form of linear bonds with respect to the elements of set A.
- ▶ That is, where the elements of set A $x_1, x_2, x_3, \dots, x_n$ are multiplied and added by constants a_1, a_2, a_3, \dots , forming $a_1x_1 + a_2x_2 + a_3x_3 + \dots + a_nx_n$, which belongs to this set A. This type of expression $x_1, x_2, x_3, \dots, x_n$ is called a linear combination.
- ▶ In this way, representing the heat of the basic element in the form of a linear bond is called linear. Linear combination is the corresponding coefficients and variables, and the solution can be obtained through this equation.
- ▶ The figure below shows the concept of linearity in mathematics and daily life.



I Creating Arrays with the linspace Function (2/2)

```
In [38]: np.linspace(1, 10, 5)
```

```
Out[38]: array([ 1. ,  3.25,  5.5 ,  7.75, 10. ])
```

```
In [39]: np.linspace(-10, 10, 20)
```

```
Out[39]: array([-10.          , -8.94736842, -7.89473684, -6.84210526,
       -5.78947368, -4.73684211, -3.68421053, -2.63157895,
       -1.57894737, -0.52631579,  0.52631579,  1.57894737,
       2.63157895,  3.68421053,  4.73684211,  5.78947368,
       6.84210526,  7.89473684,  8.94736842, 10.        ])
```

Line 38

- Five numbers in the linear space from 1 to 10.

Line 39

- 20 numbers in the linear space from 10 to 10.

| Creating Arrays with np.zeros() and np.ones()

```
In [40]: np.zeros(5)
```

```
Out[40]: array([0., 0., 0., 0., 0.])
```

```
In [41]: np.ones(5)
```

```
Out[41]: array([1., 1., 1., 1., 1.])
```

Line 40

- Be careful not to skip the “s.”

Multidimensional Arrays

A multidimensional array refers to an array of two or more dimensions. It can be made in the form of a list in the list.

```
In [42]: li1=[[1,2,3],[4,5,6],[7,8,9]]
```

```
In [43]: li1[0][0]
```

```
Out[43]: 1
```

```
In [44]: np.array(li1)
```

```
Out[44]: array([[1, 2, 3],  
                 [4, 5, 6],  
                 [7, 8, 9]])
```

```
In [45]: arr2d=np.array(li1)
```

Line 43

- The first element in this list is also a list, so the first element in that list is 1.

Line 44

- Two-dimensional Array

| Check the dimensions of the array with ndim.

```
In [46]: arr2d.ndim
```

```
Out[46]: 2
```

| Then, let's check the one-dimensional array that we learned before.

```
In [47]: arr1d=np.array([1,3,5,7,9])
```

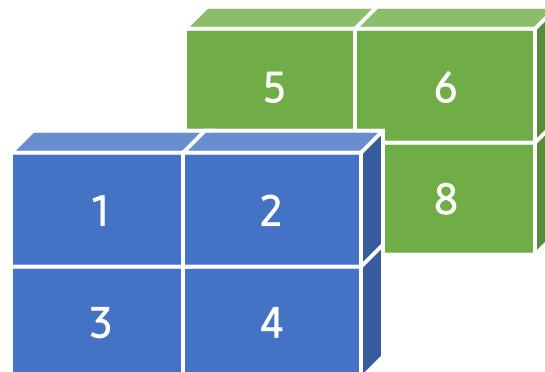
```
In [48]: arr1d.ndim
```

```
Out[48]: 1
```

We will also make a three-dimensional array.

```
In [49]: np.array([[[1,2],[3,4]], [[5,6],[7,8]]])
```

```
Out[49]: array([[[1, 2],  
[3, 4]],  
[[5, 6],  
[7, 8]])
```



I Three-Dimensional Array

```
In [50]: arr3d=np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
```

```
In [51]: arr3d.ndim
```

```
Out[51]: 3
```

```
In [52]: arr3d[0][0][0]
```

```
Out[52]: 1
```

Line 50

- A surface is two-dimensional and can be stacked. It's easy to understand if you think of the surface as a piece of paper.

Line 52

- When extracting a three-dimensional value, it's easier to think of it as a method of approaching step by step.

| Creating two-dimensional shapes using np.zeros()

```
In [53]: np.zeros((2,3))
```

```
Out[53]: array([[0., 0., 0.],  
                 [0., 0., 0.]])
```

Line 53

- Designate it as two rows and three columns in the form of a tuple.
- ▶ When created, the element within it is normally a float.

```
In [54]: arr2dZero=np.zeros((2,3))
```

```
In [55]: type(arr2dZero[0][0])
```

```
Out[55]: numpy.float64
```

| If it is created by a different type, designate the parameter name using dtype.

```
In [56]: arr2dZero2=np.zeros((2,3), dtype='int64')
```

```
In [57]: arr2dZero2
```

```
Out[57]: array([[0, 0, 0],  
                 [0, 0, 0]], dtype=int64)
```

```
In [58]: type(arr2dZero2[0][0])
```

```
Out[58]: numpy.int64
```

| Use the astype() method if you want to change the type to a float.

```
In [59]: arr2dZero2.astype('float32')
```

```
Out[59]: array([[0., 0., 0.],
 [0., 0., 0.]], dtype=float32)
```

```
In [60]: arr2dZero2=arr2dZero2.astype('float32')
```

```
In [61]: arr2dZero2
```

```
Out[61]: array([[0., 0., 0.],
 [0., 0., 0.]], dtype=float32)
```

Properties of the NumPy Array

I Explanation of Each Property

- ▶ Note that since it is not a method, () should not be attached.

```
In [62]: a=np.array([[1,2,3],[4,5,6],[7,8,9]])
```

```
In [63]: a.size
```

```
Out[63]: 9
```

```
In [64]: a.shape
```

```
Out[64]: (3, 3)
```

```
In [65]: a.ndim
```

```
Out[65]: 2
```

Line 63

- Size

Line 64

- Converting the row and columns into a tuple

| It should be noted that for one-dimensional arrays, the tuples return the number of elements.

```
In [66]: a1=np.array([2,5,1,3])
```

```
In [67]: a1.shape
```

```
Out[67]: (4,)
```

Reshape

I How to Reshape

```
In [68]: a2=np.arange(15)
```

```
In [69]: a2
```

```
Out[69]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [70]: a2.reshape((3,5))
```

```
Out[70]: array([[ 0,  1,  2,  3,  4],
   [ 5,  6,  7,  8,  9],
   [10, 11, 12, 13, 14]])
```

Line 70

- Be sure to input the shape as a tuple.

| Checking the shape of the object

```
In [71]: a2
```

```
Out[71]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [72]: a2=a2.reshape((3,5))
```

```
In [73]: a2
```

```
Out[73]: array([[ 0,  1,  2,  3,  4],  
                 [ 5,  6,  7,  8,  9],  
                 [10, 11, 12, 13, 14]])
```

Line 71

- When checked, the shape of the object remains the same. Thus, it should be assigned again to create a new object.

| The shape can be changed directly.

```
In [74]: a3=np.arange(15)
```

```
In [75]: a3
```

```
Out[75]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [76]: a3.shape=(5,3)
```

```
In [77]: a3
```

```
Out[77]: array([[ 0,  1,  2],
   [ 3,  4,  5],
   [ 6,  7,  8],
   [ 9, 10, 11],
   [12, 13, 14]])
```

Random Numbers

- | Data following the standard normal distribution (average 0, standard deviation 1) is generated as random numbers. See `help(np.random.randn)` for more detail.

```
In [78]: data=np.random.randn(2,3)
```

```
In [79]: data
```

```
Out[79]: array([[-2.14283012, -0.25184955, -0.89505598],  
                 [ 0.62480127, -0.42312272,  1.42986988]])
```

```
In [80]: np.mean(data)
```

```
Out[80]: -0.2763645366398019
```

Line 78

- The generated data follows the standard normal distribution with shapes in two rows and three columns.

| Creating data2 with 100 rows and 100 columns

```
In [81]: data2=np.random.randn(100,100)
```

```
In [82]: np.mean(data2)
```

```
Out[82]: -0.011834302170355068
```

```
In [83]: data2
```

```
Out[83]: array([[-0.18583609, -0.87218922, -1.19538247, ..., 0.18346856,
       -0.44277158, -1.19680885],
       [ 0.07765452, -1.09523774,  0.28383251, ..., 0.48052216,
       0.35715639, -0.26475694],
       [-0.178757 , -1.29948726, -1.3343085 , ..., -0.50077984,
      -2.5400508 ,  0.70543657],
       ...,
       [-0.37320999,  0.56076167, -2.54187684, ..., -1.20641404,
       0.14665862,  0.3484965 ],
       [-0.2870176 ,  0.63653664,  0.38456161, ..., -0.00658368,
      -0.12442503, -1.25921367],
       [-2.05915087,  2.22027685,  0.77467579, ..., -0.1043841 ,
      1.33209338,  0.53838072]])
```

| Go ahead and create data3 as well.

```
In [84]: data3=np.random.randn(100,100)  
np.mean(data3)
```

```
Out[84]: -0.0009906825764341783
```

```
In [85]: data3
```

```
Out[85]: array([[ 2.32654659,  0.08960113,  0.63360056, ..., -1.01526161,  
   0.5594331 , -0.55089798],  
   [ 0.08410731, -1.03712153, -0.87040639, ...,  1.16329921,  
   -0.81477079, -1.19383438],  
   [-0.21500342,  0.99080099,  0.3972265 , ...,  1.25634147,  
   0.17925074, -0.29003224],  
   ...,  
   [-0.57601374,  1.0244895 , -0.37536562, ...,  0.20769595,  
   -1.0117308 , -0.27108518],  
   [ 0.82518679,  0.75506147,  1.43082601, ..., -0.23724831,  
   1.0260131 ,  0.96915328],  
   [-0.20500495, -2.08376075,  1.30787372, ..., -0.36423519,  
   2.19855143, -0.13755722]])
```

- ▶ The average is close to zero but not completely zero. If more data is generated, then the results get closer to zero.

I Comparing Python Lists vs. NumPy Arrays: Processing Speed (1/4)

- ▶ Let's compare a Python list and a NumPy array that stores a million integers.
- ▶ If you run the example below, you can see that the code using NumPy is much faster than the code written with pure Python.
- ▶ By approaching np objects, denoted by abbreviating NumPy, you can create 1 million arrays starting from 0 using the arange method.

```
In [86]: np.arange(1000000)
```

```
Out[86]: array([    0,     1,     2, ..., 999997, 999998, 999999])
```

- ▶ Here is a data structure of the list for comparison's sake.

```
In [87]: list(range(1000000))
```

```
Out[87]: [0,  
1,  
2,  
3,  
4,  
5,  
6,  
7,  
8,
```

I Comparing Python Lists vs. NumPy Arrays: Processing Speed (2/4)

```
In [88]: type(np.arange(1000000))
```

```
Out[88]: numpy.ndarray
```

Line 88

- It returns the result in the form of NumPy.ndarray. This means that there is an array with n-dimensions.

I Comparing Python Lists vs. NumPy Arrays: Processing Speed (3/4)

- ▶ The following shows the speed of the Python array and the operation of multiplying each value of 1 million lists by 2,1000 times.

```
In [89]: %time for i in range(1000):np.arange(1000000)*2
```

Wall time: 2.69 s

- ▶ %time is an IPython magic command that returns a single execution time. This magic command is a special command designed to easily control general tasks and other operations in the IPython system.
- ▶ Magic commands are labeled with a % sign.

```
In [90]: %time for i in range(1000):list(range(1000000))*2
```

Wall time: 40.6 s

- ▶ It took 46.5 seconds. There is a difference of more than 20 times.
- ▶ Once again, it is important to understand the pros and cons of the arrays and lists we learned earlier to perform large-scale Big Data AI operations in the future. This would be a clear reason why we must learn and use NumPy when operating with such operations.

| Comparing Python Lists vs. NumPy Arrays: Processing Speed (4/4)

- ▶ The results may differ in each user's system, as we just saw on the screen.
- ▶ Wall time, also called Wall-clock time, is the sum of the entire time it takes for the program to run and end, including CPU, I/O, Sub Program, etc.
- ▶ To summarize vectorization once more, vectorization is one of the biggest reasons for using NumPy.
- ▶ Vectorization is the operation of placing data without using loops or indexing.
- ▶ NumPy's vectorization optimizes the placement operation with compiled C code by performing it out of sight. For example, the loop statement code for calculating the weight of machine learning uses NumPy vectorization.

Adding Elements to NumPy Arrays

- A value is added to the one-dimensional data. The rank of the first-dimension is 1. See help.(np.append) for more detail.

```
In [91]: a=np.array([1,2,3])
```

```
In [92]: b=np.append(a, [4,5,6])
```

```
In [93]: b
```

```
Out[93]: array([1, 2, 3, 4, 5, 6])
```

| NumPy.append() method (1/3)

- ▶ You can extend a NumPy array by calling the NumPy.append() method. Note that, unlike the lists, the plus '+' operator cannot be used for this purpose.
- ▶ When an array has orientations, you can specify the directions. For a 2D array, the axis argument set to 0 means vertical extension, while 1 means horizontal extension.
- ▶ NumPy.append() provides just a view. For the changes to remain, the result should be assigned to a variable.

I NumPy.append() method (2/3)

```
In [94]: a=np.array([[1,2],[3,4]])
```

```
In [95]: a
```

```
Out[95]: array([[1, 2],  
                 [3, 4]])
```

```
In [96]: b=np.append(a,[[9,9]] , axis=0)
```

```
In [97]: b
```

```
Out[97]: array([[1, 2],  
                 [3, 4],  
                 [9, 9]])
```

 Line 96

- axis=0 is a row.

I NumPy.append() method (3/3)

```
In [98]: c=np.append(a,[[9],[9]] , axis=1)
```

```
In [99]: c
```

```
Out[99]: array([[1, 2, 9],  
                 [3, 4, 9]])
```

Line 98

- axis=1 is a column.

Deleting Elements from NumPy Arrays

I Method of Deleting Elements from Arrays (1/2)

```
In [100]: a=np.array([[1,2,3],[4,5,6]])
```

```
In [101]: a
```

```
Out[101]: array([[1, 2, 3],  
                  [4, 5, 6]])
```

```
In [102]: np.delete(a,0)
```

```
Out[102]: array([2, 3, 4, 5, 6])
```

```
In [103]: a
```

```
Out[103]: array([[1, 2, 3],  
                  [4, 5, 6]])
```

Line 102

- Delete the corresponding index element.

Line 103

- Although the element was deleted, the array remains the same because no assignment was made.

I Method of Deleting Elements from Arrays (2/2)

```
In [106]: np.delete(a, 0, axis=0)
```

```
Out[106]: array([4, 5, 6])
```

```
In [107]: a
```

```
Out[107]: array([[1, 2, 3],  
                  [4, 5, 6]])
```

```
In [108]: np.delete(a, 1, axis=1)
```

```
Out[108]: array([[1, 3],  
                  [4, 6]])
```

Line 106

- Delete the first row.

Line 108

- Delete the second column.

Unit 1.

NumPy Array Data Structure for Optimal Computational Performance

- | 1.1. NumPy Arrays
- | 1.2. NumPy Array Basics
- | 1.3. NumPy Array Operations**

- | 1.4. NumPy Indexing and Slicing
- | 1.5. Array Transposition and Axis Swap

Basic Operations

- We learned earlier that the unique feature of the NumPy array operation is that data can be processed collectively without using a for loop statement, which is called vectorization. In this case, arithmetic operations between arrays of the same size are applied in each element unit of the array.
- ▶ For this reason, popular machine learning and deep learning libraries, such as scikit-learn, TensorFlow, PyTorch, etc., were all made based on NumPy.
 - ▶ This is a summary of the characteristics of NumPy that we've learned so far.
 - A low-level, high-performance library implemented in C
 - Supports fast, memory-effective multidimensional array ndarray operations
 - Supports various computational functions such as linear algebra, random number generator, Fourier transform, etc.

I Comparing Python Lists and NumPy Arrays: + Operator

```
In [1]: import numpy as np
```

```
In [2]: a=[1,2,3]
```

```
In [3]: b=[4,5,6]
```

```
In [4]: a+b
```

```
Out[4]: [1, 2, 3, 4, 5, 6]
```

Line 4

- In a list, the + operator means connecting the given elements.

| Comparing Python Lists and NumPy Arrays: + Operator

```
In [5]: arr1=np.array([1,2,3])  
arr2=np.array([4,5,6])
```

```
In [6]: arr1+arr2
```

```
Out[6]: array([5, 7, 9])
```

Line 6

- In a NumPy array, each element is calculated.

If the shape of the array is the same, you can perform addition, subtraction, multiplication, and division operations.

```
In [7]: print(arr1.ndim)
print(arr1.size)
print(arr1.shape)
```

```
1
3
(3,)
```

```
In [8]: print(arr2.ndim)
print(arr2.size)
print(arr2.shape)
```

```
1
3
(3,)
```

```
In [9]: print(arr1-arr2)
print(arr1*arr2)
print(arr1/arr2)
```

```
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4 0.5 ]
```

| Comparing Python Lists and NumPy Arrays: Multiplication

```
In [10]: a=[1,2,3]
3*a
```

```
Out[10]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
In [11]: b=np.array([1,2,3])
3*b
```

```
Out[11]: array([3, 6, 9])
```

Line 10

- In a list, the * operator means repetition.

Line 11

- Multiplication in a NumPy array returns the multiplication result of each element.

I repeat & tile

```
In [13]: b
```

```
Out[13]: array([1, 2, 3])
```

```
In [14]: np.repeat(b,3)
```

```
Out[14]: array([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

```
In [15]: np.tile(b,3)
```

```
Out[15]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

Line 14

- In NumPy, the repeat method is used to repeat each element.

Line 15

- In NumPy, the tile method is used for repeating the array.

| When using the tile method, the array can be repeated into a two-dimensional form.

```
In [16]: c=np.arange(9)
```

```
In [17]: c=c.reshape(3,3)
```

```
In [18]: c.shape
```

```
Out[18]: (3, 3)
```

```
In [19]: c
```

```
Out[19]: array([[0, 1, 2],  
                 [3, 4, 5],  
                 [6, 7, 8]])
```

```
In [20]: np.tile(c,3)
```

```
Out[20]: array([[0, 1, 2, 0, 1, 2, 0, 1, 2],  
                  [3, 4, 5, 3, 4, 5, 3, 4, 5],  
                  [6, 7, 8, 6, 7, 8, 6, 7, 8]])
```

Arrays can also perform exponential operations.

```
In [22]: arr2
```

```
Out[22]: array([4, 5, 6])
```

```
In [23]: arr2**2
```

```
Out[23]: array([16, 25, 36], dtype=int32)
```

```
In [24]: arr1 / (arr2**2)
```

```
Out[24]: array([0.0625    , 0.08      , 0.08333333])
```

The array can also perform comparison operations. After checking whether each element matches the conditions, it returns True if there is a match and False if otherwise.

```
In [25]: arr3=np.array([10,20,30,40])
```

```
arr3
```

```
In [26]: arr3
```

```
Out[26]: array([10, 20, 30, 40])
```

```
In [27]: arr3 >10
```

```
Out[27]: array([False,  True,  True,  True])
```

I Universal Functions (1/3)

- Universal functions operate element by element on whole arrays. See `help(np.ufunc)` for more detail.

```
In [28]: x=np.array([0,1,2,3])
```

```
In [29]: x**3
```

```
Out[29]: array([ 0,  1,  8, 27], dtype=int32)
```

```
In [30]: np.sqrt(x)
```

```
Out[30]: array([0.          , 1.          , 1.41421356, 1.73205081])
```

I Universal Functions (2/3)

- ▶ NumPy functions:

Function	Explanation	Universal function?
sin, cos, tan	Trigonometric functions	Yes
arcsin, arccos, arctan	Inverse trigonometric functions	Yes
round	Round to a given number of decimals	Yes
floor	Returns the nearest smaller integer	Yes
ceil	Returns the nearest greater integer	Yes
fix	Returns the nearest integer closer to 0	Yes
prod	Returns the product of the array elements	No
cumsum	Returns the cumulative sums of an array	No
sum, mean, var, std, median	Statistical functions	No
exp, log	Exponential and logarithmic functions	Yes
unique	Returns the unique values of an array	No
min, max, argmax, argmin	Minimum, maximum, and the corresponding indices	No

<https://docs.scipy.org/doc/NumPy-1.17.0/reference/>

I Universal Functions (3/3)

- ▶ Statistical methods of NumPy arrays:

Method	Explanation
mean	Average
var	Variance
std	Standard deviation
sum	Total sum
cumsum	The cumulative sums of an array
max, min	The maximum and the minimum of an array
argmax, argmin	Indices of the maximum and the minimum

<https://docs.scipy.org/doc/NumPy-1.17.0/reference/>

| Statistical methods of NumPy arrays:

```
In [31]: np.arange(1,11)
```

```
Out[31]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [32]: x=np.arange(1,11)
```

```
In [33]: x.sum()
```

```
Out[33]: 55
```

```
In [34]: x.mean()
```

```
Out[34]: 5.5
```

Line 33

- Sum

Line 34

- Average

| Statistical methods of NumPy arrays:

```
In [35]: x_mean=x.mean()
```

```
In [36]: x-x_mean
```

```
Out[36]: array([-4.5, -3.5, -2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5,  4.5])
```

```
In [37]: (x-x_mean)**2
```

```
Out[37]: array([20.25, 12.25,  6.25,  2.25,  0.25,  0.25,  2.25,  6.25, 12.25,
 20.25])
```

Line 35

- Average 5.5

Line 36

- Deviation (difference from average)

Line 37

- Square of Deviation

| Statistical methods of NumPy arrays:

```
In [38]: dev=(x-x_mean)**2
```

```
In [39]: dev.sum()
```

```
Out[39]: 82.5
```

Line 39

- Sum of Squared Deviation

| Statistical methods of NumPy arrays:

```
In [40]: dev.sum() / x.size  
Out[40]: 8.25
```

Line 40

- The sum of the squared deviations divided by the number of arrays is called a variance.
- ▶ This result can be calculated directly through the process above, or the var() can be used.

```
In [41]: x.var()  
Out[41]: 8.25
```

```
In [42]: np.sqrt(x.var())  
Out[42]: 2.8722813232690143
```

Line 42

- The square root of the variance is the standard deviation.

| Statistical methods of NumPy arrays:

```
In [40]: dev.sum() / x.size
```

```
Out[40]: 8.25
```

Line 40

- The sum of the squared deviations divided by the number of arrays is called a variance.
- ▶ This result can be calculated directly through the process above, or the std() method can be used.

```
In [43]: x.std()
```

```
Out[43]: 2.8722813232690143
```

| Statistical methods of NumPy arrays:

```
In [44]: x.cumsum()
```

```
Out[44]: array([ 1,  3,  6, 10, 15, 21, 28, 36, 45, 55], dtype=int32)
```

Line 44

- The cumsum() method is used to find the cumulative sum of x.

| Statistical methods of NumPy arrays:

```
In [45]: x=np.arange(1,7)
```

```
In [46]: x
```

```
Out[46]: array([1, 2, 3, 4, 5, 6])
```

```
In [47]: x.reshape(2,3)
```

```
Out[47]: array([[1, 2, 3],  
                 [4, 5, 6]])
```

```
In [48]: x.reshape(2,-1)
```

```
Out[48]: array([[1, 2, 3],  
                 [4, 5, 6]])
```

```
In [49]: x.reshape(3,-1)
```

```
Out[49]: array([[1, 2],  
                 [3, 4],  
                 [5, 6]])
```

Line 47

- In this way, you can designate it as two rows and three columns. However, if you designate only a row, use -1 to change the rest on its own.

| Statistical methods of NumPy arrays:

- ▶ Statistical functions can also be used for each row and column.

```
In [50]: x=x.reshape(2,3)
```

```
In [51]: x
```

```
Out[51]: array([[1, 2, 3],  
                 [4, 5, 6]])
```

```
In [52]: x.mean(axis=0)
```

```
Out[52]: array([2.5, 3.5, 4.5])
```

```
In [53]: x.mean(axis=1)
```

```
Out[53]: array([2., 5.])
```

Line 52

- The average of 1 and 4 is 2.5. When the axis = 0, the operation is carried out along the columns.

Line 53

- The average of 1, 2, and 3 is 2. When the axis = 1, the operation is carried out along the rows.

| About Random Numbers in NumPy

```
In [54]: np.random.seed(123)
```

Line 54

- Initializing the seed value using np.random will generate a fixed random number when generating random number values.

| About Random Numbers in NumPy

- ▶ Return random integers from 'low' (inclusive) to 'high' (exclusive). Refer to help(np.random.randint).

```
In [55]: np.random.randint(10)
```

```
Out[55]: 2
```

```
In [56]: np.random.randint(1,11)
```

```
Out[56]: 3
```

Line 55

- A random number is returned from integers 0 to 9.

Line 56

- This is how you would return a random number from integers 1 to 10.

| About Random Numbers in NumPy

- ▶ Write the code below to check if the code will return 10 every time.

```
In [57]: while True:  
    num=np.random.randint(1,11)  
    print(num)  
    if(num==10):  
        break
```

```
7  
2  
4  
10
```

| About Random Numbers in NumPy

```
In [58]: while True:  
    num=np.random.randint(1,11)  
    print(num)  
    if(num==11):  
        break
```

```
8  
7  
10  
4  
4  
2  
1
```

Line 58

- In this case, 11 is not returned, so it repeats indefinitely. Click (interrupt kernel) to stop the repetition.

| Final Summary

- ▶ Universal functions:

```
In [59]: x = np.array([0, 1, 2, 3, 4, 5])
```

```
In [60]: np.sqrt(x)
```

```
Out[60]: array([0.          , 1.41421356, 1.73205081, 2.          ,  
                2.23606798])
```

```
In [61]: np.exp(x)
```

```
Out[61]: array([ 1.          , 2.71828183, 7.3890561 , 20.08553692,  
                54.59815003, 148.4131591])
```

| Final Summary

- ▶ NumPy functions:

```
In [62]: np.random.seed(123)
a = np.random.randint(1, 11, size=1000)
```

```
In [63]: a
```

```
Out[63]: array([ 3,  3,  7,  2,  4, 10,  7,  2,  1,  2, 10,  1,  1, 10,  4,  5,  1,
  1,  5,  2,  8,  4,  3,  5,  8,  3,  5,  9,  1,  8, 10,  4,  5,  7,
  2,  6,  7,  3,  2,  9,  4,  6,  1,  3,  7,  3,  5,  5,  7,  4,  1,
  7,  5,  8,  7,  8,  2,  6,  8, 10,  3,  5,  9,  2,  3,  4,  4,  4,  9,
  6, 10,  1,  9,  2,  7,  4,  4,  6, 10,  8, 10,  3,  4,  4,  4,  4,  9,
  7, 10,  8,  7,  4, 10,  7,  7,  2,  4,  5,  4,  2,  1,  6,  9,
  7,  9, 10,  2,  1,  4,  2,  4,  5,  8,  7,  2,  5,  4,  4,  8,  7,
  9,  7,  5,  5,  8,  1,  1, 10,  9,  9,  5,  9,  7,  2,  7,  9,  8,
 10,  2,  8,  2,  8, 10,  9,  8,  2,  4,  2,  9,  8,  6,  2,  3,  6,
 3,  3, 10,  4,  3,  7,  8, 10,  2,  4,  9,  4,  8, 10, 10,  4,  4,
 6,  7,  1,  9,  8,  8,  5,  5,  6,  1,  9, 10,  3,  6,  2,  6, 10,
 3,  5,  4,  1,  4,  8,  8,  3,  6,  2,  8,  6, 10,  2,  3,  9,  6,
 1, 10,  4,  4,  2,  8,  7,  4,  2,  8,  3,  7,  6, 10,  9, 10,
 5,  8,  9,  6, 10,  3,  5,  4,  4,  2,  8,  9,  6,  8,  8,  8,  6,
 1,  1,  9,  2,  1,  5,  6,  5,  6,  1,  8,  2,  3,  3,  8,  8,  4,
 7,  2,  9,  5,  2,  7,  6, 10,  1,  4,  2,  4,  8,  8,  8,  6,  4,
 7,  4,  6,  2,  6, 10,  2,  4, 10,  2,  6,  7,  2,  6,  1,  8,  1,
 4,  1,  7,  3,  8,  6,  2,  1,  1,  3,  7,  6, 10,  2,  6,  4,  7,
 5,  8,  2,  7,  1,  4,  2,  4,  1,  9,  6,  2,  5,  4,  3,  3,  1,
 3,  1,  3,  8, 10,  1,  4, 10,  8,  1,  6,  5,  7,  1,  8,  9,  9,
 2,  1,  2,  8,  6,  4,  1,  5,  9,  5, 10,  4,  6,  2,  9,  1, 10,
 2,  6,  9,  9,  5,  7,  9,  8, 10,  3,  1,  1, 10, 10,  6,  4,  6,
 6,  3,  3,  8,  4,  2,  4, 10,  4,  7,  3,  4,  2, 10,  9,  1,  3,
 4,  8, 10,  3,  8, 10,  8,  2,  5,  8,  5,  8,  9,  1,  3,  8,  5,
 4,  8,  4,  8,  1,  8,  7,  9,  7,  9,  1,  6,  4,  3,  5,  4,  4,
10,  8,  2,  9,  1,  2,  5,  8,  3,  5,  0,  4,  5, 10,  7,  3,  8]
```

```
In [64]: a.size
```

```
Out[64]: 1000
```

| Final Summary

- ▶ NumPy functions:

```
In [64]: a.size
```

```
Out[64]: 1000
```

```
In [65]: np.unique(a)
```

```
Out[65]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [66]: np.median(a)
```

```
Out[66]: 5.0
```

```
In [67]: np.sum(a)
```

```
Out[67]: 5485
```

Line 65

- Returns only unique values that are not repeated.

| Final Summary

- ▶ NumPy functions:

```
In [68]: np.round(np.var(a),2)
```

```
Out[68]: 8.19
```

```
In [69]: np.round(np.mean(a),2)
```

```
Out[69]: 5.48
```

```
In [70]: np.round(np.std(a),2)
```

```
Out[70]: 2.86
```

Line 68

- Rounded up to the second decimal place.

| Final Summary

- ▶ NumPy functions:

```
In [71]: np.max(a)
```

```
Out[71]: 10
```

```
In [72]: np.min(a)
```

```
Out[72]: 1
```

| Final Summary

- ▶ NumPy functions:

```
In [73]: z = np.random.randint(10, size=1000)
```

```
In [74]: np.unique(z)
```

```
Out[74]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [75]: set(z)
```

```
Out[75]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

| Final Summary

- ▶ Universal functions:

```
In [76]: a = np.array([2,-3,4,5,7,0,1,-1])
```

```
In [77]: a.argmax()
```

```
Out[77]: 4
```

```
In [78]: a.argmin()
```

```
Out[78]: 1
```

Line 77

- Returns the index location with the highest value into an integer.

Line 78

- Returns the index location with the lowest value into an integer.

| Linear Algebra of the NumPy Array

- ▶ NumPy supports not only simple operations of arrays but also matrix (two-dimensional array) operations for linear algebra.
- ▶ Among the various methods, let's first look at how to obtain matrix multiplication, transpose matrix, inverse matrix, and determinant.

I Matrix Operations

- ▶ The following is the method for obtaining matrix product, transpose matrix, inverse matrix, and determinant for matrices A and B.
 - matrix product
 - transpose matrix
 - inverse matrix
 - determinant

I Matrix Operations

- ▶ For matrix operations, make 2x2 matrices A and B as follows:

```
In [79]: A = np.array([0, 1, 2, 3]).reshape(2,2)  
A
```

```
Out[79]: array([[0, 1],  
                 [2, 3]])
```

```
In [80]: B = np.array([3, 2, 0, 1]).reshape(2,2)  
B
```

```
Out[80]: array([[3, 2],  
                 [0, 1]])
```

I Matrix Operations

- ▶ Example of the product of matrices A and B ($A \cdot B$). Both methods can be used.

```
In [81]: A.dot(B)
```

```
Out[81]: array([[0, 1],  
                 [6, 7]])
```

```
In [82]: np.dot(A,B)
```

```
Out[82]: array([[0, 1],  
                 [6, 7]])
```

I Matrix Operations

- The following is an example of finding the transpose matrix of matrix A. Both methods can be used.

```
In [83]: np.transpose(A)
```

```
Out[83]: array([[0, 2],  
                 [1, 3]])
```

```
In [84]: A.transpose()
```

```
Out[84]: array([[0, 2],  
                 [1, 3]])
```

I Matrix Operations

- ▶ How to find the inverse matrix of matrix A

```
In [87]: np.linalg.inv(A)
```

```
Out[87]: array([[-1.5,  0.5],  
                 [ 1. ,  0. ]])
```

- ▶ An example of obtaining a determinant of matrix A

```
In [88]: np.linalg.det(A)
```

```
Out[88]: -2.0
```

Coding Exercise #0101



Follow practice steps on 'ex_0101.ipynb' file

Coding Exercise #0102



Follow practice steps on 'ex_0102.ipynb' file

Unit 1.

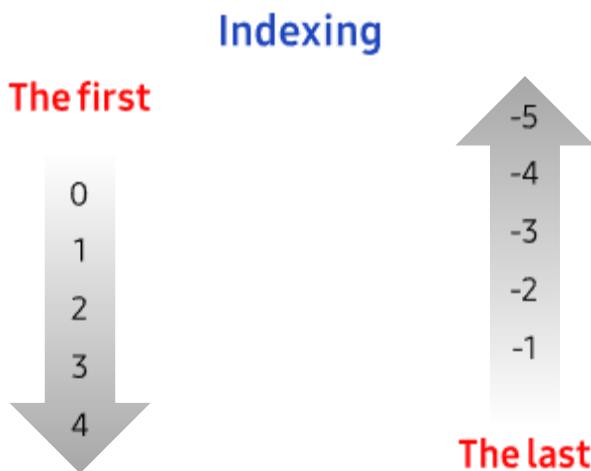
NumPy Array Data Structure for Optimal Computational Performance

- | 1.1. NumPy Arrays
- | 1.2. NumPy Array Basics
- | 1.3. NumPy Array Operations

- | 1.4. NumPy Indexing and Slicing
- | 1.5. Array Transposition and Axis Swap

Indexing and Slicing

- >Selecting elements in an array by specifying the location or condition of an array is called indexing. And selecting elements in an array by specifying a range is called slicing.
- ▶ To select an element at a specific location in a one-dimensional array, the element's position must be specified as shown below, and the element in the array starts at 0.
 - ▶ Array Name[Location]



I Indexing

- ▶ Indexing a one-dimensional array is like Python list indexing.

```
In [1]: import numpy as np
```

```
In [2]: a=np.array([0,1,2,3])
b=np.array([[0,1,2,3],
            [4,5,6,7],
            [8,9,10,11]])
```

```
In [3]: a[0]
```

```
Out[3]: 0
```

I Indexing

- ▶ Multi-dimensional arrays can also be approached like list indexing.

```
In [4]: b[0]
```

```
Out[4]: array([0, 1, 2, 3])
```

```
In [5]: b[0][1]
```

```
Out[5]: 1
```

Line 5

- Can access one specific multi-dimensional value.

I Indexing

- ▶ Unlike list indexing, dimensions can be divided into commas (,) and approached in a multidimensional array. The dimension divided by commas is called an axis. Since b is a two-dimensional array, we can treat it as a matrix. Values in columns 1 and 2 can be accessed as on the previous page.
- ▶ It can also be approached as shown below.

```
In [6]: b[0,1]
```

```
Out[6]: 1
```

I Indexing

- Not only can the array's elements be imported, but the value of the corresponding index can be changed as follows.

```
In [7]: a1=np.array([0,10,20,30,40,50])  
a1[5]
```

```
Out[7]: 50
```

```
In [8]: a1[5]=70
```

```
In [9]: a1
```

```
Out[9]: array([ 0, 10, 20, 30, 40, 70])
```

| Indexing

- ▶ To select multiple elements from a one-dimensional array, store them as follows.
- ▶ The outer brackets are brackets for indexing, and the inner brackets are brackets for the list.
- ▶ Array name [[]]
- ▶ Array name [[position 1, position 2, ..., position n]]

```
In [10]: a1[[1,3,4]]
```

```
Out[10]: array([10, 30, 40])
```

Line 10

- Elements 10, 30, and 40 located at positions 1, 3, and 4 were taken from the one-dimensional array a1.

I Indexing

- ▶ To select an element at a specific location in a two-dimensional array, specify the positions of rows and columns as follows.
- ▶ Array name [row_position, column_position]
- ▶ If only the array name [row_position] is entered without a “column_position,” the entire designated row is selected.
- ▶ It is a method of selecting and importing a specific element by indexing in a two-dimensional array.

```
In [11]: a2=np.arange(10,100,10).reshape(3,3)
a2
```

```
Out[11]: array([[10, 20, 30],
 [40, 50, 60],
 [70, 80, 90]])
```

I Indexing

- In the two-dimensional array a2, an element having a row_position of 0 and a column_position of 2 is selected and imported as follows.

```
In [12]: a2[0, 2]
```

```
Out[12]: 30
```

- As shown below, you can also change the value after selecting an element by specifying the positions of rows and columns in a two-dimensional array.

```
In [13]: a2[2,2]=95  
a2
```

```
Out[13]: array([[10, 20, 30],  
                 [40, 50, 60],  
                 [70, 80, 95]])
```

I Indexing

- A method of obtaining the entire row by specifying the “row_position” in a two-dimensional array is shown below.

```
In [14]: a2[1]
```

```
Out[14]: array([40, 50, 60])
```

- The entire row can also be changed by specifying a specific row in a two-dimensional array, as shown below.

```
In [15]: a2[2]=np.array([45,55,65])
a2
```

```
Out[15]: array([[10, 20, 30],
                 [40, 50, 60],
                 [45, 55, 65]])
```

I Indexing

- ▶ To select several elements in a two-dimensional array, do as follows.
- ▶ Array name [[row_position 1, row_position 2, ..., row_position n],[column_position n], column_position 2, ..., column_position n]]
- ▶ The following is an example of selecting multiple elements by specifying the positions of rows and columns in a two-dimensional array.

```
In [16]: a2
```

```
Out[16]: array([[10, 20, 30],  
                 [40, 50, 60],  
                 [45, 55, 65]])
```

```
In [17]: a2[[0,2],[0,1]]
```

```
Out[17]: array([10, 55])
```

Line 17

- If you select the row first from [10, 20, 30] and the column from [45, 55, 65], then 10 and 55 are returned.

I Selecting the Array by Specifying the Conditions

- ▶ Array Name[Condition]

```
In [18]: a=np.array([1,2,3,4,5,6])
```

```
In [19]: a[a>3]
```

```
Out[19]: array([4, 5, 6])
```

```
In [20]: a[a%2==0]
```

```
Out[20]: array([2, 4, 6])
```

Line 19

- Only elements that meet the conditions of “`a>3`” are returned.

Line 20

- Only even numbers are returned.

Array Slicing

- Instead of selecting one element through indexing, slicing selects a portion of the array by specifying a range.
 - For a one-dimensional array, slicing specifies the positions of the beginning and the end, as shown below.
 - Array[start_position]:end_position]
 - If the start position is not specified, the start position becomes 0, and the range becomes "0 to end position -1." If the end position is not specified, the "end position" becomes the array's length, and the range becomes "start_position to end of the array."

```
In [21]: b1=np.array([0,10,20,30,40,50])
          b1[1:4]

Out[21]: array([10, 20, 30])
```

| Slicing

- In a one-dimensional array, slicing is performed without specifying a "start_position" and an "end_position," as shown below.

```
In [22]: b1[:3]
```

```
Out[22]: array([ 0, 10, 20])
```

```
In [23]: b1[2:]
```

```
Out[23]: array([20, 30, 40, 50])
```

Slicing

- Now, let's consider the case for a two-dimensional array.

```
In [25]: arr2d=np.arange(1,10).reshape(3,3)
```

```
In [26]: arr2d
```

```
Out[26]: array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])
```

- Try slicing the array above.

```
In [27]: arr2d[:2]
```

```
Out[27]: array([[1, 2, 3],
 [4, 5, 6]])
```

Line 27

- Choose from the beginning to the second line of "arr2d."

Slicing

- It is also possible to slice multi-dimensional arrays by crossing several indices.

```
In [28]: arr2d[:2, 1:]
```

```
Out[28]: array([[2, 3],  
                 [5, 6]])
```

- As shown above, slicing always gives a view of the array in the same dimension. An integer index and a slice can be used together to obtain a lower-dimension slice.
- For example, if you want to select only the first two columns in the second row, do as follows.

```
In [29]: arr2d[1, :2]
```

```
Out[29]: array([4, 5])
```

Slicing

- If you select only the third column in the first two rows, do as follows.

```
In [30]: arr2d[:,2]
```

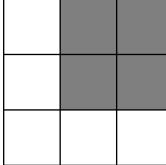
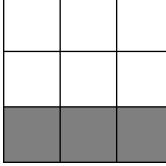
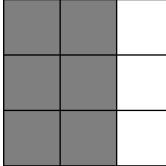
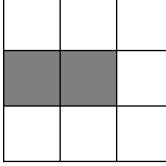
```
Out[30]: array([3, 6])
```

- If you just use a colon, you choose the entire axis, so the slice of the original dimension is returned.

```
In [31]: arr2d[:,1]
```

```
Out[31]: array([[1],  
                 [4],  
                 [7]])
```

| Slicing

	Code	Form of Slice
	<code>arr[:2, 1:]</code>	(2,2)
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	(3,) (3,) (1,3)
	<code>arr[:, :2]</code>	(3,2)
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	(2,) (1,2)

I Selecting in Boolean Values

```
In [32]: names=np.array(['Bob','Joe','Tom','Bob','Tom','Joe','Joe'])
```

```
In [33]: names
```

```
Out[33]: array(['Bob', 'Joe', 'Tom', 'Bob', 'Tom', 'Joe', 'Joe'], dtype='<U3')
```

- ▶ This Boolean array determines whether the name “Bob” is in the names array.

```
In [34]: names=='Bob'
```

```
Out[34]: array([ True, False, False,  True, False, False])
```

- ▶ By indexing under this condition, only those whose name is “Bob” can be returned.

```
In [35]: names[names=='Bob']
```

```
Out[35]: array(['Bob', 'Bob'], dtype='<U3')
```

Fancy Indexing

| Return a new array of given shape and type without initializing entries. See help(np.empty) for more detail.

```
In [36]: np.empty((8,4))
```

```
Out[36]: array([[0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.]])
```

```
In [37]: arr=np.empty((8,4))
```

```
In [38]: for i in range(8):
 arr[i]=i
```

```
In [39]: arr
```

```
Out[39]: array([[0., 0., 0., 0.],
 [1., 1., 1., 1.],
 [2., 2., 2., 2.],
 [3., 3., 3., 3.],
 [4., 4., 4., 4.],
 [5., 5., 5., 5.],
 [6., 6., 6., 6.],
 [7., 7., 7., 7.]])
```

I Fancy Indexing

- ▶ Selecting rows in a specific order can skip the ndarray or the list containing the desired order.

```
In [40]: arr[[4,3,0,6]]
```

```
Out[40]: array([[4., 4., 4., 4.],
 [3., 3., 3., 3.],
 [0., 0., 0., 0.],
 [6., 6., 6., 6.]])
```

- ▶ If you use negative numbers as an index, select a row from the end.

```
In [41]: arr[[-3, -5]]
```

```
Out[41]: array([[5., 5., 5., 5.],
 [3., 3., 3., 3.]])
```

I Fancy Indexing

- To index values that are only multiples of 5, the Boolean array for the condition can be put into the variable and obtained.

```
In [42]: arr2=np.arange(20)
```

```
In [43]: arrMask=(arr2%5==0)
```

```
In [44]: arr2[arrMask]
```

```
Out[44]: array([ 0,  5, 10, 15])
```

I Fancy Indexing

- ▶ It can also be obtained by combining two conditions.

```
In [45]: arr2>10
```

```
Out[45]: array([False, False, False, False, False, False, False, False,
   False, False, True, True, True, True, True, True, True, True,
   True, True])
```

```
In [46]: arrMask2=(arr2>=10)
```

```
In [47]: arr2[arrMask & arrMask2]
```

```
Out[47]: array([10, 15])
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
TRUE										TRUE									

Unit 1.

NumPy Array Data Structure for Optimal Computational Performance

- | 1.1. NumPy Arrays
- | 1.2. NumPy Array Basics
- | 1.3. NumPy Array Operations
- | 1.4. NumPy Indexing and Slicing
- | **1.5. Array Transposition and Axis Swap**

Array Transposition

Array Transposition is a special function that returns a view in which the data shape has changed without copying the data. ndarray has a transpose method and a special property named T.

```
In [1]: import numpy as np
```

```
In [2]: arr=np.arange(15).reshape((3,5))
```

Array Transposition

- Transpose can change the dimensional order of the ndarray. T can be used to reverse the order of all dimensions with transpose. Since this is a frequently used function, it was made into a shortcut function.

```
In [3]: arr
```

```
Out[3]: array([[ 0,  1,  2,  3,  4],  
               [ 5,  6,  7,  8,  9],  
               [10, 11, 12, 13, 14]])
```

```
In [4]: arr.transpose()
```

```
Out[4]: array([[ 0,  5, 10],  
               [ 1,  6, 11],  
               [ 2,  7, 12],  
               [ 3,  8, 13],  
               [ 4,  9, 14]])
```

```
In [5]: arr
```

```
Out[5]: array([[ 0,  1,  2,  3,  4],  
               [ 5,  6,  7,  8,  9],  
               [10, 11, 12, 13, 14]])
```

```
In [6]: arr.T
```

```
Out[6]: array([[ 0,  5, 10],  
               [ 1,  6, 11],  
               [ 2,  7, 12],  
               [ 3,  8, 13],  
               [ 4,  9, 14]])
```

Array Transposition

- ▶ Linear algebra, such as matrix multiplication, division, determinant, and square matrix, is an important part of the library dealing with arrays.
- ▶ Multiplying two two-dimensional arrays by * operators yields the product of each corresponding element, not the multiplication of the matrix.
- ▶ Matrix multiplication is calculated using a dot function in the NumPy namespace and an array method.

```
In [7]: arr=np.random.randn(6,3)
```

```
In [8]: arr
```

```
Out[8]: array([[ 1.73124749, -2.06641206, -1.21970588],
   [ 1.24537674, -0.04426672, -0.48196356],
   [-0.22442593,  0.69946937,  1.28283584],
   [ 0.21520254, -0.77688742, -1.07946701],
   [ 1.21499758, -1.2265046 ,  0.29900669],
   [ 1.79874526,  0.43687671,  2.48195604]])
```

```
In [9]: arr.T
```

```
Out[9]: array([[ 1.73124749,  1.24537674, -0.22442593,  0.21520254,  1.21499758,
   1.79874526],
   [-2.06641206, -0.04426672,  0.69946937, -0.77688742, -1.2265046 ,
   0.43687671],
   [-1.21970588, -0.48196356,  1.28283584, -1.07946701,  0.29900669,
   2.48195604]])
```

| Array Transposition

- ▶ It is often used in matrix calculation, and np.dot is also used to find the inner part of the matrix.

```
In [10]: np.dot(arr.T,arr)
```

```
Out[10]: array([[ 9.35656387, -4.66113685,  1.59565446],
 [-4.66113685,  7.0600046 ,  4.99525431],
 [ 1.59565446,  4.99525431, 10.78039891]])
```

Array Transposition

- ▶ First, let's reshape 3 rows and 4 columns in 2D into 4 rows and 3 columns.

```
In [11]: arr_1=np.arange(12).reshape(3,4)
```

```
In [12]: arr_1
```

```
Out[12]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])
```

```
In [13]: arr_1.reshape((4,3))
```

```
Out[13]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])
```

```
In [14]: arr_1.transpose((0,1))
```

```
Out[14]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])
```

```
In [15]: arr_1.transpose((1,0))
```

```
Out[15]: array([[ 0,  4,  8],
                 [ 1,  5,  9],
                 [ 2,  6, 10],
                 [ 3,  7, 11]])
```

| Array Transposition

- ▶ This time, let's try with a three-dimensional array.
- ▶ Even in a three-dimensional array, the transpose method receives the tuple axis number and replaces it.

```
In [16]: arr=np.arange(16).reshape((2,2,4))
```

```
In [17]: arr
```

```
Out[17]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],
                 [[ 8,  9, 10, 11],
                  [12, 13, 14, 15]]])
```

Line 16

- Second page, second row, fourth row.

I Array Transposition

```
In [18]: arr.transpose((1,0,2))
```

```
Out[18]: array([[[ 0,  1,  2,  3],
                  [ 8,  9, 10, 11],
                  [[ 4,  5,  6,  7],
                   [12, 13, 14, 15]]])
```

```
In [19]: arr.transpose((0,1,2))
```

```
Out[19]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [[ 8,  9, 10, 11],
                   [12, 13, 14, 15]]])
```

```
In [20]: arr.transpose((2,1,0))
```

```
Out[20]: array([[[ 0,  8],
                  [ 4, 12]],
                  [[ 1,  9],
                   [ 5, 13]],
                  [[ 2, 10],
                   [ 6, 14]],
                  [[ 3, 11],
                   [ 7, 15]]])
```

Line 18

- The order of the first and second axes was reversed, and the last axis remained the same.

Array Transposition

- There is a method called `swapaxes` in the `ndarray`, which receives two axis numbers and reverses the array.

```
In [21]: arr
```

```
Out[21]: array([[[ 0,  1,  2,  3],
   [ 4,  5,  6,  7],
   [[ 8,  9, 10, 11],
   [12, 13, 14, 15]]])
```

```
In [22]: help(arr.swapaxes)
```

Help on built-in function `swapaxes`:

`swapaxes(...)` method of `numpy.ndarray` instance
`a.swapaxes(axis1, axis2)`

Return a view of the array with `'axis1'` and `'axis2'` interchanged.

Refer to `'numpy.swapaxes'` for full documentation.

See Also

`numpy.swapaxes` : equivalent function

I Array Transposition

```
In [23]: arr.swapaxes(1,2)
```

```
Out[23]: array([[[], [0, 4],  
                  [[1, 5],  
                   [2, 6],  
                   [3, 7]],  
                  [[8, 12],  
                   [9, 13],  
                   [10, 14],  
                   [11, 15]]]])
```

```
In [24]: arr.swapaxes(0,1)
```

```
Out[24]: array([[[0, 1, 2, 3],  
                  [8, 9, 10, 11]],  
                  [[4, 5, 6, 7],  
                   [12, 13, 14, 15]]])
```

Coding Exercise #0103



Follow practice steps on 'ex_0103.ipynb' file

Unit 2.

Optimal Data Exploration Through Pandas

- | 2.1. Pipelines: Data Structures According to Data Types
- | 2.2. Pandas Series and DataFrames
- | 2.3. Merging and Binding DataFrames
- | 2.4. DataFrame Sorting and Multi-Index
- | 2.5. Examining the Characteristics of Data Through Descriptive Statistics and Data Samples

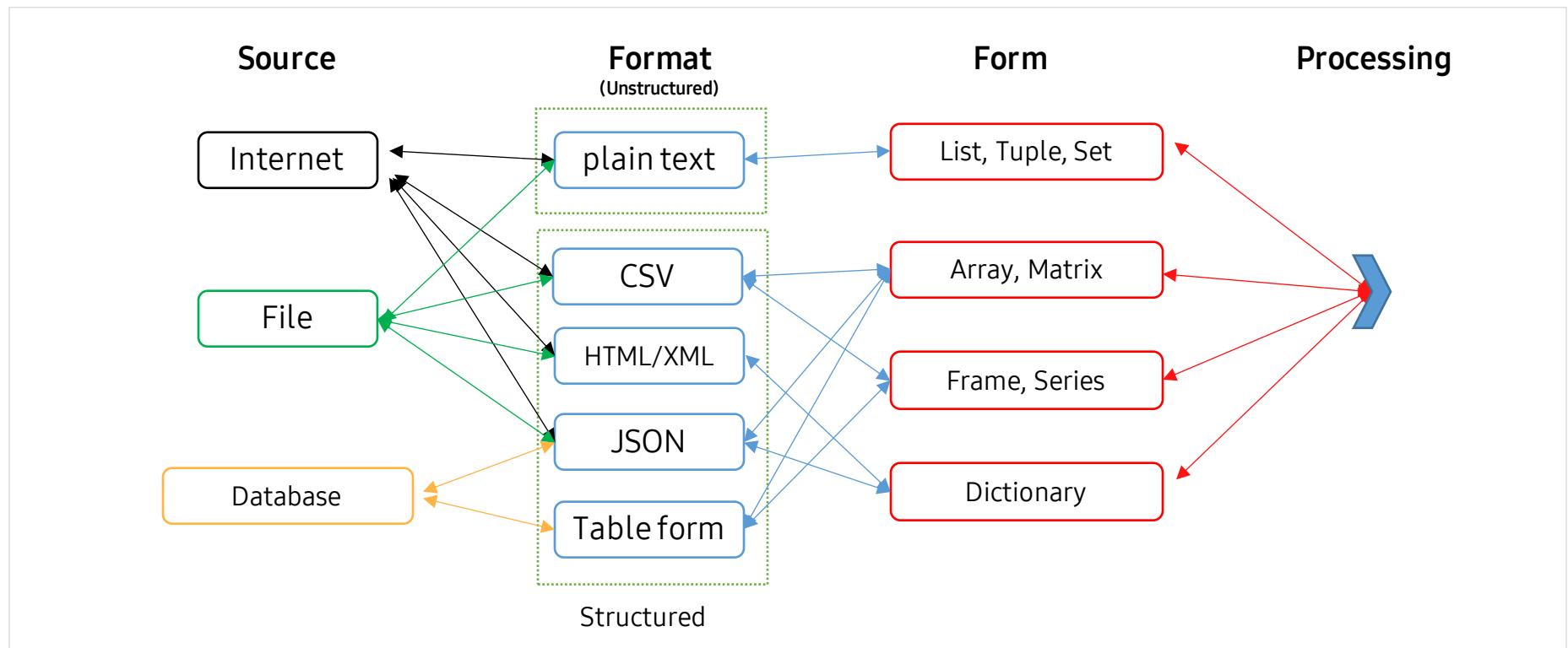
Pipelines

I Structural Perspective of Data Types

- ▶ Collected data, from a structural perspective (schema structure or computability), can be divided into three categories: structured, unstructured, and semi-structured data.
 - Structured data refers to data that has a structure-based form of the structured schema (form) and is stored in fixed fields such as RDB and spreadsheet and is consistent in value and format.
 - Unstructured data refers to data that does not have a schema structure and is not stored in fixed fields such as social media, web bulletin boards, and NoSQL.
 - Semi-Structured data refers to data that has a schema(formal) structure and contains metadata such as XML, HTML, weblogs, system logs, and alarms and is inconsistent in value and format.

| Python Data Structure Pipeline

- ▶ Data collected from various sources have various forms and properties, as shown in the figure below. Thus, for analysis, it is necessary to integrate various data types in the same format so that computers can understand them.



Unit 2.

Optimal Data Exploration Through Pandas

- | 2.1. Pipelines: Data Structures According to Data Types
- | **2.2. Pandas Series and DataFrames**
- | 2.3. Merging and Binding DataFrames
- | 2.4. DataFrame Sorting and Multi-Index
- | 2.5. Examining the Characteristics of Data Through Descriptive Statistics and Data Samples

Pandas Outline

- | A table is the most optimal form of data that a person can understand. Therefore, the ability to handle tabular data well is the basis of analysis. In Python, the table form is called a DataFrame and is implemented as a pandas library.
 - ▶ The pandas library is built based on NumPy but specializes in more complex data analysis.
 - ▶ While NumPy processes only the same array of data types, Pandas can process different data types.
 - ▶ The pandas library is an optimal tool for collecting and organizing data.
 - ▶ Pandas is an important tool that can handle most of the work in data science and is essential for data scientists.

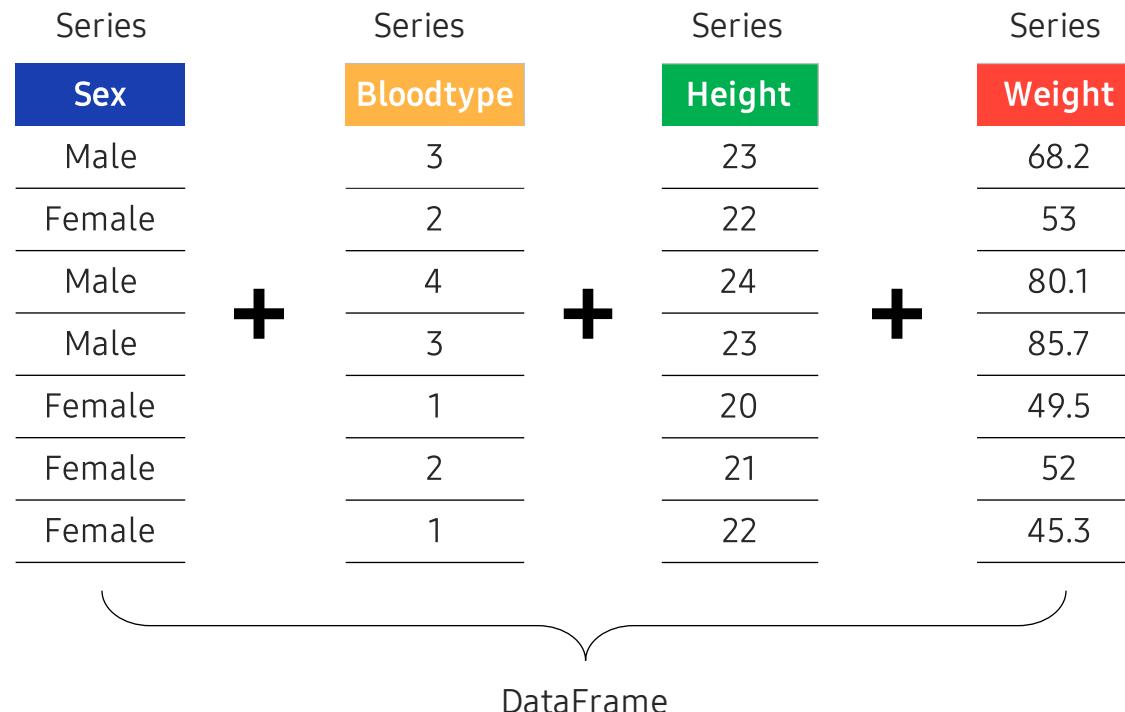
Pandas for Data Analysis

| Essential Library for Data Analysis

- ▶ Statistics-based effective representation and data collection summary
- ▶ Data alignment and manipulation to merge and weave various types of data
- ▶ Processing data such as collection, transformation, and function application can be applied to the entire data bundle.
- ▶ Since NumPy is a general arithmetic data processing-based library, pandas must be used to process tabular data for statistics or analysis. Pandas also provides time series processing capabilities that NumPy lacks.

| About Pandas Data Structure:

- To learn about pandas, it is important to familiarize oneself with Series and DataFrame data structures.



- The basic data structures of pandas include Series (1D) and DataFrame (2D). DataFrame is a container for Series, and Series is a container for scalar (0D). They can add or delete data in a dictionary manner.

Series

A series is a form of sequentially listed one-dimensional arrays.

- ▶ Since it is an array, all elements in the series must belong to one data type.
- ▶ Simple series can be created from sequences such as lists, tuples, and arrays.
- ▶ The word ‘series’ is singular AND plural. Its Latin origin, ‘serere,’ means to join or connect.
- ▶ The default value of the series is an integer index. The label of the first item is 0, the label of the second item is 1, and it increases in this manner.
- ▶ The attribute value of the series is the list of all values in the series.
- ▶ The attribute value of the index means an index of a series.
- ▶ The attribute value of the index.values is an array of all index values.

| How to Create a Series

```
In [1]: import pandas as pd
```

- ▶ Creating from a List

```
In [2]: pd.Series(['Male', 'Female', 'Male','Male','Female', 'Female', 'Female'])
```

```
Out[2]: 0      Male
        1    Female
        2      Male
        3      Male
        4    Female
        5    Female
        6    Female
       dtype: object
```

| How to Create a Series

```
In [3]: import numpy as np
```

- ▶ Creating from NumPy Arrays. See help(pd.Series) for more detail.

```
In [4]: pd.Series(np.array(['Male', 'Female', 'Male','Male','Female', 'Female', 'Female']))
```

```
Out[4]: 0      Male
        1    Female
        2      Male
        3      Male
        4    Female
        5    Female
        6    Female
       dtype: object
```

| How to Create a Series

- ▶ Series can also be created with array-like, Iterable, dict, or scalar value.

```
In [5]: ser=pd.Series(np.array(['Male', 'Female', 'Male','Male','Female', 'Female', 'Female']))
```

```
In [6]: type(ser)
```

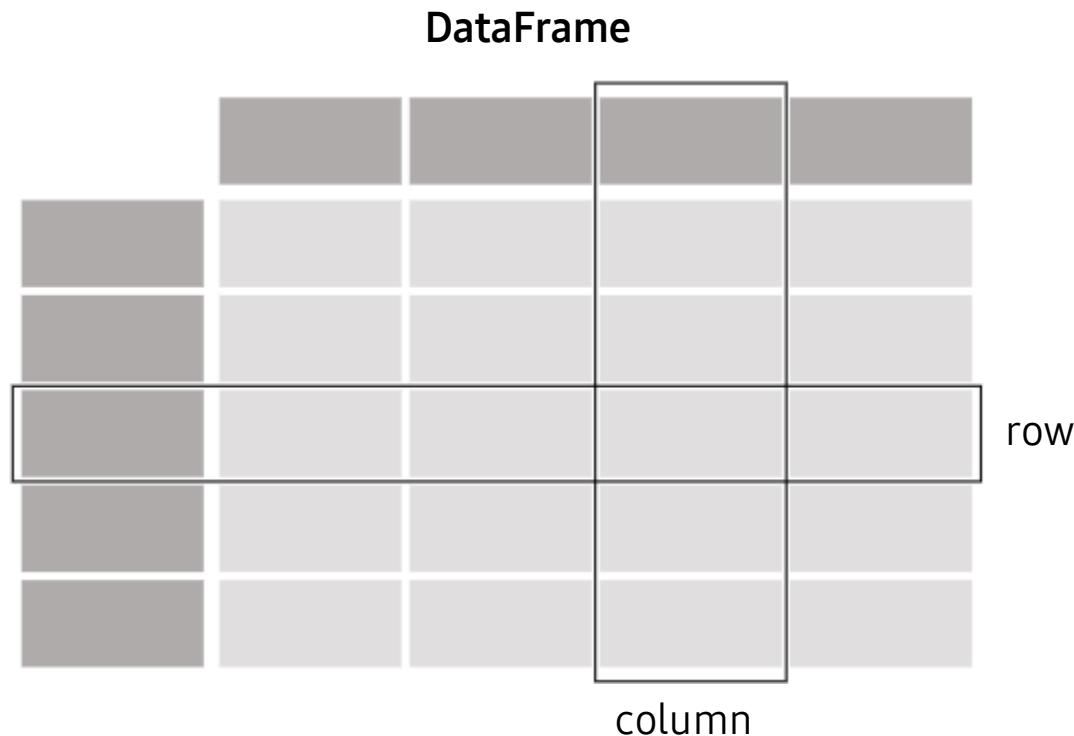
```
Out[6]: pandas.core.series.Series
```

Line 6

- We can know that the data type is a series.

| How to Create a Series

- ▶ To deal with DataFrames in the future, you must first understand the series data and think of the column part as a series in the figure below.



| Creating a Series with Dictionary

```
In [7]: dict_data={'a':1,'b':2,'c':3}
```

```
In [8]: pd.Series(dict_data)
```

```
Out[8]: a    1  
        b    2  
        c    3  
       dtype: int64
```

```
In [9]: ser1=pd.Series(dict_data)
```

```
In [10]: type(ser1)
```

```
Out[10]: pandas.core.series.Series
```

| Creating a Series with Dictionary

- ▶ Index arrays can be selected separately using index attributes of the series class.

```
In [11]: ser1.index
```

```
Out[11]: Index(['a', 'b', 'c'], dtype='object')
```

- ▶ Selecting the data value array separately is also possible. At that time, the values attribute of the series class is used.

```
In [12]: ser1.values
```

```
Out[12]: array([1, 2, 3], dtype=int64)
```

| Creating a Series with List

```
In [13]: list_data=['2019-01-02', 3.14, 'ABC', 100, True]
```

```
In [14]: ser2=pd.Series(list_data)
```

```
In [15]: ser2
```

```
Out[15]: 0    2019-01-02  
1        3.14  
2        ABC  
3        100  
4       True  
dtype: object
```

```
In [16]: ser2.index
```

```
Out[16]: RangeIndex(start=0, stop=5, step=1)
```

Line16

- The index is represented by an integer-like RangeIndex object in the range of 0 and 4. At this time, the last value is not included.
- ▶ The data value array maintains the order of the list element array of list_data, which is the original data.

| DataFrame and Series

- Let's create a simple DataFrame and learn how to manipulate data in a series. This data stores passenger data of the Titanic. It is assumed that the passenger's name (character), age (integer), and gender (male/female) data are known.

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

| DataFrame and Series

- ▶ A data frame must be used to view the proceeding table and store data in the form of a table. The column name, also called a column header, is mainly set as a key value of a dictionary.

```
In [17]: pd.DataFrame({ "Name": [ "Braund, Mr. Owen Harris",
                                "Allen, Mr. William Henry",
                                "Bonnel, Miss. Elizabeth"],
                                "Age": [22, 35, 58],
                                "Sex": [ "male", "male", "female"]})
```

Out [17]:

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

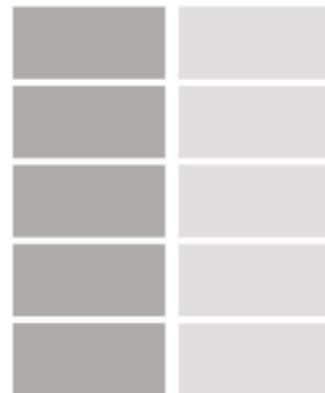
| DataFrame and Series

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

- ▶ A DataFrame is a two-dimensional data structure that stores various data types (letters, integers, floating point values, categorical data, etc.) in a column.
- ▶ Each table has three columns with a column label. The column labels are “Name,” “Age,” and “Sex.”
- ▶ Column “Name” consists of text data in which each value is a string, column “Age” is a silver number, and column “Sex” is text data.
- ▶ It is similar to data table representation in a spreadsheet.

| The column of a DataFrame is a series.

Series



| The column of a DataFrame is a series.

- ▶ If you are only interested in the "Age" column and want to pull out the column, use the dictionary key value.

```
In [18]: df=pd.DataFrame({"Name": ["Braund, Mr. Owen Harris",
                               "Allen, Mr. William Henry",
                               "Bonnell, Miss. Elizabeth"],
                           "Age":[22, 35, 58],
                           "Sex":["male", "male", "female"]})
```

```
In [19]: df['Age']
```

```
Out[19]: 0    22
          1    35
          2    58
          Name: Age, dtype: int64
```

```
In [20]: type(df['Age'])
```

```
Out[20]: pandas.core.series.Series
```

Line 19

- If you are familiar with dictionary, selecting a single column is very similar to selecting a dictionary value based on a key.

| The column of a DataFrame is a series.

- ▶ If you are only interested in the “Age” column and want to pull out the column, use the dictionary key value.

```
In [21]: df['Age'][0]
```

```
Out[21]: 22
```

 Line 21

- It is possible to index a series.

| The column of a DataFrame is a series.

- ▶ Let's make a DataFrame into a list of list types and organize it in a two-dimensional form.

```
In [22]: df=pd.DataFrame([[1,2,3],[4,5,6],[7,8,0]])
```

```
In [23]: df.columns
```

```
Out[23]: RangeIndex(start=0, stop=3, step=1)
```

```
In [24]: df[0]
```

```
Out[24]: 0    1  
         1    4  
         2    7  
Name: 0, dtype: int64
```

```
In [25]: type(df[0])
```

```
Out[25]: pandas.core.series.Series
```

Line 23

- Since the column name is not specified, it appears as a RangeIndex object.

| Creating a series by designating index

```
In [28]: eye=pd.Series({'Brown':220, 'Blue':215, 'Hazel':93, 'Green':64})
```

```
In [29]: eye.index
```

```
Out[29]: Index(['Brown', 'Blue', 'Hazel', 'Green'], dtype='object')
```

```
In [30]: type(eye)
```

```
Out[30]: pandas.core.series.Series
```

Line 29

- Note that string data types in the series are recognized in the form of objects.

| Creating a series by designating index

```
In [31]: eye.name
```

- ▶ The attributes of the series are as follows.

```
In [32]: eye.name="eye_color"
```

```
In [33]: eye
```

```
Out[33]: Brown    220
          Blue     215
          Hazel     93
          Green      64
          Name: eye_color, dtype: int32
```

Line 32

- Designate the index name with the name attribute.

| Creating a series by designating index

```
In [34]: eye.name  
Out[34]: 'eye_color'  
  
In [35]: eye.index  
Out[35]: Index(['Brown', 'Blue', 'Hazel', 'Green'], dtype='object')  
  
In [36]: eye.values  
Out[36]: array([220, 215, 93, 64])
```

| Creating a series by designating index

- ▶ The methods of the series are as follows.

```
In [37]: eye.sort_values()
```

```
Out[37]: Green    64
Hazel    93
Blue     215
Brown    220
Name: eye_color, dtype: int32
```

```
In [38]: eye.sort_index()
```

```
Out[38]: Blue     215
Brown    220
Green    64
Hazel    93
Name: eye_color, dtype: int32
```

| The methods of the series are as follows.

```
In [39]: eye.unique()
```

```
Out[39]: array([220, 215, 93, 64])
```

```
In [40]: eye.nunique()
```

```
Out[40]: 4
```

Line 39

- Only non-duplicate values are returned.

Line 40

- This is the number of values that are not duplicates.

| The methods of the series are as follows.

- ▶ Now, let's try and duplicate the values.

```
In [41]: my_data2=[220, 215, 93, 64, 64]
```

```
In [42]: eye2=pd.Series(data=my_data2, index=['Brown', 'Blue','Blue','Hazel','Green'])
```

```
In [43]: eye2
```

```
Out[43]: Brown    220
          Blue     215
          Blue      93
          Hazel     64
          Green     64
          dtype: int64
```

| The methods of the series are as follows.

- ▶ Now, let's try and duplicate the values.

```
In [44]: eye2.unique()
```

```
Out[44]: array([220, 215, 93, 64], dtype=int64)
```

```
In [45]: eye2.nunique()
```

```
Out[45]: 4
```

```
In [46]: eye2.value_counts()
```

```
Out[46]:
```

64	2
220	1
93	1
215	1

```
dtype: int64
```

Line 44

- Only non-duplicate values are returned.

Line 46

- Using the value_counts() method, the same value is added to the number. It's a method that is used quite often, so be sure to remember it.

| The methods of the series are as follows.

- ▶ Series can be indexed and sliced like a NumPy array.

```
In [47]: ser=pd.Series([0,10,20,30,40], index=['a','b','c','d','e'])
```

```
In [48]: ser[1]
```

```
Out[48]: 10
```

```
In [49]: ser['b']
```

```
Out[49]: 10
```

```
In [50]: ser[1:3]
```

```
Out[50]: b    10  
         c    20  
         dtype: int64
```

| The methods of the series are as follows.

- ▶ Operations in the series are also possible.

```
In [51]: ser1=pd.Series([0,1,2,3,4], index=[0,1,2,3,4])
ser2=pd.Series([0,1,2,3,4], index=[4,3,2,1,0])
```

```
In [52]: ser1
```

```
Out[52]: 0    0
1    1
2    2
3    3
4    4
dtype: int64
```

```
In [53]: ser2
```

```
Out[53]: 4    0
3    1
2    2
1    3
0    4
dtype: int64
```

| The methods of the series are as follows.

- ▶ Operations in the series are also possible.

```
In [54]: ser1+ser2
```

```
Out[54]: 0    4  
1    4  
2    4  
3    4  
4    4  
dtype: int64
```

ser1	ser2	ser1+ser2
0	4	4
1	3	4
2	2	4
3	1	4
4	0	4

| The methods of the series are as follows.

- ▶ Operations in the series are also possible.

```
In [55]: ser1*ser2
```

```
Out[55]: 0    0
         1    3
         2    4
         3    3
         4    0
dtype: int64
```

ser1	ser2	ser1*ser2
0	4	0
1	3	3
2	2	4
3	1	3
4	0	0

| The methods of the series are as follows.

- ▶ Operations in the series are also possible.

```
In [56]: ser1/ser2
```

```
Out[56]: 0    0.000000
          1    0.333333
          2    1.000000
          3    3.000000
          4      inf
dtype: float64
```

| The methods of the series are as follows.

- ▶ Methods Related to Statistics in the Series

```
In [57]: ser1.sum()
```

```
Out[57]: 10
```

```
In [58]: ser1.mean()
```

```
Out[58]: 2.0
```

```
In [59]: ser1.median()
```

```
Out[59]: 2.0
```

| The methods of the series are as follows.

- ▶ Methods Related to Statistics in the Series

```
In [60]: ser1.max()
```

```
Out[60]: 4
```

```
In [61]: ser1.min()
```

```
Out[61]: 0
```

```
In [62]: ser1.std()
```

```
Out[62]: 1.5811388300841898
```

```
In [63]: ser1.sort_values()
```

```
Out[63]: 0    0  
1    1  
2    2  
3    3  
4    4  
dtype: int64
```

The methods of the series are as follows.

- ▶ The lambda function can be applied to the series using the apply function.

```
In [64]: ser_height=pd.Series([160,170,180],name='height' )
```

```
In [65]: plus_10=lambda x:x+10
```

```
In [66]: plus_10(5)
```

```
Out[66]: 15
```

```
In [67]: ser_height.apply(lambda x:x+10 )
```

```
Out[67]: 0    170  
1    180  
2    190  
Name: height, dtype: int64
```

Line 65

- We will replace the lambda function with what we learned in Python Basics.

Line 67

- Add 10 to each element and return it to the series.

Pandas Series and DataFrame Practice

| Two-Dimensional Array

- ▶ It was mentioned that DataFrame is a two-dimensional array consisting of rows and columns. The two-dimensional array is the most commonly used form of data for computers.
- ▶ In the field of image processing and data analysis, most of the data are searched and processed in the form of a two-dimensional array. The DataFrame data structure of pandas originated from the DataFrame of R, a popular statistical package.
- ▶ It is similar to the two-dimensional NumPy array. The data type of the column may be different. It is ideal for storing data from CSV files, Excel spreadsheets, SQL tables, etc. There are properties such as columns, indices, etc.
- ▶ DataFrame can be thought of as a collection of Series objects. Pandas DataFrames allows for data representation similar to Excel spreadsheets.
- ▶ DataFrame consists of rows and columns. Rows are observations or instances. Columns are variables.

```
In [1]: import pandas as pd
```

| Pandas Series & DataFrame Practice

- ▶ pandas DataFrame is very convenient when reading an external data set, especially when data is stored in a comma-separated (CSV) format.

```
In [2]: pd.read_csv('data_iris.csv')
```

```
Out[2]:
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

I Pandas Series & DataFrame Practice

```
In [3]: iris_df=pd.read_csv('data_iris.csv')
```

```
In [4]: type(iris_df)
```

```
Out[4]: pandas.core.frame.DataFrame
```

Pandas Series & DataFrame Practice

- ▶ DataFrame objects can be created from dictionary objects.

```
In [4]: data = {'NAME': ['Jake', 'Jeniffer', 'Paul', 'Andrew'], 'AGE': [24, 21, 25, 19], 'GENDER': ['M', 'F', 'M', 'M']}
```

```
In [5]: pd.DataFrame(data)
```

```
Out [5]:
```

	NAME	AGE	GENDER
0	Jake	24	M
1	Jeniffer	21	F
2	Paul	25	M
3	Andrew	19	M

```
In [6]: df=pd.DataFrame(data)
```

I Things to Check After Bringing in the DataFrame

- 1) df.info()
- 2) df.head()

▶ This method prints information about a DataFrame, including the index dtype and columns, non-null values, and memory usage. See help(df.info) for more detail.

```
In [8]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   NAME    4 non-null      object 
 1   AGE     4 non-null      int64  
 2   GENDER  4 non-null      object 
dtypes: int64(1), object(2)
memory usage: 224.0+ bytes
```

| Things to Check After Bringing in the DataFrame

```
In [9]: iris_df.head()
```

Out [9]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Line 9

- It only shows the first five data. It is used to quickly view the state or value of the data.

| Things to Check After Bringing in the DataFrame

```
In [10]: iris_df.tail()
```

Out[10]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

Line 10

- It only shows the last five data.

| Things to Check After Bringing in the DataFrame

```
In [11]: iris_df.columns
```

```
Out[11]: Index(['Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width',
       'Species'],
      dtype='object')
```

Line11

- Used to check the column name.
- iris_df.columns() #caution: Since it is not a method, () is not used.

I Things to Check After Bringing in the DataFrame

```
In [12]: iris_df.index
```

```
Out[12]: RangeIndex(start=0, stop=150, step=1)
```

Line 12

- Used the check index.

Things to Check After Bringing in the DataFrame

- You can also change the column name, as shown below.

```
In [13]: iris_df.columns=['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width','Species']
```

```
In [14]: iris_df
```

```
Out[14]:
```

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...

Line 14

- You can see that it has changed from . to _.

I Things to Check After Bringing in the DataFrame

- ▶ Indexing and slicing are also possible in DataFrames.

```
In [15]: iris_df.Sepal_Length
```

```
Out[15]: 0    5.1
          1    4.9
          2    4.7
          3    4.6
          4    5.0
          ...
         145   6.7
         146   6.3
         147   6.5
         148   6.2
         149   5.9
Name: Sepal_Length, Length: 150, dtype: float64
```

```
In [16]: iris_df['Sepal_Length']
```

```
Out[16]: 0    5.1
          1    4.9
          2    4.7
          3    4.6
          4    5.0
          ...
         145   6.7
         146   6.3
         147   6.5
         148   6.2
         149   5.9
Name: Sepal_Length, Length: 150, dtype: float64
```

Several columns are brought into the two-dimensional form, as shown below.

```
In [17]: iris_df[['Sepal_Length', 'Sepal_Width']]
```

```
Out[17]:
```

	Sepal_Length	Sepal_Width
0	5.1	3.5
1	4.9	3.0
2	4.7	3.2
3	4.6	3.1
4	5.0	3.6
...
145	6.7	3.0
146	6.3	2.5
147	6.5	3.0
148	6.2	3.4
149	5.9	3.0

150 rows × 2 columns

| Pandas Series & DataFrame Practice

- ▶ A DataFrame can also be created into a two-dimensional array. Let's make the table below into a two-dimensional array of DataFrames.

Name	Age	Sex	School
Tom	15	Male	middle
Alice	10	Female	elementary

I Pandas Series & DataFrame Practice

```
In [18]: df=pd.DataFrame([['Name','Age','Sex','School'],['Tom',15,'Male','middle'],['Alice',10,'Female','elementary']])
```

- As shown above, the column name is not recognized when the data is entered. Thus, you have to set the columns yourself and create a DataFrame.

```
In [19]: df
```

```
Out[19]:
```

	0	1	2	3
0	Name	Age	Sex	School
1	Tom	15	Male	middle
2	Alice	10	Female	elementary

I Pandas Series & DataFrame Practice

```
In [20]: df=pd.DataFrame([['Tom',15,'Male','middle'],['Alice',10,'Female','elementary']], columns=['Name','Age','Sex','School'])
```

```
In [21]: df
```

```
Out[21]:
```

	Name	Age	Sex	School
0	Tom	15	Male	middle
1	Alice	10	Female	elementary

| Pandas Series & DataFrame Practice

- ▶ The row index and column name object of the df of the DataFrame, represented as df.index and df.columns, can be changed by assigning a new list to the attributes of the row index and column.
 - #Row Index Change: DataFrame Object.index = new row index list
 - #Column Name Change: DataFrame Object.columns = new column name list
- ▶ There is no index at this time.

```
In [22]: df.index
```

```
Out[22]: RangeIndex(start=0, stop=2, step=1)
```

I Pandas Series & DataFrame Practice

```
In [23]: df.index=['stu1','stu2']
```

```
In [24]: df
```

```
Out[24]:
```

	Name	Age	Sex	School
stu1	Tom	15	Male	middle
stu2	Alice	10	Female	elementary

```
In [25]: df.index
```

```
Out[25]: Index(['stu1', 'stu2'], dtype='object')
```

| Pandas Series & DataFrame Practice

- ▶ Columns can also be changed.

```
In [26]: df.columns=['student_name','years','sex2', 'school2']
```

```
In [27]: df
```

```
Out[27]:
```

	student_name	years	sex2	school2
stu1	Tom	15	Male	middle
stu2	Alice	10	FeMale	elementary

| Pandas Series & DataFrame Practice

- ▶ It is also possible to select and replace a row index or a part of the column name using the rename method in the DataFrame. It should be noted, however, that the original object is not directly modified, but a new DataFrame object is returned. Use the inplace= True option to change the original object.
 - # Row Index Change: DataFrame Object.rename(index={Existing index: New index,...})
 - # Column Name Change: DataFrame Object.rename(columns={Existing Name: New Name,...})

I Pandas Series & DataFrame Practice

```
In [28]: df.rename(index={'stu1':'student1'})
```

```
Out[28]:
```

	student_name	years	sex2	school2
student1	Tom	15	Male	middle
stu2	Alice	10	FeMale	elementary

```
In [29]: df
```

```
Out[29]:
```

	student_name	years	sex2	school2
stu1	Tom	15	Male	middle
stu2	Alice	10	FeMale	elementary

 Line 29

- Looking at it again, notice that it didn't change. Let's try the inplace=True option.

I Pandas Series & DataFrame Practice

```
In [30]: df.rename(index={'stu1':'student1'}, inplace=True)
```

```
In [31]: df
```

```
Out[31]:
```

	student_name	years	sex2	school2
student1	Tom	15	Male	middle
stu2	Alice	10	FeMale	elementary

| Pandas Series & DataFrame Practice

- ▶ If you don't want to use the `inplace=True` option, you can reassign new objects with the same name, but it's a bit cumbersome.
- ▶ If you want to change the index 'stu2' to 'student2', do it as follows.

```
In [32]: df=df.rename(index={'stu2':'student2'})
```

```
In [33]: df
```

```
Out[33]:
```

	student_name	years	sex2	school2
student1	Tom	15	Male	middle
student2	Alice	10	FeMale	elementary

| Pandas Series & DataFrame Practice

- ▶ Column names can be created in the same way.
- ▶ When changing 'student_name' to 'stu_name,' the following two methods can be used.

```
In [34]: df.rename(columns={'student_name':'stu_name'}, inplace=True)
```

```
In [35]: df
```

```
Out[35]:
```

	stu_name	years	sex2	school2
student1	Tom	15	Male	middle
student2	Alice	10	Female	elementary

I Deleting Rows and Columns

- When deleting a row or column of a DataFrame, use the drop() method. When deleting a row, enter Axis=0 as the axis option, or do not enter anything at all. On the other hand, if Axis=1 is entered as an axis option, the column is deleted. If you want to delete multiple rows or columns at the same time, enter them in the form of a list.
- Delete Row: DataFrame Object.drop(row index or list, axis=0)
- Delete Column: DataFrame Object.drop(column name or list, axis=1)

```
In [36]: exam_data = {'math' : [ 90, 80, 70], 'eng' : [ 98, 89, 95],  
                  'music' : [ 85, 95, 100], 'science' : [ 100, 90, 90]}
```

```
In [37]: exam_data
```

```
Out[37]: {'math': [90, 80, 70],  
          'eng': [98, 89, 95],  
          'music': [85, 95, 100],  
          'science': [100, 90, 90]}
```

Line 36

- Converting a DataFrame with a DataFrame() function. Save to variable df.

I Deleting Rows and Columns

- Let's create a DataFrame with an index as follows.

```
In [38]: df = pd.DataFrame(exam_data, index=['stu1', 'stu2', 'stu3'])
```

```
In [39]: df
```

```
Out[39]:
```

	math	eng	music	science
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

- Always be in the habit of copying the original before deleting the row. This is because if the data is incorrectly deleted when checking with the data prior to deleting, it needs to be brought back from the original. This is the only way to save time if the amount of data is large.

I Deleting Rows and Columns

```
In [40]: df2=df[:]
```

```
In [41]: df2.drop('stu2', inplace=True)
```

I Deleting Rows and Columns

```
In [42]: df2
```

```
Out[42]:
```

	math	eng	music	science
stu1	90	98	85	100
stu3	70	95	100	90

```
In [43]: df
```

```
Out[43]:
```

	math	eng	music	science
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

Line 43

- The original remains.

I Deleting Rows and Columns

```
In [44]: df3=df[:]
```

```
In [45]: df3.drop(['stu2','stu3'], axis=0, inplace=True)
```

```
In [46]: df3
```

```
Out[46]:
```

	math	eng	music	science
stu1	90	98	85	100

I Deleting Rows and Columns

- ▶ The same goes for deleting columns.

```
In [47]: df
```

```
Out[47]:
```

	math	eng	music	science
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

```
In [49]: df4 = df[:]
```

Line 49

- Replicate the DataFrame df and store it in the variable df4. Delete one column of df4.

I Deleting Rows and Columns

- ▶ The same goes for deleting columns.

```
In [50]: df4.drop('math', axis=1, inplace=True)
```

```
In [51]: df4
```

```
Out[51]:
```

	eng	music	science
stu1	98	85	100
stu2	89	95	90
stu3	95	100	90

I Deleting Rows and Columns

- ▶ The same goes for deleting columns.

```
In [52]: df
```

```
Out[52]:
```

	math	eng	music	science
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

```
In [53]: df5 = df[:]
```

I Deleting Rows and Columns

- ▶ The same goes for deleting columns.

```
In [54]: df5.drop(['eng', 'science'], axis=1, inplace=True)
```

```
In [55]: df5
```

```
Out[55]:
```

	math	music
stu1	90	85
stu2	80	95
stu3	70	100

I Deleting Rows and Columns

- ▶ Use the row index to select one row.

```
In [56]: df
```

```
Out[56]:
```

	math	eng	music	science
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

```
In [57]: 1 label1 = df.loc['stu1']
          2 position1 = df.iloc[0]
```

Line 57-1

- Use the loc indexer.

Line 57-2

- Use the iloc indexer.

I Deleting Rows and Columns

- ▶ Use the row index to select one row.

```
In [58]: label1
```

```
Out[58]: math      90
          eng       98
          music     85
          science   100
          Name: stu1, dtype: int64
```

```
In [59]: position1
```

```
Out[59]: math      90
          eng       98
          music     85
          science   100
          Name: stu1, dtype: int64
```

I Deleting Rows and Columns

- ▶ Use the row index to select one row.

```
In [60]: label2 = df.loc[['stu1', 'stu2']]  
position2 = df.iloc[[0, 1]]
```

```
In [61]: label2
```

```
Out[61]:
```

	math	eng	music	science
stu1	90	98	85	100
stu2	80	89	95	90

```
In [62]: position2
```

```
Out[62]:
```

	math	eng	music	science
stu1	90	98	85	100
stu2	80	89	95	90

| Selecting Columns

```
In [63]: math1 = df['math']
```

```
In [64]: math1
```

```
Out[64]: stu1    90
stu2    80
stu3    70
Name: math, dtype: int64
```

Line 63

- Select only the “math” score data. Save it to variable math1.

I Selecting Columns

```
In [65]: english = df.eng
```

```
In [66]: english
```

```
Out[66]: stu1    98  
stu2    89  
stu3    95  
Name: eng, dtype: int64
```

Line 65

- Select only the “english” score data. Save it to variable english.

I Selecting Columns

- ▶ Select “music” and “science” score data. Save it to variable music_sci.

```
In [67]: music_sci=df[['music', 'science']]
```

```
In [68]: music_sci
```

```
Out[68]:
```

	music	science
stu1	85	100
stu2	95	90
stu3	100	90

I Selecting Columns

- If you want to store it in a two-dimensional form, and there is only one column, use the list form[].

```
In [69]: df['math']
```

```
Out[69]: stu1    90
stu2    80
stu3    70
Name: math, dtype: int64
```

```
In [70]: df[['math']]
```

```
Out[70]:
```

	math
stu1	90
stu2	80
stu3	70

| Designating Index

```
In [71]: exam_data = {'name' : [ 'hongildong', 'hongeedong', 'hongsamdong'],
                  'math' : [ 90, 80, 70],
                  'eng' : [ 98, 89, 95],
                  'music' : [ 85, 95, 100],
                  'phys_tra' : [ 100, 90, 90]}
```

```
In [72]: df = pd.DataFrame(exam_data)
```

```
In [73]: df.index
```

```
Out[73]: RangeIndex(start=0, stop=3, step=1)
```

| Designating Index

- ▶ There is no index, but an index may be selected among the columns.
- ▶ Designate the column “Name” as a new index and reflect the changes to the df object.

```
In [74]: df.set_index('name', inplace=True)
```

```
In [75]: df.index
```

```
Out[75]: Index(['hongildong', 'hongeedException', 'hongsamdong'], dtype='object', name='name')
```

```
In [76]: df
```

```
Out[76]:
```

	math	eng	music	phys_tra
name				
hongildong	90	98	85	100
hongeedException	80	89	95	90
hongsamdong	70	95	100	90

Line 74

- If the name part moves down, it is set as an index.

I Designating Index

```
In [77]: a = df.loc['hongildong', 'music']
```

```
In [78]: a
```

```
Out[78]: 85
```

Line 77

- Select one specific element of the DataFrame df (“music” score of “honggildong”).

| Designating Index

```
In [79]: c=df.loc['hongildong', ['music', 'phys_tra']]
```

```
In [80]: c
```

```
Out[80]: music    85  
          phys_tra  100  
          Name: hongildong, dtype: int64
```

Line 79

- Select two or more elements of the DataFrame df (“music” and “phys_tra” scores of “honggildong”).

| Designating Index

```
In [81]: d = df.iloc[0, [2, 3]]  
  
In [82]: d  
  
Out[82]: music      85  
          phys_tra  100  
          Name: hongildong, dtype: int64
```

Line 81

- This can also be done with the integer index iloc.

| Designating Index

```
In [83]: e = df.loc['hongildong', 'music':'phys_tra']
```

```
In [84]: e
```

```
Out[84]: music    85
          phys_tra  100
Name: hongildong, dtype: int64
```

Line 83

- It is also possible through slicing.

| Designating Index

```
In [85]: f = df.iloc[0, 2:]
```

```
In [86]: f
```

```
Out[86]: music      85  
phys_tra    100  
Name: hongildong, dtype: int64
```

I Designating Index

```
In [87]: g = df.loc[['hongildong', 'hongeedong'], ['music', 'phys_tra']]
```

```
In [88]: g
```

```
Out[88]:
```

	music	phys_tra
name		
hongildong	85	100
hongeedong	95	90

	music	phys_tra
name		
hongildong	85	100
hongeedong	95	90

Line 87

- Select an element from two or more rows and columns of df ("music" and "phys_tra" scores of "honggildong" and "hongeedong").

| Designating Index

```
In [89]: h = df.iloc[[0, 1], [2, 3]]
```

```
In [90]: h
```

```
Out[90]:
```

	music	phys_tra
name		
hongildong	85	100
hongeedException	95	90

| Designating Index

```
In [91]: i = df.loc['hongildong':'hongeedong', 'music':'phys_tra']
```

```
In [92]: i
```

```
Out[92]:
```

	music	phys_tra
name		
hongildong	85	100
hongeedong	95	90

| Designating Index

```
In [93]: df
```

```
Out[93]:
```

	math	eng	music	phys_tra
--	------	-----	-------	----------

name	math	eng	music	phys_tra
hongildong	90	98	85	100
hongeedong	80	89	95	90
hongsamdong	70	95	100	90

Add a Column

```
In [94]: df['kor'] = 80
```

```
In [95]: df
```

```
Out[95]:
```

	math	eng	music	phys	tra	kor
--	------	-----	-------	------	-----	-----

name	math	eng	music	phys	tra	kor
hongildong	90	98	85	100	80	80
hongeedException	80	89	95	90	80	80
hongsamdong	70	95	100	90	80	80

Line 94

- Add a "kor" score column to the DataFrame df. The value of the data is 80.

I Add a Row

In [96]: df

Out [96]:

	math	eng	music	phys_tra	kor
name					
hongildong	90	98	85	100	80
hongeedong	80	89	95	90	80
hongsamdong	70	95	100	90	80

	math	eng	music	phys_tra	kor
name					
hongildong	90	98	85	100	80
hongeedong	80	89	95	90	80
hongsamdong	70	95	100	90	80

Add a Row

```
In [97]: df.loc[3] = 0
```

```
In [98]: df
```

```
Out[98]:
```

	math	eng	music	phys	tra	kor
name						
hongildong	90	98	85	100	80	
hongeedong	80	89	95	90	80	
hongsamdong	70	95	100	90	80	
3	0	0	0	0	0	0

	math	eng	music	phys	tra	kor
name						
hongildong	90	98	85	100	80	
hongeedong	80	89	95	90	80	
hongsamdong	70	95	100	90	80	
3	0	0	0	0	0	0

Line 97

- Add a new row – enter the same element value.

Add a Row

```
In [99]: df.loc[3] = [90, 80, 70, 60, 77]
```

```
In [100]: df
```

```
Out[100]:
```

	math	eng	music	phys_tra	kor
--	------	-----	-------	----------	-----

name	math	eng	music	phys_tra	kor
hongildong	90	98	85	100	80
hongeedong	80	89	95	90	80
hongsamdong	70	95	100	90	80
3	90	80	70	60	77

Line 99

- Add a new row – enter an array of element values.

| Add a Row

```
In [101]: df.index
```

```
Out[101]: Index(['hongildong', 'hongeedong', 'hongsamdong', 3], dtype='object', name='name')
```

```
In [102]: df.index[3]
```

```
Out[102]: 3
```

| Add a Row

- ▶ `help(df.reset_index)`

```
In [103]: df
```

```
Out[103]:
```

	math	eng	music	phys_tra	kor
name					
hongildong	90	98	85	100	80
hongeedong	80	89	95	90	80
hongsamdong	70	95	100	90	80
3	90	80	70	60	77

```
In [104]: df=df.reset_index()
```

```
In [105]: df
```

```
Out[105]:
```

	name	math	eng	music	phys_tra	kor
0	hongildong	90	98	85	100	80
1	hongeedong	80	89	95	90	80
2	hongsamdong	70	95	100	90	80
3	3	90	80	70	60	77

| Add a Row

```
In [106]: df.iloc[3, 0]='hongsadong'
```

```
In [107]: df
```

```
Out[107]:
```

	name	math	eng	music	phys_tra	kor
0	hongildong	90	98	85	100	80
1	hongeedong	80	89	95	90	80
2	hongsamdong	70	95	100	90	80
3	hongsadong	90	80	70	60	77

Add a Row

```
In [108]: df.set_index('name', inplace=True)
```

```
In [109]: df
```

```
Out[109]:
```

	math	eng	music	phys_tra	kor
--	------	-----	-------	----------	-----

name	math	eng	music	phys_tra	kor
hongildong	90	98	85	100	80
hongeedong	80	89	95	90	80
hongsamdong	70	95	100	90	80
hongsadong	90	80	70	60	77

Line 109

- When the index is set this way, pay close attention when adding rows.

| Change the DataFrame Element

```
In [110]: exam_data = {'name' : ['stu1', 'stu2', 'stu3'],
                  'math' : [ 90, 80, 70],
                  'eng' : [ 98, 89, 95],
                  'mus' : [ 85, 95, 100],
                  'phy' : [ 100, 90, 90]}
df = pd.DataFrame(exam_data)
```

```
In [111]: df
```

```
Out[111]:
```

	name	math	eng	mus	phy
0	stu1	90	98	85	100
1	stu2	80	89	95	90
2	stu3	70	95	100	90

```
In [112]: df.index
```

```
Out[112]: RangeIndex(start=0, stop=3, step=1)
```

| Change the DataFrame Element

```
In [113]: df.set_index('name', inplace=True)
```

```
In [114]: df
```

```
Out[114]:
```

	math	eng	mus	phy
--	------	-----	-----	-----

name	math	eng	mus	phy
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

Line 113

- Designate the column “name” as a new index and reflect the changes to the df object.

| Change the DataFrame Element

```
In [115]: df.iloc[0][3] = 80
```

```
In [116]: df
```

```
Out[116]:
```

	math	eng	mus	phy
--	------	-----	-----	-----

name	math	eng	mus	phy
stu1	90	98	85	80
stu2	80	89	95	90
stu3	70	95	100	90

Line 115

- A method for changing a specific element for the DataFrame df: There are various methods for changing a “phy” score of “stu1.”

| Change the DataFrame Element

```
In [117]: df.loc['stu1']['phy'] = 90
```

```
In [118]: df
```

```
Out[118]:
```

	math	eng	mus	phy
name				
stu1	90	98	85	90
stu2	80	89	95	90
stu3	70	95	100	90

| Change the DataFrame Element

```
In [119]: df.loc['stu1', 'phy'] = 100
```

```
In [120]: df
```

```
Out[120]:
```

	math	eng	mus	phy
name				
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

| Change the DataFrame Element

```
In [121]: df.loc['stu1', ['mus', 'phy']] = 50
```

```
In [122]: df
```

```
Out[122]:
```

	math	eng	mus	phy
name				
stu1	90	98	50	50
stu2	80	89	95	90
stu3	70	95	100	90

Line 121

- A method for changing several elements of stu1 df: “mus” and “phy” scores of “stu1.”

| Change the DataFrame Element

```
In [123]: df.loc['stu1', ['mus', 'phy']] = 100, 100
```

```
In [124]: df
```

```
Out[124]:
```

	math	eng	mus	phy
--	------	-----	-----	-----

name				
stu1	90	98	100	100
stu2	80	89	95	90
stu3	70	95	100	90

| Transpose the DataFrame

```
In [125]: df.reset_index(inplace=True)
```

```
In [126]: df
```

```
Out[126]:
```

	name	math	eng	mus	phy
0	stu1	90	98	100	100
1	stu2	80	89	95	90
2	stu3	70	95	100	90

| Transpose the DataFrame

```
In [127]: dff=df.transpose()
```

```
In [128]: dff
```

```
Out[128]:
```

	0	1	2
name	stu1	stu2	stu3
math	90	80	70
eng	98	89	95
mus	100	95	100
phy	100	90	90

Line 127

- Transposing the DataFrame df (using the method).

| Transpose the DataFrame

```
In [129]: dfff = df.T
```

```
In [130]: dfff
```

```
Out[130]:
```

	name	math	eng	mus	phy
0	stu1	90	98	100	100
1	stu2	80	89	95	90
2	stu3	70	95	100	90

Line 129

- Transposing the DataFrame df again (using class attributes).

| Transpose the DataFrame

▶ Index Setting

```
In [131]: exam_data = {'name' : ['stu1', 'stu2', 'stu3'],
                 'math' : [ 90, 80, 70],
                 'eng' : [ 98, 89, 95],
                 'mus' : [ 85, 95, 100],
                 'phy' : [ 100, 90, 90]}
df = pd.DataFrame(exam_data)
```

```
In [132]: ndf = df.set_index(['name'])
```

```
In [133]: ndf
```

```
Out[133]:
```

	math	eng	mus	phy
name				
stu1	90	98	85	100
stu2	80	89	95	90
stu3	70	95	100	90

Line 132

- A specific column is set as a row index of a DataFrame.

| Transpose the DataFrame

```
In [134]: ndf2 = ndf.set_index('mus')
```

```
In [135]: ndf2
```

```
Out[135]:
```

	math	eng	phy
--	------	-----	-----

mus			
85	90	98	100
95	80	89	90
100	70	95	90

| Transpose the DataFrame

```
In [136]: ndf3 = ndf.set_index(['math', 'mus'])
```

```
In [137]: ndf3
```

```
Out[137]:
```

		eng	phy
math	mus		
90	85	98	100
80	95	89	90
70	100	95	90

Line 136

- Multi-index

| Transpose the DataFrame

▶ Index

```
In [138]: 1 dict_data = {'c0':[1,2,3], 'c1':[4,5,6], 'c2':[7,8,9], 'c3':[10,11,12], 'c4':[13,14,15]}\n2 df = pd.DataFrame(dict_data, index=['r0', 'r1', 'r2'])
```

Line 138-1

- Definition of the dictionary

Line 138-2

- Converting the dictionary into a DataFrame. Specify the index as [r0, r1, r2].

| Transpose the DataFrame

```
In [139]: new_index = ['r0', 'r1', 'r2', 'r3', 'r4']
ndf = df.reindex(new_index)
ndf
```

Out[139]:

	c0	c1	c2	c3	c4
r0	1.0	4.0	7.0	10.0	13.0
r1	2.0	5.0	8.0	11.0	14.0
r2	3.0	6.0	9.0	12.0	15.0
r3	NaN	NaN	NaN	NaN	NaN
r4	NaN	NaN	NaN	NaN	NaN

Line 139

- Redesignate the index to [r0, r1, r2, r3, r4].

| Transpose the DataFrame

```
In [140]: new_index = ['r0', 'r1', 'r2', 'r3', 'r4']
ndf2 = df.reindex(new_index, fill_value=0)
```

```
In [141]: ndf2
```

```
Out[141]:
```

	c0	c1	c2	c3	c4
r0	1	4	7	10	13
r1	2	5	8	11	14
r2	3	6	9	12	15
r3	0	0	0	0	0
r4	0	0	0	0	0

Line 140

- Fill in the NaN value generated by reindexing with the number 0.

| Transpose the DataFrame

```
In [142]: 1 dict_data = {'c0':[1,2,3], 'c1':[4,5,6], 'c2':[7,8,9], 'c3':[10,11,12], 'c4':[13,14,15]}\n2 df = pd.DataFrame(dict_data, index=['r0', 'r1', 'r2'])\n3 df
```

Out[142]:

	c0	c1	c2	c3	c4
r0	1	4	7	10	13
r1	2	5	8	11	14
r2	3	6	9	12	15

Line 142-1

- Definition of the dictionary

Line 142-2

- Converting the dictionary into a DataFrame. Specify the index as [r0, r1, r2].

| Transpose the DataFrame

```
In [143]: ndf = df.reset_index()
```

```
In [144]: ndf
```

```
Out[144]:
```

	index	c0	c1	c2	c3	c4
0	r0	1	4	7	10	13
1	r1	2	5	8	11	14
2	r2	3	6	9	12	15

Line 143

- Reset the row index to integer.

| Transpose the DataFrame

```
In [145]: 1 dict_data = {'c0':[1,2,3], 'c1':[4,5,6], 'c2':[7,8,9], 'c3':[10,11,12], 'c4':[13,14,15]}\n2 df = pd.DataFrame(dict_data, index=['r0', 'r1', 'r2'])\n3 df
```

Out [145]:

	c0	c1	c2	c3	c4
r0	1	4	7	10	13
r1	2	5	8	11	14
r2	3	6	9	12	15

Line 145-1

- Definition of the dictionary

Line 145-2

- Converting the dictionary into a DataFrame. Specify the index as [r0, r1, r2].

| Transpose the DataFrame

```
In [146]: ndf = df.sort_index(ascending=False)  
ndf
```

Out[146]:

	c0	c1	c2	c3	c4
r2	3	6	9	12	15
r1	2	5	8	11	14
r0	1	4	7	10	13

Line 146

- Sort row indices in descending order.

| Transpose the DataFrame

```
In [147]: 1 dict_data = {'c0':[1,2,3], 'c1':[4,5,6], 'c2':[7,8,9], 'c3':[10,11,12], 'c4':[13,14,15]}
2 df = pd.DataFrame(dict_data, index=['r0', 'r1', 'r2'])
3 df
```

Out[147]:

	c0	c1	c2	c3	c4
r0	1	4	7	10	13
r1	2	5	8	11	14
r2	3	6	9	12	15

Line 147-1

- Definition of the dictionary

Line 147-2

- Converting the dictionary into a DataFrame. Specify the index as [r0, r1, r2].

| Transpose the DataFrame

```
In [148]: ndf = df.sort_values(by='c1', ascending=False)
print(ndf)
```

	c0	c1	c2	c3	c4
r2	3	6	9	12	15
r1	2	5	8	11	14
r0	1	4	7	10	13



Line 148

- Sort in descending order based on column c1.

Series Operations

| Series vs. Numbers

- ▶ Adding a number to a series object adds a number to each of the series' individual elements and converts the calculated result into a series object.

```
In [1]: import numpy as np
```

```
In [2]: import pandas as pd
```

| Series vs. Numbers

- ▶ Adding a number to a series object adds a number to each of the series' individual elements and converts the calculated result into a series object.

```
In [3]: student1 = pd.Series({'kor':100, 'eng':80, 'math':90})  
student1
```

```
Out[3]: kor    100  
         eng     80  
         math    90  
        dtype: int64
```

```
In [4]: percentage = student1 / 200  
percentage
```

```
Out[4]: kor    0.50  
         eng    0.40  
         math   0.45  
        dtype: float64
```

Line 3

- Create a pandas series with data from a dictionary.

Line 4

- Divide the student's scores by 200 per subject.

| Series vs. Series

- ▶ For all indices in the series, elements with the same index are calculated.

```
In [5]: student1 = pd.Series({'kor':100, 'eng':80, 'math':90})
student2 = pd.Series({'math':80, 'kor':90, 'eng':80})

addition = student1 + student2
subtraction = student1 - student2
multiplication = student1 * student2
division = student1 / student2
```

Line 5

- Create a pandas series with data from a dictionary.

Line 5-4 ~ 5-7

- Perform the four fundamental arithmetic calculations based on the scores of each student per subject.

| Series vs. Series

```
In [6]: result = pd.DataFrame([addition, subtraction, multiplication, division],  
                           index=['addition', 'subtraction', 'multiplication', 'division'])  
result
```

Out[6]:

	kor	math	eng
addition	190.000000	170.000	160.0
subtraction	10.000000	10.000	0.0
Multiplication	9000.000000	7200.000	6400.0
division	1.111111	1.125	1.0

Line 6

- Combine the results of the arithmetic operation into a DataFrame (series → DataFrame).
- ▶ In the example above, the order of subject names given by index is different. However, pandas finds and sorts the same subject name (index), and it adds scores of the same subject name (index).

| Series vs. Series

- If the number of elements in the two series is different or the size of the series is the same, it is treated as a not-a-number (NaN), meaning that there is no valid value.

```
In [7]: student1 = pd.Series({'kor':np.nan, 'eng':80, 'math':90})
student2 = pd.Series({'math':80, 'kor':90})

addition = student1 + student2
subtraction = student1 - student2
multiplication = student1 * student2
division = student1 / student2
```

Line 7

- Create a pandas series with data from a dictionary.

Line 7-4 ~ 7-7

- Perform the four fundamental arithmetic calculations based on the scores of each student per subject.
(Series vs. Series)

| Series vs. Series

```
In [8]: result = pd.DataFrame([addition, subtraction, multiplication, division],  
                           index=['addition', 'subtraction', 'multiplication', 'division'])
```

```
In [9]: result
```

Out[9]:

	kor	math	eng
addition	NaN	170.000	NaN
subtraction	NaN	10.000	NaN
Multiplication	NaN	7200.000	NaN
division	NaN	1.125	NaN

Line 8

- Combine the results of the arithmetic operation into a DataFrame (series -> DataFrame).

I Operation Method

- As you learned in previous slides, when there is no common index or NAN in values, it returns NaN. To not occur this result, set the fill value for the data set.

```
In [10]: 1 student1 = pd.Series({'kor':np.nan, 'eng':80, 'math':90})  
2 student2 = pd.Series({'math':80, 'kor':90})  
3  
4 sr_add = student1.add(student2, fill_value=0)  
5 sr_sub = student1.sub(student2, fill_value=0)  
6 sr_mul = student1.mul(student2, fill_value=0)  
7 sr_div = student1.div(student2, fill_value=0)
```

Line 10

- Create a pandas series with data from a dictionary.

Line 10~4 ~ 5~7

- Perform the four fundamental arithmetic calculations based on the scores of each student per subject.
(Use the Operation Method)

I Operation Method

```
In [11]: result = pd.DataFrame([sr_add, sr_sub, sr_mul, sr_div],  
                           index=['addition', 'subtraction', 'multiplication', 'division'])  
result
```

Out[11]:

	kor	math	eng
addition	90.0	170.000	80.0
subtraction	-90.0	10.000	80.0
Multiplication	0.0	7200.000	0.0
division	0.0	1.125	inf



Line 11

- Combine the results of the arithmetic operation into a DataFrame (series -> DataFrame).

DataFrame Operations

- DataFrames can be understood as concepts that expand series operations. First, it is sorted based on the row/column index and calculated between corresponding elements one by one.

| DataFrame vs. Numbers

- ▶ The seaborn library will be covered later in visualization, but here, it was used to import data.

```
In [12]: import seaborn as sns  
  
titanic = sns.load_dataset('titanic')
```

Line 12

- Create a DataFrame by selecting two columns, age and fare, from the titanic dataset.

| DataFrame vs. Numbers

```
In [13]: 1 titanic = sns.load_dataset('titanic')
2 df = titanic.loc[:, ['age','fare']]
3 df.head()
```

Out[13]:

	age	fare
0	22.0	7.2500
1	38.0	71.2833
2	26.0	7.9250
3	35.0	53.1000
4	35.0	8.0500

Line 13-2

- Create a DataFrame by selecting two columns, age and fare, from the titanic dataset.

Line 13-3

- Shows only the first five lines.

| DataFrame vs. Numbers

```
In [14]: 1 addition = df + 10  
         2 addition.head()
```

Out[14]:

	age	fare
0	32.0	17.2500
1	48.0	81.2833
2	36.0	17.9250
3	45.0	63.1000
4	45.0	18.0500

Line 14-1

- Add 10 to the DataFrame.

Line 14-2

- Shows only the first five lines.

- ▶ While maintaining the form of the existing DataFrame, only the element value is replaced with a new calculated value and returned as a new DataFrame object.

| DataFrame vs. DataFrame

- ▶ Elements in the same row and column positions of each DataFrame are calculated. If the element is not present on either side or NaN, the calculation result is treated as NaN.

```
In [15]: titanic = sns.load_dataset('titanic')
df = titanic.loc[:, ['age','fare']]
```

Line 15

- Create a DataFrame by selecting two columns, age and fare, from the titanic dataset.

| DataFrame vs. Numbers

```
In [16]: addition = df + 10  
addition
```

```
Out[16]:
```

	age	fare
0	32.0	17.2500
1	48.0	81.2833
2	36.0	17.9250
3	45.0	63.1000
4	45.0	18.0500
...

Line 16

- Add 10 to the DataFrame.

| DataFrame vs. Numbers

```
In [17]: subtraction = addition - df  
subtraction.tail()
```

Out[17]:

	age	fare
886	10.0	10.0
887	10.0	10.0
888	NaN	10.0
889	10.0	10.0
890	10.0	10.0

Line 17-1

- Calculate between DataFrames (additon - df).

Line 17-2

- Shows only the last five lines.

Coding Exercise #0104



Follow practice steps on 'ex_0104.ipynb' file

Coding Exercise #0105_Edited



Follow practice steps on 'ex_0105.ipynb' file

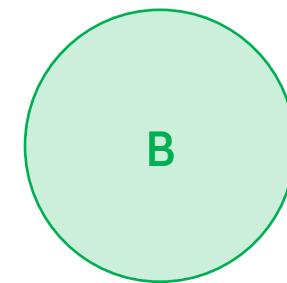
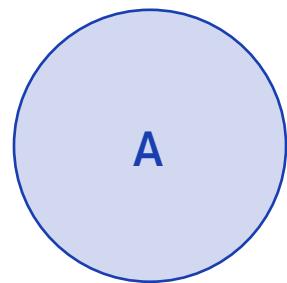
Unit 2.

Optimal Data Exploration Through Pandas

- | 2.1. Pipelines: Data Structures According to Data Types
- | 2.2. Pandas Series and DataFrames
- | **2.3. Merging and Binding DataFrames**
- | 2.4. DataFrame Sorting and Multi-Index
- | 2.5. Examining the Characteristics of Data Through Descriptive Statistics and Data Samples

DataFrame Manipulation

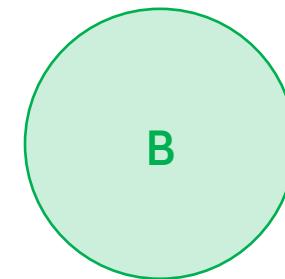
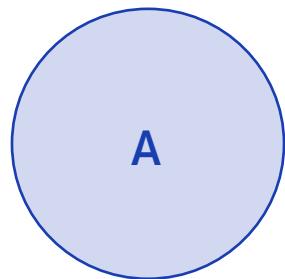
I Merging DataFrames:



Name	Gender	Age
Harry Potter	Male	23
David Baker	Male	31
John Smith	Male	22
Juan Martinez	Male	36
Jane Connor	Female	30

Name	Position	Wage
John Smith	Intern	25000
Alex Du Bois	Team Lead	75000
Joanne Rowling	Manager	90000
Jane Connor	Manager	70000

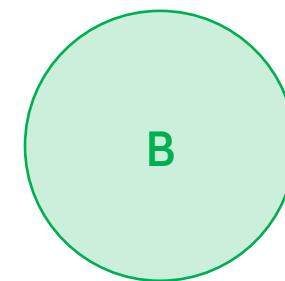
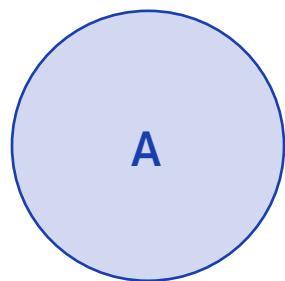
| Merging DataFrames: A.Name and B.Name used as keys



Name	Gender	Age
Harry Potter	Male	23
David Baker	Male	31
John Smith	Male	22
Juan Martinez	Male	36
Jane Connor	Female	30

Name	Position	Wage
John Smith	Intern	25000
Alex Du Bois	Team Lead	75000
Joanne Rowling	Manager	90000
Jane Connor	Manager	70000

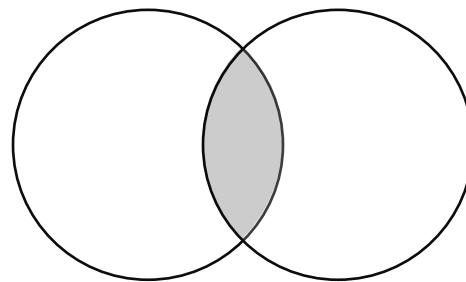
I Merging DataFrames: Inner join



Name	Gender	Age
Harry Potter	Male	23
David Baker	Male	31
John Smith	Male	22
Juan Martinez	Male	36
Jane Connor	Female	30

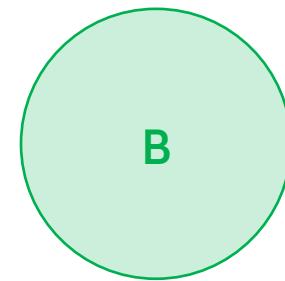
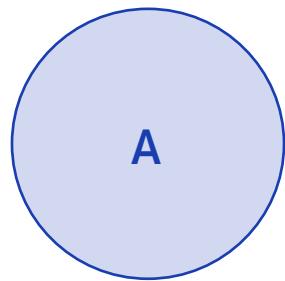
Name	Position	Wage
John Smith	Intern	25000
Alex Du Bois	Team Lead	75000
Joanne Rowling	Manager	90000
Jane Connor	Manager	70000

I Merging DataFrames: Inner join



A.Name	Gender	Age	B.Name	Position	Wage
John Smith	Male	22	John Smith	Intern	25000
Jane Connor	Female	30	Jane Connor	Manager	70000

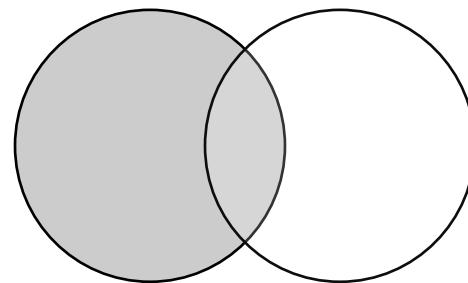
I Merging DataFrames: Left join



Name	Gender	Age
Harry Potter	Male	23
David Baker	Male	31
John Smith	Male	22
Juan Martinez	Male	36
Jane Connor	Female	30

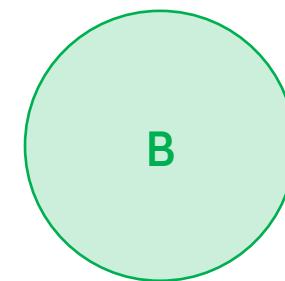
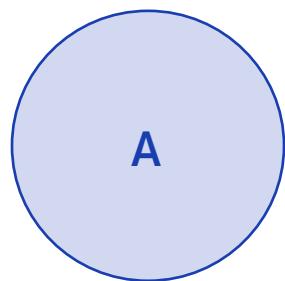
Name	Position	Wage
John Smith	Intern	25000
Alex Du Bois	Team Lead	75000
Joanne Rowling	Manager	90000
Jane Connor	Manager	70000

I Merging DataFrames: Left join



A.Name	Gender	Age	B.Name	Position	Wage
Harry Potter	Male	23	NA	NA	NA
David Baker	Male	31	NA	NA	NA
John Smith	Male	22	John Smith	Intern	25000
Juan Martinez	Male	36	NA	NA	NA
Jane Connor	Female	30	Jane Connor	Manager	70000

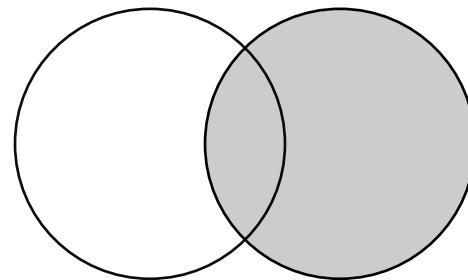
Merging DataFrames: Right join



Name	Gender	Age
Harry Potter	Male	23
David Baker	Male	31
John Smith	Male	22
Juan Martinez	Male	36
Jane Connor	Female	30

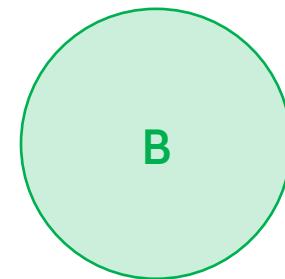
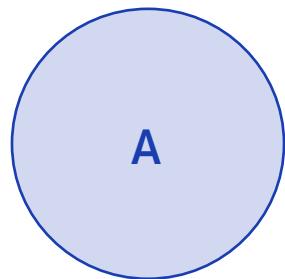
Name	Position	Wage
John Smith	Intern	25000
Alex Du Bois	Team Lead	75000
Joanne Rowling	Manager	90000
Jane Connor	Manager	70000

| Merging DataFrames: Right join

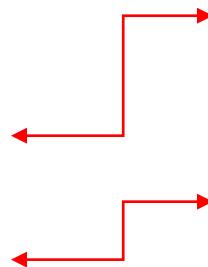


A.Name	Gender	Age	B.Name	Position	Wage
John Smith	Male	22	John Smith	Intern	25000
NA	NA	NA	Alex Du Bois	Team Lead	75000
NA	NA	NA	Joanne Rowling	Manager	90000
Jane Connor	Female	30	Jane Connor	Manager	70000

I Merging DataFrames: Full outer join

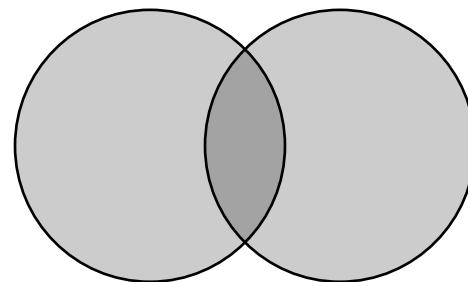


Name	Gender	Age
Harry Potter	Male	23
David Baker	Male	31
John Smith	Male	22
Juan Martinez	Male	36
Jane Connor	Female	30



Name	Position	Wage
John Smith	Intern	25000
Alex Du Bois	Team Lead	75000
Joanne Rowling	Manager	90000
Jane Connor	Manager	70000

I Merging DataFrames: Full outer join



A.Name	Gender	Age	B.Name	Position	Wage
Harry Potter	Male	23	NA	NA	NA
David Baker	Male	31	NA	NA	NA
John Smith	Male	22	John Smith	Intern	25000
Juan Martinez	Male	36	NA	NA	NA
Jane Connor	Female	30	Jane Connor	Manager	70000
NA	NA	NA	Alex Du Bois	Team Lead	75000
NA	NA	NA	Joanne Rowling	Manager	90000

Merging and Binding DataFrames:

```
In [ ]: 1 pd.merge(df_left, df_right, on='NAME')
2 pd.merge(df_left, df_right, left_on='NAME', right_on = 'NAME', how='inner')
3 pd.merge(df_left, df_right, left_on='NAME', right_on = 'NAME', how='left')
4 pd.merge(df_left, df_right, left_on='NAME', right_on = 'NAME', how='right')
5 pd.merge(df_left, df_right, left_on='NAME', right_on = 'NAME', how='outer')
```

Line 1, 2

- Inner join

Line 3

- Left join

Line 4

- Right join

Line 5

- Full outer join

I Merging and Binding DataFrames:

```
In [ ]: 1 pd.concat([df_A, df_B], sort=True)
         2 pd.concat([df_A, df_B], axis=1, sort=True)
```

Line 1

- Bind vertically by matching the column names.

Line 2

- Bind horizontally by matching the indices.

| Binding DataFrames

- ▶ When data is divided into several places, it may be necessary to combine them into one or bind the data. In pandas, functions used to combine or bind DataFrames include concat(), merge(), and join.
- ▶ `pandas.concat(list of DataFrames)`
- ▶ If the axial direction is not specified, the default option `axis=0` is applied and connected in the up/down row direction.

| Binding DataFrames

```
In [1]: import pandas as pd
```

```
In [2]: df1 = pd.DataFrame({'a': ['a0', 'a1', 'a2', 'a3'],
                         'b': ['b0', 'b1', 'b2', 'b3'],
                         'c': ['c0', 'c1', 'c2', 'c3']},
                        index=[0, 1, 2, 3])
```

```
In [3]: df1
```

```
Out[3]:
```

	a	b	c
0	a0	b0	c0
1	a1	b1	c1
2	a2	b2	c2
3	a3	b3	c3

| Binding DataFrames

```
In [4]: df2 = pd.DataFrame({'a': ['a2', 'a3', 'a4', 'a5'],
                           'b': ['b2', 'b3', 'b4', 'b5'],
                           'c': ['c2', 'c3', 'c4', 'c5'],
                           'd': ['d2', 'd3', 'd4', 'd5']},
                           index=[2, 3, 4, 5])
```

```
In [5]: df2
```

Out[5]:

	a	b	c	d
2	a2	b2	c2	d2
3	a3	b3	c3	d3
4	a4	b4	c4	d4
5	a5	b5	c5	d5

| Binding DataFrames

- ▶ Rows 0, 1, 2, and 3 derived from df1 are entered as NaN because there is no column "d."

```
In [6]: pd.concat([df1,df2])
```

```
Out[6]:
```

	a	b	c	d
0	a0	b0	c0	NaN
1	a1	b1	c1	NaN
2	a2	b2	c2	NaN
3	a3	b3	c3	NaN
2	a2	b2	c2	d2
3	a3	b3	c3	d3
4	a4	b4	c4	d4
5	a5	b5	c5	d5

| Binding DataFrames

- ▶ help(pd.concat). ignore_index

```
In [7]: pd.concat([df1,df2], ignore_index=True)
```

```
Out[7]:
```

	a	b	c	d
0	a0	b0	c0	NaN
1	a1	b1	c1	NaN
2	a2	b2	c2	NaN
3	a3	b3	c3	NaN
4	a2	b2	c2	d2
5	a3	b3	c3	d3
6	a4	b4	c4	d4
7	a5	b5	c5	d5

| Binding DataFrames

- ▶ Axis=1 option binds the DataFrame in the left/right column directions.

```
In [8]: pd.concat([df1,df2], axis=1)
```

```
Out[8]:
```

	a	b	c	a	b	c	d
0	a0	b0	c0	NaN	NaN	NaN	NaN
1	a1	b1	c1	NaN	NaN	NaN	NaN
2	a2	b2	c2	a2	b2	c2	d2
3	a3	b3	c3	a3	b3	c3	d3
4	NaN	NaN	NaN	a4	b4	c4	d4
5	NaN	NaN	NaN	a5	b5	c5	d5

Merging DataFrames

- The concat() function is a concept that merges the two DataFrames by a certain criterion in a manner similar to SQL's join command. In this case, the column or index that is the reference is referred to as a key. The key must exist in both DataFrames.

```
In [9]: # Creating a DataFrame with Stock Market Data
df1 = pd.read_excel('./stock price.xlsx')
df2 = pd.read_excel('./stock valuation.xlsx')
```

```
In [10]: df1.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   id          10 non-null    int64  
 1   stock_name  10 non-null    object  
 2   value        10 non-null    float64 
 3   price        10 non-null    int64  
dtypes: float64(1), int64(2), object(1)
memory usage: 448.0+ bytes
```

I Merging DataFrames

```
In [11]: df1.head()
```

```
Out[11]:
```

	id	stock_name	value	price
0	128940	Hanmi Pharmaceutical	59385.666667	421000
1	130960	CJ E&M	58540.666667	98900
2	138250	NS Shopping	14558.666667	13200
3	139480	E-mart	239230.833333	254500
4	142280	Green Cross Medical Science Corporation	468.833333	10200

I Merging DataFrames

```
In [12]: df2.head()
```

Out[12]:

	id	name	eps	bps	per	pbr
0	130960	CJ E&M	6301.333333	54068	15.695091	1.829178
1	136480	Harim Co.	274.166667	3551	11.489362	0.887074
2	138040	Meritz Financial Group	2122.333333	14894	6.313806	0.899691
3	139480	E-mart	18268.166667	295780	13.931338	0.860437
4	145990	Samyang	5741.000000	108090	14.283226	0.758627

- ▶ The key here must be an id with a non-overlapping value.

Merging DataFrames

- ▶ See help(pd.merge) for more detail. Let's just put two DataFrames as parameters and merge them.

```
In [13]: pd.merge(df1, df2)
```

Out[13]:

	id	stock_name	value	price	name	eps	bps	per	pbr
0	130960	CJ E&M	58540.666667	98900	CJ E&M	6301.333333	54068	15.695091	1.829178
1	139480	E-mart	239230.833333	254500	E-mart	18268.166667	295780	13.931338	0.860437
2	145990	Samyang	82750.000000	82000	Samyang	5741.000000	108090	14.283226	0.758627
3	185750	Chong Kun Dang	40293.666667	100500	Chong Kun Dang	3990.333333	40684	25.185866	2.470259
4	204210	Mode Tour Reit	3093.333333	3475	Mode Tour Reit	85.166667	5335	40.802348	0.651359

I Merging DataFrames

```
In [14]: df_1 = pd.read_excel('./stock price.xlsx')
df_2 = pd.read_excel('./stock valuation.xlsx')
```

```
In [15]: pd.merge(df_1, df_2)
```

Out[15]:

	id	stock_name	value	price	name	eps	bps	per	pbr
0	130960	CJ E&M	58540.666667	98900	CJ E&M	6301.333333	54068	15.695091	1.829178
1	139480	E-mart	239230.833333	254500	E-mart	18268.166667	295780	13.931338	0.860437
2	145990	Samyang	82750.000000	82000	Samyang	5741.000000	108090	14.283226	0.758627
3	185750	Chong Kun Dang	40293.666667	100500	Chong Kun Dang	3990.333333	40684	25.185866	2.470259
4	204210	Mode Tour Reit	3093.333333	3475	Mode Tour Reit	85.166667	5335	40.802348	0.651359

- ▶ If there are no common column names, an error occurs.

Merging DataFrames

- ▶ Here is how to solve the error.
- ▶ Bring the file again.

```
In [16]: df1 = pd.read_excel('./stock price.xlsx')
df2 = pd.read_excel('./stock valuation.xlsx')
```

```
In [17]: help(pd.merge)
```

Help on function merge in module pandas.core.reshape.merge:

```
merge(left, right, how: str = 'inner', on=None, left_on=None, right_on=None, left_index: bool = False, right_index: bool = False, sort: bool = False, suffixes=('_x', '_y'), copy: bool = True, indicator: bool = False, validate=None) -> 'DataFrame'
```

Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes ***will be ignored***. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.
When performing a cross merge, no column specifications to merge on are allowed.

Merging DataFrames

- ▶ Here is how to solve the error.

```
In [18]: pd.merge(df_1, df_2)
```

Out[18]:

	id	stock_name	value	price	name	eps	bps	per	pbr
0	130960	CJ E&M	58540.666667	98900	CJ E&M	6301.333333	54068	15.695091	1.829178
1	139480	E-mart	239230.833333	254500	E-mart	18268.166667	295780	13.931338	0.860437
2	145990	Samyang	82750.000000	82000	Samyang	5741.000000	108090	14.283226	0.758627
3	185750	Chong Kun Dang	40293.666667	100500	Chong Kun Dang	3990.333333	40684	25.185866	2.470259
4	204210	Mode Tour Reit	3093.333333	3475	Mode Tour Reit	85.166667	5335	40.802348	0.651359

Merging DataFrames

- ▶ The `on=None` and `how='inner'` options are applied as default values. The `on=None` option means merging all columns that belong in common to the two DataFrames into a reference (key).
- ▶ The `how='inner'` option means that data in the reference column is extracted only when the data is an intersection common to both DataFrames.
- ▶ We merged and returned five commonly existing stocks based on the column "id."

Merging DataFrames

- Let's deliberately change and see what happens if there are columns "id_1" and "name_1" on the left side of the DataFrame and if there are columns "id_1" and "name_1" on its right side.

```
In [19]: df1_3 = pd.read_excel('./stock price.xlsx')
df2_3 = pd.read_excel('./stock valuation.xlsx')
```

```
In [20]: pd.merge(df1_3, df2_3)
```

Out[20]:

	id	stock_name	value	price	name	eps	bps	per	pbr
0	130960	CJ E&M	58540.666667	98900	CJ E&M	6301.333333	54068	15.695091	1.829178
1	139480	E-mart	239230.833333	254500	E-mart	18268.166667	295780	13.931338	0.860437
2	145990	Samyang	82750.000000	82000	Samyang	5741.000000	108090	14.283226	0.758627
3	185750	Chong Kun Dang	40293.666667	100500	Chong Kun Dang	3990.333333	40684	25.185866	2.470259
4	204210	Mode Tour Reit	3093.333333	3475	Mode Tour Reit	85.166667	5335	40.802348	0.651359

- When the two columns are the same, the results are merged based on the values in the common column name.

Merging DataFrames

- If you use `how='left'`, all the companies in the left DataFrame will be returned, and those not on the right will be treated as NaN.

```
In [21]: df1 = pd.read_excel('./stock price.xlsx')
df2 = pd.read_excel('./stock valuation.xlsx')
```

```
In [22]: pd.merge(df1, df2, on='id', how='left')
```

Out[22]:

		id	stock_name	value	price	name	eps	bps	per	pbr
0	128940		Hanmi Pharmaceutical	59385.666667	421000		NaN	NaN	NaN	NaN
1	130960		CJ E&M	58540.666667	98900	CJ E&M	6301.333333	54068.0	15.695091	1.829178
2	138250		NS Shopping	14558.666667	13200		NaN	NaN	NaN	NaN
3	139480		E-mart	239230.833333	254500	E-mart	18268.166667	295780.0	13.931338	0.860437
4	142280	Green Cross Medical Science Corporation		468.833333	10200		NaN	NaN	NaN	NaN
5	145990		Samyang	82750.000000	82000	Samyang	5741.000000	108090.0	14.283226	0.758627
6	185750		Chong Kun Dang	40293.666667	100500	Chong Kun Dang	3990.333333	40684.0	25.185866	2.470259
7	192400		Cuckoo Holdings	179204.666667	177500		NaN	NaN	NaN	NaN
8	199800		Toolgen	-2514.333333	115400		NaN	NaN	NaN	NaN
9	204210		Mode Tour Reit	3093.333333	3475	Mode Tour Reit	85.166667	5335.0	40.802348	0.651359

Merging DataFrames

- If you `how='right'`, all the companies in the right DataFrame will be returned, and those not on the left will be treated as NaN.

```
In [23]: pd.merge(df1, df2, on='id', how='right')
```

	id	stock_name	value	price		name	eps	bps	per	pbr
0	130960	CJ E&M	58540.666667	98900.0		CJ E&M	6301.333333	54068	15.695091	1.829178
1	136480	NaN	NaN	NaN		Harim Co.	274.166667	3551	11.489362	0.887074
2	138040	NaN	NaN	NaN	Meritz Financial Group	2122.333333	14894	6.313806	0.899691	
3	139480	E-mart	239230.833333	254500.0		E-mart	18268.166667	295780	13.931338	0.860437
4	145990	Samyang	82750.000000	82000.0		Samyang	5741.000000	108090	14.283226	0.758627
5	161390	NaN	NaN	NaN		Hankook Tire	5648.500000	51341	7.453306	0.820007
6	181710	NaN	NaN	NaN	NHN Entertainment	2110.166667	78434	30.755864	0.827447	
7	185750	Chong Kun Dang	40293.666667	100500.0		Chong Kun Dang	3990.333333	40684	25.185866	2.470259
8	204210	Mode Tour Reit	3093.333333	3475.0		Mode Tour Reit	85.166667	5335	40.802348	0.651359
9	207940	NaN	NaN	NaN	Samsung BioLogics	4644.166667	60099	89.790059	6.938551	

Merging DataFrames

- If you use how="outer," all the data on the left and right is returned.

```
In [24]: pd.merge(df1, df2, on='id', how='outer')
```

Out[24]:

	id	stock_name	value	price	name	eps	bps	per	pbr
0	128940	Hanmi Pharmaceutical	59385.666667	421000.0	NaN	NaN	NaN	NaN	NaN
1	130960	CJ E&M	58540.666667	98900.0	CJ E&M	6301.333333	54068.0	15.695091	1.829178
2	138250	NS Shopping	14558.666667	13200.0	NaN	NaN	NaN	NaN	NaN
3	139480	E-mart	239230.833333	254500.0	E-mart	18268.166667	295780.0	13.931338	0.860437
4	142280	Green Cross Medical Science Corporation	468.833333	10200.0	NaN	NaN	NaN	NaN	NaN
5	145990	Samyang	82750.000000	82000.0	Samyang	5741.000000	108090.0	14.283226	0.758627
6	185750	Chong Kun Dang	40293.666667	100500.0	Chong Kun Dang	3990.333333	40684.0	25.185866	2.470259
7	192400	Cuckoo Holdings	179204.666667	177500.0	NaN	NaN	NaN	NaN	NaN
8	199800	Toolgen	-2514.333333	115400.0	NaN	NaN	NaN	NaN	NaN
9	204210	Mode Tour Reit	3093.333333	3475.0	Mode Tour Reit	85.166667	5335.0	40.802348	0.651359
10	136480	NaN	NaN	NaN	Harim Co.	274.166667	3551.0	11.489362	0.887074
11	138040	NaN	NaN	NaN	Meritz Financial Group	2122.333333	14894.0	6.313806	0.899691
12	161390	NaN	NaN	NaN	Hankook Tire	5648.500000	51341.0	7.453306	0.820007
13	181710	NaN	NaN	NaN	NHN Entertainment	2110.166667	78434.0	30.755864	0.827447
14	207940	NaN	NaN	NaN	Samsung BioLogics	4644.166667	60099.0	89.790059	6.938551

Coding Exercise #0106



Follow practice steps on 'ex_0106.ipynb' file

Coding Exercise #0107



Follow practice steps on 'ex_0107.ipynb' file

Unit 2.

Optimal Data Exploration Through Pandas

- | 2.1. Pipelines: Data Structures According to Data Types
- | 2.2. Pandas Series and DataFrames
- | 2.3. Merging and Binding DataFrames
- | **2.4. DataFrame Sorting and Multi-Index**
- | 2.5. Examining the Characteristics of Data Through Descriptive Statistics and Data Samples

DataFrame Manipulation

I Sorting:

- ▶ It is possible to sort the rows of a DataFrame using one or more columns.

```
In [ ]: 1 df.sort_values(by='bloodtype')
          2 df.sort_values(by='bloodtype', ascending=False)
          3 df.sort_values(by=['bloodtype', 'gender'])
```

Line1

- Sort in ascending order.

Line2

- Sort in descending order.

Line3

- Sort using two columns.

| Hierarchical indexing with MultiIndex:

```
In [1]: import numpy as np  
import pandas as pd
```

```
In [2]: 1 my_header = ['a', 'b', 'c']  
2 my_index_out = ['G1']*3 + ['G2']*3  
3 my_index_in = [1,2,3]*2  
4 my_index_zipped = list(zip(my_index_out, my_index_in))  
5 my_index = pd.MultiIndex.from_tuples(my_index_zipped)  
6 df = pd.DataFrame(data=np.random.randn(6,3), index=my_index, columns=my_header)  
7 df
```

Line 2-1

- Column names

Line 2-2

- Labels for the outer layer

Line 2-3

- Labels for the inner layer

| Hierarchical indexing with MultiIndex:

```
In [1]: import numpy as np  
import pandas as pd
```

```
In [2]: 1 my_header = ['a', 'b', 'c']  
2 my_index_out = ['G1']*3 + ['G2']*3  
3 my_index_in = [1,2,3]*2  
4 my_index_zipped = list(zip(my_index_out, my_index_in))  
5 my_index = pd.MultiIndex.from_tuples(my_index_zipped)  
6 df = pd.DataFrame(data=np.random.randn(6,3), index=my_index, columns=my_header)  
7 df
```

Line 2-4

- Create a list of tuples with the labels.

Line 2-5

- Create the MultiIndex.

Line 2-6

- Apply the MultiIndex.

| Hierarchical indexing with MultiIndex:

Out [2]:

		a	b	c
	G1	1 -0.955838	0.743064	0.444978
		2 1.079126	-1.212418	-0.816950
		3 1.368649	0.904057	-2.603676
	G2	1 -0.126599	-1.603466	1.456226
		2 1.505663	-1.034911	-2.629897
		3 0.459346	3.236755	-0.651125

Unit 2.

Optimal Data Exploration Through Pandas

- | 2.1. Pipelines: Data Structures According to Data Types
- | 2.2. Pandas Series and DataFrames
- | 2.3. Merging and Binding DataFrames
- | 2.4. DataFrame Sorting and Multi-Index
- | **2.5. Examining the Characteristics of Data Through Descriptive Statistics and Data Samples**

DataFrame Summarization

| Grouping and Summarizing:

```
In [1]: import pandas as pd
```

```
In [2]: df=pd.read_csv('data_studentlist.csv')
```

```
In [3]: df.head(3)
```

```
Out[3]:
```

		name	gender	age	grade	absence	bloodtype	height	weight
0	Jared	Diamond	M	23	3	Y	0	165.3	68.2
1	Sarah	O'Donnell	F	22	2	N	AB	170.1	53.0
2	Brian	Martin	M	24	4	N	B	175.0	80.1

```
In [4]: df.groupby(['gender','bloodtype'])['height'].mean()
```

```
Out[4]:
```

gender	bloodtype	height
F	A	172.450000
	AB	170.100000
	B	158.200000
M	O	164.433333
	A	165.700000
	AB	181.050000
	B	174.550000
O	166.200000	

Name: height, dtype: float64

Pivoting:

- Manipulate the indices and the columns and then summarize.

```
In [7]: my_dict = {"Size": ["L", "L", "M", "M", "M", "S", "S", "S", "S"],  
                 "Type": ["A", "A", "A", "B", "B", "A", "A", "B", "B"],  
                 "Location": ["L1", "L1", "L1", "L2", "L2", "L1", "L2", "L2", "L1"],  
                 "A": [1, 2, 2, 3, 3, 4, 5, 6, 7],  
                 "B": [2, 4, 5, 5, 6, 6, 8, 9, 9]}  
df = pd.DataFrame(my_dict)  
df
```

Out[7]:

	Size	Type	Location	A	B
0	L	A	L1	1	2
1	L	A	L1	2	4
2	M	A	L1	2	5
3	M	B	L2	3	5
4	M	B	L2	3	6
5	S	A	L1	4	6
6	S	A	L2	5	8
7	S	B	L2	6	9
8	S	B	L1	7	9

Pivoting:

- Index by 'Size' and 'Type.' Columns by 'Location.' Values provided by the 'B' column.

```
In [9]: dfr = pd.pivot_table(df, index=['Size','Type'], columns='Location', values='B')  
dfr
```

```
Out[9]:
```

		Location	L1	L2
Size	Type			
L	A	3.0	NaN	
	A	5.0	NaN	
M	B	NaN	5.5	
	A	6.0	8.0	
S	B	9.0	9.0	

| Pivoting:

```
In [10]: dfr.columns
```

```
Out[10]: Index(['L1', 'L2'], dtype='object', name='Location')
```

```
In [11]: dfr.index
```

```
Out[11]: MultiIndex([(L, 'A'),  
                      (M, 'A'),  
                      (M, 'B'),  
                      (S, 'A'),  
                      (S, 'B')],  
                     names=['Size', 'Type'])
```

Pivoting:

```
In [12]: pd.pivot_table(df, index=['Size', 'Type'], columns='Location', values='B', fill_value=0)
```

```
Out[12]:
```

Size	Type	Location	
		L1	L2
L	A	3	0.0
	A	5	0.0
M	B	0	5.5
	A	6	8.0
S	B	9	9.0

```
Out[9]:
```

Size	Type	Location	
		L1	L2
L	A	3.0	NaN
	A	5.0	NaN
M	B	NaN	5.5
	A	6.0	8.0
S	B	9.0	9.0

- The same as the graph on the right, but fill the missing values with 0.

Pivoting:

```
In [14]: import numpy as np
```

```
In [15]: pd.pivot_table(df, index=['Size','Type'], columns='Location', values='B', aggfunc = np.mean, fill_value=0)
```

```
Out[15]:
```

		Location	L1	L2
Size	Type			
L	A	3	0.0	
	A	5	0.0	
M	B	0	5.5	
	A	6	8.0	
S	B	9	9.0	
	A	3	0.0	

```
Out[12]:
```

		Location	L1	L2
Size	Type			
L	A	3	0.0	
	A	5	0.0	
M	B	0	5.5	
	A	6	8.0	
S	B	9	9.0	
	A	3	0.0	

- The same as the graph on the right with the aggregation function specified.

Pivoting:

- ▶ Index by 'Location.' Columns by 'Size' and 'Type.' Values provided by the 'B' column.

```
In [16]: dfr = pd.pivot_table(df, index='Location', columns=['Size','Type'], values='B')  
dfr
```

```
Out[16]:
```

	Size	L		M		S		
		Type	A	A	B	A	B	
Location								
L1	3.0	5.0	NaN	6.0	9.0			
L2	NaN	NaN	5.5	8.0	9.0			

Pivoting:

```
In [17]: dfr.index
```

```
Out[17]: Index(['L1', 'L2'], dtype='object', name='Location')
```

```
In [18]: dfr.columns
```

```
Out[18]: MultiIndex([['L', 'A'],
                      ['M', 'A'],
                      ['M', 'B'],
                      ['S', 'A'],
                      ['S', 'B']],
                     names=['Size', 'Type'])
```

Line 18

- Now, MultiIndex object for the columns.

Pivoting:

- The aggregation function is NumPy.median().

```
In [19]: pd.pivot_table(df, index=['Size','Type'], columns='Location', values='B', aggfunc = np.median, fill_value=0)
```

Out[19]:

		Location	
		L1	L2
Size	Type		
L	A	3	0.0
	A	5	0.0
M	B	0	5.5
	A	6	8.0
S	B	9	9.0

Pivoting:

- ▶ Group averages of the columns 'A' and 'B'

```
In [20]: pd.pivot_table(df, index=['Size', 'Type'], values=['A', 'B'], aggfunc=np.mean)
```

```
Out[20]:
```

		A	B
Size	Type		
L	A	1.5	3.0
	A	2.0	5.0
M	B	3.0	5.5
	A	4.5	7.0
S	B	6.5	9.0

Pivoting:

- Now, with groupby() method. The result is the same.

```
In [21]: df.groupby(['Size', 'Type'])[['A', 'B']].mean()
```

```
Out[21]:
```

		A	B
	Size	Type	
L	A	1.5	3.0
	B	2.0	5.0
M	A	3.0	5.5
	B	4.5	7.0
S	A	6.5	9.0
	B		

Pivoting:

- ▶ Aggregate the columns 'A' and 'B' differently.

```
In [22]: pd.pivot_table(df, index=['Size', 'Type'], values=['A', 'B'], aggfunc={'A':np.max, 'B':np.min})
```

```
Out[22]:
```

		A	B
Size	Type		
L	A	2	2
	B	3	5
M	A	2	5
	B	3	5
S	A	5	6
	B	7	9

I Statistics:

```
In [ ]: 1 df.sum(axis=0)
          2 df.sum(axis=1)
          3 df.mean(axis=0, skipna=False)
          4 df.describe()
          5 df.count(axis=0)
          6 df.A.corr(df.B)
          7 df.corr()
          8 df.corrwith(df.A)
```

Line1

- Column sums

Line2

- Row sums

Line3

- Column averages without skipping the missing values

Line4

- Descriptive statistics of the columns (variables)

I Statistics:

```
In [ ]: 1 df.sum(axis=0)
          2 df.sum(axis=1)
          3 df.mean(axis=0, skipna=False)
          4 df.describe()
          5 df.count(axis=0)
          6 df.A.corr(df.B)
          7 df.corr()
          8 df.corrwith(df.A)
```

Line 5

- Non-missing values along the columns

Line 6

- Correlation between the column 'A' and the column 'B'

Line 7

- Correlation matrix taking the numeric variables pair-wise

Line 8

- Correlations between 'A' and the other numeric variables

I Missing value detection and processing

```
In [ ]: 1 df.isnull()  
2 (df.isnull()).sum(axis=0)  
3 (df.isnull()).mean(axis=0)  
4 df.dropna(axis = 0)  
5 df.dropna(axis = 1)  
6 df.dropna(axis=0, thresh = 3)  
7 df.fillna(value=0)
```

Line 1

- A DataFrame with **True** where missing values are found.

Line 2

- Count the missing values for each column.

Line 3

- Proportions of the missing values for each column.

Line 4

- Drop the rows where one or more missing values are found.

I Missing value detection and processing.

```
In [ ]: 1 df.isnull()  
2 (df.isnull()).sum(axis=0)  
3 (df.isnull()).mean(axis=0)  
4 df.dropna(axis = 0)  
5 df.dropna(axis = 1)  
6 df.dropna(axis=0, thresh = 3)  
7 df.fillna(value=0)
```

Line 5

- Drop the columns where one or more missing values are found.

Line 6

- Drop the rows with less than 3 normal values.

Line 7

- Fill the missing values with 0.

Coding Exercise #0108



Follow practice steps on 'ex_0108.ipynb' file

Coding Exercise #0109



Follow practice steps on 'ex_0109.ipynb' file

Unit 3.

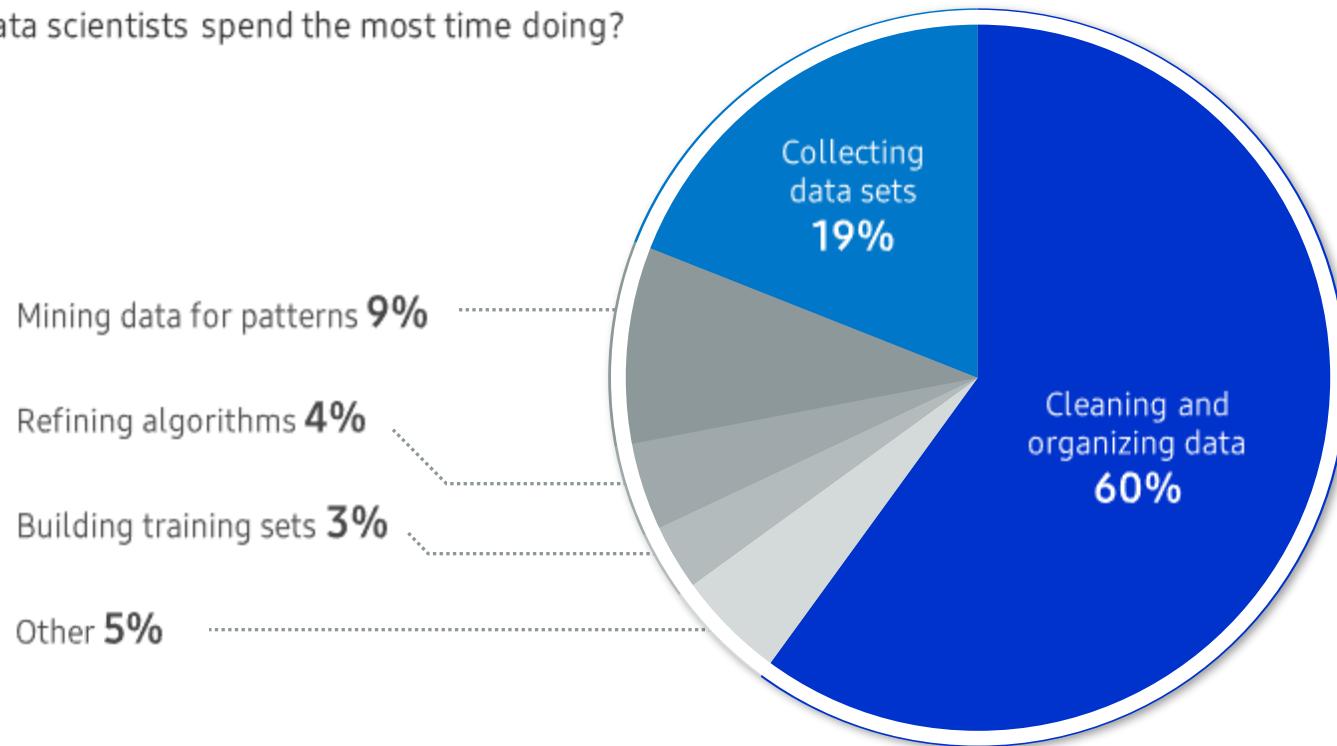
Pandas Data Preprocessing for Optimal Model Execution

- | 3.1. Data Preprocessing
- | 3.2. Identifying Data Properties
- | 3.3. Checking for Missing Data
- | 3.4. Checking and Processing Duplicate Data
- | 3.5. Data Feature Engineering

Data Preprocessing

| Data scientist survey:

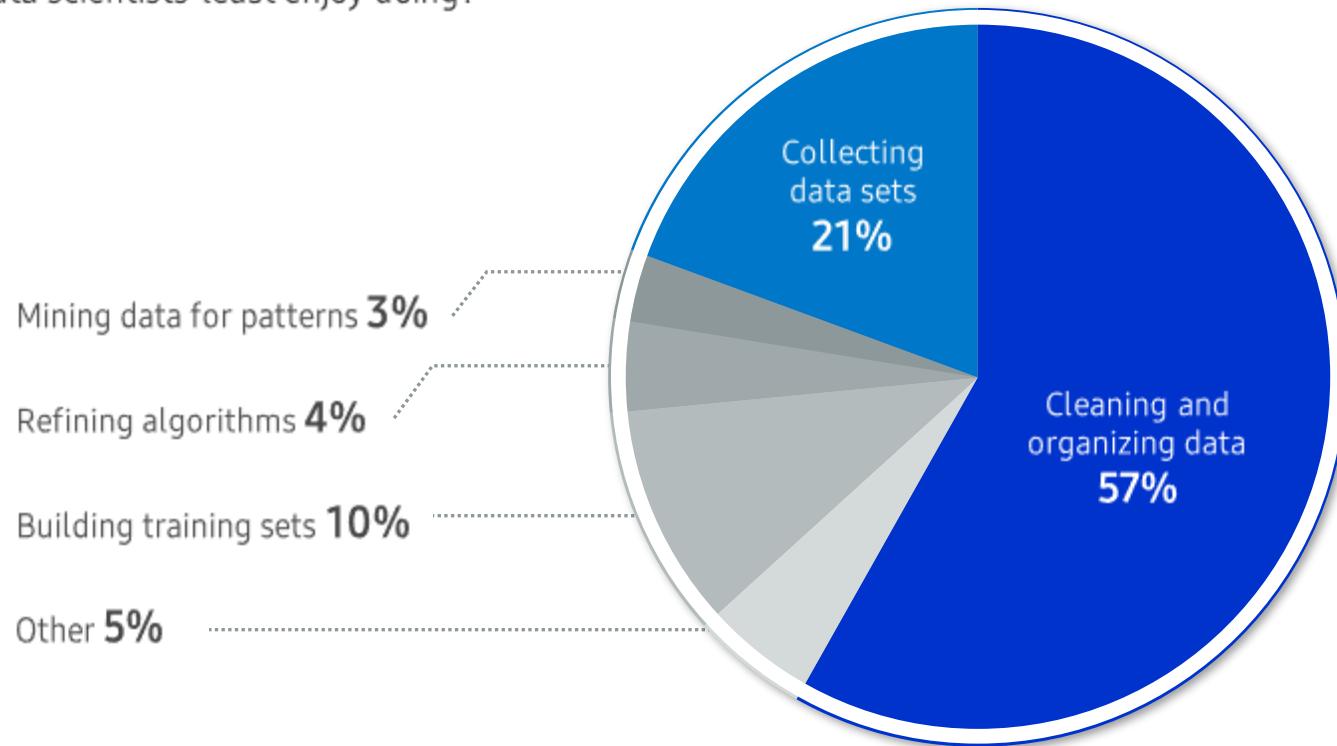
- ▶ What do data scientists spend the most time doing?



<https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#790d18c36f63>

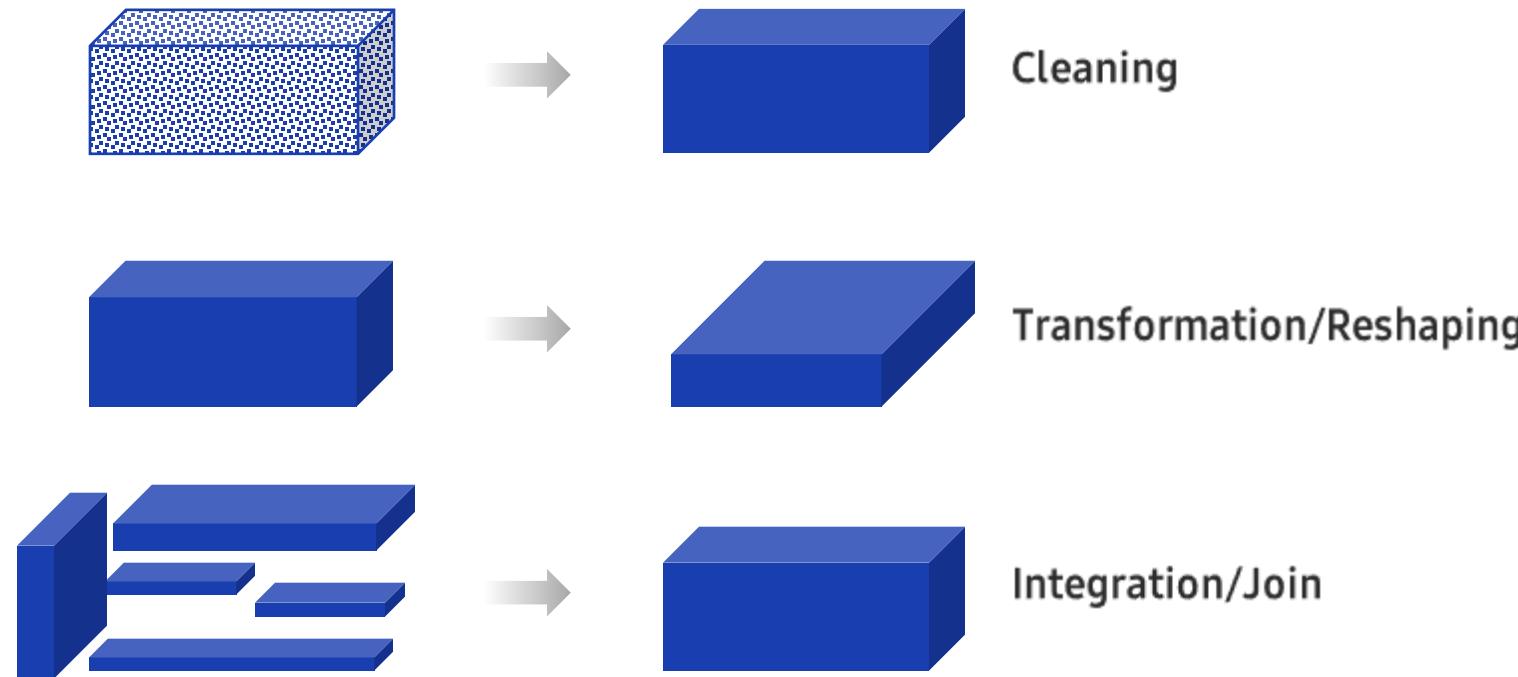
| Data scientist survey:

- ▶ What do data scientists least enjoy doing?

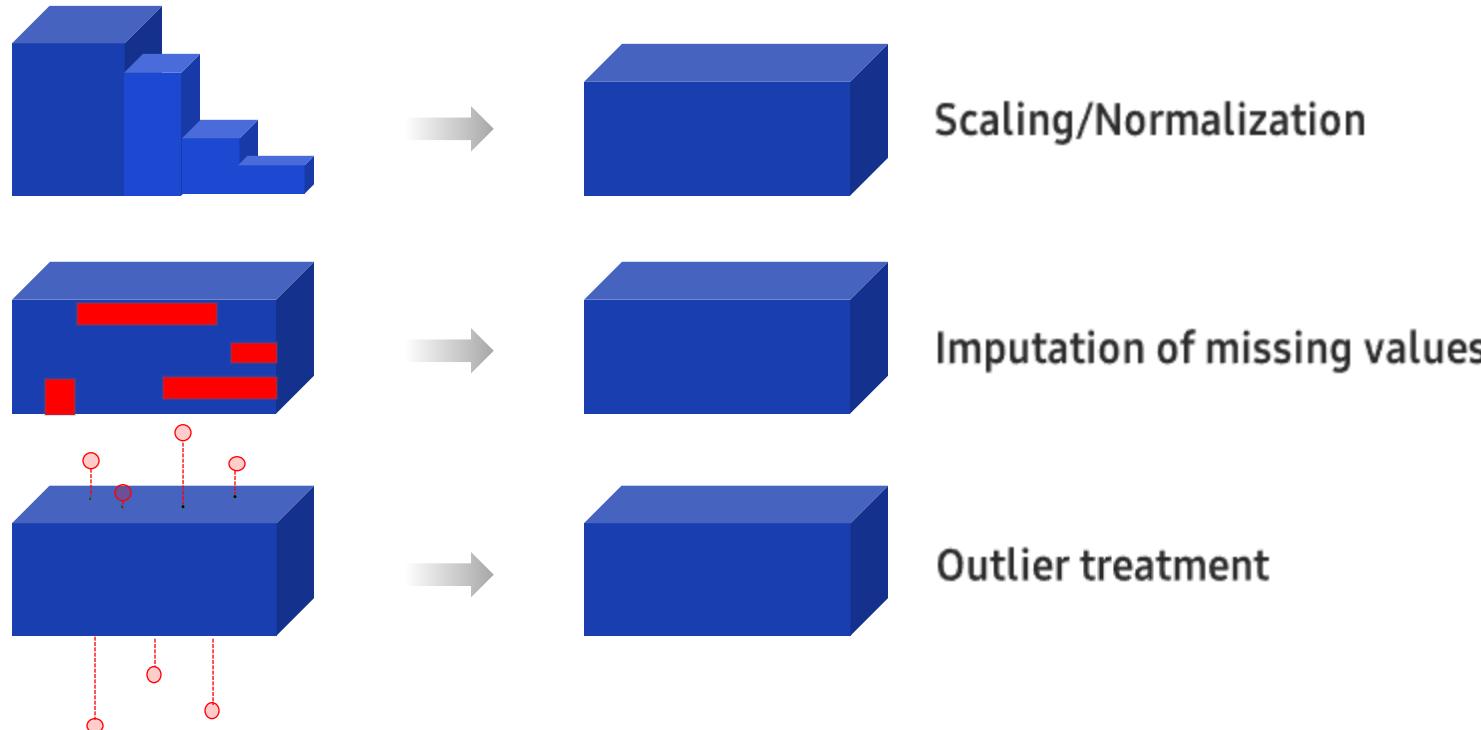


<https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#790d18c36f63>

I Operations:



I Operations:



| Algorithm considerations:

- ▶ Often, machine learning algorithms can be categorized into “Tree-like” and “non-Tree-like.”
 - a) Tree-like algorithms: Tree, Random Forest, AdaBoost, XGBoost, etc.
 - b) Non-Tree-like algorithms: Linear Regression, Logistic Regression, SVM, Neural Network, etc.
- ▶ Scaling/Normalization and Outlier treatment can be needed for non-Tree-like algorithms, but not for Tree-like algorithms.
- ▶ This is because Tree-like algorithms partition the configuration space into “patches,” mostly unaffected by scales or outliers.
- ▶ Other preprocessing operations are equally applicable for both Tree-like and non-Tree-like.

Unit 3.

Pandas Data Preprocessing for Optimal Model Execution

- | 3.1. Data Preprocessing
- | **3.2. Identifying Data Properties**
- | 3.3. Checking for Missing Data
- | 3.4. Checking and Processing Duplicate Data
- | 3.5. Data Feature Engineering

Scale

| Data Classification

- ▶ Continuous Scale data and Categorical data are the most basic types of structured data.
- ▶ Continuous data include continuity data, such as wind speed and duration, and discrete data, such as the frequency of occurrence of events.
- ▶ Categorical data refers to the city names of each country (Washington, New York, Los Angeles) or the type of car (bus, taxi, truck).
- ▶ Binary data is a special case of having either value, such as 0 and 1, yes/no, or true/false, among categorical types.
- ▶ Among the categorical types, the ratings (1,2,3,4,5) in which the values within the category are ranked are called Ordinal data.

I Why Classify Data?

- ▶ Data science software such as Python utilizes this data type information for computational performance.
- ▶ The type of data determines how to perform the calculation related to the variable.
- ▶ For example, in R or Python, ordinal data is classified into ordered.factor and is used to maintain the order desired by users in charts, tables, and statistical models.

Unit 3.

Pandas Data Preprocessing for Optimal Model Execution

- | 3.1. Data Preprocessing
- | 3.2. Identifying Data Properties
- | **3.3. Checking for Missing Data**
- | 3.4. Checking and Processing Duplicate Data
- | 3.5. Data Feature Engineering

Missing Value

Missing Value

- ▶ Missing value means refers to missing data and empty data.
- ▶ It is impossible to distort the analysis results or apply functions.
- ▶ In some cases, the missing values generated on the variable are not related to other variables, so the missing values must be deleted and replaced according to the situation.
- ▶ Assuming that each variable follows a specific probability distribution, the distribution parameters are estimated and replaced.
- ▶ There are mean replacement, median replacement, and mode replacement.

I Checking for Missing Data

```
In [1]: 1 import seaborn as sns  
2 df = sns.load_dataset('titanic')  
3 nan_deck = df['deck'].value_counts(dropna=False)  
4 print(nan_deck)
```

```
NaN      688  
C        59  
B        47  
D        33  
E        32  
A        15  
F        13  
G         4  
Name: deck, dtype: int64
```

Line 1-1

- Import Library.

Line 1-2

- Load the Titanic Dataset.

Checking for Missing Data

```
In [1]: 1 import seaborn as sns  
2 df = sns.load_dataset('titanic')  
3 nan_deck = df['deck'].value_counts(dropna=False)  
4 print(nan_deck)
```

```
NaN      688  
C        59  
B        47  
D        33  
E        32  
A        15  
F        13  
G         4  
Name: deck, dtype: int64
```

Line 1-3

- Calculate the number of NaNs in the Deck column.

I Checking for Missing Data

```
In [2]: print(df.head().isnull())
```

```
survived  pclass   sex   age  sibsp  parch  fare  embarked  class \
0      False  False  False  False  False  False  False  False  False
1      False  False  False  False  False  False  False  False  False
2      False  False  False  False  False  False  False  False  False
3      False  False  False  False  False  False  False  False  False
4      False  False  False  False  False  False  False  False  False

      who  adult_male  deck  embark_town  alive  alone
0  False        False  True        False  False  False
1  False        False  False        False  False  False
2  False        False  True        False  False  False
3  False        False  False        False  False  False
4  False        False  True        False  False  False
```

Line 2

- Find the missing data using the isnull() method.

Checking for Missing Data

```
In [3]: print(df.head().notnull())
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	\
0	True	True	True	True	True	True	True	True	True	True	
1	True	True	True	True	True	True	True	True	True	True	
2	True	True	True	True	True	True	True	True	True	True	
3	True	True	True	True	True	True	True	True	True	True	
4	True	True	True	True	True	True	True	True	True	True	

	adult_male	deck	embark_town	alive	alone	
0	True	False		True	True	
1	True	True		True	True	
2	True	False		True	True	
3	True	True		True	True	
4	True	False		True	True	

Line 3

- Find the missing data using the notnull() method.

I Checking for Missing Data

```
In [4]: print(df.head().isnull().sum(axis=0))
```

```
survived      0
pclass        0
sex           0
age           0
sibsp         0
parch         0
fare          0
embarked      0
class         0
who           0
adult_male    0
deck          3
embark_town   0
alive         0
alone         0
dtype: int64
```



Line 4

- Calculate the number of missing data using the isnull() method.

| Applying the threshold=500 option to the dropna() method deletes all columns with 500 or more NaN values.

```
In [5]: df_thresh = df.dropna(axis=1, thresh=500)
print(df_thresh.columns)

Index(['survived', 'pclass', 'sex', 'age', 'sibsp', 'parch', 'fare',
       'embarked', 'class', 'who', 'adult_male', 'embark_town', 'alive',
       'alone'],
      dtype='object')
```

Line 5

- Delete all columns with more than 500 NaN values. Deck column (688 NaN values out of 891).

| Limit the subset to the column 'age.'

```
In [6]: df_age = df.dropna(subset=['age'], how='any', axis=0)
print(len(df_age))
```

714

Line 6

- Delete all rows without age data in the age column. Age column (177 NaN values out of 891).

Replacing Missing Values

Replacing the Missing Data with the Mean

```
In [7]: 1 mean_age = df['age'].mean(axis=0)
2 df['age'].fillna(mean_age, inplace=True)
3 print(df['age'].head(10))
```

```
0    22.000000
1    38.000000
2    26.000000
3    35.000000
4    35.000000
5    29.699118
6    54.000000
7    2.000000
8    27.000000
9    14.000000
Name: age, dtype: float64
```

Line 7-1

- Delete all rows without age data in the age column. Age Column (177 NaN values out of 891).

| Replacing the Missing Data with the Mean

```
In [7]: 1 mean_age = df['age'].mean(axis=0)
2 df['age'].fillna(mean_age, inplace=True)
3 print(df['age'].head(10))
```

```
0    22.000000
1    38.000000
2    26.000000
3    35.000000
4    35.000000
5    29.699118
6    54.000000
7     2.000000
8    27.000000
9    14.000000
Name: age, dtype: float64
```

Line 7-2

- Calculate the mean of the age column. (Excluding NaN values)

Line 7-3

- Print the first 10 data in the age column. (In row 5, NaN values are replaced by the mean.)

Practice with Missing Data

Checking for Missing Data

```
In [8]: 1 df = sns.load_dataset('titanic')
2 print(df['embark_town'][825:830])
3 print('\n')
```

```
825    Queenstown
826    Southampton
827    Cherbourg
828    Queenstown
829      NaN
Name: embark_town, dtype: object
```

Line 8-1

- Load the titanic dataset.

Line 8-2

- Print the NaN data of column embark_town and row 829.

| Replacing the Missing Data with the Mode

```
In [9]: most_freq = df['embark_town'].value_counts(dropna=True).idxmax()
print(most_freq)
print('\n')
```

Southampton

Line 9

- The NaN value of the embark_town column is replaced with the value that appears the most among the boarding cities.

| Replacing the Missing Data with the Mode

```
In [10]: df['embark_town'].fillna(most_freq, inplace=True)  
print(df['embark_town'][825:830])
```

```
825    Queenstown  
826    Southampton  
827    Cherbourg  
828    Queenstown  
829    Southampton  
Name: embark_town, dtype: object
```

Line 10

- Print the NaN data of row 829 and column embark_town. (NaN value is replaced by value most_freq.)

| Replacing the Missing Data with the Mode

```
In [11]: df = sns.load_dataset('titanic')
df['embark_town'].fillna(method='ffill', inplace=True)
print(df['embark_town'][825:830])
```

```
825    Queenstown
826    Southampton
827    Cherbourg
828    Queenstown
829    Queenstown
Name: embark_town, dtype: object
```

Line 11

- Change the NaN value of embark_town to the immediately preceding value of row 828.

Unit 3.

Pandas Data Preprocessing for Optimal Model Execution

- | 3.1. Data Preprocessing
- | 3.2. Identifying Data Properties
- | 3.3. Checking for Missing Data
- | 3.4. Checking and Processing Duplicate Data
- | 3.5. Data Feature Engineering

Processing Duplicate Data

Checking for Duplicate Data

```
In [12]: import pandas as pd
df = pd.DataFrame({'c1':['a','a','b','a','b'],
                    'c2':[1,1,1,2,2],
                    'c3':[1,1,2,2,2]})
print(df)
```

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

Line 12

- Create a DataFrame with duplicate data.

I Checking for Duplicate Data

```
In [13]: df_dup = df.duplicated()  
print(df_dup)
```

```
0    False  
1    True  
2   False  
3   False  
4   False  
dtype: bool
```

Line 13

- Find the duplicate values among the entire row data of the DataFrame.

| The data in line 1 becomes True because it overlaps with the previous row 0.

```
In [14]: col_dup = df['c2'].duplicated()  
print(col_dup)
```

```
0    False  
1    True  
2    True  
3    False  
4    True  
Name: c2, dtype: bool
```

Line14

- Find the duplicate value in the specific column data of the DataFrame.

I drop_duplicates() Method

```
In [15]: df2 = df.drop_duplicates()  
print(df2)
```

	c1	c2	c3
0	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

Line 15

- Find the duplicate value in the specific column data of the DataFrame.

| Column Reference Corresponding to the Subset Option

```
In [16]: df3 = df.drop_duplicates(subset=['c2', 'c3'])  
print(df3)
```

	c1	c2	c3
0	a	1	1
2	b	1	2
3	a	2	2

Line 16

- Remove duplicate rows based on columns c2 and c3.

Unit 3.

Pandas Data Preprocessing for Optimal Model Execution

- | 3.1. Data Preprocessing
- | 3.2. Identifying Data Properties
- | 3.3. Checking for Missing Data
- | 3.4. Checking and Processing Duplicate Data
- | **3.5. Data Feature Engineering**

Feature Engineering

Derived variables:

- ▶ Create one or more variables based on other variable(s).
Ex From the “Date” variable, derive the “Year,” “Month,” “Day,” “Weekday,” etc.

Date
2009-8-21
2013-8-27
2019-2-11



Year	Month	Day	Weekday
2009	8	21	Friday
2013	8	27	Tuesday
2019	2	11	Monday

| Derived variables:

- ▶ Create one or more variables based on other variable(s).

Ex From the “X_coordinate” and “Y_coordinate,” derive the “Distance” variable:

$$\text{Distance} = \sqrt{\text{X_coordinate}^2 + \text{Y_coordinate}^2}$$

X_coordinate	Y_coordinate
3	4
5	12
9	12



Distance
5
13
15

| Derived variables:

- ▶ Create one or more variables based on other variable(s).

Ex From the “Price” and “Area(m2),” derive the “PricePerArea” variable:

$$\text{PricePerArea} = \text{Price} / \text{Area}$$

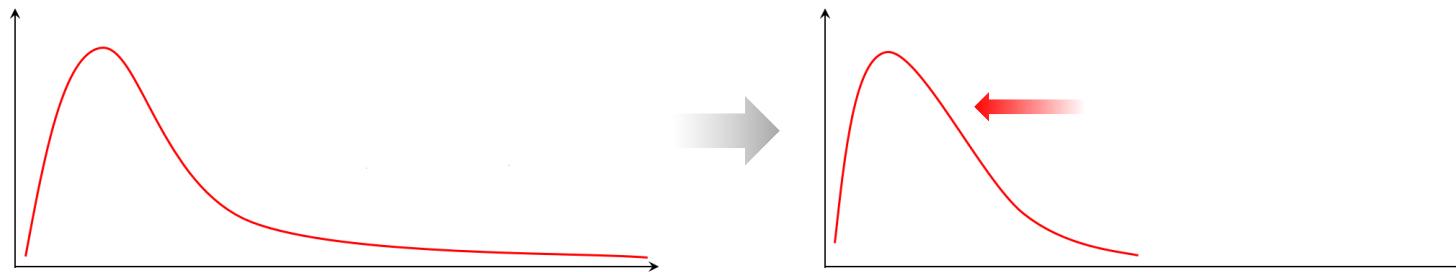
Price	Area
300,000	100
525,000	150
160,000	80



PricePerArea
3,000
3,500
2,000

| Derived variables:

- ▶ Principal components with or without dimensional reduction can be regarded as derived variables.
- ▶ In general, rotated coordinates can be regarded as derived variables.
- ▶ We can apply mathematical functions such as `log()` to a variable to create its derived variable.



With `log()`, we can ameliorate a positively skewed distribution making it more “normal.”

I Dealing with categorical variables: **Dummy variables**

- ▶ A dummy variable is a derived variable that can have only 0 or 1 as a value.
- ▶ A categorical variable generates *Category count* – 1 dummy variable.

Ex “Gender” variable with two category values: “male” and “female.”

→ Only one dummy variable is required: “Gender_male.”

Gender	Gender_male
female	0
male	1

I Dealing with categorical variables: **Dummy variables**

- ▶ A dummy variable is a derived variable that can have only 0 or 1 as a value.
- ▶ A categorical variable generates *Category count* – 1 dummy variable.

Ex “Species” variable with three category values: “setosa,” “versicolor,” and “virginica.”

→ Only two dummy variables are required: “Species_versicolor” and “Species_virginica.”

Species	Species_versicolor	Species_virginica
setosa	0	0
versicolor	1	0
virginica	0	1

| Dealing with categorical variables: **Dummy variables**

- ▶ Similar to dummy variables.
- ▶ In this case, there are as many dummy variables as the count of unique category values.

Ex One hot encoding for a variable that can have integer values 0~9.

→ Integer variables often represent category or class rather than the numeric value.

X0	X1	X2	X3	X4	X5	X6	X7	X8	X9
0	0	0	0	0	1	0	0	0	0

↑
5

I Dealing with categorical variables: **Label encoding**

- ▶ Encodes a categorical variable by assigning integers to the category values.
- ▶ Creates a new numeric variable (derived variable).
- ▶ OK with “Tree-like” algorithms, but **unsafe** with “non-Tree-like” algorithms.

Ex “Capital” with five category values [“London”, “Moscow”, “Paris”, “Seoul”, “Washington”] can be encoded as an integer variable “Capital_int” where $0 \leftrightarrow \text{“London”}$, $1 \leftrightarrow \text{“Moscow”}$, $2 \leftrightarrow \text{“Paris”}$, $3 \leftrightarrow \text{“Seoul”}$, $4 \leftrightarrow \text{“Washington”}$.

| Turning numerical variables into categorical:

- ▶ From a numerical variable, we can derive a categorical one that contains intervals as values.
- ▶ These intervals can be quantiles or custom-made.
- ▶ From the categorical variable, the corresponding dummy variables can be generated as usual.

Ex A numerical "Age" variable can be converted into "Age_10," "Age_20," etc.

Age	Age_10	Age_20	Age_30	Age_40	Age_50	Age_60	Age_70	Age_80
8	0	0	0	0	0	0	0	0
17	1	0	0	0	0	0	0	0
26	0	1	0	0	0	0	0	0
63	0	0	0	0	0	1	0	0

Matching the Same Measurement Unit Equally

| Convert miles per gallon into kilometers per liter (km/L).

```
In [1]: import pandas as pd  
import numpy as np
```

```
In [2]: 1 df = pd.read_csv('./auto-mpg.csv', header=None)  
2  
3 df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',  
4                 'acceleration', 'model year', 'origin', 'name']  
5 print(df.head(3))  
6 print('\n')
```

```
mpg cylinders displacement horsepower weight acceleration model year \
0 18.0          8         307.0       130.0   3504.0        12.0        70
1 15.0          8         350.0       165.0   3693.0        11.5        70
2 18.0          8         318.0       150.0   3436.0        11.0        70
```

```
origin name
0      1 chevrolet chevelle malibu
1      1           buick skylark 320
2      1           plymouth satellite
```



- Generate df with the read_csv() function.

| Convert miles per gallon into kilometers per liter (km/L).

```
In [1]: import pandas as pd  
import numpy as np
```

```
In [2]: 1 df = pd.read_csv('./auto-mpg.csv', header=None)  
2  
3 df.columns = ['mpg','cylinders','displacement','horsepower','weight',  
4                 'acceleration','model year','origin','name']  
5 print(df.head(3))  
6 print('\n')
```

```
mpg cylinders displacement horsepower weight acceleration model year \
0 18.0          8         307.0      130.0  3504.0        12.0       70
1 15.0          8         350.0      165.0  3693.0        11.5       70
2 18.0          8         318.0      150.0  3436.0        11.0       70
```

```
origin           name
0      1  chevrolet chevelle malibu
1      1            buick skylark 320
2      1  plymouth satellite
```

Line 2-3

- Designate the column name.

| The round (2) command rounds up the second digit below the decimal point.

```
In [3]: 1 mpg_to_kpl = 1.60934 / 3.78541
2
3 df['kpl'] = df['mpg'] * mpg_to_kpl
4 print(df.head(3))
5 print('\n')
6
7 df['kpl'] = df['kpl'].round(2)
8 print(df.head(3))
```

Line 3-1

- Convert mile per gallon to kilometer per liter ($\text{mpg_to_kpl} = 0.425$).

Line 3-3

- Add the result of multiplying the mpg column by 0.425 to the new column (kpl).

Line 3-7

- Round up the kpl column to the second place below the decimal point.

| The round(2) command rounds up the second digit below the decimal point.

```
mpg cylinders displacement horsepower weight acceleration model year \
0 18.0          8           307.0       130.0   3504.0        12.0      70
1 15.0          8           350.0       165.0   3693.0        11.5      70
2 18.0          8           318.0       150.0   3436.0        11.0      70
```

```
origin          name     kpl
0      1  chevrolet chevelle malibu 7.652571
1      1          buick skylark 320 6.377143
2      1  plymouth satellite    7.652571
```

```
mpg cylinders displacement horsepower weight acceleration model year \
0 18.0          8           307.0       130.0   3504.0        12.0      70
1 15.0          8           350.0       165.0   3693.0        11.5      70
2 18.0          8           318.0       150.0   3436.0        11.0      70
```

```
origin          name     kpl
0      1  chevrolet chevelle malibu 7.65
1      1          buick skylark 320 6.38
2      1  plymouth satellite    7.65
```

Data Type Conversion

| Check the original data type.

```
In [4]: 1 print(df.dtypes)
          2 print('\n')
          3
          4 print(df['horsepower'].unique())
          5 print('\n')
```

```
mpg           float64
cylinders     int64
displacement  float64
horsepower    object
weight         float64
acceleration  float64
model year    int64
origin         int64
name           object
kpl            float64
dtype: object
```

Line 4-1

- Check the data type of each column.

| Check the original data type.

```
In [4]: 1 print(df.dtypes)
2 print('\n')
3
4 print(df['horsepower'].unique())
5 print('\n')
```

```
mpg           float64
cylinders     int64
displacement  float64
horsepower    object
weight         float64
acceleration  float64
model year    int64
origin         int64
name           object
kpl            float64
dtype: object
```

Line 4-4

- Check the original value of the horsepower column.

| Check the original data type.

```
[ '130.0' '165.0' '150.0' '140.0' '198.0' '220.0' '215.0' '225.0' '190.0'  
 '170.0' '160.0' '95.00' '97.00' '85.00' '88.00' '46.00' '87.00' '90.00'  
 '113.0' '200.0' '210.0' '193.0' '?' '100.0' '105.0' '175.0' '153.0'  
 '180.0' '110.0' '72.00' '86.00' '70.00' '76.00' '65.00' '69.00' '60.00'  
 '80.00' '54.00' '208.0' '155.0' '112.0' '92.00' '145.0' '137.0' '158.0'  
 '167.0' '94.00' '107.0' '230.0' '49.00' '75.00' '91.00' '122.0' '67.00'  
 '83.00' '78.00' '52.00' '61.00' '93.00' '148.0' '129.0' '96.00' '71.00'  
 '98.00' '115.0' '53.00' '81.00' '79.00' '120.0' '152.0' '102.0' '108.0'  
 '68.00' '58.00' '149.0' '89.00' '63.00' '48.00' '66.00' '139.0' '103.0'  
 '125.0' '133.0' '138.0' '135.0' '142.0' '77.00' '62.00' '132.0' '84.00'  
 '64.00' '74.00' '116.0' '82.00']
```

| Convert the integer data type into a character data type using dictionary.

```
In [5]: print(df['origin'].unique())
```

```
[1 3 2]
```

```
In [6]: 1 df['origin'].replace({1:'USA', 2:'EU', 3:'JAPAN'}, inplace=True)
2
3 print(df['origin'].unique())
4 print(df['origin'].dtypes)
5 print('\n')
```

```
['USA' 'JAPAN' 'EU']
object
```

Line 5

- Check the original value of the origin column.

| Convert the integer data type into a character data type using dictionary.

```
In [5]: print(df['origin'].unique())
```

```
[1 3 2]
```

```
In [6]: 1 df['origin'].replace({1:'USA', 2:'EU', 3:'JAPAN'}, inplace=True)
2
3 print(df['origin'].unique())
4 print(df['origin'].dtypes)
5 print('\n')
```

```
['USA' 'JAPAN' 'EU']
object
```

Line 6-1

- Convert the integer data type into a character data type.

Line 6-3~6-4

- Check the original value and the data type of the origin column.

| Convert the string into a categorical type.

```
In [7]: 1 df['origin'] = df['origin'].astype('category')
2 print(df['origin'].dtypes)
3
4 df['origin'] = df['origin'].astype('str')
5 print(df['origin'].dtypes)
```

```
category
object
```

Line 7-1

- Convert the string data type of the origin column into a categorical type.

Line 7-4

- Convert the categorical type back to the string type.

| Convert the string into a categorical type.

```
In [8]: print(df['model year'].sample(3))
df['model year'] = df['model year'].astype('category')
print(df['model year'].sample(3))

264    78
15     70
194    76
Name: model year, dtype: int64
120    73
105    73
221    77
Name: model year, dtype: category
Categories (13, int64): [70, 71, 72, 73, ..., 79, 80, 81, 82]
```

Line 8

- Convert the integer type of the model year column into a categorical type.

Categorical Data Conversion (Division of Sections)

| Convert continuous variables into categorical discrete variables.

```
In [9]: 1 df = pd.read_csv('./auto-mpg.csv', header=None)
2
3 df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
4               'acceleration', 'model year', 'origin', 'name']
5
6 df['horsepower'].replace('?', np.nan, inplace=True)
7 df.dropna(subset=['horsepower'], axis=0, inplace=True)
8 df['horsepower'] = df['horsepower'].astype('float')
```

```
In [10]: count, bin_dividers = np.histogram(df['horsepower'], bins=3)
print(bin_dividers)

[ 46.          107.33333333 168.66666667 230. ]
```

Line 9-1

- Generate df with the read_csv() function.

| Convert continuous variables into categorical discrete variables.

```
In [9]: 1 df = pd.read_csv('./auto-mpg.csv', header=None)
2
3 df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
4               'acceleration', 'model year', 'origin', 'name']
5
6 df['horsepower'].replace('?', np.nan, inplace=True)
7 df.dropna(subset=['horsepower'], axis=0, inplace=True)
8 df['horsepower'] = df['horsepower'].astype('float')
```

```
In [10]: count, bin_dividers = np.histogram(df['horsepower'], bins=3)
print(bin_dividers)

[ 46.          107.33333333 168.66666667 230.          ]
```

Line 9-3~9-4

- Designate the column name.

Line 9-6~9-8

- Find the missing data ("?") of the horsepower column and convert it into a float.

| Convert continuous variables into categorical discrete variables.

```
In [9]: 1 df = pd.read_csv('./auto-mpg.csv', header=None)
2
3 df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
4               'acceleration', 'model year', 'origin', 'name']
5
6 df['horsepower'].replace('?', np.nan, inplace=True)
7 df.dropna(subset=['horsepower'], axis=0, inplace=True)
8 df['horsepower'] = df['horsepower'].astype('float')
```

```
In [10]: count, bin_dividers = np.histogram(df['horsepower'], bins=3)
print(bin_dividers)

[ 46.          107.33333333 168.66666667 230. ]
```

Line 9-6

- Change “?” to np.nan.

Line 9-7

- Delete the missing data row.

| Convert continuous variables into categorical discrete variables.

```
In [9]: 1 df = pd.read_csv('./auto-mpg.csv', header=None)
2
3 df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
4               'acceleration', 'model year', 'origin', 'name']
5
6 df['horsepower'].replace('?', np.nan, inplace=True)
7 df.dropna(subset=['horsepower'], axis=0, inplace=True)
8 df['horsepower'] = df['horsepower'].astype('float')
```

```
In [10]: count, bin_dividers = np.histogram(df['horsepower'], bins=3)
print(bin_dividers)

[ 46.          107.33333333 168.66666667 230. ]
```

Line 9-8

- Convert the string to a float.

Line 10

- Find a list of boundary values divided by three bin by the np.histogram function.

| Use the include_lowest=True option to include the low boundary values between countries.

```
In [11]: bin_names = ['Low output', 'Normal output', 'High output']

df['hp_bin'] = pd.cut(x=df['horsepower'],
                      bins=bin_dividers,
                      labels=bin_names,
                      include_lowest=True)

print(df[['horsepower', 'hp_bin']].head(15))
```

Line 11-1

- Designate the names for the 3 bins.

Line 11-3 ~ 11-6

- Each data corresponds to three bins as a pd.cut function.

Line 11-3

- Data array

| Use the include_lowest=True option to include the low boundary values between countries.

```
In [11]: bin_names = ['Low output', 'Normal output', 'High output']

df['hp_bin'] = pd.cut(x=df['horsepower'],
                      bins=bin_dividers,
                      labels=bin_names,
                      include_lowest=True)

print(df[['horsepower', 'hp_bin']].head(15))
```

 Line 11-4

- Boundary value list

 Line 11-5

- Bin names

| Use the include_lowest=True option to include the low boundary values between countries.

```
In [11]: bin_names = ['Low output', 'Normal output', 'High output']

df['hp_bin'] = pd.cut(x=df['horsepower'],
                      bins=bin_dividers,
                      labels=bin_names,
                      include_lowest=True)

print(df[['horsepower', 'hp_bin']].head(15))
```

Line 11-6

- Include the first boundary value.

Line 11-8

- Print the first 15 rows of the horse bower column, hp_bin column.

Categorical Data Conversion (Divisions of Sections)

Use the include_lowest=True option to include the low boundary values between countries.

	horsepower	hp_bin
0	130.0	Normal output
1	165.0	Normal output
2	150.0	Normal output
3	150.0	Normal output
4	140.0	Normal output
5	198.0	High output
6	220.0	High output
7	215.0	High output
8	225.0	High output
9	190.0	High output
10	170.0	High output
11	160.0	Normal output
12	150.0	Normal output
13	225.0	High output
14	95.0	Low output

Dummy Variable

- To use categorical data representing categories in machine learning algorithms, convert them into dummy variables represented by the number 0 or 1.

```
In [12]: df = pd.read_csv('./auto-mpg.csv', header=None)

df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'model year', 'origin', 'name']

df['horsepower'].replace('?', np.nan, inplace=True)
df.dropna(subset=['horsepower'], axis=0, inplace=True)
df['horsepower'] = df['horsepower'].astype('float')

count, bin_dividers = np.histogram(df['horsepower'], bins=3)
bin_names = ['Low output', 'Normal output', 'High output']
```



Line 12-1

- Generate df with the read_csv() function.

| To use categorical data representing categories in machine learning algorithms, convert them into dummy variables represented by the number 0 or 1.

```
In [12]: df = pd.read_csv('./auto-mpg.csv', header=None)

df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'model year', 'origin', 'name']

df['horsepower'].replace('?', np.nan, inplace=True)
df.dropna(subset=['horsepower'], axis=0, inplace=True)
df['horsepower'] = df['horsepower'].astype('float')

count, bin_dividers = np.histogram(df['horsepower'], bins=3)
bin_names = ['Low output', 'Normal output', 'High output']
```

Line 12-3~12-4

- Designate the column name.

Line 12-6~12-8

- Delete the missing data “?” in the horsepower column and convert it into a float.

| To use categorical data representing categories in machine learning algorithms, convert them into dummy variables represented by the number 0 or 1.

```
In [12]: df = pd.read_csv('./auto-mpg.csv', header=None)

df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'model year', 'origin', 'name']

df['horsepower'].replace('?', np.nan, inplace=True)
df.dropna(subset=['horsepower'], axis=0, inplace=True)
df['horsepower'] = df['horsepower'].astype('float')

count, bin_dividers = np.histogram(df['horsepower'], bins=3)
bin_names = ['Low output', 'Normal output', 'High output']
```

Line 12-6

- Change "?" to np.nan.

Line 12-7

- Delete the missing data row.

To use categorical data representing categories in machine learning algorithms, convert them into dummy variables represented by the number 0 or 1.

```
In [12]: df = pd.read_csv('./auto-mpg.csv', header=None)

df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'model year', 'origin', 'name']

df['horsepower'].replace('?', np.nan, inplace=True)
df.dropna(subset=['horsepower'], axis=0, inplace=True)
df['horsepower'] = df['horsepower'].astype('float')

count, bin_dividers = np.histogram(df['horsepower'], bins=3)
bin_names = ['Low output', 'Normal output', 'High output']
```

Line 12-8

- Convert the string into a float.

Line 10

- Find a list of boundary values divided by three bins using np.histogram.

To use categorical data representing categories in machine learning algorithms, convert them into dummy variables represented by the number 0 or 1.

```
In [12]: df = pd.read_csv('./auto-mpg.csv', header=None)

df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'model year', 'origin', 'name']

df['horsepower'].replace('?', np.nan, inplace=True)
df.dropna(subset=['horsepower'], axis=0, inplace=True)
df['horsepower'] = df['horsepower'].astype('float')

count, bin_dividers = np.histogram(df['horsepower'], bins=3)
bin_names = ['Low output', 'Normal output', 'High output']
```

Line 12-12

- Designate the name for three bins.

One-Hot Vector

- If the get_dummies() method is used, all original values of the categorical variables are each converted into new dummy variables.

```
In [13]: df['hp_bin'] = pd.cut(x=df['horsepower'],
                           bins=bin_dividers,
                           labels=bin_names,
                           include_lowest=True)

horsepower_dummies = pd.get_dummies(df['hp_bin']).astype('int')
print(horsepower_dummies.head(15))
```

Line 13-1 ~ 13-4

- Assign each data to three bins using pd.cut.

Line 13-1

- Data array

Line 13-2

- Boundary value list

- If the `get_dummies()` method is used, all original values of the categorical variables are each converted into new dummy variables.

```
In [13]: df['hp_bin'] = pd.cut(x=df['horsepower'],
                           bins=bin_dividers,
                           labels=bin_names,
                           include_lowest=True)

horsepower_dummies = pd.get_dummies(df['hp_bin']).astype('int')
print(horsepower_dummies.head(15))
```

Line 13-3

- Bin names

Line 13-4

- Include the first boundary value.

Line 13-6

- Convert the categorical data in the `hp_bin` column into dummy variables.

- If the `get_dummies()` method is used, all original values of the categorical variables are each converted into new dummy variables.

	Low output	Normal output	High output
0	0	1	0
1	0	1	0
2	0	1	0
3	0	1	0
4	0	1	0
5	0	0	1
6	0	0	1
7	0	0	1
8	0	0	1
9	0	0	1
10	0	0	1
11	0	1	0
12	0	1	0
13	0	0	1
14	1	0	0

Normalization

- The DataFrames are normalized so that decreased performance, due to the difference in the relative size of numerical data in the column (each variable), does not occur.

```
In [14]: 1 print(df.horsepower.describe())
2 print('\n')
3
4 df.horsepower = df.horsepower / abs(df.horsepower.max())
5
6 print(df.horsepower.head())
7 print('\n')
8 print(df.horsepower.describe())
```

Line 14-1

- Check the maximum value with statistical summary information of the horsepower column.

Line 14-4

- All data is divided and stored as the absolute value of the maximum value of the horsepower column.

- | The DataFrames are normalized so that decreased performance, due to the difference in the relative size of numerical data in the column (each variable), does not occur.

```
count    392.000000  
mean     104.469388  
std      38.491160  
min      46.000000  
25%     75.000000  
50%     93.500000  
75%    126.000000  
max    230.000000  
Name: horsepower, dtype: float64
```

```
count    392.000000  
mean     0.454215  
std      0.167353  
min      0.200000  
25%     0.326087  
50%     0.406522  
75%     0.547826  
max      1.000000  
Name: horsepower, dtype: float64
```

```
0    0.565217  
1    0.717391  
2    0.652174  
3    0.652174  
4    0.608696  
Name: horsepower, dtype: float64
```

- | The DataFrames are normalized so that decreased performance, due to the difference in the relative size of numerical data in the column (each variable), does not occur.

```
In [15]: 1 print(df.horsepower.describe())
2 print('\n')
3
4 min_x = df.horsepower - df.horsepower.min()
5 min_max = df.horsepower.max() - df.horsepower.min()
6 df.horsepower = min_x / min_max
7
8 print(df.horsepower.head())
9 print('\n')
10 print(df.horsepower.describe())
```

Line 15-1

- The maximum value (max) and the minimum value (min) are checked by the statistical summary information of the horsepower column.

Line 15-4 ~ 15-6

- All data is divided and stored as the absolute value of the maximum value of the horsepower column.

- | The DataFrames are normalized so that decreased performance, due to the difference in the relative size of numerical data in the column (each variable), does not occur.

```
count    392.000000  
mean     0.454215  
std      0.167353  
min      0.200000  
25%      0.326087  
50%      0.406522  
75%      0.547826  
max      1.000000  
Name: horsepower, dtype: float64
```

```
count    392.000000  
mean     0.317768  
std      0.209191  
min      0.000000  
25%      0.157609  
50%      0.258152  
75%      0.434783  
max      1.000000  
Name: horsepower, dtype: float64
```

```
0    0.456522  
1    0.646739  
2    0.565217  
3    0.565217  
4    0.510870  
Name: horsepower, dtype: float64
```

Coding Exercise #0110



Follow practice steps on 'ex_0110.ipynb' file

Unit 4.

Data Visualization for Various Data Scales

| 4.1. Intro to Data Visualization

| 4.2. Graphs for Continuous Data Summary

| 4.3. Graphs for Categorical Data Summary

| 4.4. Visualization for Matplotlib & Pandas

| 4.5. Advanced Graphing with Seaborn

Exploring Univariate Data and Multivariate Data

Refers to data with one variable and is divided into quantitative and qualitative data.

Variable	Variable Type	Growth Rate
Univariate (One Variable)	Continuous Data	<ul style="list-style-type: none">▪ Histogram▪ Box Plot▪ Violin Plot▪ Kernel Density Curve
	Categorical Data (nominal, ordinal)	<ul style="list-style-type: none">▪ Bar Chart▪ Pie Chart

| Refers to data with one variable and is divided into quantitative and qualitative data.

Variable	Variable Type	Growth Rate
Univariate (Two or More Variables)	Continuous Data	<ul style="list-style-type: none">▪ Scatter Plot▪ Line Plot▪ Time Series Plot
	Categorical Data (nominal, ordinal)	<ul style="list-style-type: none">▪ Mosaic Chart

Unit 4.

Data Visualization for Various Data Scales

| 4.1. Intro to Data Visualization

| **4.2. Graphs for Continuous Data Summary**

| 4.3. Graphs for Categorical Data Summary

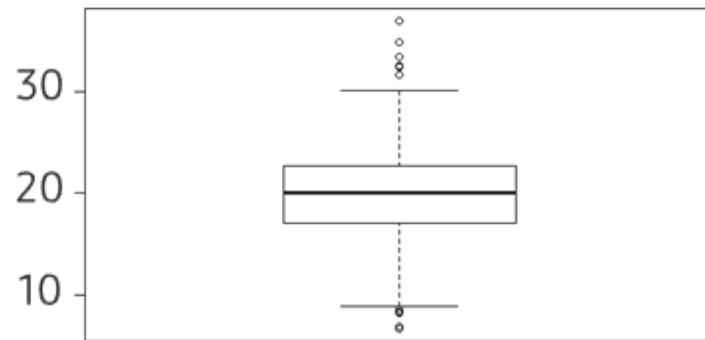
| 4.4. Visualization for Matplotlib & Pandas

| 4.5. Advanced Graphing with Seaborn

Basic Visualization Types

| Univariate visualization:

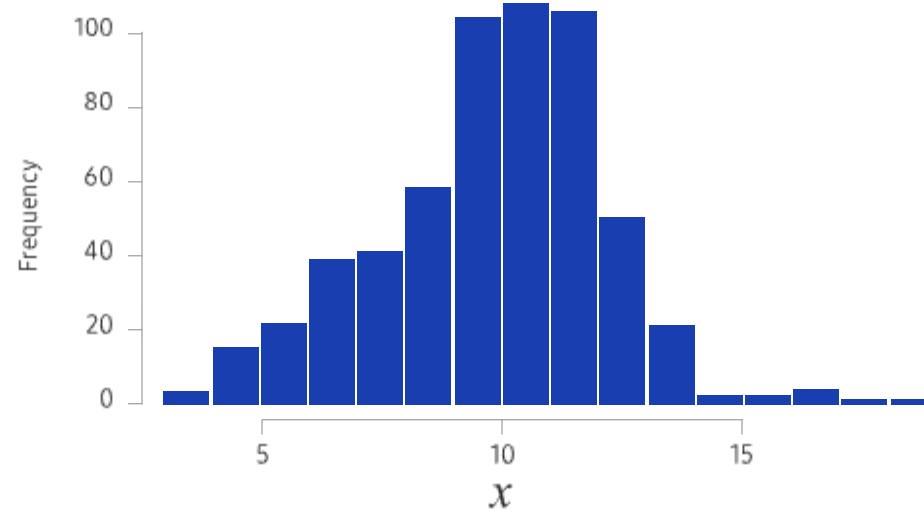
- ▶ One continuous numeric variable: **Boxplot**



- ▶ A boxplot is composed of a box, whiskers, outliers, etc.

| Univariate visualization:

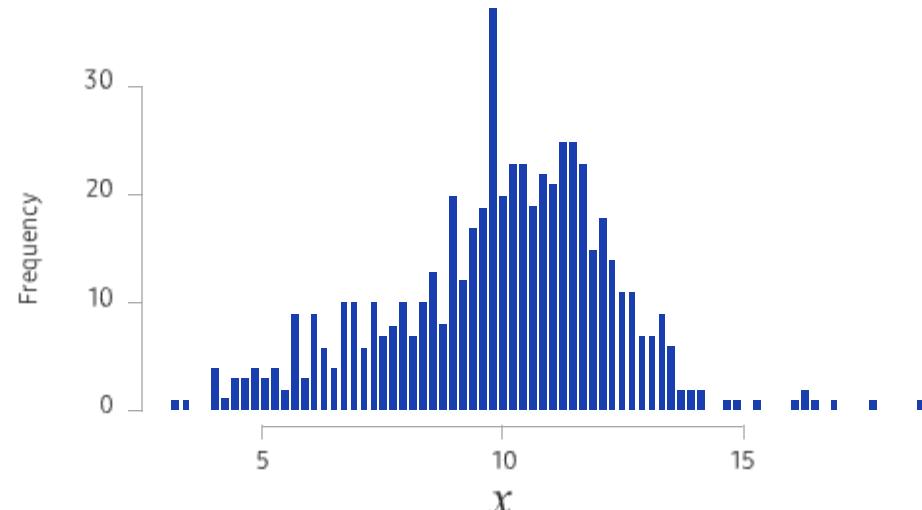
- ▶ One continuous numeric variable: **Histogram**



- ▶ Shows the absolute or relative frequencies of each interval.

| Univariate visualization:

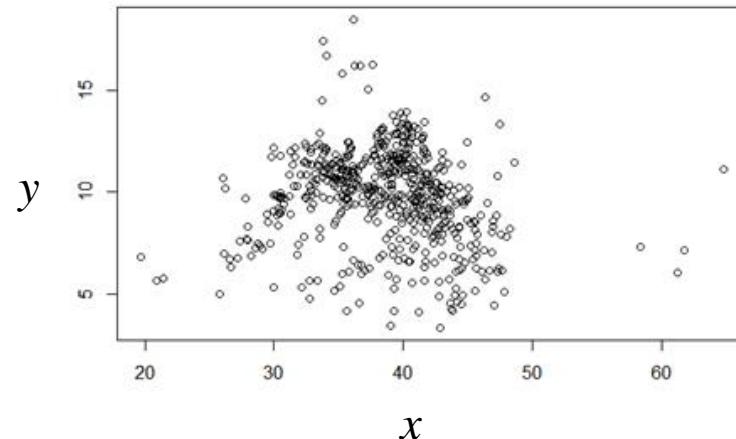
- ▶ One continuous numeric variable: **Histogram**



- ▶ The interval width can be adjusted.

| Multivariate visualization:

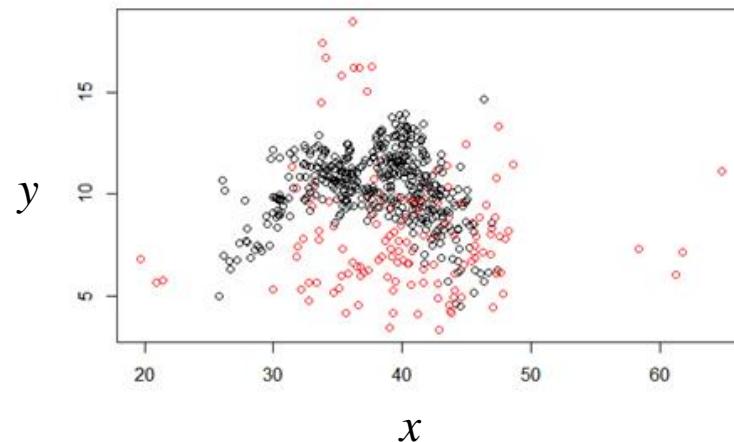
- ▶ Two continuous numeric variables: **Scatter plot**



- ▶ Identify whether a linear relation exists between the two variables.

| Multivariate visualization:

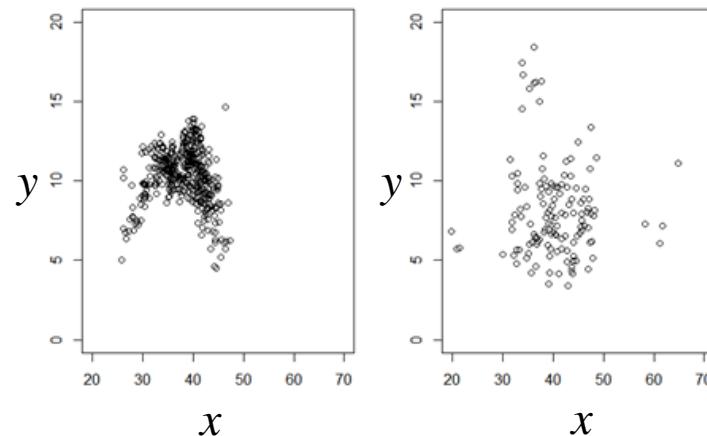
- ▶ Two continuous numeric variables: **Scatter plot**



- ▶ Different categories can be denoted by different colors or markers (symbols).

| Multivariate visualization:

- ▶ Two continuous numeric variables and one categorical variable: **Multiple scatter plots**



- ▶ Different categories can be plotted separately.
- ▶ You should make sure that the axis ranges match for proper comparison.

Unit 4.

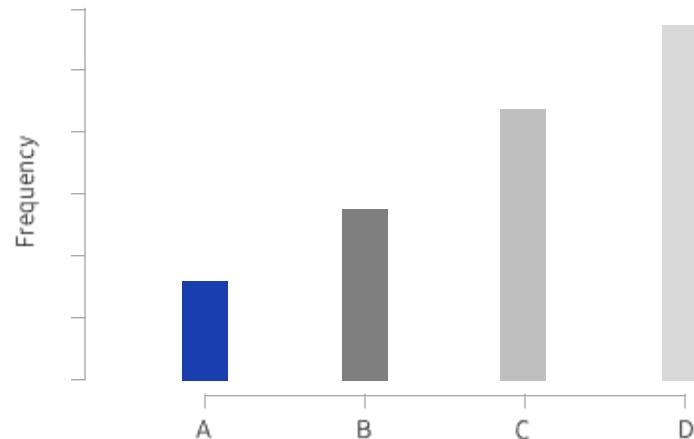
Data Visualization for Various Data Scales

- | 4.1. Intro to Data Visualization
- | 4.2. Graphs for Continuous Data Summary
- | **4.3. Graphs for Categorical Data Summary**
- | 4.4. Visualization for Matplotlib & Pandas
- | 4.5. Advanced Graphing with Seaborn

Basic Visualization Types

| Univariate visualization:

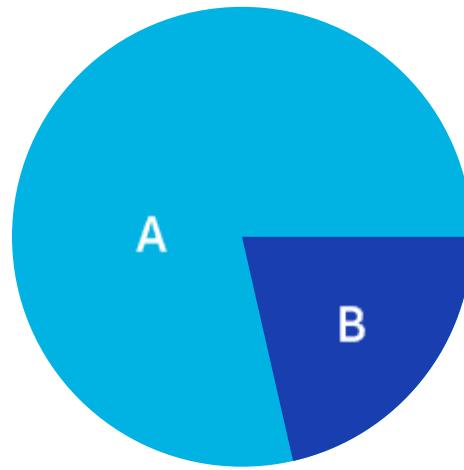
- ▶ One categorical variable: **Bar plot**



- ▶ Shows the absolute or relative frequencies of each category (type).

| Multivariate visualization:

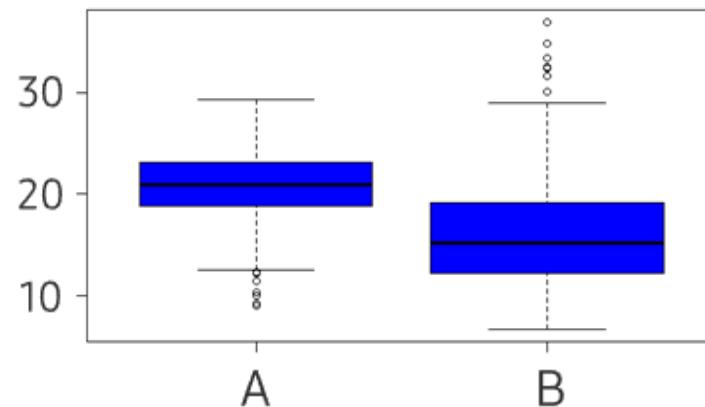
- ▶ One categorical variable: **Pie chart**



- ▶ Shows the proportions of each category (type).

| Bivariate visualization:

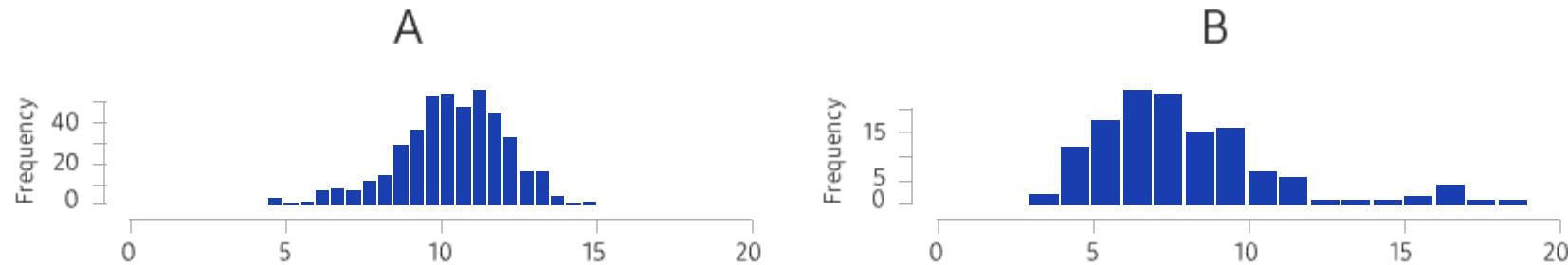
- ▶ One continuous numeric variable & one categorical variable: **Multiple boxplots**



- ▶ The number of categories (types) = the number of boxplots

| Bivariate visualization:

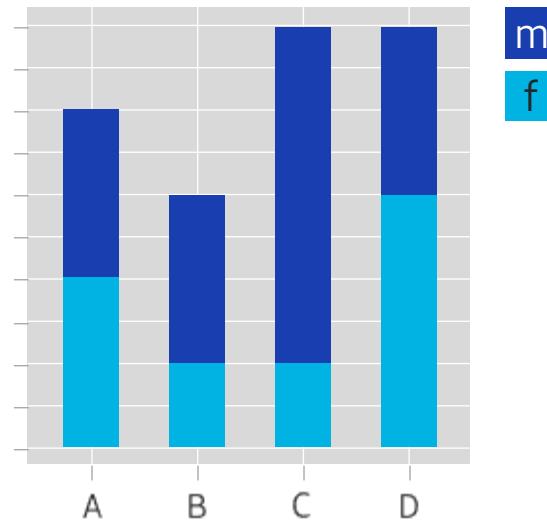
- ▶ One continuous numeric variable & one categorical variable: **Multiple histograms**



- ▶ The number of categories (types) = the number of histograms
- ▶ You should make sure that the axis ranges match for proper comparison.

| Bivariate visualization:

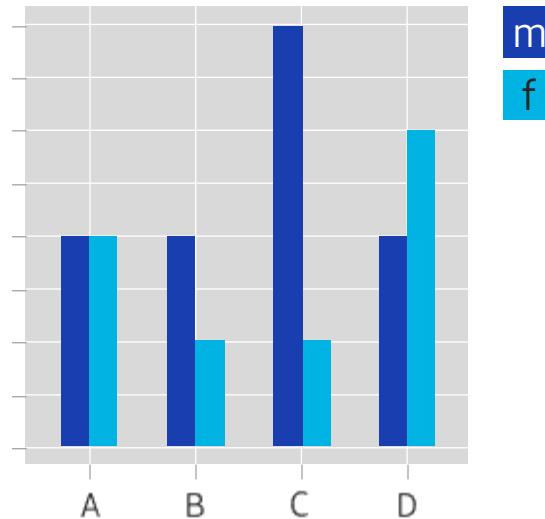
- ▶ Two categorical variables: **Bar plot**



- ▶ Use color to distinguish the categories of the secondary variable.

| Bivariate visualization:

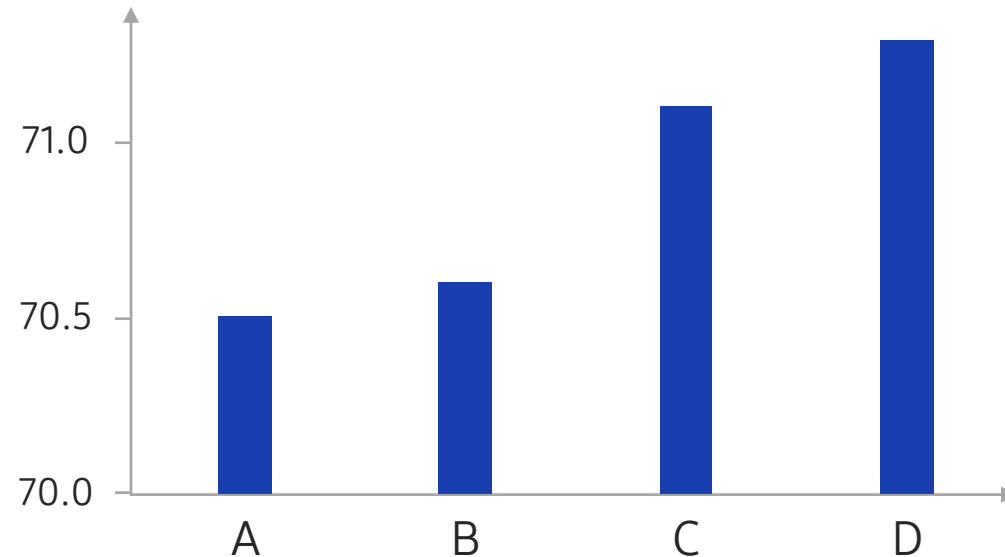
- ▶ Two categorical variables: **Bar plot**



- ▶ Use color and dodged bars to distinguish the categories of the secondary variable.

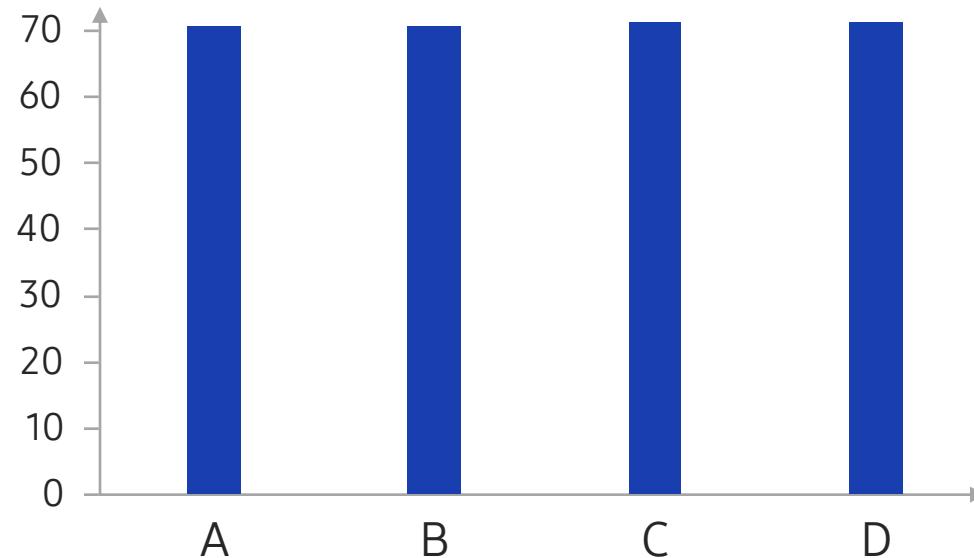
Recommendations

- | In the following bar plot, can you see big difference among the categories?
- ▶ Apparently, yes?



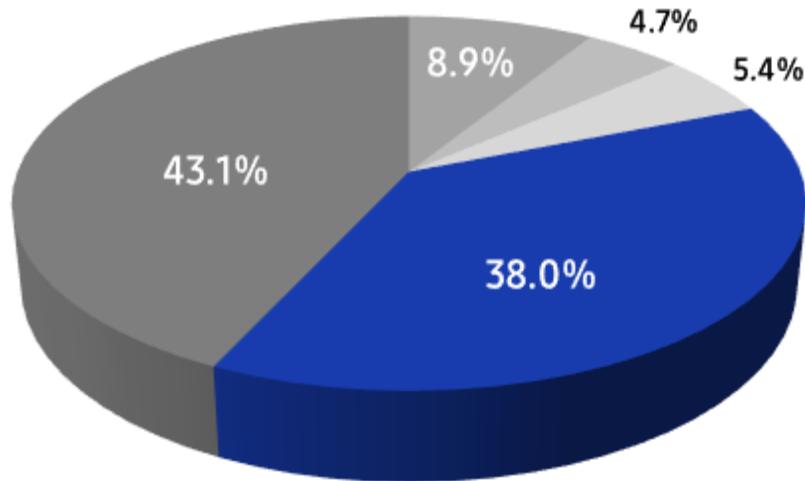
In the following bar plot, can you see big difference between the categories?

- ▶ In this case where the vertical zero is shown, you see little difference.



| Sometimes, 3D effects should be avoided.

- ▶ In a 3D pie chart, it is hard to distinguish the relative proportions due to the perspective.



Unit 4.

Data Visualization for Various Data Scales

- | 4.1. Intro to Data Visualization
- | 4.2. Graphs for Continuous Data Summary
- | 4.3. Graphs for Categorical Data Summary

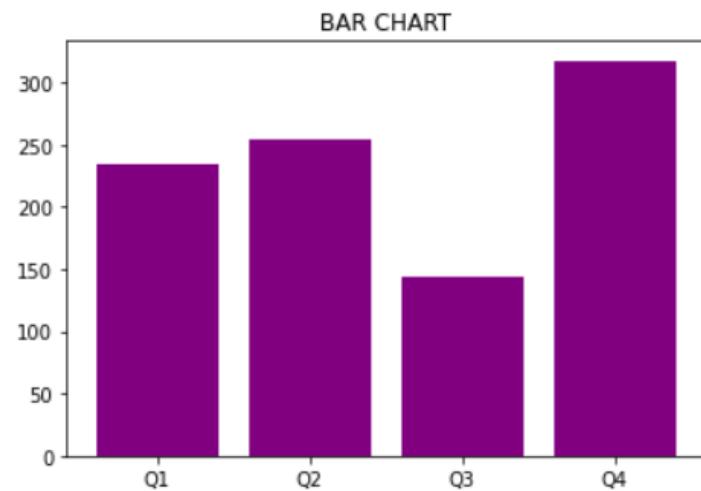
- | 4.4. Visualization for Matplotlib & Pandas
- | 4.5. Advanced Graphing with Seaborn

Basic Matplotlib Visualization

I Bar plot

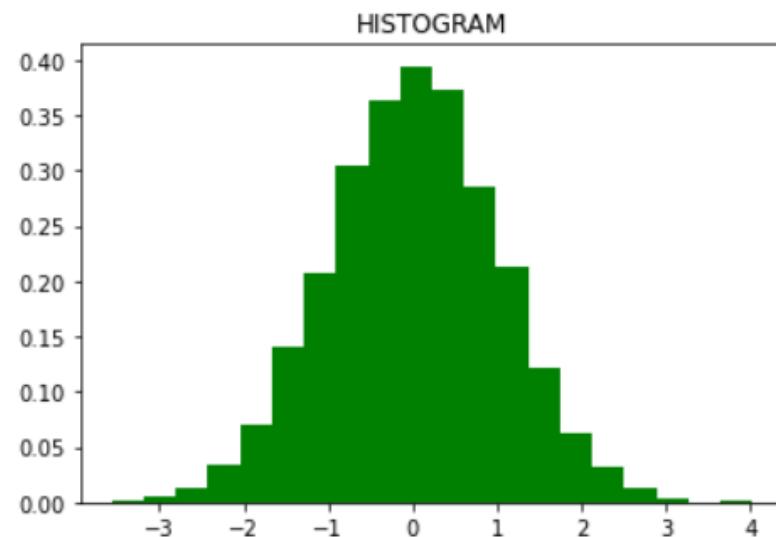
```
In [1]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

x = np.array(['Q1', 'Q2', 'Q3', 'Q4'])
y = np.array([ 234.0, 254.7, 144.6, 317.6])
plt.bar(x,y,color = 'purple')
plt.title('BAR CHART')
plt.show()
```



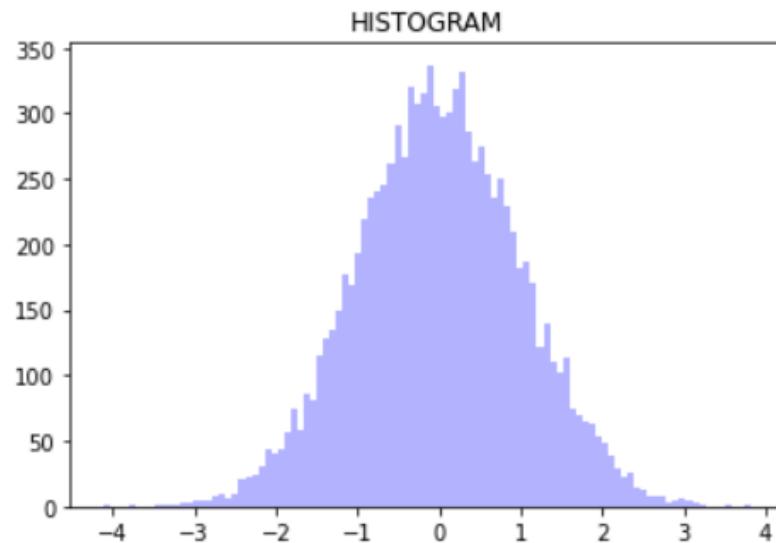
Histogram

```
In [2]: x = np.random.randn(10000)
plt.hist(x,bins=20,color='green',density=True)
plt.title('HISTOGRAM')
plt.show()
```



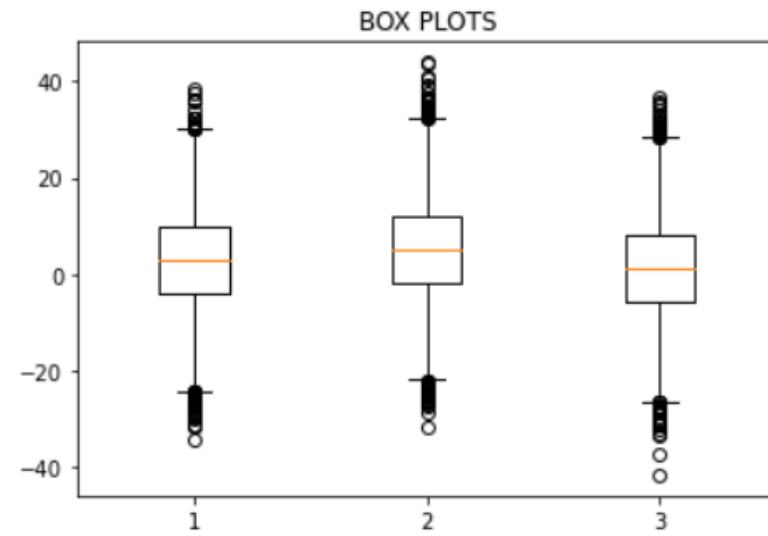
Histogram

```
In [3]: x = np.random.randn(10000)
plt.hist(x,bins=100,color='blue',density=False,alpha=0.3)
plt.title('HISTOGRAM')
plt.show()
```



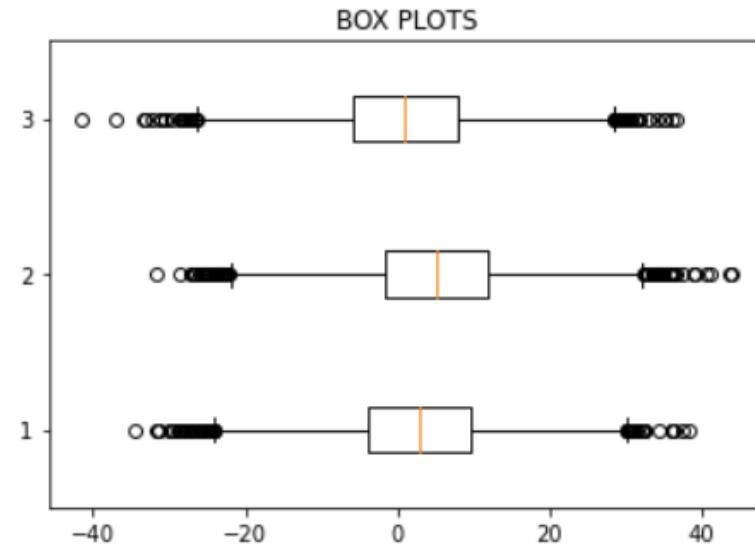
I Multiple boxplots

```
In [4]: x = np.random.randn(10000)*10+3  
y = np.random.randn(10000)*10+5  
z = np.random.randn(10000)*10+1  
plt.boxplot([x,y,z],0)  
plt.title('BOX PLOTS')  
plt.show()
```



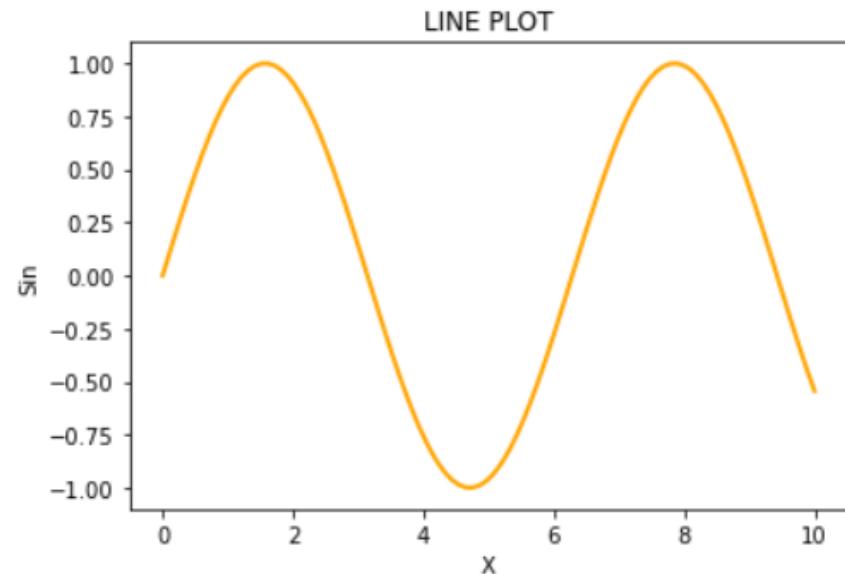
| Multiple boxplots

```
In [5]: plt.boxplot([x,y,z],0,vert=False)
plt.title('BOX PLOTS')
plt.show()
```



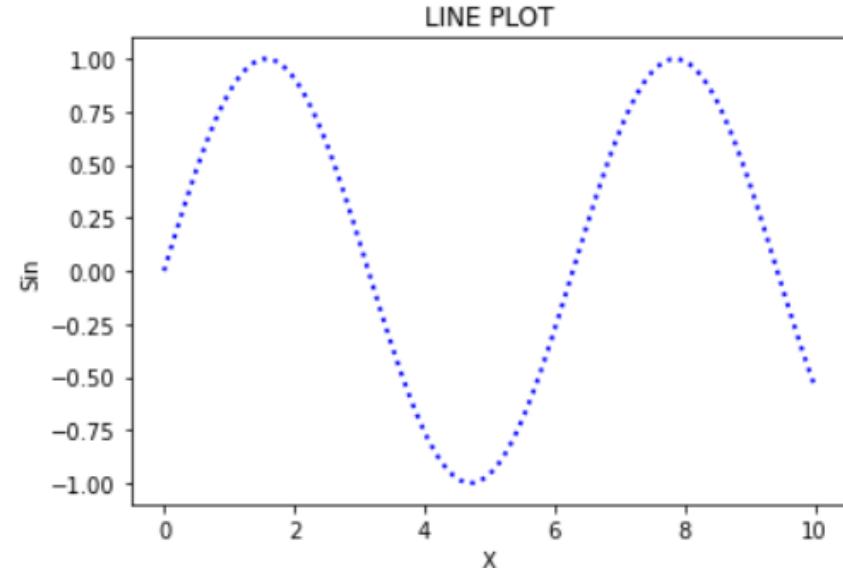
I Line plot

```
In [6]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,color='orange',linewidth=2)                      # linestyle = '-'
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('LINE PLOT')
plt.show()
```



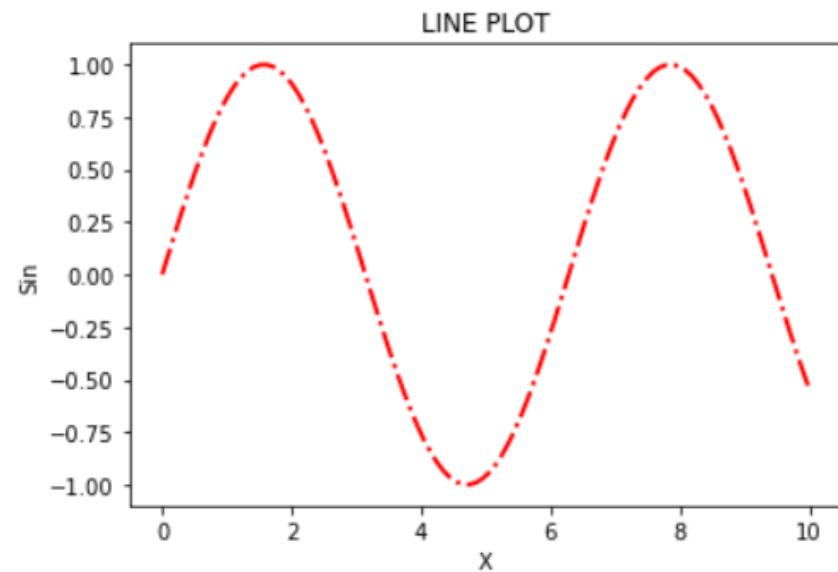
I Line plot

```
In [7]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,color='blue',linestyle=':', linewidth=2)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('LINE PLOT')
plt.show()
```



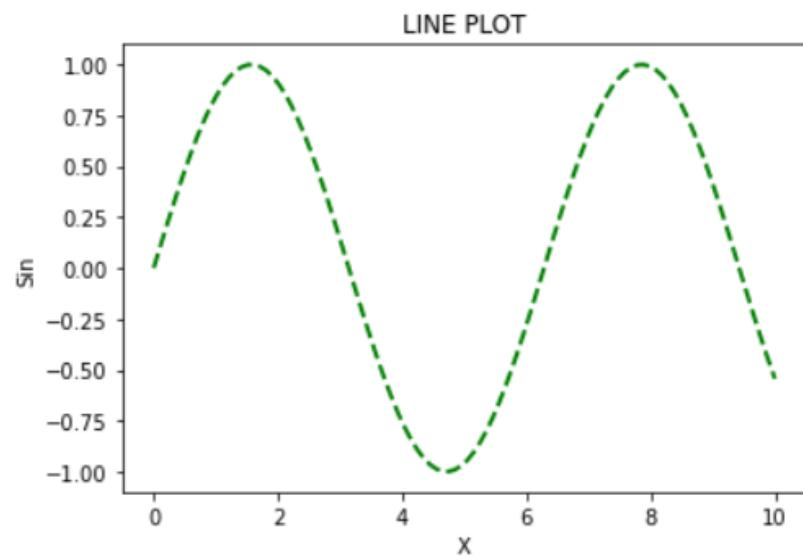
| Line plot

```
In [8]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,color='red',linestyle='-.',linewidth=2)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('LINE PLOT')
plt.show()
```



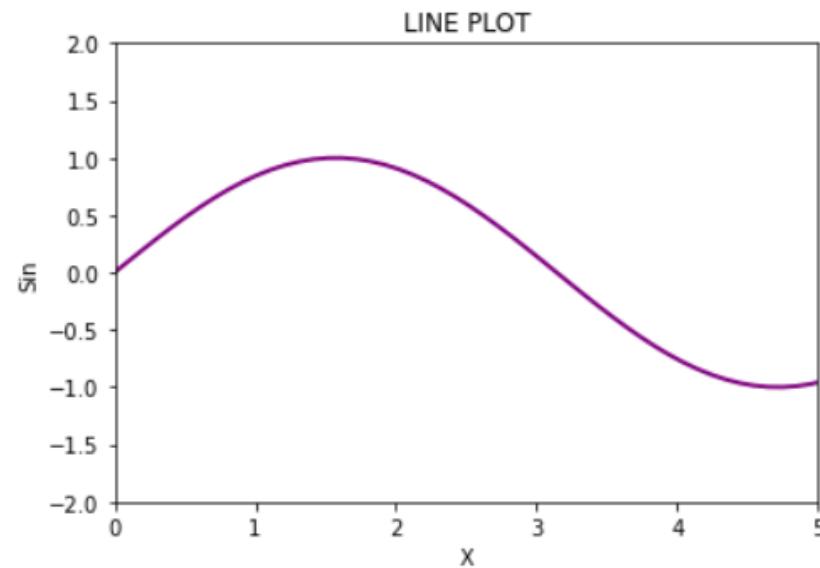
| Line plot

```
In [9]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,color='green',linestyle='--',linewidth=2)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('LINE PLOT')
plt.show()
```



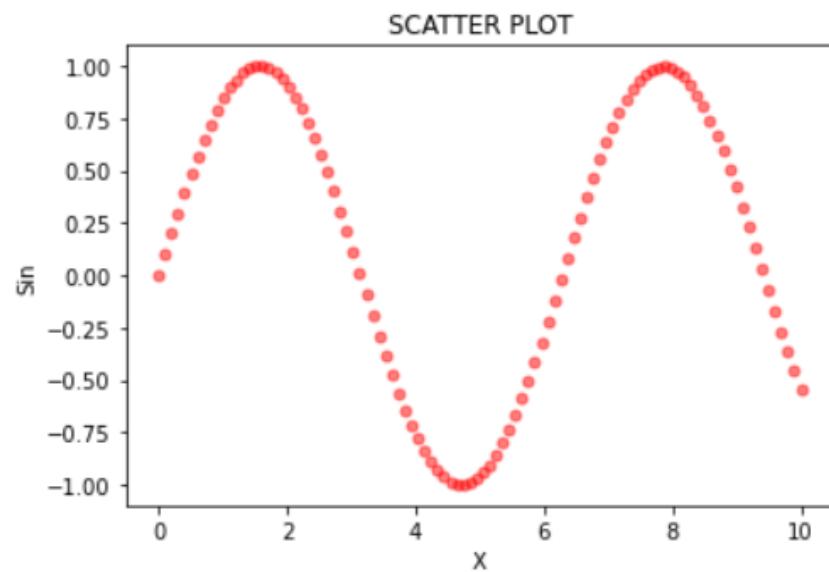
| Line plot

```
In [10]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,color='purple',linestyle='-',linewidth=2)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('LINE PLOT')
plt.xlim([0,5])                                # Horizontal axis limits.
plt.ylim([-2,+2])                            # Vertical axis limits.
plt.show()
```



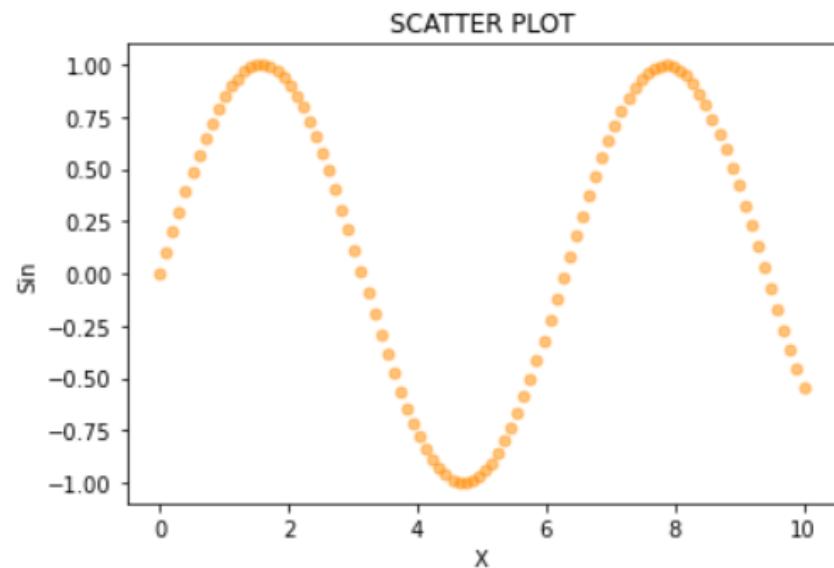
| Scatter plots with plot() function and linestyle ='none':

```
In [11]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,color='red',marker='o',linestyle='none',markersize=5, alpha=0.5)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('SCATTER PLOT')
plt.show()
```



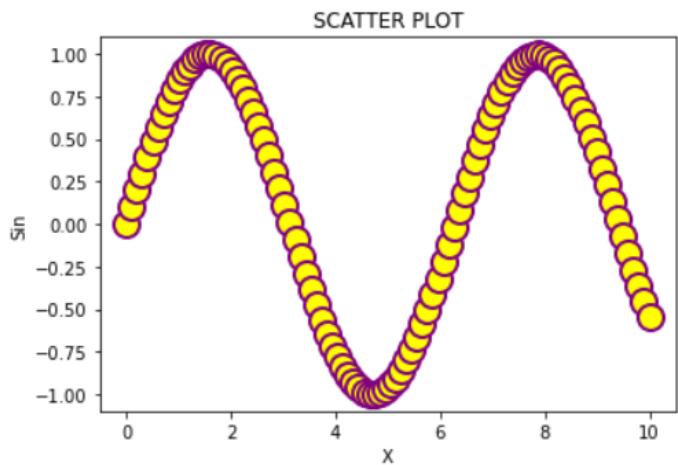
| Scatter plots with plot() function and linestyle ='none':

```
In [12]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,color='#FF8C00',marker='o',linestyle='none',markersize=5, alpha=0.5)      # RGB color.
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('SCATTER PLOT')
plt.show()
```



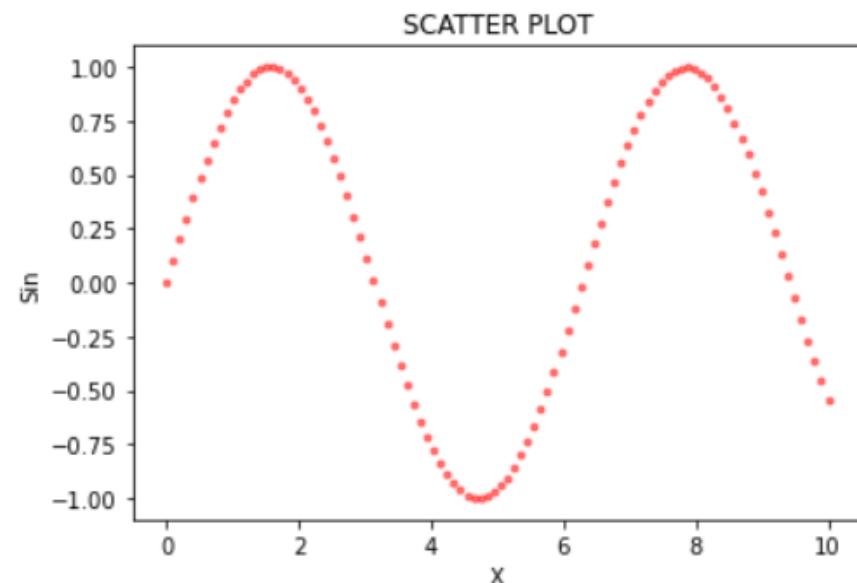
| Scatter plots with plot() function and linestyle ='none':

```
In [13]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.plot(x,y,marker='o',linestyle='none',markersize=15,markerfacecolor='yellow',markeredgecolor='purple',markeredgewidth=2)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('SCATTER PLOT')
plt.show()
```



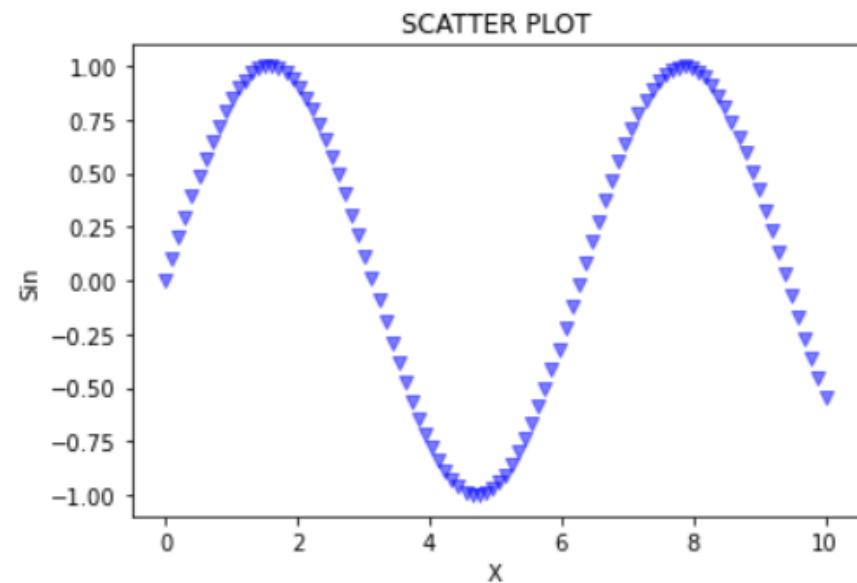
| Scatter plots with scatter() function:

```
In [14]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.scatter(x,y,c='red',marker='.',alpha=0.5)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('SCATTER PLOT')
plt.show()
```



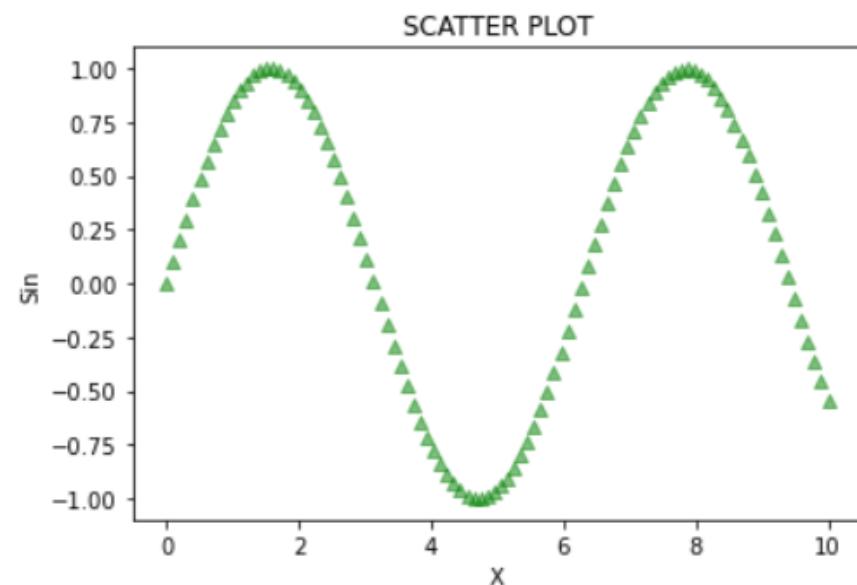
| Scatter plots with scatter() function:

```
In [15]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.scatter(x,y,c='blue',marker='v',alpha=0.5)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('SCATTER PLOT')
plt.show()
```



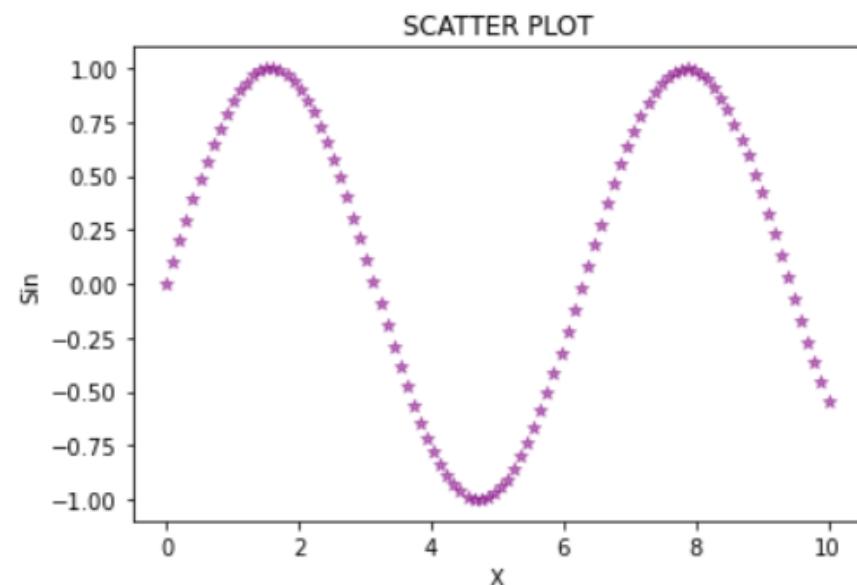
| Scatter plots with scatter() function:

```
In [16]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.scatter(x,y,c='green',marker='^',alpha=0.5)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('SCATTER PLOT')
plt.show()
```



| Scatter plots with scatter() function:

```
In [18]: x = np.linspace(0,10,100)
y = np.sin(x)
plt.scatter(x,y,c='purple',marker='*',alpha=0.5)
plt.xlabel('X')
plt.ylabel('Sin')
plt.title('SCATTER PLOT')
plt.show()
```



I Arguments of the plot() function:

Argument	Explanation
color	Color
alpha	Transparency
linewidth	Line width
linestyle	Line style
marker	Marker type
markersize	Marker size
markerfacecolor	Marker color inside
markeredgecolor	Color of the marker edge
markeredgewidth	Width of the marker edge

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html

| Values of the `linestyle` argument:

linestyle	Explanation
'none'	No line
'.'	Dotted line
'--'	Dashed line
'-'	Dash dot
'_'	Continuous line
'steps'	In steps

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html

| Values of the `marker` argument:

marker	Explanation
'.'	Point
',	Pixel
'o'	Circle
'^'	Triangle up
'v'	Triangle down
's'	Square
'*'	Star
'+'	Plus sign
'x'	X character
'D'	Diamond
'p'	Pentagon

https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html

Matplotlib Visualization with Objects

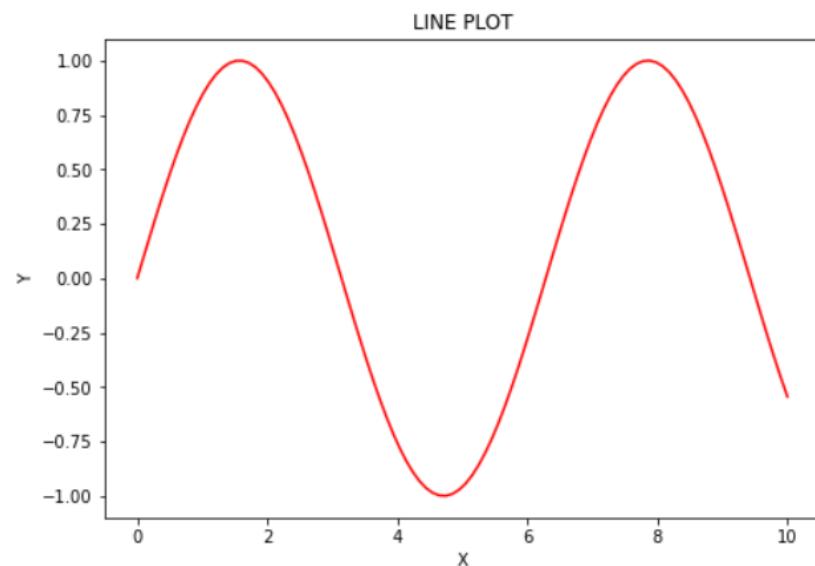
| Import required modules.

```
In [1]: import matplotlib.pyplot as plt  
import numpy as np  
%matplotlib inline
```

| Visualization with a figure object:

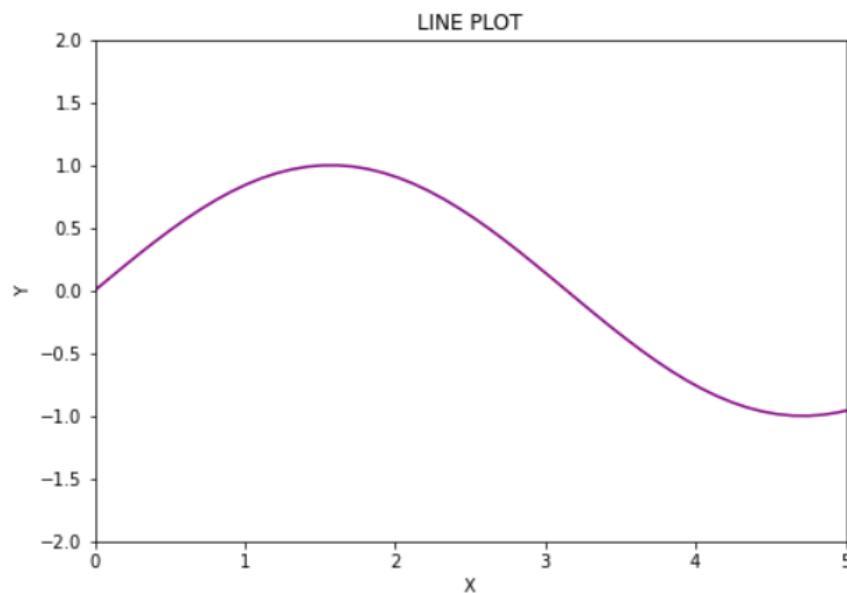
```
In [2]: x = np.linspace(0,10,100)
y = np.sin(x)
z = np.cos(x)
```

```
In [3]: fig0=plt.figure()
axes0 = fig0.add_axes([0,0,1,1])           # Left, bottom, width, height.
axes0.plot(x,y,color='red',linestyle='-')
axes0.set_xlabel('X')
axes0.set_ylabel('Y')
axes0.set_title('LINE PLOT')
plt.show()
```



| Visualization with a figure object:

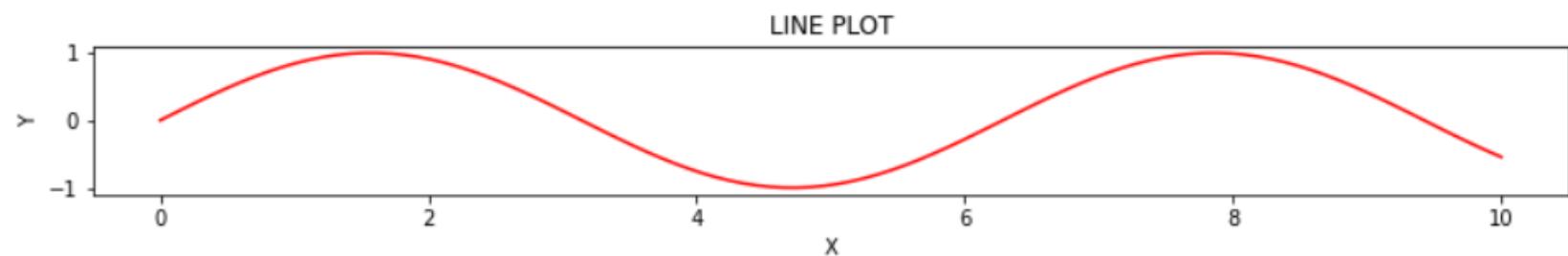
```
In [4]: fig0=plt.figure()
axes0 = fig0.add_axes([0,0,1,1])           # Left, bottom, width, height.
axes0.plot(x,y,color='purple',linestyle='-')
axes0.set_xlabel('X')
axes0.set_ylabel('Y')
axes0.set_title('LINE PLOT')
axes0.set_xlim([0,5])                      # Horizontal axis limits.
axes0.set_ylim([-2,2])                      # Vertical axis limits.
plt.show()
```



| Visualization with a figure object:

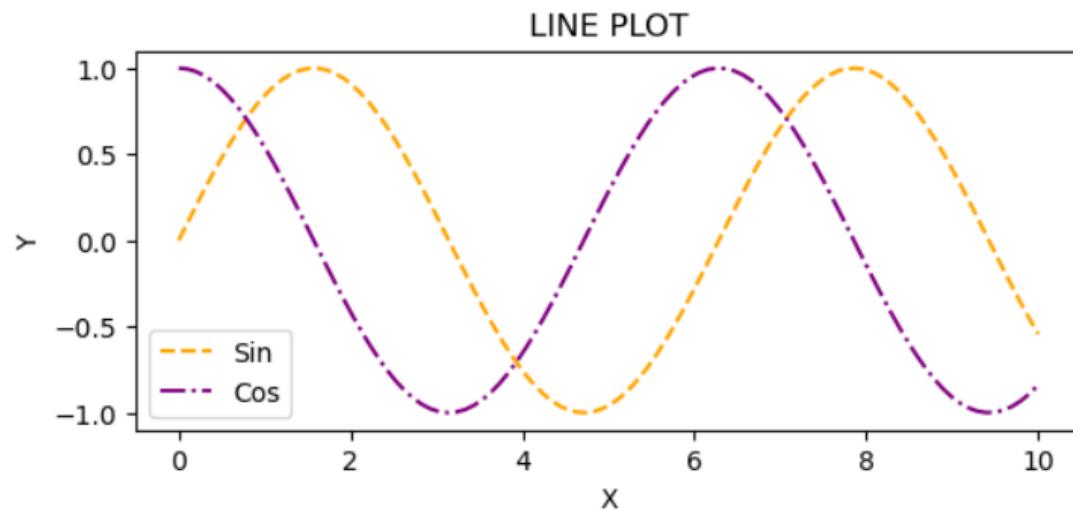
```
In [5]: fig0=plt.figure(figsize=(10,1))  
axes0 = fig0.add_axes([0,0,1,1])  
axes0.plot(x,y,color='red',linestyle='-')  
axes0.set_xlabel('X')  
axes0.set_ylabel('Y')  
axes0.set_title('LINE PLOT')  
plt.show()
```

Width and height specified with figsize.
Left, bottom, width, height.



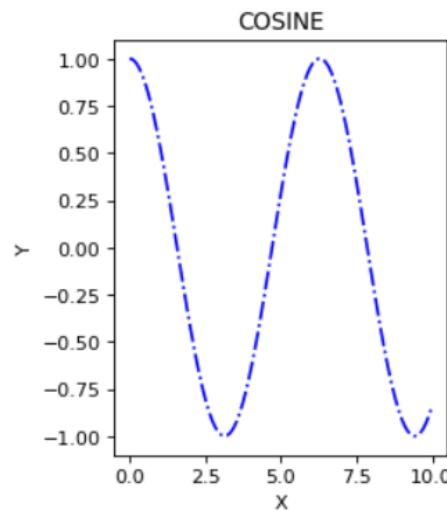
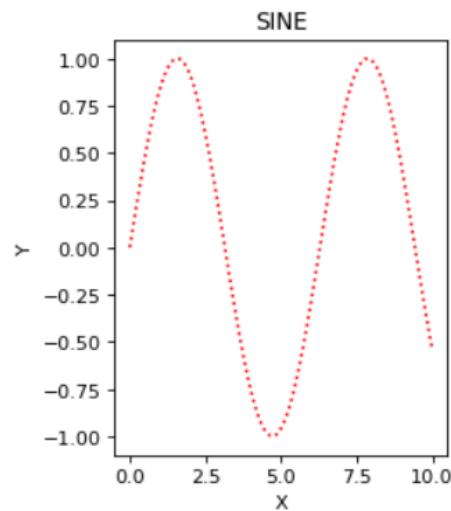
| Multiple plots within the same axes:

```
In [6]: fig0=plt.figure(figsize=(5,2), dpi=100)      # Width, height and DPI setting.  
axes0 = fig0.add_axes([0,0,1,1])                 # Left, bottom, width, height.  
axes0.plot(x,y,color='orange',linestyle='--', label='Sin')    # 'label' for the legend.  
axes0.plot(x,z,color='purple',linestyle='-.', label='Cos')    # 'label' for the legend.  
axes0.legend(loc=0)                                # Legend to be located at the top-right corner.  
axes0.set_xlabel('X')  
axes0.set_ylabel('Y')  
axes0.set_title('LINE PLOT')  
plt.show()
```



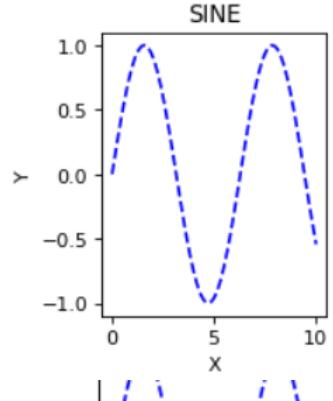
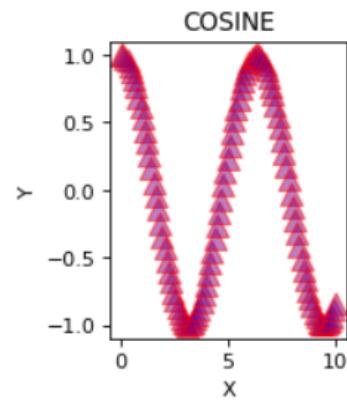
| Multiple plots in separate axes:

```
In [7]: fig0=plt.figure(figsize=(8,3), dpi=80)           # Width, height and DPI setting.  
axes1 = fig0.add_axes([0,0,0.3,1])                  # Left, bottom, width, height.  
axes2 = fig0.add_axes([0.5,0,0.3,1])                # Left, bottom, width, height.  
axes1.plot(x,y,color='red',linestyle=': ')  
axes2.plot(x,z,color='blue',linestyle='-.')  
axes1.set_xlabel('X')  
axes1.set_ylabel('Y')  
axes1.set_title('SINE')  
axes2.set_xlabel('X')  
axes2.set_ylabel('Y')  
axes2.set_title('COSINE')  
plt.show()
```



| Multiple plots in separate axes:

```
In [8]: fig0=plt.figure(figsize=(8,4), dpi=80)           # Width, height and DPI setting.  
axes1 = fig0.add_axes([0,0.5,0.2,0.5])            # Left, bottom, width, height.  
axes2 = fig0.add_axes([0.5,0,0.2,0.5])            # Left, bottom, width, height.  
axes1.plot(x,z,linestyle='none',marker='^',markerSize=10,markerfacecolor='purple',markeredgecolor='red',alpha=0.5)  
axes2.plot(x,y,color='blue',linestyle='--')  
axes1.set_xlabel('X')  
axes1.set_ylabel('Y')  
axes1.set_title('COSINE')  
axes2.set_xlabel('X')  
axes2.set_ylabel('Y')  
axes2.set_title('SINE')  
plt.show()
```

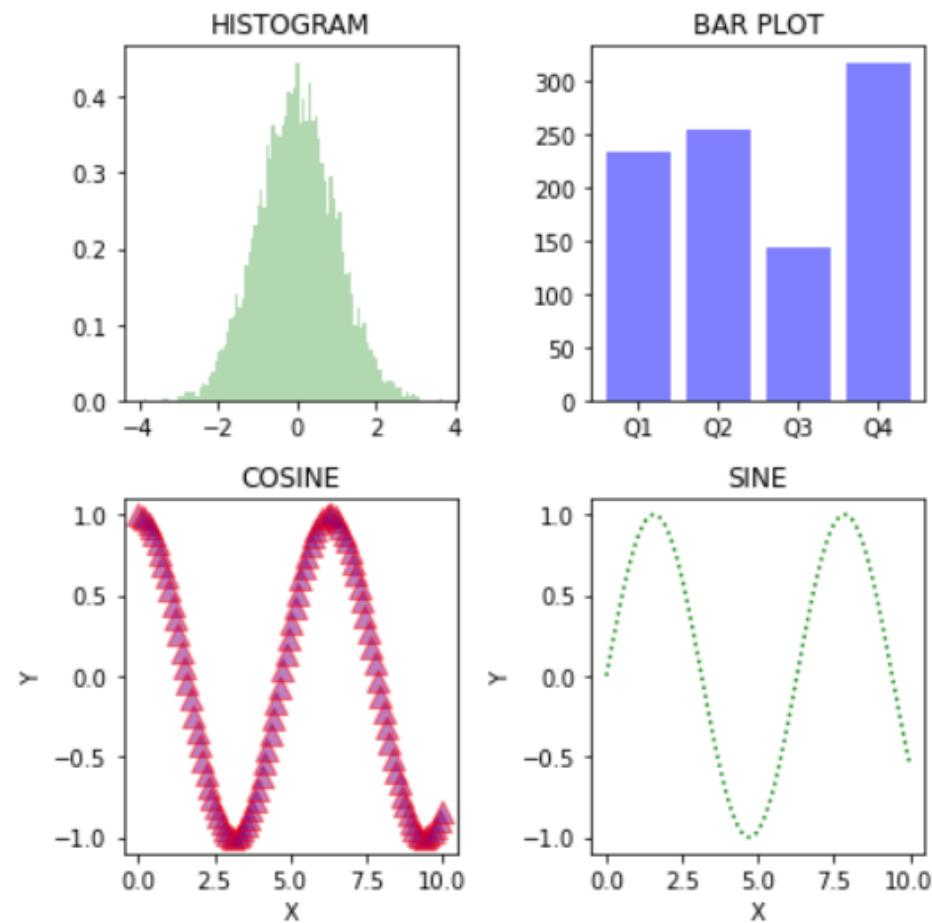


| Multiple plots in an array of axes:

```
In [9]: fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(6,6))
# (0,0)
c = np.random.randn(10000)
axes[0,0].hist(c,bins=100,color='green',density=True,alpha=0.3)
axes[0,0].set_title('HISTOGRAM')
# (1,0)
x = np.linspace(0, 10, 100)
y = np.sin(x)
axes[1,0].plot(x,y,color='red',linestyle='solid',marker='^',markersize=10,markerfacecolor='purple',markeredgecolor='red',alpha=0.5)
axes[1,0].set_xlabel('X')
axes[1,0].set_ylabel('Y')
axes[1,0].set_title('SINE')
# (0,1)
a = np.array(['Q1', 'Q2', 'Q3','Q4'])
b = np.array([ 234.0, 254.7, 144.6, 317.6])
axes[0,1].bar(a,b,color='blue',alpha=0.5)
axes[0,1].set_title('BAR PLOT')
# (1,1)
x = np.linspace(0, 10, 100)
y = np.sin(x)
axes[1,1].plot(x,y,color='green',linestyle='dotted')
axes[1,1].set_xlabel('X')
axes[1,1].set_ylabel('Y')
axes[1,1].set_title('SINE')

plt.tight_layout()                                # Avoid overlapping.
plt.show()
```

| Multiple plots in an array of axes:



Pandas Visualization

| Import modules and data.

```
In [1]: import pandas as pd  
import numpy as np  
import os  
import matplotlib.pyplot as plt  
%matplotlib inline
```

```
In [2]: df = pd.read_csv('data_iris.csv', header='infer')
```

```
In [3]: df.shape
```

```
Out[3]: (150, 5)
```

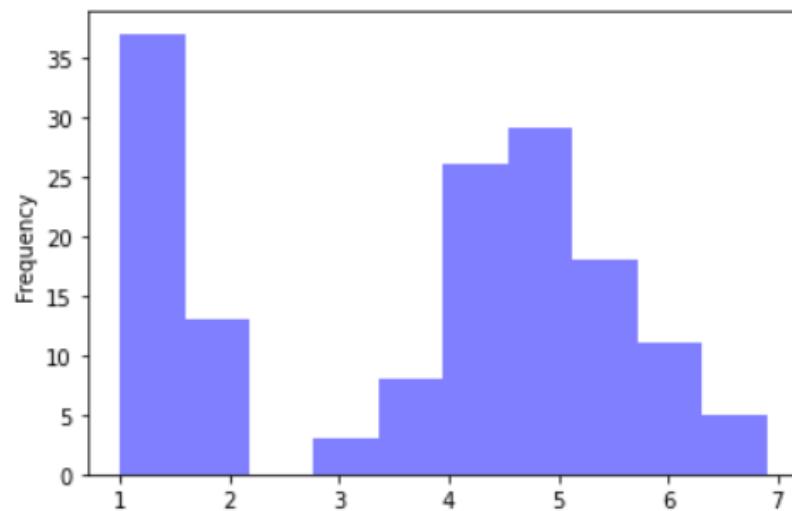
```
In [4]: df.head(5)
```

```
Out[4]:
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

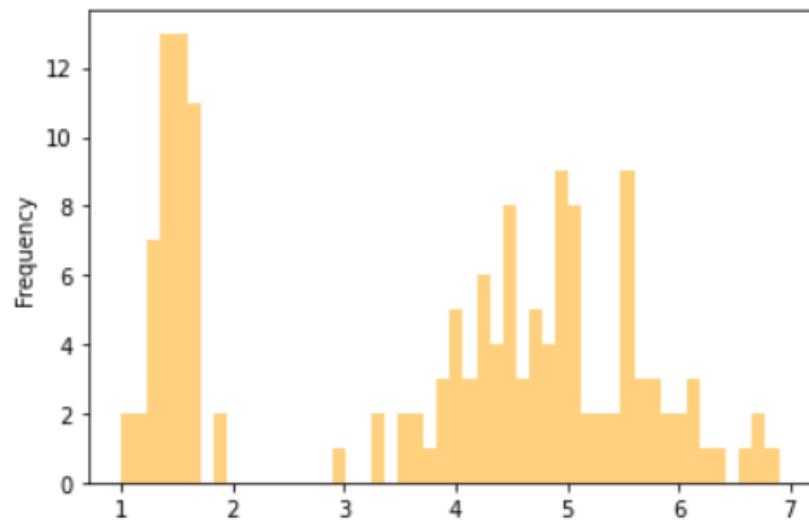
| Histogram:

```
In [5]: df.loc[:, 'Petal.Length'].plot.hist(color='blue', alpha=0.5)  
plt.show()
```



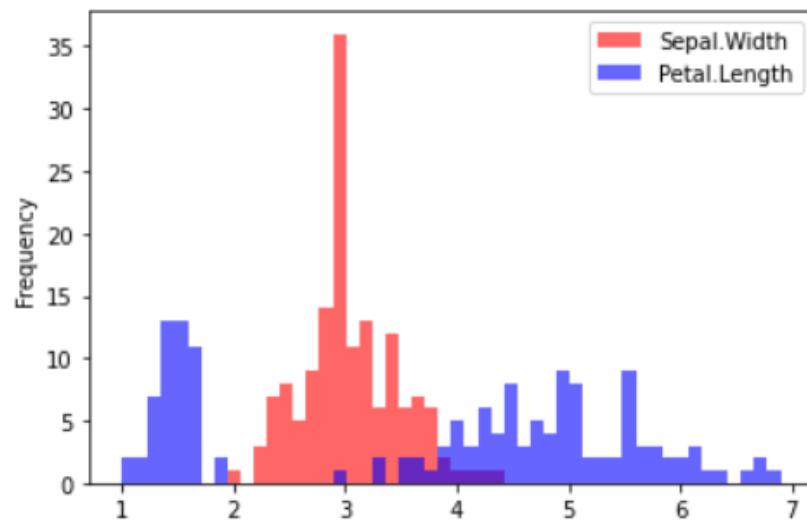
| Histogram:

```
In [6]: df.loc[:, 'Petal.Length'].plot.hist(bins=50,color='orange',alpha=0.5)  
plt.show()
```



Histogram:

```
In [7]: df.loc[:,['Sepal.Width','Petal.Length']].plot.hist(bins=50,color=['red','blue'],alpha=0.6)  
plt.show()
```



| Bar plot:

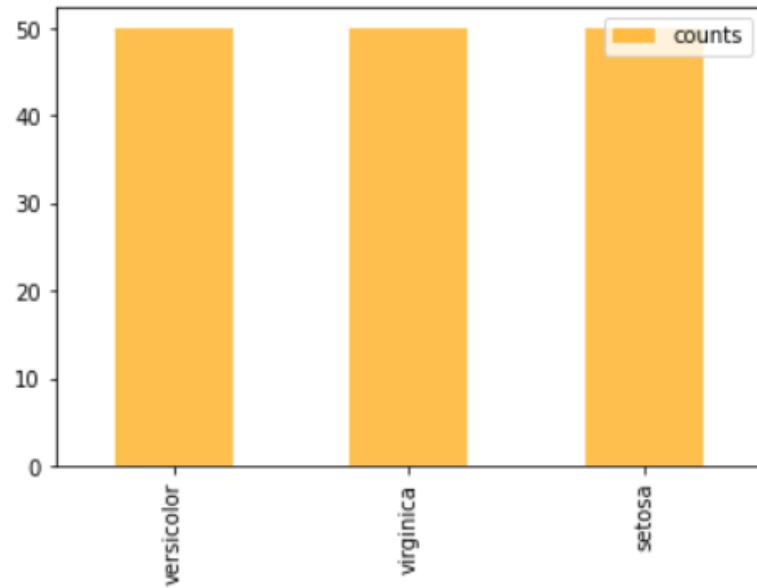
```
In [8]: frequencies = df.Species.value_counts()  
my_counts = list(frequencies.values)  
my_labels = list(frequencies.index)  
df2 = pd.DataFrame( {'counts':my_counts}, index = my_labels)  
df2
```

Out[8]:

	counts
versicolor	50
virginica	50
setosa	50

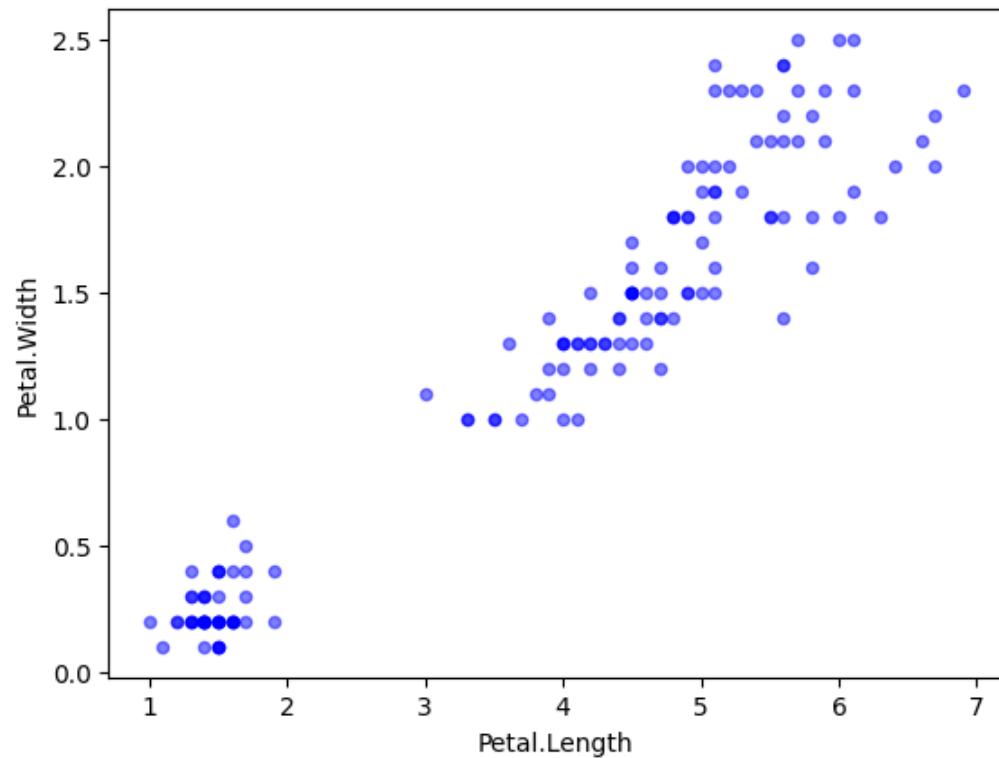
| Bar plot:

```
In [9]: df2.plot.bar(color='orange', alpha=0.7)  
plt.show()
```



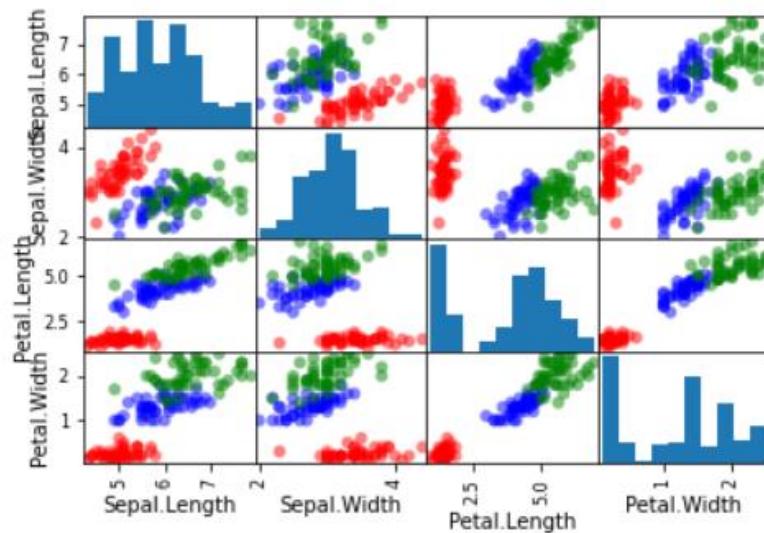
| Single scatter plot:

```
In [10]: df.plot.scatter(x='Petal.Length', y='Petal.Width', color='blue', alpha=0.5, marker='o')  
plt.show()
```



| An array of scatter plots:

```
In [11]: my_cols_dict = {'setosa':'red', 'virginica':'green', 'versicolor':'blue'}
my_cols = df['Species'].apply(lambda x: my_cols_dict[x])
pd.plotting.scatter_matrix(df, c=my_cols, marker='o', alpha=0.5)
plt.show()
```



Coding Exercise #0111



Follow practice steps on 'ex_0111.ipynb' file

Coding Exercise #0112



Follow practice steps on 'ex_0112.ipynb' file

Unit 4.

Data Visualization for Various Data Scales

- | 4.1. Intro to Data Visualization
- | 4.2. Graphs for Continuous Data Summary
- | 4.3. Graphs for Categorical Data Summary

- | 4.4. Visualization for Matplotlib & Pandas
- | **4.5. Advanced Graphing with Seaborn**

Seaborn Visualization Library

I What the Seaborn library provides:

- ▶ Internal dataset: `load_dataset()`
- ▶ Basic graphic types: `distplot()`, `jointplot()`, `kdeplot()`, `rugplot()`, `barplot()`, `countplot()`, etc.
- ▶ Arrays: `pairplot()`, `PairGrid()`, `FacetGrid()`, etc.
- ▶ With regression (trend) line: `lmplot()`, `jointplot()`, etc.
- ▶ Special graphic types: `heatmap()`, `clustermap()`, etc.
- ▶ Applied graph types: `violinplot()`, `swarmplot()`, `stripplot()`, etc.

I Install the Seaborn library.

- ▶ Remember to execute “!pip install seaborn” command first for Seaborn library installation.
- ▶ Import modules.

```
In [1]: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import os
import warnings
%matplotlib inline
warnings.filterwarnings('ignore') # Turn off the warnings.
```

| Import data set.

- ▶ Read in the data set 'mpg' from the library.

```
In [2]: dat = sns.load_dataset('mpg')
```

```
In [3]: dat.shape
```

```
Out[3]: (398, 9)
```

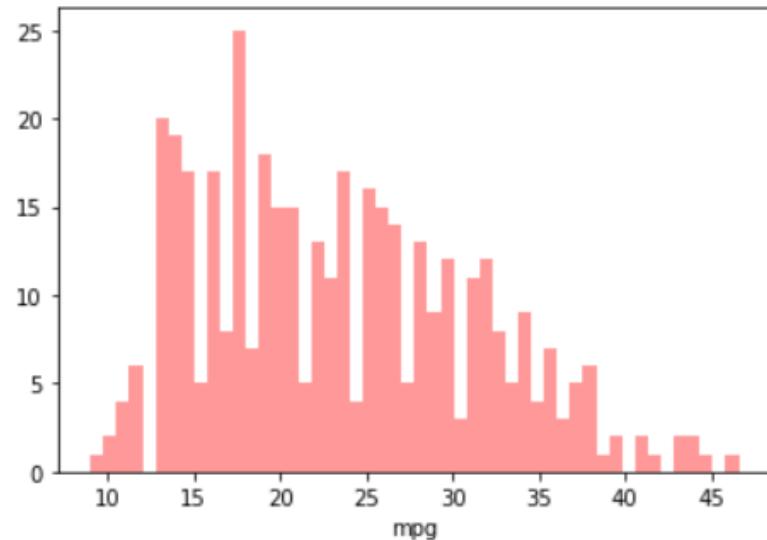
```
In [4]: dat.head(5)
```

```
Out[4]:
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino

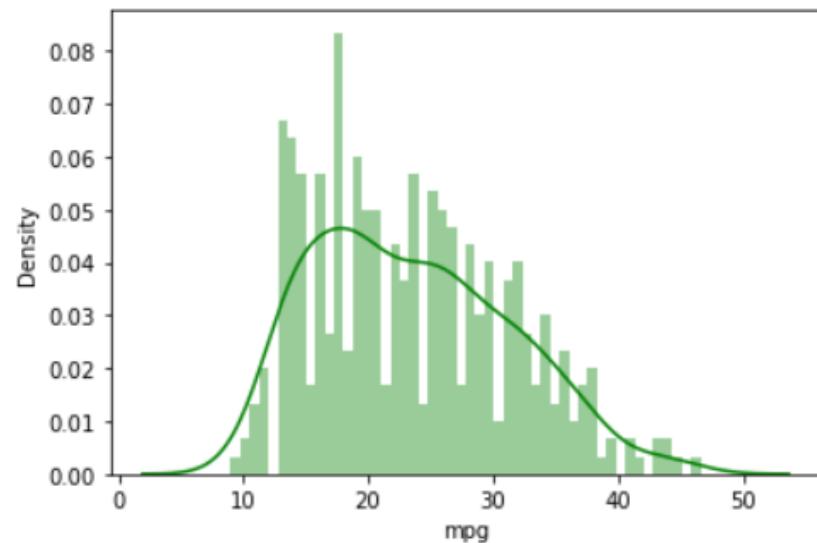
| Histogram:

```
In [5]: sns.distplot(dat.mpg, kde=False, rug=False, bins=50, color='red')  
plt.show()
```



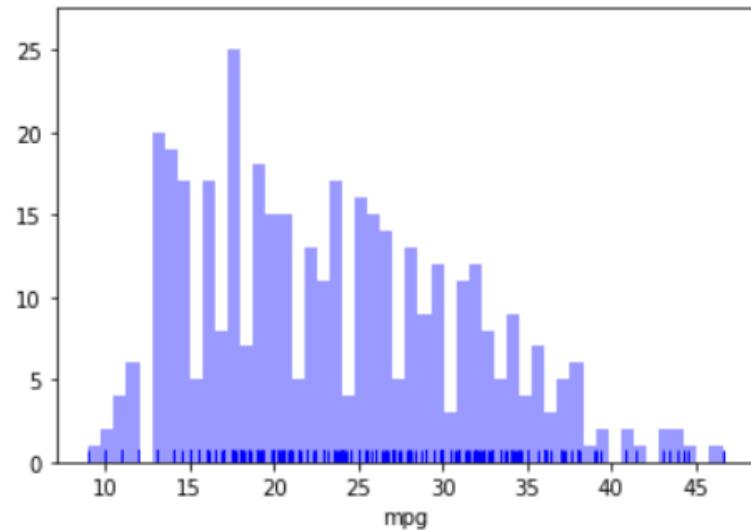
| Histogram + KDE:

```
In [6]: sns.distplot(dat.mpg, kde=True, rug=False, bins=50, color='green')
plt.show()
```



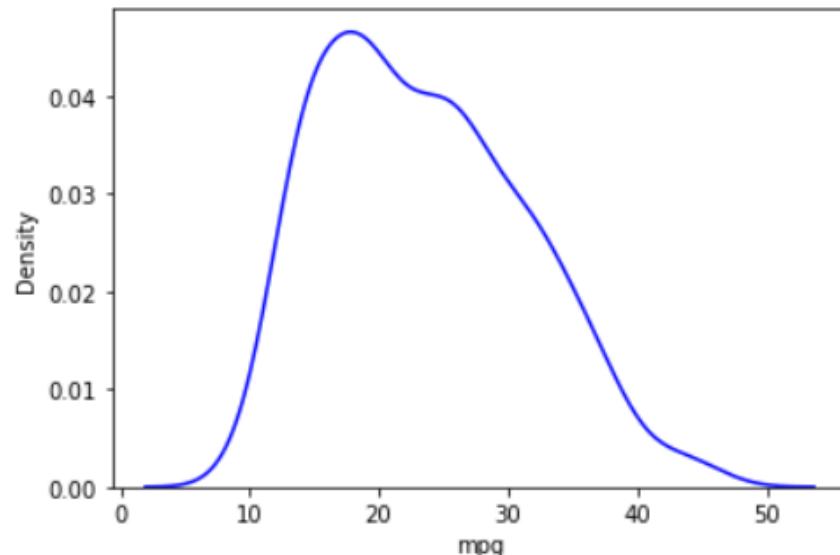
| Histogram + Rug:

```
In [7]: sns.distplot(dat.mpg, kde=False, rug=True, bins=50, color='blue')  
plt.show()
```



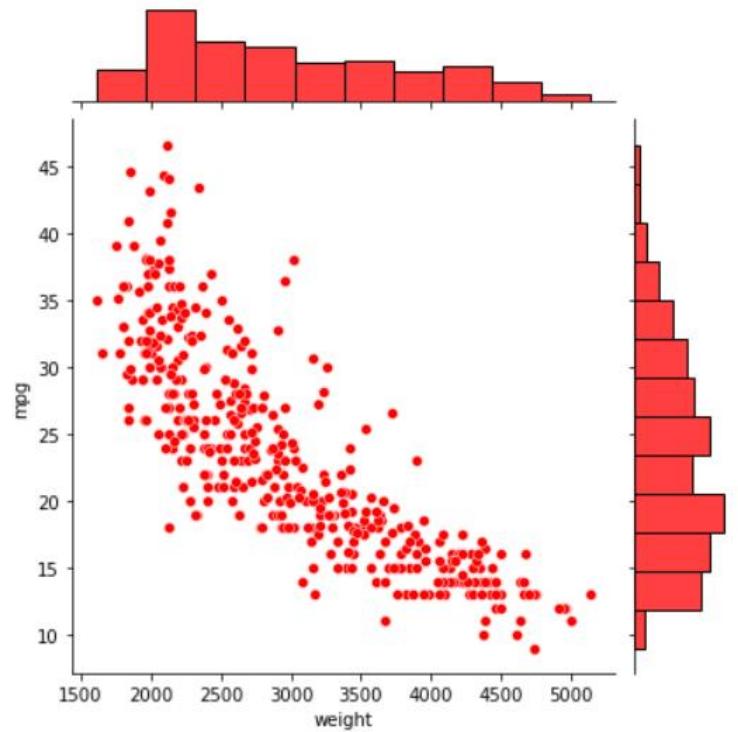
| KDE (Kernel Density Estimation):

```
In [8]: sns.kdeplot(dat.mpg, color='blue')
plt.show()
```



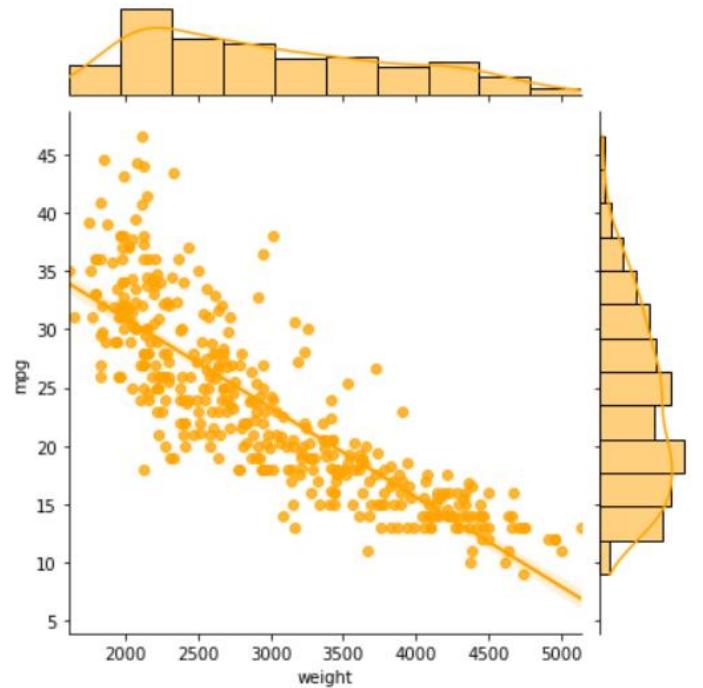
| Scatter plot:

```
In [9]: sns.jointplot(x='weight', y='mpg', data=dat, color='red', kind='scatter')
plt.show()
```



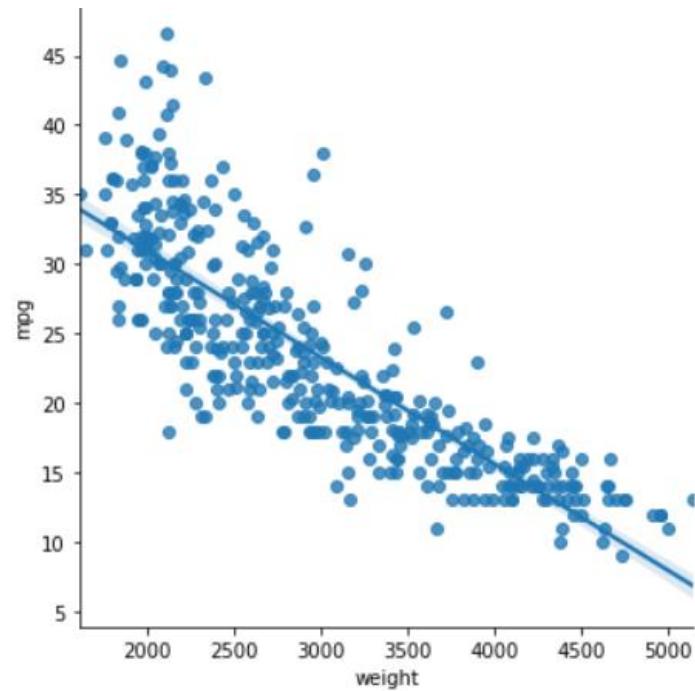
| Scatter plot + regression line:

```
In [10]: sns.jointplot(x='weight', y='mpg', data=dat, color='orange', kind='reg')  
plt.show()
```



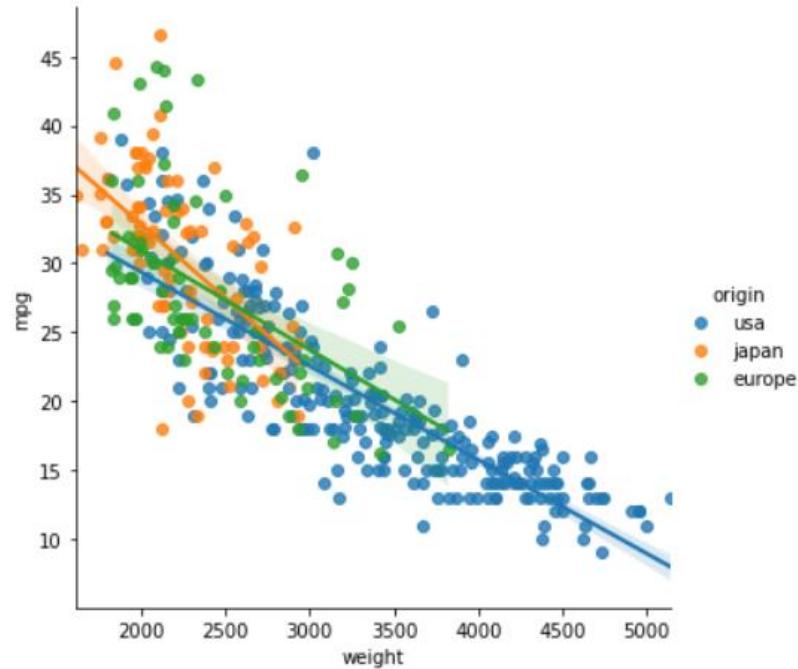
| Scatter plot + regression line:

```
In [11]: sns.lmplot(data=dat, x= 'weight', y='mpg')  
plt.show()
```



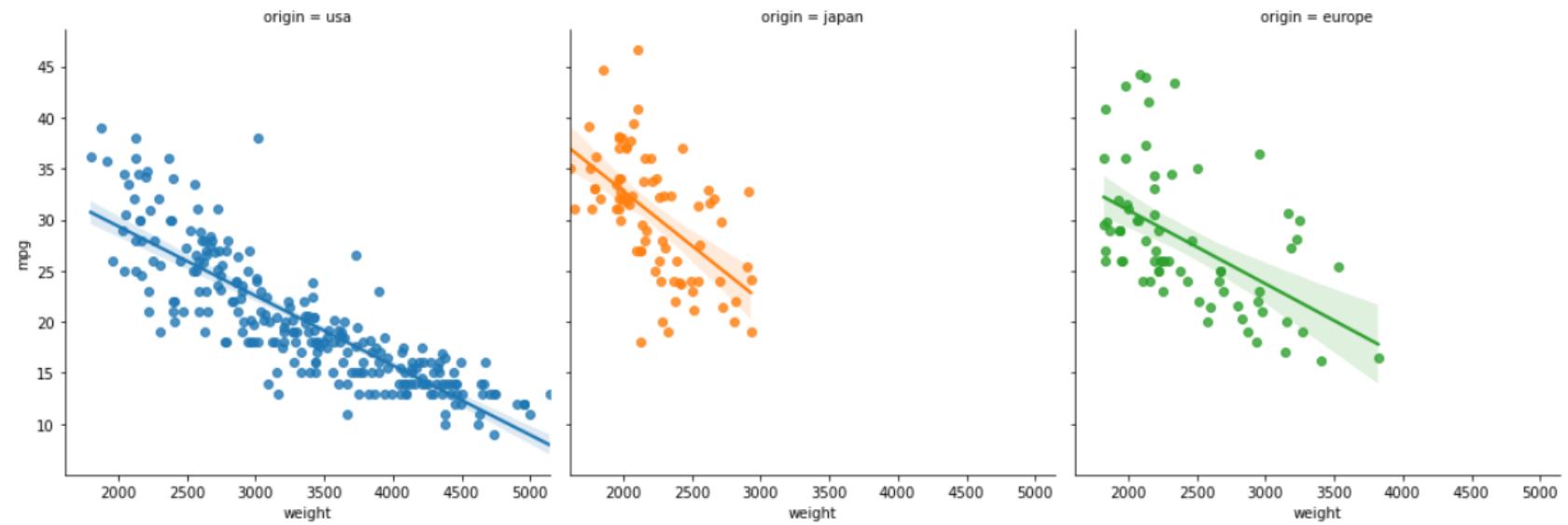
| Scatter plot + regression line (using 'hue' argument):

```
In [12]: sns.lmplot(data=dat, x='weight', y='mpg', hue = 'origin')
plt.show()
```



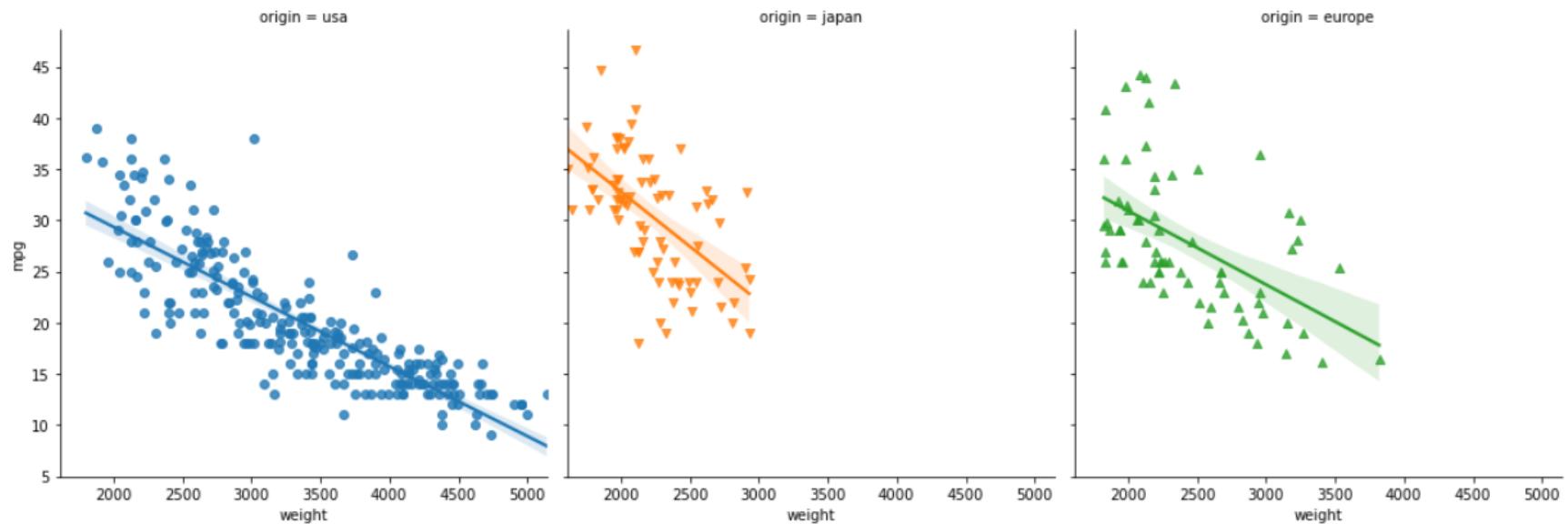
| Scatter plot + regression line (multiple of plots):

```
In [13]: sns.lmplot(data=dat, x='weight', y='mpg', col = 'origin', hue = 'origin')
plt.show()
```



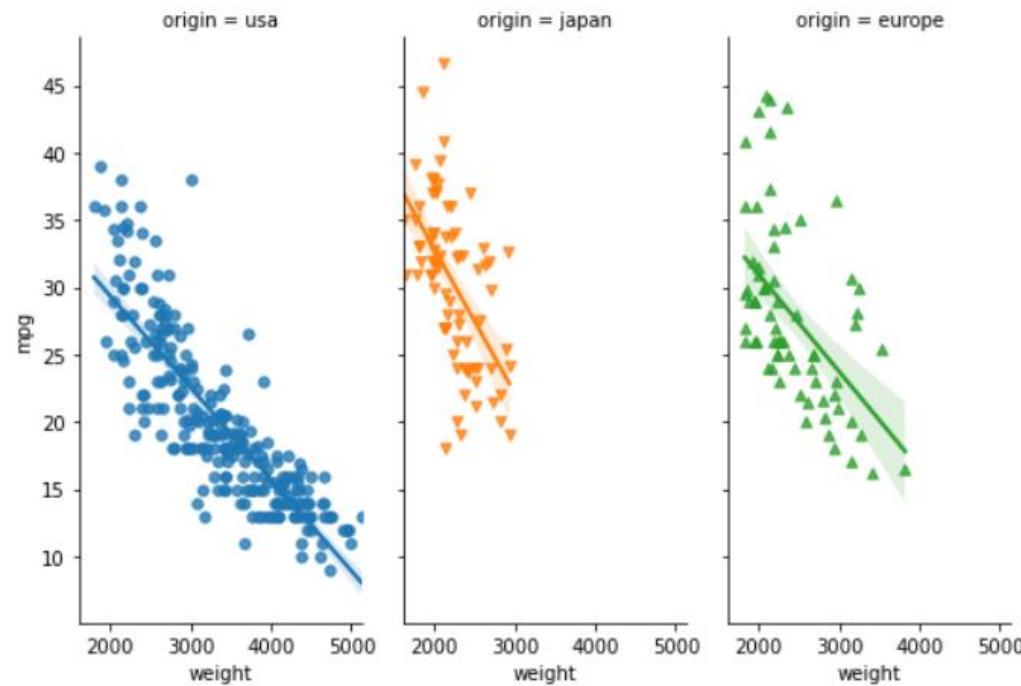
| Scatter plot + regression line (multiple of plots):

```
In [14]: sns.lmplot(data=dat, x='weight', y='mpg', col = 'origin', hue = 'origin', markers=['o','v','^'])  
plt.show()
```



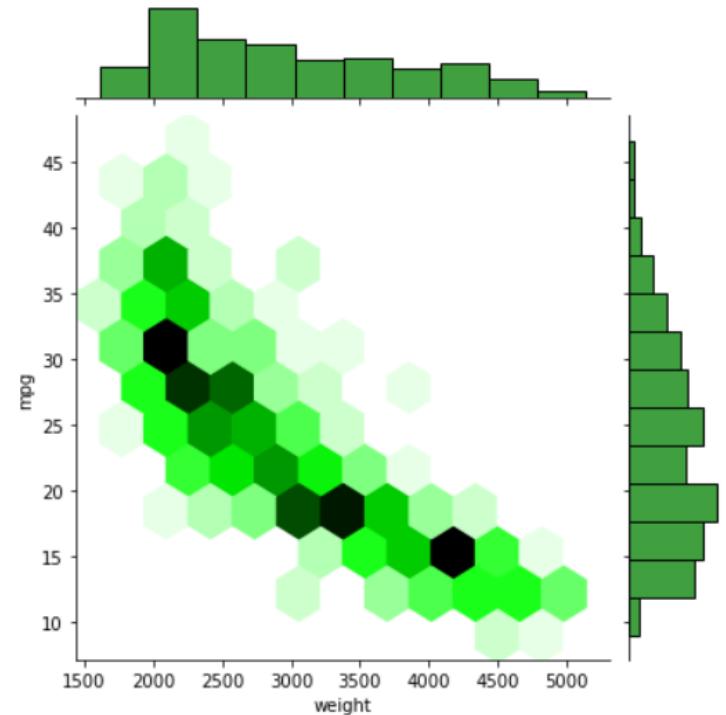
| Scatter plot + regression line (multiple of plots):

```
In [15]: sns.lmplot(data=dat, x='weight', y='mpg', col='origin', hue='origin', markers=['o', 'v', '^'],
                  aspect=0.5)
plt.show()
```



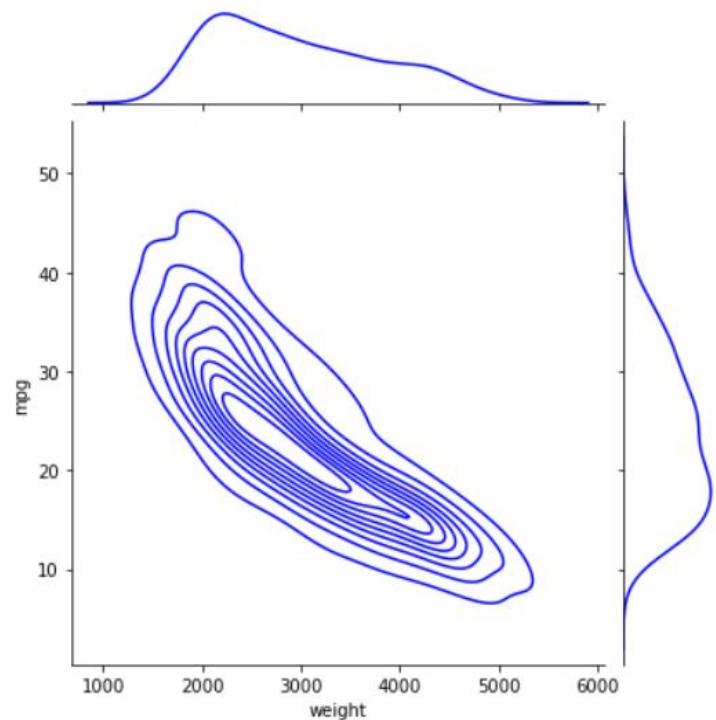
| Hex:

```
In [16]: sns.jointplot(x='weight', y='mpg', data=dat, color='green', kind='hex')
plt.show()
```



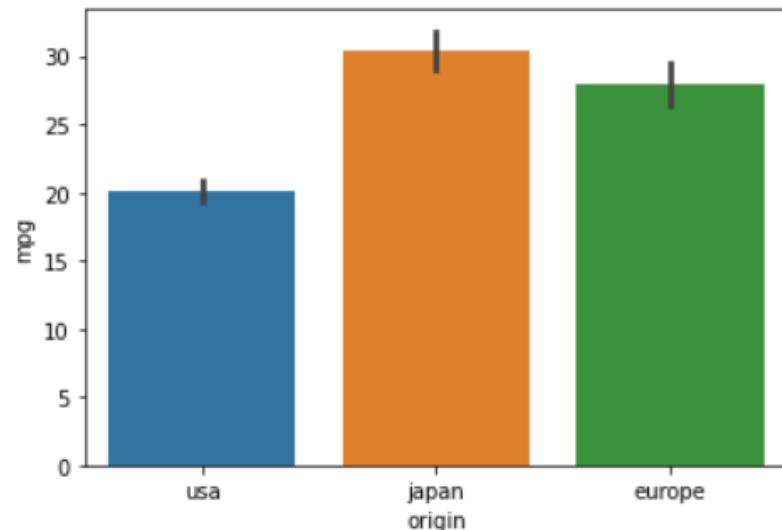
| KDE (Kernel Density Estimation):

```
In [17]: sns.jointplot(x='weight', y='mpg', data=dat, color='blue', kind='kde')
plt.show()
```



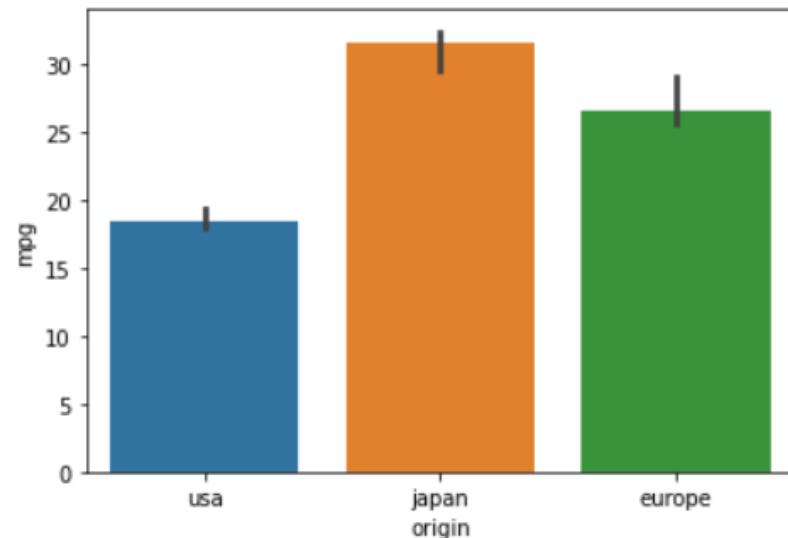
| One categorical variable + one numeric variable. Aggregated by the mean.

```
In [18]: sns.barplot(x='origin',y='mpg',data=dat)  
plt.show()
```



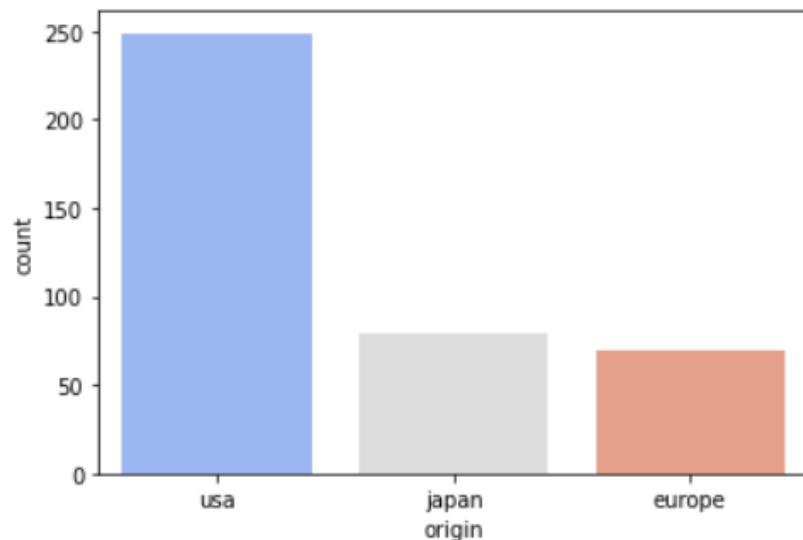
| One categorical variable + one numeric variable. Aggregated by the median.

```
In [19]: sns.barplot(x='origin',y='mpg',data=dat,estimator=np.median)  
plt.show()
```



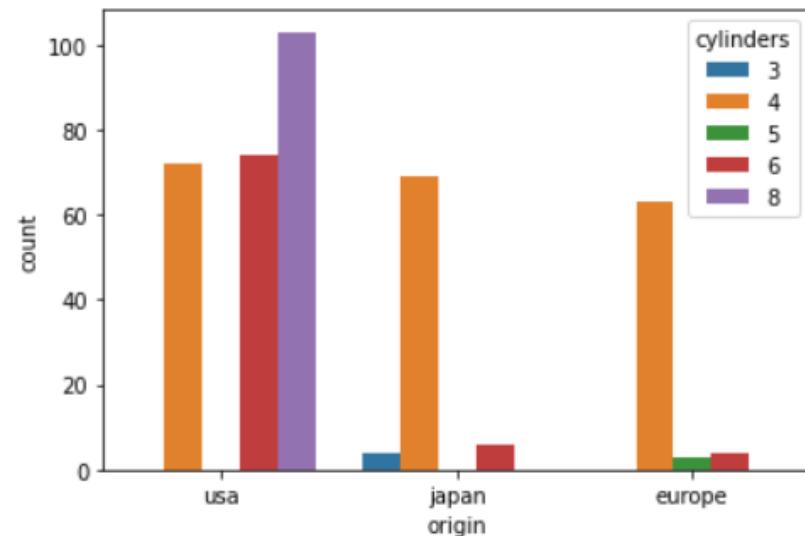
| One categorical variable only. Frequency table is implicitly calculated.

```
In [20]: sns.countplot(x='origin', data=dat, palette='coolwarm')
plt.show()
```



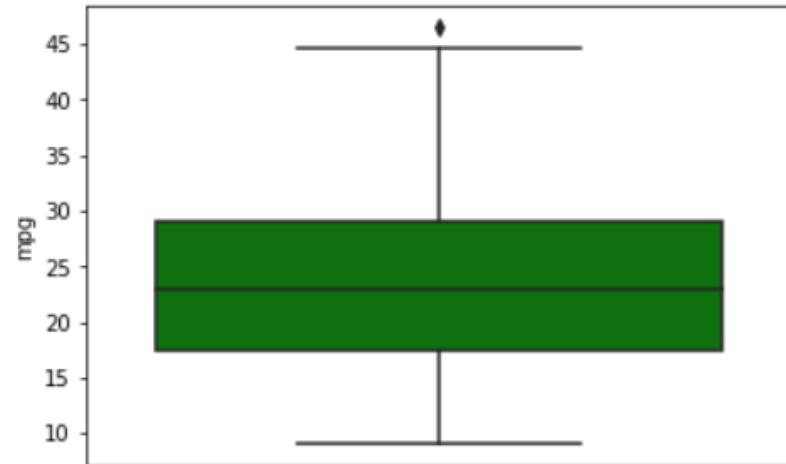
| Two categorical variables. Use of 'hue' argument.

```
In [21]: sns.countplot(x='origin', data=dat, hue='cylinders')
plt.show()
```



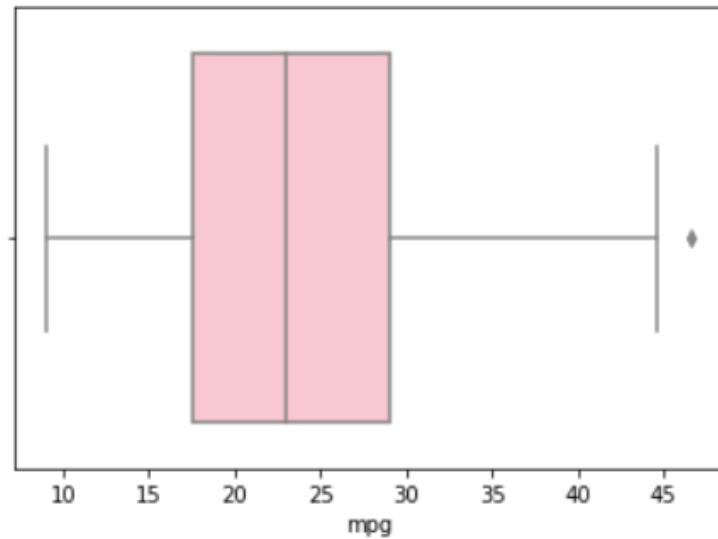
| Horizontal boxplot:

```
In [22]: sns.boxplot(y='mpg', data=dat, color='green')
plt.show()
```



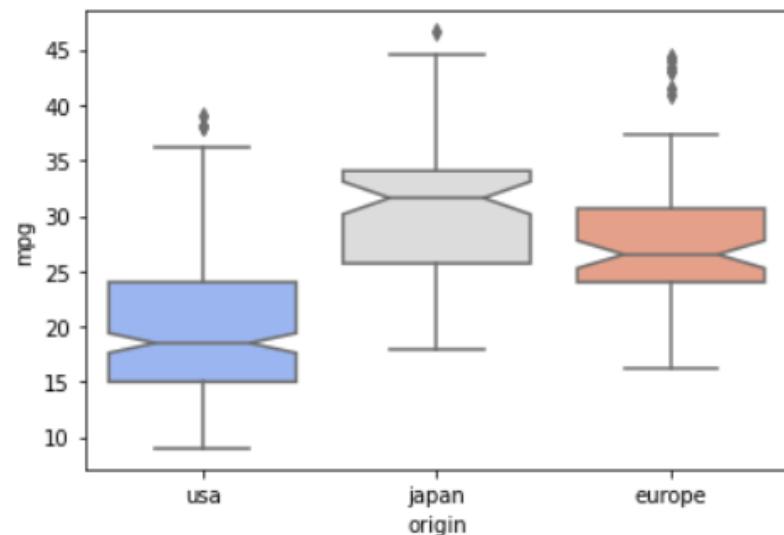
| Vertical boxplot:

```
In [23]: sns.boxplot(x='mpg', data=dat, color='pink')
plt.show()
```



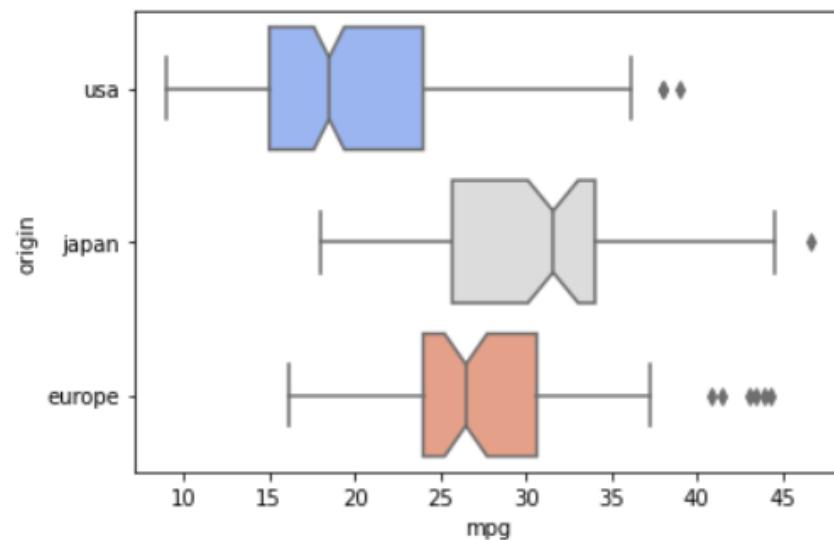
| Multiple boxplots:

```
In [24]: sns.boxplot(x='origin', y='mpg', data=dat, palette='coolwarm', notch=True)  
plt.show()
```



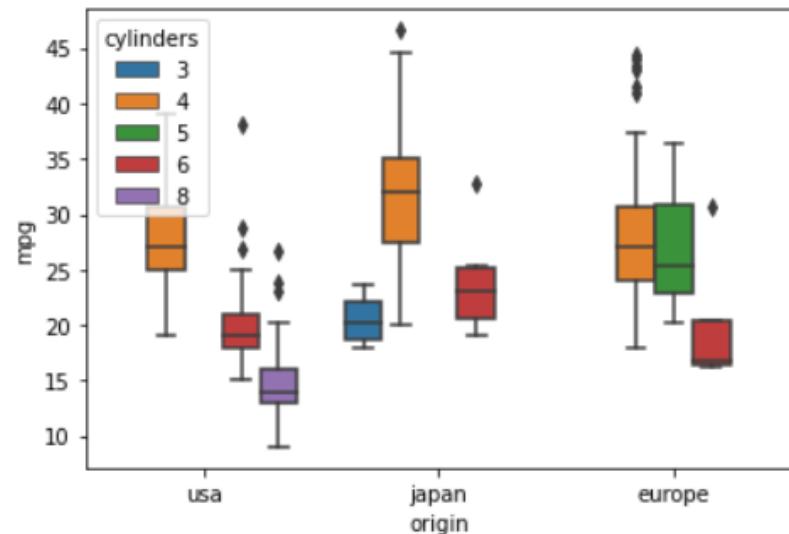
| Multiple boxplots:

```
In [25]: sns.boxplot(x='mpg', y='origin', data=dat, palette='coolwarm', notch=True)  
plt.show()
```



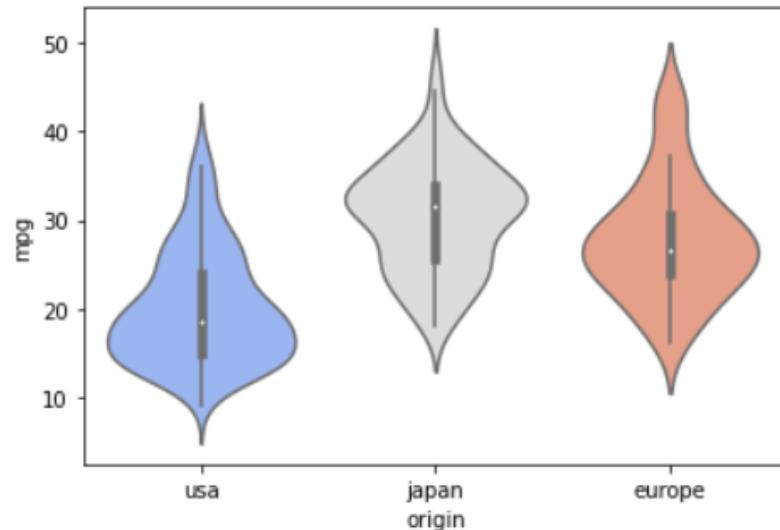
| Multiple boxplots. Two categorical variables + one numeric variable. Use of 'hue' argument.

```
In [26]: sns.boxplot(x='origin', y='mpg', data=dat, hue='cylinders')
plt.show()
```



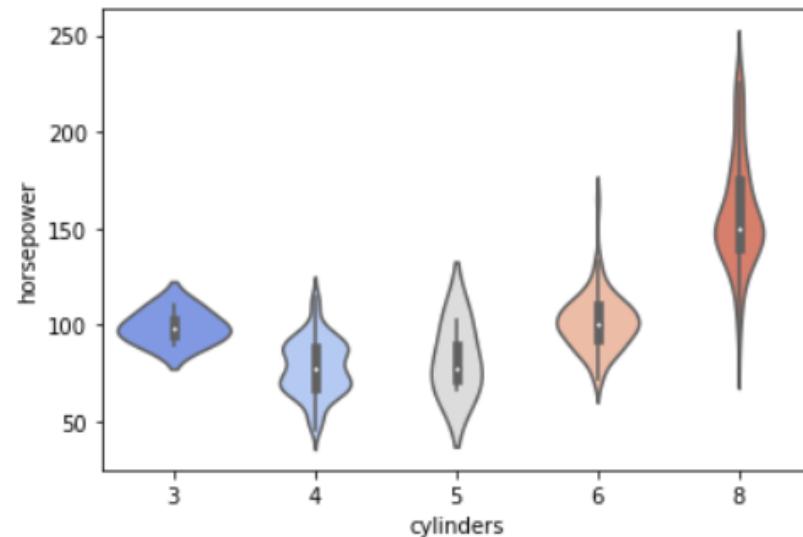
| Multiple violin plots:

```
In [27]: sns.violinplot(x='origin', y='mpg', data=dat, palette='coolwarm')
plt.show()
```



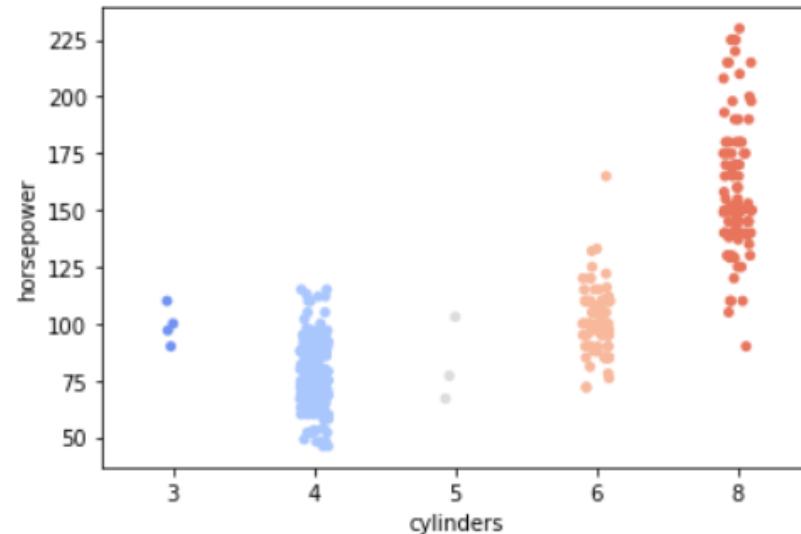
| Multiple violin plots:

```
In [28]: sns.violinplot(x='cylinders', y='horsepower', data=dat, palette='coolwarm')
plt.show()
```



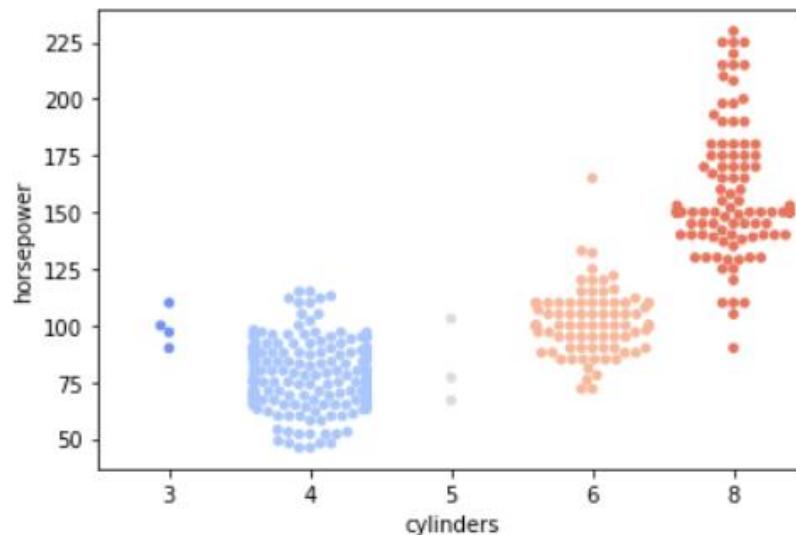
| Strip plot:

```
In [29]: sns.stripplot(x='cylinders', y='horsepower', data=dat, palette='coolwarm')
plt.show()
```



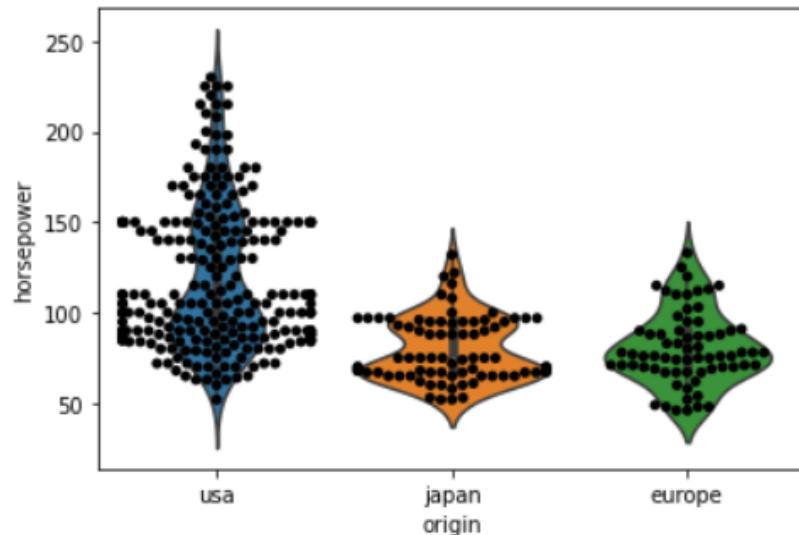
| Swarm plot:

```
In [30]: sns.swarmplot(x='cylinders', y='horsepower', data=dat, palette='coolwarm')
plt.show()
```



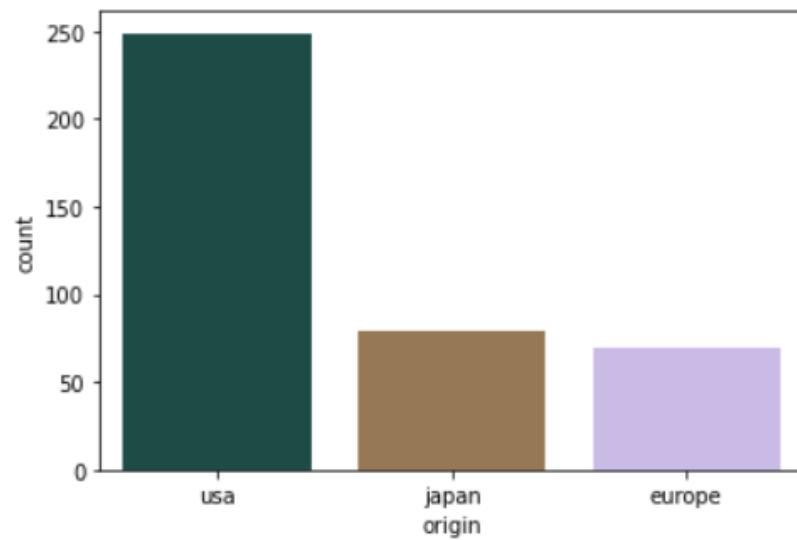
| Violin plot + Swarm plot:

```
In [31]: sns.violinplot(x='origin', y='horsepower', data=dat)
sns.swarmplot(x='origin', y='horsepower', data=dat, color='black')
plt.show()
```



| Color palette (Frequency table shown as bar plot):

```
In [32]: sns.countplot(x='origin', data=dat, palette='cubebeelix')
# sns.countplot(x='origin', data=dat, palette='coolwarm')
# sns.countplot(x='origin', data=dat, palette='magma')
# sns.countplot(x='origin', data=dat, palette='seismic')
plt.show()
```



Seaborn Visualization Library (with more examples)

| Import modules.

```
In [1]: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import os
import warnings
%matplotlib inline
warnings.filterwarnings('ignore')          # Turn off the warnings.
```

| Scatter plot array:

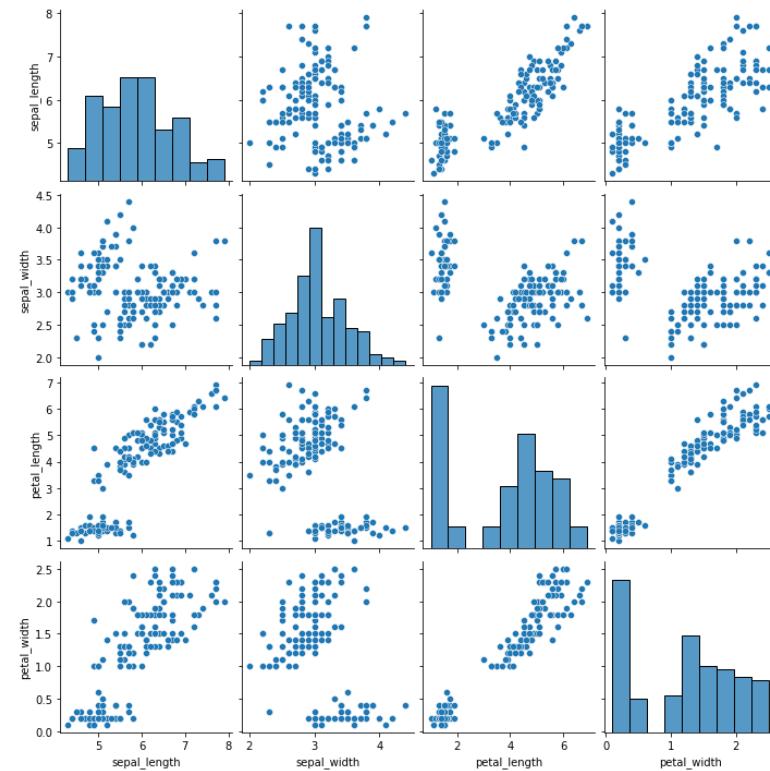
```
In [2]: dat = sns.load_dataset('iris')
```

Line 2

- Read in the 'iris' data from the library.

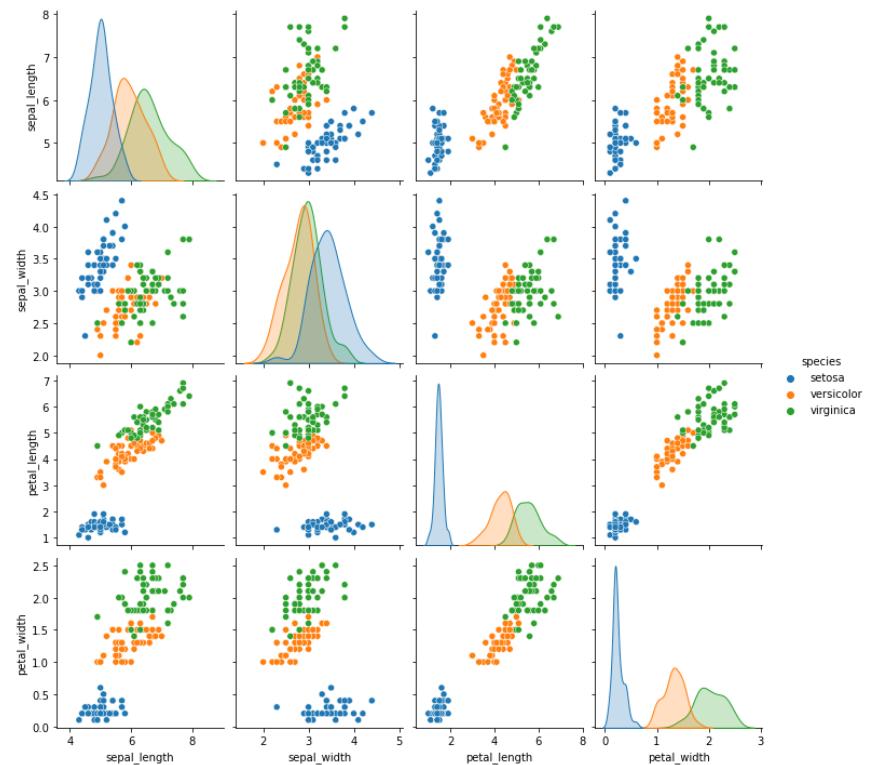
| Scatter plot array:

```
In [3]: sns.pairplot(dat)  
plt.show()
```



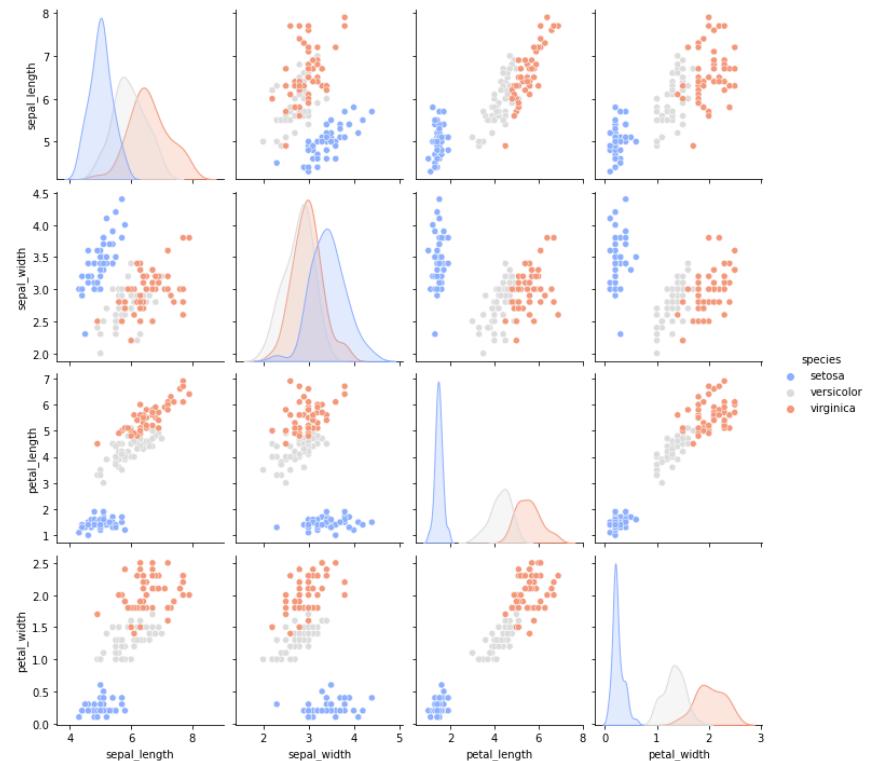
| Scatter plot array (Color the markers with 'species.'):

```
In [4]: sns.pairplot(dat, hue='species')  
plt.show()
```



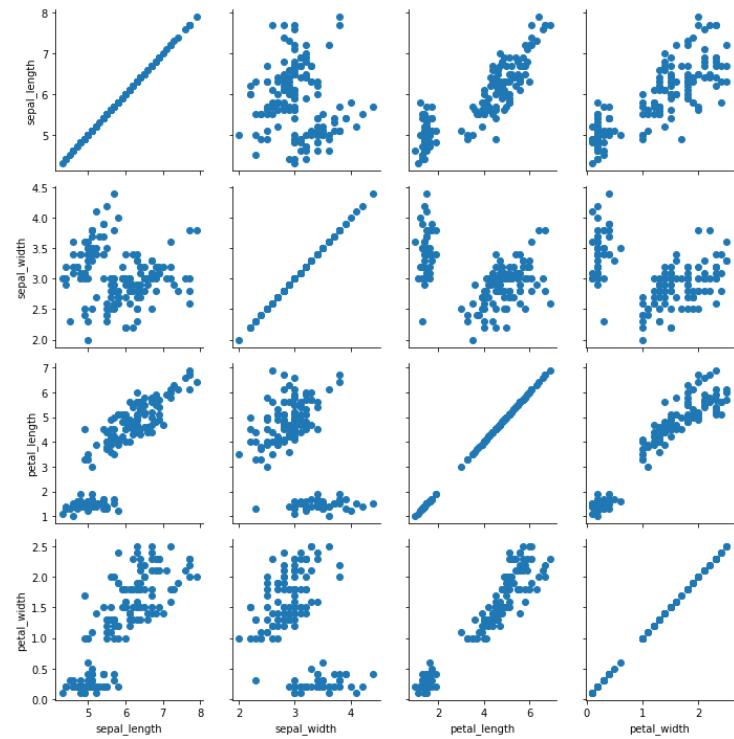
| Scatter plot array (Color the markers with 'species.' Apply palette.):

```
In [5]: sns.pairplot(dat, hue='species', palette='coolwarm')
plt.show()
```



| PairGrid (An array of scatter plots):

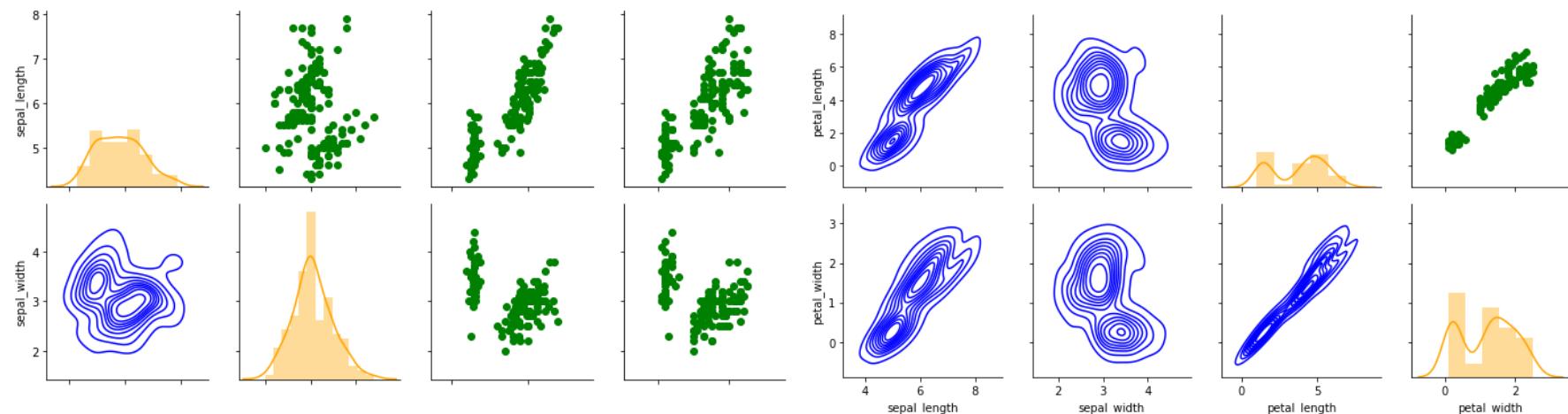
```
In [6]: g=sns.PairGrid(dat)
g.map(plt.scatter)
plt.show()
```



| PairGrid (An array of different visualization types):

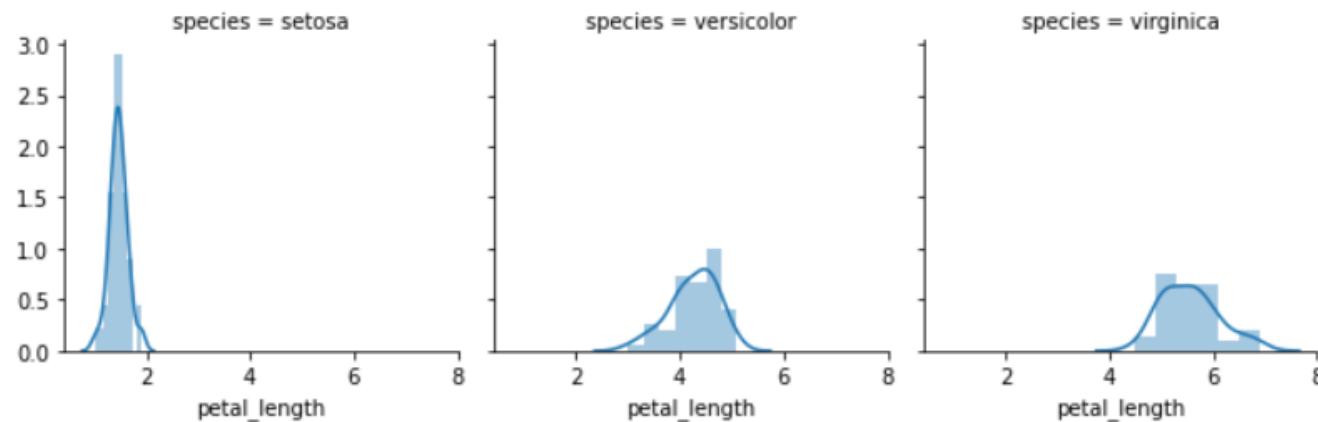
```
In [7]: g=sns.PairGrid(dat)
g.map_diag(sns.distplot, color='orange')
g.map_upper(plt.scatter, color='green')
g.map_lower(sns.kdeplot, color='blue')
plt.show()
```

Diagonal = histogram.
Upper triangle = scatter plot.
Lower triangle = KDE.



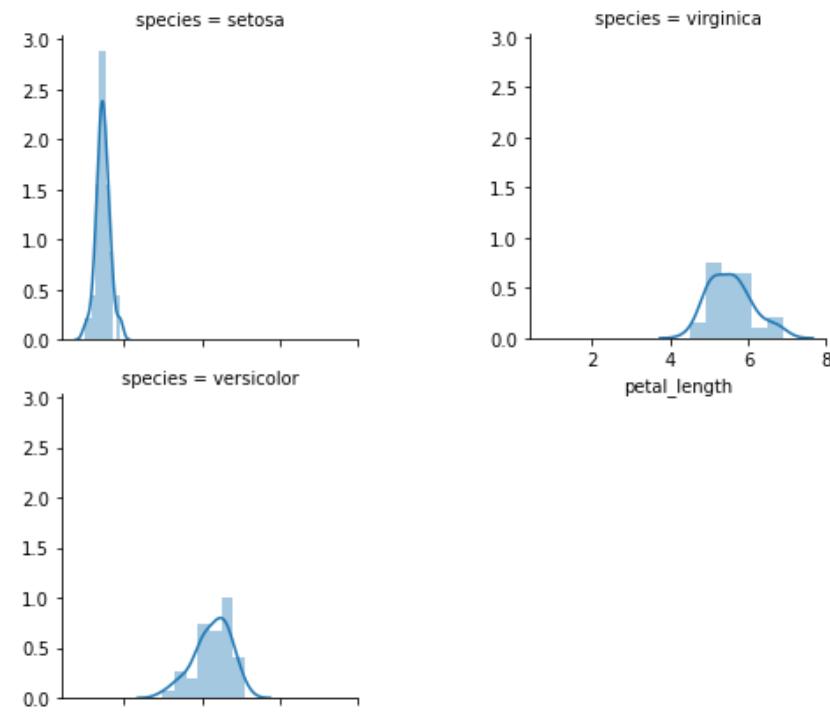
| FacetGrid (Multiple histograms):

```
In [8]: g=sns.FacetGrid(data=dat, col='species')
g.map(sns.distplot, 'petal_length')
plt.show()
```



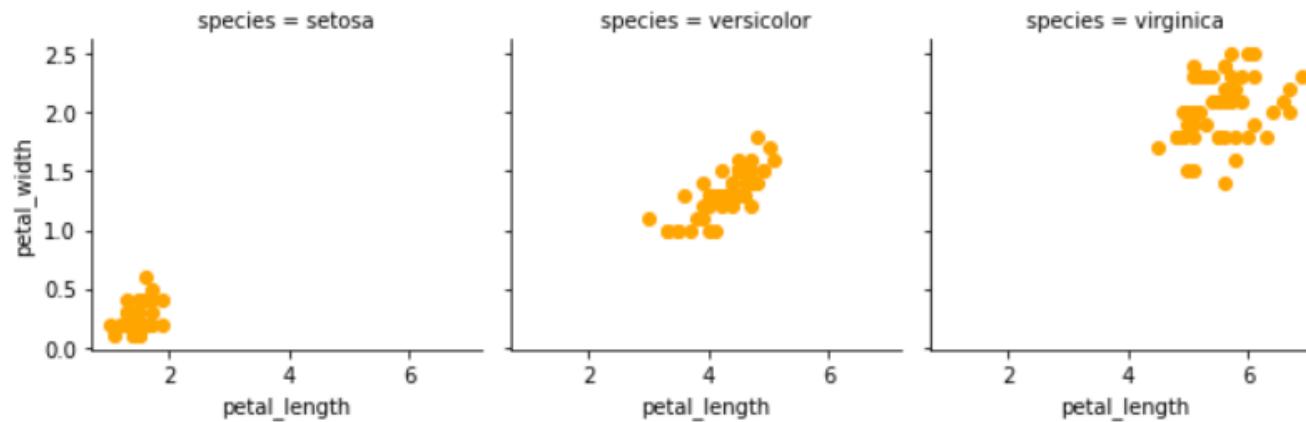
| FacetGrid (Multiple histograms):

```
In [9]: g=sns.FacetGrid(data=dat, row='species')
g.map(sns.distplot, 'petal_length')
plt.show()
```



| FacetGrid (Multiple scatter plots):

```
In [10]: g=sns.FacetGrid(data=dat, col='species')
g.map(plt.scatter, 'petal_length', 'petal_width',color='orange')
plt.show()
```



| Heatmap:

```
In [11]: dat = sns.load_dataset('mpg')
dat.drop(columns=['origin', 'name'], inplace=True)
```

Line 11

- Read in 'mpg' data from the library.

| Heat map (Correlation matrix):

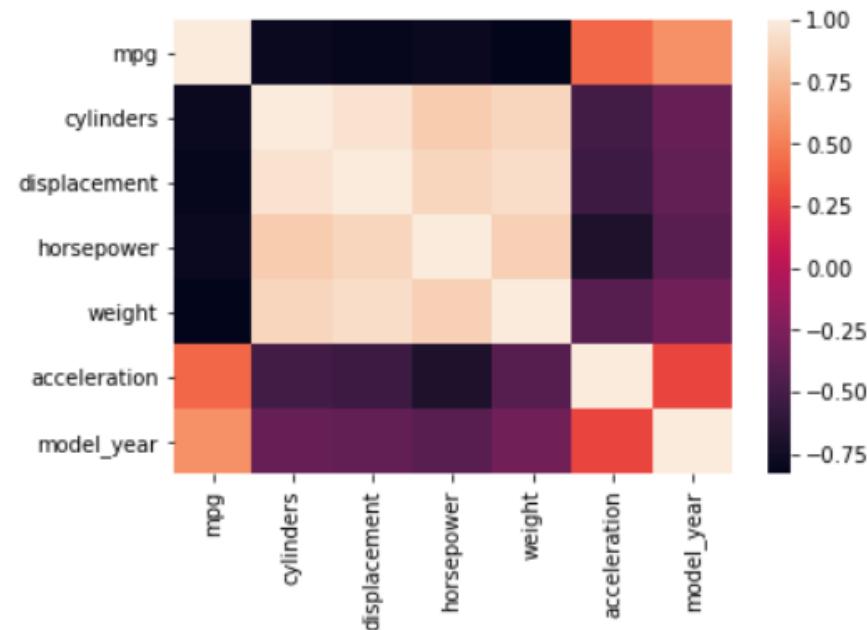
```
In [12]: x = dat.corr()  
x
```

Out[12]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year
mpg	1.000000	-0.775396	-0.804203	-0.778427	-0.831741	0.420289	0.579267
cylinders	-0.775396	1.000000	0.950721	0.842983	0.896017	-0.505419	-0.348746
displacement	-0.804203	0.950721	1.000000	0.897257	0.932824	-0.543684	-0.370164
horsepower	-0.778427	0.842983	0.897257	1.000000	0.864538	-0.689196	-0.416361
weight	-0.831741	0.896017	0.932824	0.864538	1.000000	-0.417457	-0.306564
acceleration	0.420289	-0.505419	-0.543684	-0.689196	-0.417457	1.000000	0.288137
model_year	0.579267	-0.348746	-0.370164	-0.416361	-0.306564	0.288137	1.000000

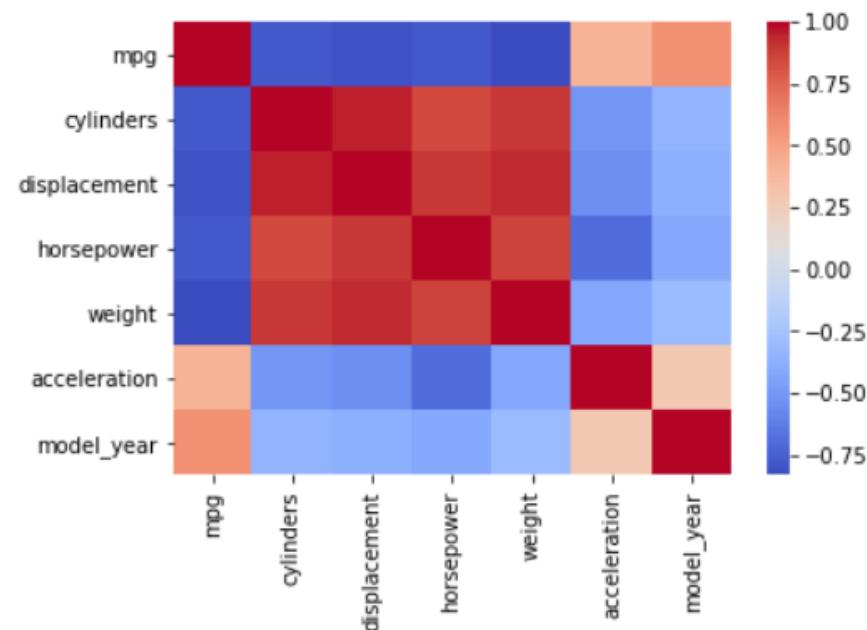
| Heat map (Default heatmap):

```
In [13]: sns.heatmap(x)  
plt.show()
```



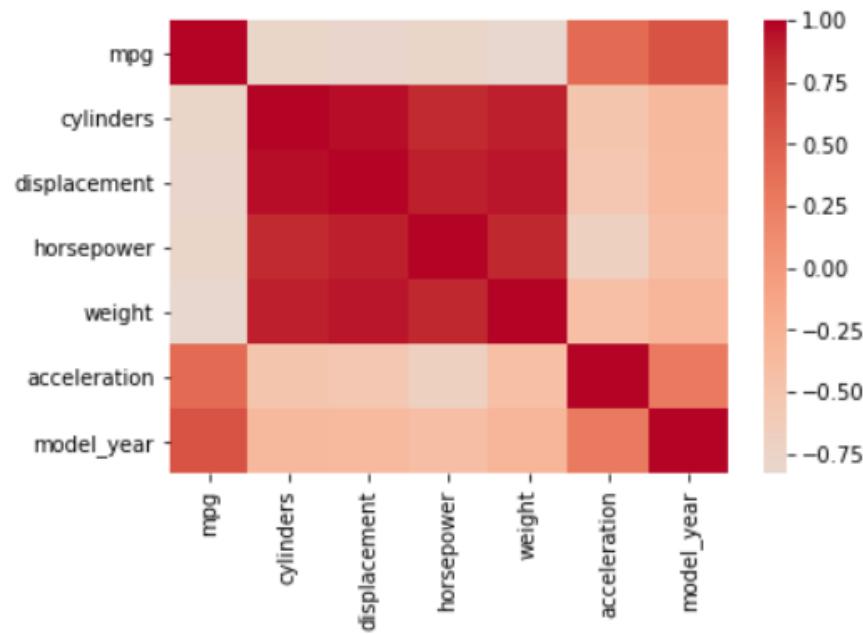
| Heat map (Use 'cmap' instead of 'palette.'):

```
In [14]: sns.heatmap(x, cmap = 'coolwarm')
plt.show()
```



| Heat map (Adjust the center.):

```
In [15]: sns.heatmap(x, cmap = 'coolwarm', center=-1)  
plt.show()
```



Coding Exercise #0113



Follow practice steps on 'ex_0113.ipynb' file

A photograph of a person working at a desk. They are wearing an orange long-sleeved shirt and are holding a brown paper coffee cup with a black lid in their right hand. Their left hand is on a black computer keyboard. In front of them is a large monitor displaying a dark interface with text. To the left of the monitor, there's a stack of papers or books. A pair of glasses sits on the desk next to the papers. The background shows a window with vertical blinds.

End of Document



Together for Tomorrow! Enabling People

Education for Future Generations

©2023 SAMSUNG. All rights reserved.

Samsung Electronics Corporate Citizenship Office holds the copyright of book.

This book is a literary property protected by copyright law so reprint and reproduction without permission are prohibited.

To use this book other than the curriculum of Samsung Innovation Campus or to use the entire or part of this book, you must receive written consent from copyright holder.