I.4: Numpy

- One of the most important foundational packages for numerical computing in Python
- It provides efficient multidimensional array type `ndarray` with fast array-oriented arithmetic operations.
- It allows executing fast operations on entire arrays of data without having to write loops.
- It provides Linear algebra operations, random number generation, and Fourier transform capabilities.
- it is designed for efficiency on large arrays of data.
- NumPy internally stores data in a contiguous block of memory, independent of other built-in Python object
- It is a computational foundation for general numerical data processing for many other packages such as `pandas` and `SciPy`.

```
import sys
import numpy as np

py_arr = list(range(1000000)) #standard python array
np_arr = np.arange(1000000) #numpy array

#print time consumed in multiplicatios
%time for _ in range(10): np_arr * 2/2
%time for _ in range(10): [x * 2/2 for x in py_arr]

CPU times: user 31.8 ms, sys: 6.08 ms, total: 37.9 ms
Wall time: 51.8 ms
CPU times: user 1.57 s, sys: 180 ms, total: 1.75 s
Wall time: 2.26 s
```

`NumPy`-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

NumPy ndarray (Multidimensional Array)

- a key feature of NumPy is its N-dimensional array object, or ndarray.
- It is a fast, flexible container for large datasets in Python.
- It enables you to perform mathematical operations on whole blocks of data similarly to the operations between scalars.
- It is a generic multidimensional container for homogeneous data(same type).
- It has `shape`, a tuple indicating the size of each dimension, and `dtype`, an object describing the data typeof the array

```
import numpy as np
np_data = np.random.randn(4,3) # create an array of random values with
dims 4x3
print("np_data: \n", np_data)

#do some math operations with the array
np_data = (np_data + np_data) * 2
```

```
print("\n(np_data + np_data) * 2: \n", np_data)

#print shape and dtype of the array
print("\n shape: ", np_data.shape, "; dtype: ", np_arr.dtype)

np_data:
 [[-0.97964204 -0.25464528  1.14784297]
 [-0.19705887 -1.20501532  0.6985225 ]
 [-0.29363815 -0.35362777 -0.67977467]
 [-1.11033485  1.26349186  0.63231134]]

(np_data + np_data) * 2:
 [[-3.91856818 -1.01858113  4.59137189]
 [-0.78823548 -4.82006128  2.79409   ]
 [-1.17455261 -1.41451109 -2.71909869]
 [-4.44133941  5.05396746  2.52924536]]

 shape:  (4, 3) ; dtype:  int64
```

Basic operarions on ndarray

The easiest way to create an array is to use `array` function

```
arr1 = np.array([1.5, 5.1, 6, 12, 3.4])
#create an array 1d
print('\n arr1:\n', arr1)

arr2 = np.array([arr1, arr1, [0, 1, 2, 3, 8]])
#create an array 2d
print('\n arr2:\n', arr2)

arr3 = np.array([arr1,  [0.1, 1.2, 2.3, 3.4, 8.9]], dtype= np.int32)
#create an array given the type
print('\n arr3:\n', arr3)


 arr1:
 [ 1.5  5.1  6.  12.   3.4]

 arr2:
 [[ 1.5  5.1  6.  12.   3.4]
 [ 1.5  5.1  6.  12.   3.4]
 [ 0.   1.   2.   3.   8. ]]

 arr3:
 [[ 1  5  6 12  3]
 [ 0  1  2  3  8]]
```

- There are a number of other functions for creating new arrays.

- As examples, `zeros` and `ones` create arrays of 0s or 1s, respectively, with a given length or shape.
- `empty` creates an array without initializing its values to any particular value.
- To create a higher dimensional array with these methods, pass a tuple for the shape:

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

```
arr1 = np.zeros((2,6))
print('\n arr1:\n', arr1)

arr2 = np.empty((2,3,2))
print('\n arr2:\n', arr2)

arr3 = np.arange(20)
print('\n arr3:\n', arr3)
```

```
 arr1:
 [[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]

 arr2:
 [[[0. 0.]
  [0. 0.]
  [0. 0.]]

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

 arr3:
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Arithmetic with NumPy Arrays

- Arrays enable you to express batch operations on data without writing any forloops. NumPy users call this vectorization.
- Any arithmetic operations between equal-size arrays applies the operation element-wise.
- avoid using loops as match as you can while dealing with `ndarray`

Example 1: The following example show the diffrence between function based on a loop and another one based on nparray `ufuncs`

```
import numpy as np

#define a standard function for calcualting reciprocals using loop
def calc_rec(vect):
  res = np.empty(len(vect))
  for i in range(len(vect)):
       res[i] = 1.0 / vect[i]
  return res

vect = np.random.randint(1,20,5)
print("vect: \n", vect)
print("Reciprocals using our function: \n", calc_rec(vect))
print("Reciprocals using numpy ufuncs: \n", 1/vect)

vect:
 [ 2  3  9  6 14]
Reciprocals using our function:
 [0.5        0.33333333 0.11111111 0.16666667 0.07142857]
Reciprocals using numpy ufuncs:
 [0.5        0.33333333 0.11111111 0.16666667 0.07142857]

# compare the approaches in terms of execution time

big_vect = np.random.randint(1,20,1000000)

%timeit calc_rec(big_vect)
%timeit (1/big_vect)

1 loop, best of 5: 2.22 s per loop
1000 loops, best of 5: 1.54 ms per loop
```

- Vectorized operations in NumPy are implemented via ufuncs.
- ufuncs's main purpose is to quickly execute repeated operations on values in NumPy arrays without needing loops.
- Ufuncs are extremely flexible (operations can be performed between scalar/ array and array/array.

Example 2:

```
import numpy as np

arr1 = np.ones((2,3))/(3 * np.random.randn(2,3)) #scaler/array and
array/array operations
arr2 = np.power(np.reshape(arr1,(1,6), order='F'), 2)
arr3 = arr1+arr1
arr4 = arr3<0

print(' arr1:\n', arr1)
print('\n arr2:\n', arr2)
print('\n arr3:\n', arr3)
print('\n arr4:\n', arr4)

 arr1:
 [[-0.58760428 -4.14201903 -2.68074686]
 [-1.95918352  0.24392213 -0.3310625 ]]

 arr2:
 [[ 0.34527879  3.83840008 17.15632164  0.05949801  7.18640374
0.10960238]]

 arr3:
 [[-1.17520856 -8.28403806 -5.36149373]
 [-3.91836705  0.48784426 -0.662125  ]]

 arr4:
 [[ True  True  True]
 [ True False  True]]
```

Example 3: Here ae some other common ufuncs used with Numpy arrays. Each of these arithmetic operations are simply wrappers around specific functions built into `NumPy`; for example, the `+` operator is a wrapper for the `numpy.add` function

```
x = np.arange(4)
print("x      =", x)
print("x + 5  =", x + 5)
print("x - 5  =", x - 5)
print("x * 2  =", x * 2)
print("x / 2  =", x / 2)
print("x // 2 =", x // 2)  # floor division
print("-x     =", -x)
print("abs(-x)=", abs(-x))
print("x ** 2 =", x ** 2)
print("x % 2  =", x % 2)
print("np.multiply.outer(x, x)", np.multiply.outer(x, x)) #vwT

x      = [0 1 2 3]
x + 5  = [5 6 7 8]
x - 5  = [-5 -4 -3 -2]
x * 2  = [0 2 4 6]
```

```
x / 2  = [0.  0.5 1.  1.5]
x // 2 = [0 0 1 1]
-x     = [ 0 -1 -2 -3]
abs(-x)= [0 1 2 3]
x ** 2 = [0 1 4 9]
x % 2  = [0 1 0 1]
np.multiply.outer(x, x) [[0 0 0 0]
 [0 1 2 3]
 [0 2 4 6]
 [0 3 6 9]]
```

```
# Trigonometric functions
theta = np.round(np.linspace(0, np.pi, 4), 2)
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta) )
print("cos(theta) = ", np.cos(theta) )
print("tan(theta) = ", np.tan(theta) )
```

```
theta      =  [0.   1.05 2.09 3.14]
sin(theta) =  [0.         0.86742323 0.86821458 0.00159265]
cos(theta) =  [ 1.          0.49757105 -0.49618891 -0.99999873]
tan(theta) =  [ 0.00000000e+00  1.74331531e+00 -1.74976619e+00 -
1.59265494e-03]
```

```
#Exponents and logarithms
x = [1, 2, 3]
print("x        =", x)
print("e^x      =", np.exp(x))
print("2^x      =", np.exp2(x))
print("3^x      =", np.power(3, x))

print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
x        = [1, 2, 3]
e^x      = [ 2.71828183  7.3890561  20.08553692]
2^x      = [2. 4. 8.]
3^x      = [ 3  9 27]
ln(x)    = [0.         0.69314718 1.09861229]
log2(x)  = [0.        1.        1.5849625]
log10(x) = [0.         0.30103    0.47712125]
```

Specialized ufuncs:

- NumPy has many more ufuncs available, look through the NumPy documentation to learn more.
- Another excellent source for more specialized ufuncs is the submodule `scipy.special`

```
from scipy import special
```

```
# Gamma functions (generalized factorials) and related functions
x = [1, 3, 5]
print("gamma(x)     =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2)   =", special.beta(x, 2))

# Error function (integral of Gaussian), its complement, and its
inverse
x = np.array([0, 0.3, 0.7, 1.0])
print("\nerf(x)  =", special.erf(x))
print("erfc(x) =", special.erfc(x))
print("erfinv(x) =", special.erfinv(x))
```

```
gamma(x)     = [ 1.  2. 24.]
ln|gamma(x)| = [0.         0.69314718 3.17805383]
beta(x, 2)   = [0.5        0.08333333 0.03333333]

erf(x)  = [0.         0.32862676 0.67780119 0.84270079]
erfc(x) = [1.         0.67137324 0.32219881 0.15729921]
erfinv(x) = [0.         0.27246271 0.73286908        inf]
```

Indexing and Slicing

- There are many ways you may want to select a subset of data or individual elements.
- One-dimensional ndarrays samply act similarly to Python lists.
- With Numpy, data is not copied, and any modifications to a view will be reflected in the source array. to copy a slice of an ndarray instead of a view, you will need `.copy`

```
import numpy as np

arr = np.arange(5) * 2
print('arr:      ', arr)
print('arr[2]:   ', arr[2])
print('arr[1:4]:', arr[1:4])
arr[1:4] = 1
print('arr:      ', arr)
```

```
arr:      [0 2 4 6 8]
arr[2]:   4
arr[1:4]: [2 4 6]
arr:      [0 1 1 1 8]
```

```
# data modification
arr1 = np.arange(5) + 2
arr2 = arr1[1:4]
arr2[:] = 0
print('arr2:      ', arr2)
print('arr1:      ', arr1)
arr2 = arr1[1:4].copy() #this solves the problem of modifynig the
original data
arr2[:] = 1
```

```
print('-------------------------')
print('arr2:     ', arr2)
print('arr1:     ', arr1)
```

```
arr2:      [0 0 0]
arr1:      [2 0 0 0 6]
-------------------------
arr2:      [1 1 1]
arr1:      [2 0 0 0 6]
```

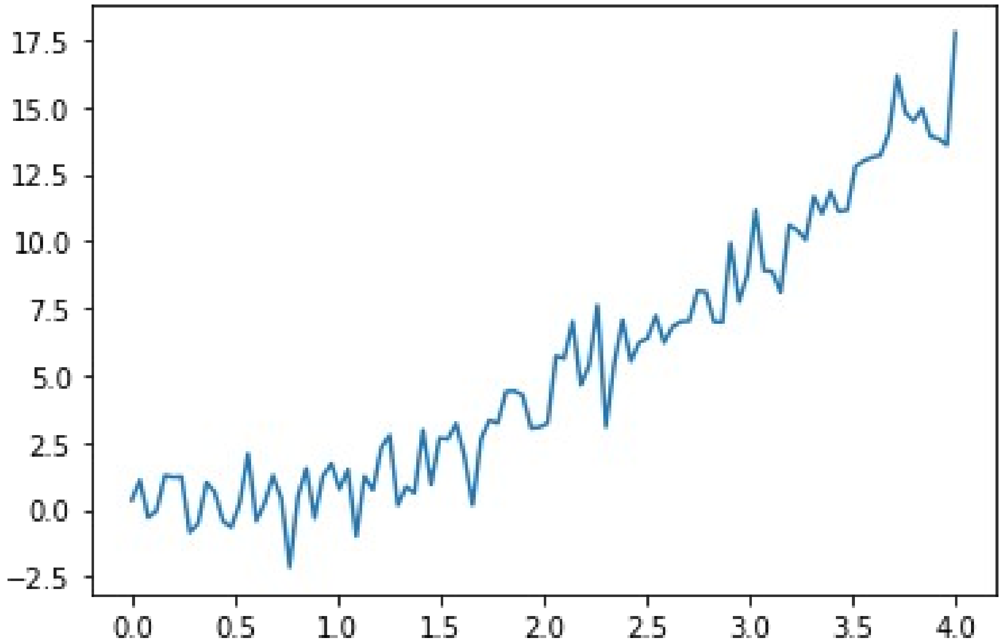
- With higher dimensional arrays, indices are no longer scalars but rather one-dimensional arrays.
- pass a comma-separated list of indices to select individual elements

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print('arr2d[2]:     ', arr2d[2])
print('arr2d[3,2]:   ', arr2d[2,1])

arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

arr3d_2 = arr3d[1]
print('\narr3d_2:\n    ', arr3d_2)
print('\narr3d[1,1]:\n  ', arr3d[1,1])
print('\narr3d[1,1,2]:\n  ', arr3d[1,1,2])
```

```
arr2d[2]:      [7 8 9]
arr2d[3,2]:    8

arr3d_2:
     [[ 7  8  9]
 [10 11 12]]

arr3d[1,1]:
   [10 11 12]

arr3d[1,1,2]:
   12
```

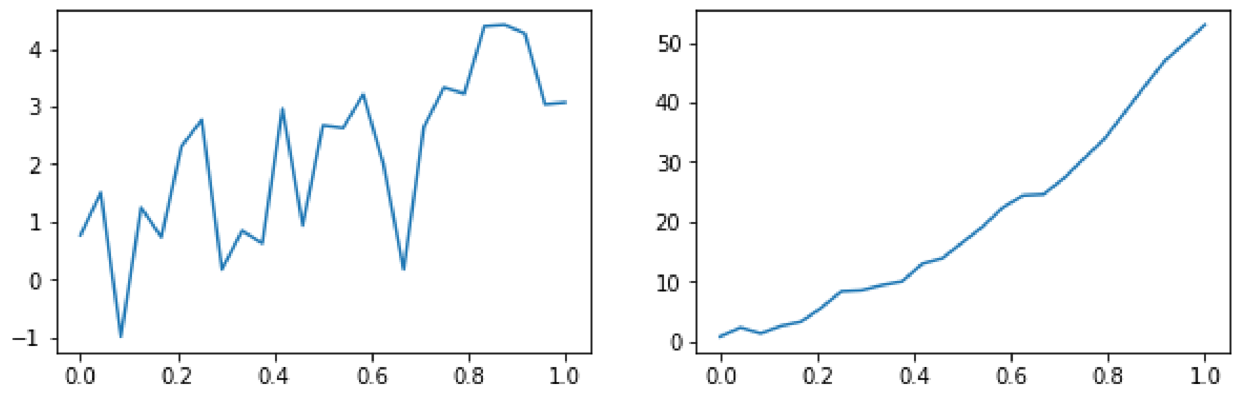
```
import numpy as np
import matplotlib.pyplot as plt

#Let's supose that we have the following ploted profit of 4 days
profit = np.linspace(0,4,100)**2
profit = profit + np.random.randn(100)
plt.plot(np.linspace(0,4,100), profit)
```

```
[<matplotlib.lines.Line2D at 0x7f1e45ed2a10>]
```




```
#we want to plot the profit of day 2
fig = plt.figure(figsize=(10, 3))
profit_day2 = profit[25:50]

fig.add_subplot(1,2,1) # plot the slice
plt.plot(np.linspace(0,1,25), profit_day2)

fig.add_subplot(1,2,2) #plot the accumative sum
plt.plot(np.linspace(0,1,25), profit_day2.cumsum())

#show some statistics
print("max profit: ", profit_day2.max())
print("min profit: ", profit_day2.min())
print("mean profit: ", profit_day2.mean())
print("std profit: ", profit_day2.std())
```

```
max profit:  4.41976198733571
min profit:  -0.9933785966128232
mean profit:  2.117912064412047
std profit:  1.4166553214281168
```



Boolean Indexing Logical operators can be used to slice Numpy arrays ***Example:*** supose that the following students recieved marks as follow

	Khaled	Selma	Djaber
PSD	10	12	9
DL	15	10	11
English	12	12	14

```
students = np.array(['Khaled', 'Selma', 'Djaber'])
modules = np.array(['PSD', 'DL', 'English'])
marks=np.array([[10,12,9], [15,10,11], [12,12,14]])


print("--students == 'Selma':         ", students == 'Selma')
print("--marks[modules == 'DL']:        ", marks[modules == 'DL'])
print("--marks[:,students == 'Selma']: ", marks[:,students ==
'Selma'])
print("--marks[:,students != 'Selma']: ", marks[:,students !=
'Selma'])
print("--marks[modules == 'PSD',students == 'Djaber']: ",
marks[modules == 'PSD',students == 'Djaber'])
marks[marks>10] = marks[marks>10] + 2
```

```
--students == 'Selma':          [False  True False]
--marks[modules == 'DL']:        [[15 10 11]]
--marks[:,students == 'Selma']:  [[12]
 [10]
 [12]]
--marks[:,students != 'Selma']:  [[10  9]
 [15 11]
 [12 14]]
--marks[modules == 'PSD',students == 'Djaber']:  [9]
```

Fancy Indexing

- indexing using integer arrays.
- Fancy indexing, unlike slicing, always copies the data into a new array.

```
data = np.arange(20).reshape((5, 4))
print("---data:\n",data)
print("---data[[1,3,4]]:\n", data[[1,3,4]])
print("---data[[-4,-2,-1]]:\n", data[[-4,-2,-1]])
print("---data[[-8,-6,-1], [0,2,-1]]\n:", data[[-4,-2,-1], [0,2,-1]])
```

```
---data:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
---data[[1,3,4]]:
 [[ 4  5  6  7]
 [12 13 14 15]
 [16 17 18 19]]
---data[[-4,-2,-1]]:
 [[ 4  5  6  7]
 [12 13 14 15]
 [16 17 18 19]]
---data[[-8,-6,-1], [0,2,-1]]
: [ 4 14 19]
```

Aggregations and sorting

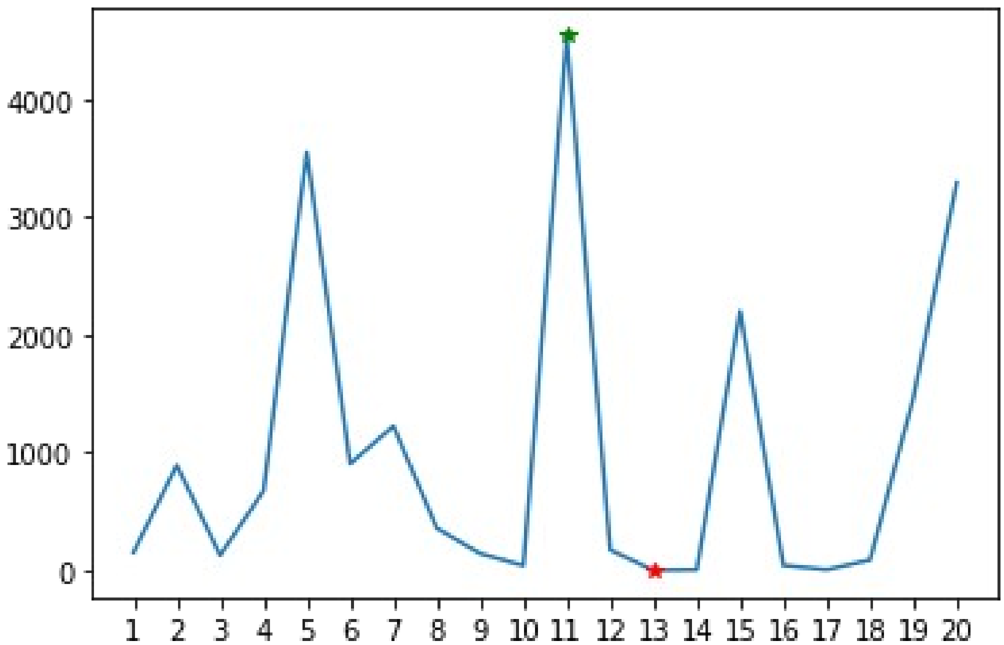
Aggregation:

- When faced with a large amount of data, a first step is to compute summary statistics
- The most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset.
- Other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.)

```
import numpy as np
import matplotlib.pyplot as plt

#generate profit per day data
profit = 1000*np.random.randn(20)**2
plt.plot(np.arange(1,21, 1), profit)
plt.xticks(np.arange(1,21, 1))

#some aggeragations
mean = np.mean(profit);
std = np.std(profit);
median = np.median(profit)
print("mean: {mean:.2f}, standaed deviation: {std:.2f}, and med:
{mean:.2f}".format(mean=mean, std=std, med = median))

#other aggregations
min = np.min(profit)
```

```
x_min = np.argmin(profit) + 1
plt.plot(x_min, min, 'r*')

max = np.max(profit)
x_max = np.argmax(profit) + 1
plt.plot(x_max, max, 'g*');
```

```
mean: 994.76, standaed deviation: 1323.06, and med: 994.76
```



For multi dimentional arrays, the aggregtions are estimated over a specefied axis

```
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print('arr:\n', arr)
arr2 = arr.cumsum(axis=0)
print('arr2:\n', arr2)
np.max(arr, 0)
```

```
arr:
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
arr2:
 [[ 0  1  2]
 [ 3  5  7]
 [ 9 12 15]]

array([6, 7, 8])
```

Sorting

- This subsection covers algorithms related to sorting values in NumPy arrays.
- These algorithms are of fondamuntal topics in introductory computer science courses (insertion sorts, selection sorts, merge sorts, quick sorts, bubble sorts, etc.)
- NumPy arrays can be sorted in-place with the `sort` method of oeder $O[N \log N]$.
- `argsort` is a related function, which instead returns the indices of the sorted elements.

```
import numpy as np
#sorting one dimention array
arr        = np.abs(np.random.randn(5))
sorted_arr = np.sort(arr)
sorted_indices = np.argsort(arr)

print('arr:            ', arr)
print('sorted array:   ', sorted_arr)
print('sorted indices:', sorted_indices)
print('\n\n')

#sorting n dimention array
arr = np.random.randint(0, 10, (3, 4))
sorted_arr_over_rows = np.sort(arr, axis=0)
sorted_arr_over_cls  = np.sort(arr, axis=1)
print('arr:\n', arr)
print('sorted array over rows:\n', sorted_arr_over_rows)
print('sorted array over columns:\n', sorted_arr_over_cls)
print('\n\n')
```

```
arr:            [0.84778948 0.06092322 0.62543832 1.57583266
0.59100793]
sorted array:   [0.06092322 0.59100793 0.62543832 0.84778948
1.57583266]
sorted indices: [1 4 2 0 3]



arr:
 [[6 3 4 1]
 [0 6 7 7]
 [9 6 0 7]]
sorted array over rows:
 [[0 3 0 1]
 [6 6 4 7]
 [9 6 7 7]]
sorted array over columns:
 [[1 3 4 6]
 [0 6 7 7]
 [0 6 7 9]]
```

- somtimes we are interested in finding and sorting the k smallest values in the array.

- `np.partition` takes an array and a number *K*, and produces a new array with the smallest *K* values to the left of the partition, and the remaining values to the right, in arbitrary order.

```
x = np.array([7, 2, 3, 1, 5, 8, 4])
np.partition(x, 3)

array([2, 1, 3, 4, 5, 8, 7])
```

Example: k-Nearest Neighbors:

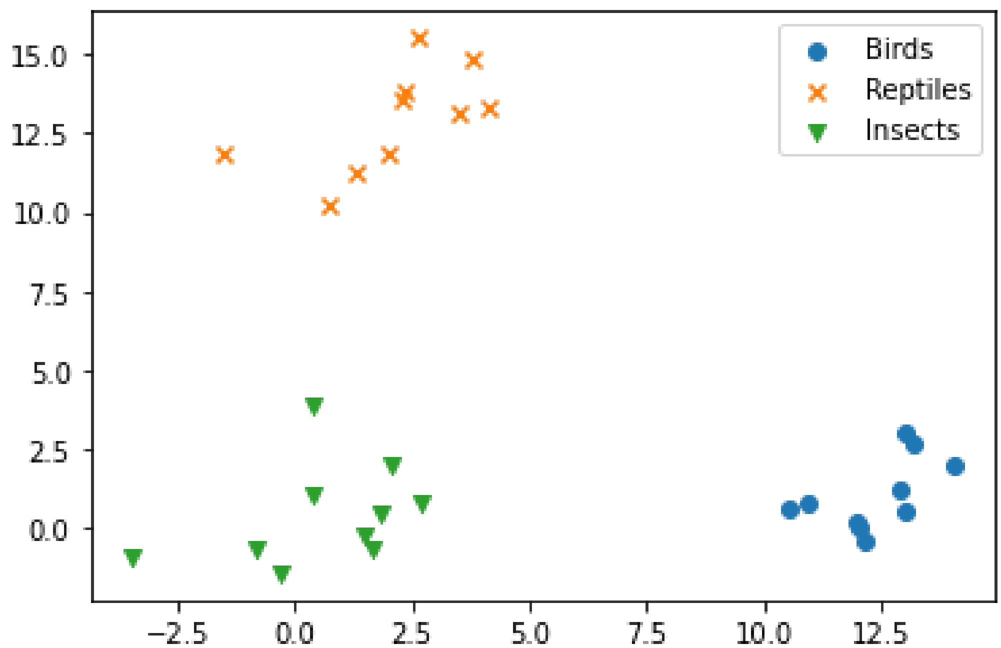
- The k-nearest neighbors (KNN) is a simple supervised machine learning algorithm that can be used to solve classification and regression problems.
- KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other.
- For a given point in the space (image, packet, word, etc), KNN aims at finding the most similar points(i.o.w., finding neighbors after sorting).

```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
fig = plt.figure(figsize=(4, 4))

#Generate Data
Birds = np.random.multivariate_normal([12, 1], [[1,2],[2,1]], 10).T
Reptiles = np.random.multivariate_normal([1, 12], [[1,2],[2,1]], 10).T
Insects = np.random.multivariate_normal([1, 1], [[1,2],[2,1]], 10).T

print("Birds:\n", Birds)
print("Repriles:\n", Reptiles)
print("Insects:\n", Insects)

Birds:
 [[12.90167044 14.07200497 10.54703882 13.19117433 13.0424816
13.02076662
  10.96055607 12.01516425 12.16171476 11.99964493]
 [ 1.25530601  2.04573506  0.60441732  2.6614238   3.0181768
0.56147608
   0.81673676  0.07888133 -0.38632499  0.20938731]]
Repriles:
 [[ 0.71545065  3.765862    1.30666699  2.62239463  2.33971954 -
1.54288147
   2.25728551  3.49278675  4.15338558  1.99934912]
 [10.2127597  14.84638237 11.23207262 15.5107982  13.85921634
11.86678895
  13.5720774  13.1557101  13.34748752 11.82452219]]
Insects:
 [[ 1.45795908  0.35836242  1.83584422  0.40669918  2.70512557
2.02988764
  -0.85182524 -0.31542024  1.63233144 -3.49929197]
```

```
 [-0.25284019  1.10457218  0.42908635  3.87453712  0.80574787
1.98689088
  -0.675192   -1.41049149 -0.65751082 -0.8707069 ]]

<Figure size 288x288 with 0 Axes>
```

To get an idea of how these points look, let's quickly scatter plot them.

```
plt.scatter(Birds[0], Birds[1], marker='o', label='Birds' )
plt.scatter(Reptiles[0], Reptiles[1], marker='x' , label='Reptiles')
plt.scatter(Insects[0], Insects[1], marker='v' , label='Insects');
plt.legend()

<matplotlib.legend.Legend at 0x7f1e46ca9250>
```

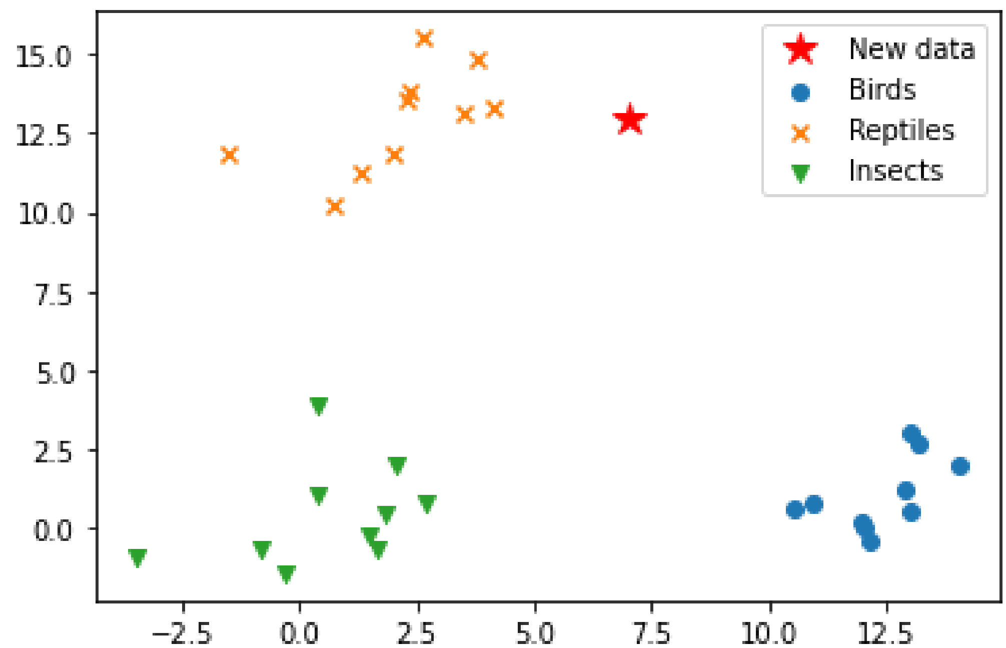


Let's supose that we have some new data we want to recognize

```
new_point = np.random.randint(-2,14,2)

#plot clasees
plt.scatter(Birds[0], Birds[1], marker='o', label='Birds' )
plt.scatter(Reptiles[0], Reptiles[1], marker='x' , label='Reptiles')
plt.scatter(Insects[0], Insects[1], marker='v' , label='Insects');


#plot the new point
plt.plot(new_point[0], new_point[1],   "*r", markersize=12, label='New
data')
```

```
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f1e466fb050>
```

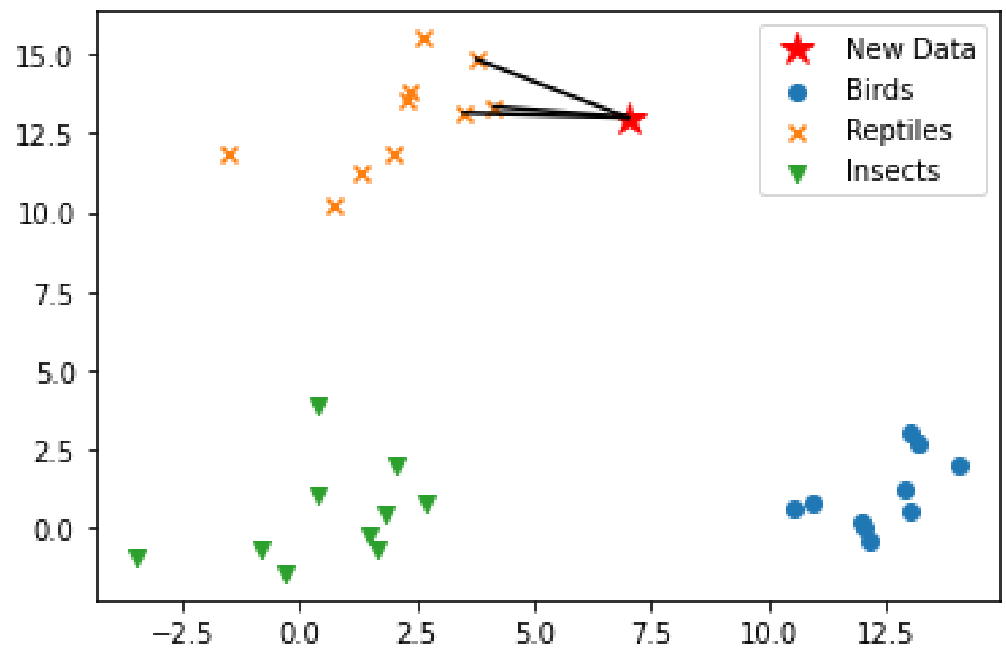


The first step is to calculate the distance between this point all point of the three classes.

```
#merge the data into one single array
K = 3
data =      np.concatenate( (Birds,Reptiles,Insects), axis=1)
distances = np.sum((data - new_point[:,np.newaxis])**2, axis=0)
sorted_points = np.argpartition(distances, K+1)
print(sorted_points)
k_nearest = sorted_points[0:K]


#plot the result
plt.scatter(Birds[0], Birds[1], marker='o', label ='Birds' )
plt.scatter(Reptiles[0], Reptiles[1], marker='x', label ='Reptiles' )
plt.scatter(Insects[0], Insects[1], marker='v' , label ='Insects');

#plot the point
plt.plot(new_point[0], new_point[1],   "*r", markersize=12, label
='New Data')
plt.legend()
#plot lines to the closest points
for i in range(3):
  plt.plot(*zip(new_point, data[:,k_nearest[i]]), c='k')
```

```
#print the predicted class
unique, counts = np.unique(np.uint8(k_nearest/10), return_counts=True)
c = unique[np.argmax(counts)]
print("The class of the new data is: \x1b[1;31m", (['Birds',
'Reptiles', 'Insects'])[c], "\x1b[0m")
```

```
[18 17 11 14 16 19 13 12 10 15 23  3 24  6  1 25  4  2  0  9 20 21 22
8
  7  5 26 27 28 29]
The class of the new data is:  Reptiles

array([1], dtype=uint8)
```



1.6: Data manipulation with Pandas

- Pandas is a package built upon NumPy providing an efficient implementation of a `DataFrame`.
- DataFrames are essentially n-dimensional arrays with row/column labels, and often with heterogeneous types and/or missing data.
- While numpy serves its purpose very well, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.).

```
import pandas as pd
```

Pandas Series

To create a data `Series`

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print("values : ", data.values)
print("indices: ", data.index)
```

```
values :  [0.25 0.5  0.75 1.  ]
indices:  RangeIndex(start=0, stop=4, step=1)
```

data can be accessed as with numpy

```
print("data[1]:     ", data[1])
print("data[1:-1]: ", data[1:-1].values)
```

```
data[1]:      0.5
data[1:-1]:  [0.5  0.75]
```

Numpy Array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print("data[1]:     ", data['a'])
print("data[-1:-3]: ", data['b':'d'].values)
```

```
data[1]:      0.25
data[-1:-3]:  [0.5  0.75 1.  ]
```

- Pandas `Series` are a bit like Python dictionary.
- A dictionary is a structure that maps **arbitrary** keys to a set of arbitrary values,
- `Series` is a structure which maps **typed** keys to a set of typed values.
- This typing makes `Series` more efficient than a Python lists for certain operations.

```
pop_dict = {'Constantine': 938475,
                   'Biskra': 547137,
                   'Ouargla': 311337,
                   'Eloued': 647548,
                   'Adrar': 261258}
population = pd.Series(pop_dict)
print('population[Ouargla]                                   :',
population['Ouargla'])
print("population[['Biskra', 'Eloued', 'Adrar']]             :",
population[['Biskra', 'Eloued', 'Adrar']].values)
print('population[0:2]                                       :',
population[0:2].values)
print('population[(population>500000) & (population<700000)] :',
population[(population>500000) & (population<700000)].index.values)
```

```
population[Ouargla]                                   : 311337
population[['Biskra', 'Eloued', 'Adrar']]             : [547137 647548
261258]
```

```
population[0:2]                                       : [938475
547137]
population[(population>500000) & (population<700000)] : ['Biskra'
'Eloued']
```

Constructing Series: The main instruction for constructing series is `pd.Series(data, index=index)`. Other variantes of this isntruction can be used:

```
pd.Series([5.6,  13.1,  19])

0    2
1    4
2    6
dtype: int64

pd.Series('Hello', index=[10, 20, 50])

10    Hello
20    Hello
50    Hello
dtype: object

pd.Series({2:'X', 5:'Y', 1:'Z'})

2    X
5    Y
1    Z
dtype: object

pd.Series({2:'a', 1:'b', 3:'c', 9:'d'}, index=[3, 2, 9])

3    c
2    a
9    d
dtype: object
```

Indexers: loc, iloc, and ix: if your Series has an explicit integer index, the *indexing* uses explicit while *slicing* use the implicit indexing.

```
data = pd.Series(['a', 'b', 'c', 'd'], index=[2, 3, 5, 6])
print('data[1]   :',data[2])            # explicit index when indexing
print('data[1:3] :', data[2:4].values) # implicit index when slicing

data[1]   : a
data[1:3] : ['c' 'd']
```

`loc` attribute allows indexing and slicing that always references the explicit index:

```
print('data.loc[1]   :',data.loc[2])
print('data.loc[1:3] :', data.loc[2:5].values)
```

```
data.loc[1]   : a
data.loc[1:3] : ['a' 'b' 'c']
```

iloc attribute allows implicit indexing and slicing

```
print('data.iloc[1]   :',data.iloc[1])
print('data.iloc[1:3] :', data.iloc[1:3].values)

data.iloc[1]   : b
data.iloc[1:3] : ['b' 'c']
```

- One guiding principle of Python code is that "explicit is better than implicit."
- Use both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

Pandas DataFrame

- `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names.
- You can think of a `DataFrame` as a sequence of aligned `Series` objects.

Example:

```
import pandas as pd

pop_dict    = {'Constantine': 938475, 'Biskra': 547137, 'Ouargla':
311337,'Eloued': 647548, 'Adrar': 261258}
area_dict   = {'Constantine': 2187, 'Biskra': 9576, 'Ouargla': 194552
     ,'Eloued': 45738, 'Adrar': 254471}
population  = pd.Series(pop_dict)
area        = pd.Series(area_dict)

#constract a Dataframe
wilayas = pd.DataFrame({'population': population, 'area': area})
print("indices: ",wilayas.index)
print("columns: ",wilayas.columns)
wilayas

indices:  Index(['Constantine', 'Biskra', 'Ouargla', 'Eloued',
'Adrar'], dtype='object')
columns:  Index(['population', 'area'], dtype='object')

             population    area
Constantine      938475    2187
Biskra           547137    9576
Ouargla          311337  194552
Eloued           647548   45738
Adrar            261258  254471
```

we can pick one column from the `DataFrame`

```
wilayas['area']
```

```
Constantine      2187
Biskra           9576
Ouargla        194552
Eloued          45738
Adrar          254471
Name: area, dtype: int64
```

In case some keys are missing, **Pandas** will fill them in with NaN

```
pd.DataFrame([{'a': 1, 'b': 2, 'c': 5}, {'b': 3, 'c': 4, 'd':12}])
```

```
     a  b  c     d
0  1.0  2  5   NaN
1  NaN  3  4  12.0
```

We can create `DataFrame`, from any numpy array, with any specified column and index names

```
import numpy as np

pd.DataFrame(np.random.rand(5, 3),
             columns=['income', 'outcome', 'lose'],
             index=['day 1', 'day 2', 'day 3', 'day 4', 'day 5'])
```

```
         income   outcome      lose
day 1  0.175215  0.437155  0.053180
day 2  0.632874  0.782488  0.187896
day 3  0.393341  0.215869  0.123302
day 4  0.163537  0.711649  0.676054
day 5  0.695072  0.392600  0.372434
```

Data Indexing and Slicing

```
import pandas as pd

pop_dict    = {'Constantine': 938475, 'Biskra': 547137, 'Ouargla':
311337,'Eloued': 647548, 'Adrar': 261258}
area_dict   = {'Constantine': 2187, 'Biskra': 9576, 'Ouargla': 194552
     ,'Eloued': 45738, 'Adrar': 254471}

wilayas = pd.DataFrame({'population': pop_dict, 'area': area_dict})
#add a column
wilayas['density'] = wilayas['population']/ wilayas['area']
wilayas
```

```
             population    area     density
Constantine      938475    2187  429.115226
Biskra           547137    9576   57.136278
Ouargla          311337  194552    1.600277
```

```
Eloued            647548   45738   14.157768
Adrar             261258  254471    1.026671
```

One can transpose the full DataFrame to swap rows and columns

```
wilayas.T

              Constantine         Biskra  ...         Eloued
Adrar
population  938475.000000  547137.000000  ...  647548.000000
261258.000000
area          2187.000000    9576.000000  ...   45738.000000
254471.000000
density        429.115226      57.136278  ...      14.157768
1.026671

[3 rows x 5 columns]
```

Indexing & slicing

```
print('wilayas.values[0]    :', wilayas.values[0])
print("\nwilayas['density']   :\n", wilayas['density'])
print("\nwilayas.iloc[0:3, 1:]   :\n", wilayas.iloc[0:3, 1:]) #
slicing the array as if it is a simple NumPy array using loc
print("\nwilayas.loc['Ouargla':, :'area']   :\n",
wilayas.loc['Ouargla':, :'area']) # slicing using loc
print("\nwilayas.loc['Ouargla':, :'area']   :\n",
      wilayas.loc[wilayas.density > 50, ['population', 'area']])
#fancy indexing

wilayas.values[0]    : [9.38475000e+05 2.18700000e+03 4.29115226e+02]

wilayas['density']   :
 Constantine    429.115226
Biskra          57.136278
Ouargla          1.600277
Eloued          14.157768
Adrar            1.026671
Name: density, dtype: float64

wilayas.iloc[0:3, 1:]   :
                area     density
Constantine    2187  429.115226
Biskra         9576   57.136278
Ouargla      194552    1.600277

wilayas.loc['Ouargla':, :'area']   :
          population    area
Ouargla      311337  194552
```

```
Eloued        647548   45738
Adrar         261258  254471

wilayas.loc['Ouargla':, :'area']   :
              population  area
Constantine      938475  2187
Biskra           547137  9576
```

- Any of these indexing/slicing approaches may also be used to set or modify values;
- The `ix` indexer allows a hybrid of these two approaches. Try it yourselves.

Operations

- Pandas inherits much functionalities from NumPy

```
import numpy as np

ds = pd.Series({'a':10, 'b':2, 'c':5, 'd':7})
df = pd.DataFrame({'x':ds, 'y':(ds + ds)**2/3})
np.exp(data_1)
np.sin(df * np.pi / 4)

32.66666666666667
```

Operating on Null Values:

```
import numpy as np
data = pd.Series([1, np.nan, 'hello', None])
print(data.isnull().values)
print(data[data.notnull()].values)
data.dropna()

[False  True False  True]
[1 'hello']

0        1
2    hello
dtype: object
```

Hierarchical Indexing

- Often it is useful to go beyond this and store higher-dimensional data with more than one or two keys.
- a common pattern in practice is to make use of hierarchical indexing to incorporate multiple index levels within a single index.

```
import pandas as pd

pop_dict    = {('Constantine', 2010): 538475, ('Constantine', 2020): 
938475,
               ('Biskra', 2010): 247137,  ('Biskra', 2020): 247137,
```

```
               ('Ouargla', 2010): 111337, ('Ouargla', 2020): 311337}
# area_dict   = {'Constantine': 2187, 'Biskra': 9576, 'Ouargla':
194552     }

wilayas = pd.Series(pop_dict)
wilayas.index.names=('wilaya', 'year')#Sometimes it is convenient to
name the levels of the MultiIndex
print("wilayas.loc[[('Biskra', 2010),('Constantine', 2020)]]: \n",
wilayas.loc[[('Biskra', 2010),('Constantine', 2020)]], '\n')
wilayas
```

```
wilayas.loc[[('Biskra', 2010),('Constantine', 2020)]]:
 wilaya       year
Biskra       2010    247137
Constantine  2020    938475
dtype: int64


wilaya       year
Constantine  2010    538475
             2020    938475
Biskra       2010    247137
             2020    247137
Ouargla      2010    111337
             2020    311337
dtype: int64
```

How to select all populations in 2010?

```
print(wilayas.loc[[i for i in wilayas.index if i[1] == 2010]],'\n')
print(wilayas[:,2010],'\n')
```

```
Constantine  2010    538475
Biskra       2010    247137
Ouargla      2010    111337
dtype: int64

Constantine    538475
Biskra         247137
Ouargla        111337
dtype: int64
```

The `unstack()` method will quickly convert a multiply indexed `Series` into a conventionally indexed `DataFrame`. Naturally, the `stack()` method provides the opposite operation

```
wilayas.unstack()
```

```
               2010    2020
Biskra       247137  247137
```

```
Constantine  538475  938475
Ouargla      111337  311337
```

why would we would bother with hierarchical indexing at all?

- to use multi-indexing to represent two-dimensional data within a one-dimensional `Series`,
- we can also use it to represent data of three or more dimensions in a `Series` or `DataFrame`.
- Each extra level in a multi-index represents an extra dimension of data;

```
df = pd.DataFrame({'total': wilayas, 'under 10': wilayas.values//5})
df
```

```
                   total  under 10
Constantine 2010  538475    107695
            2020  938475    187695
Biskra      2010  247137     49427
            2020  247137     49427
Ouargla     2010  111337     22267
            2020  311337     62267
```

```
(df['under 10']/df['total']).unstack()
```

```
                 2010      2020
Biskra       0.199998  0.199998
Constantine  0.200000  0.200000
Ouargla      0.199996  0.199999
```

Combining Datasets: Concat and Append

Some of the most interesting studies of data come from combining different data sources

```
import numpy as np
import pandas as pd


#with numpy
x = [[1, 2], [3, 4]]
print('np.concatenate([x, x], axis=1):\n', np.concatenate([x, x],
axis=1))

#with pandas Series
ser1 = pd.Series(['A', 'B'], index=[1, 2])
ser2 = pd.Series(['D', 'E'], index=[4, 5])
print('\npd.concat([ser1, ser2]):\n', pd.concat([ser1, ser2]))

#with pandas Dataframes
df1 = pd.DataFrame({(1,2), (3,4)}, index=[0,1], columns=('C1', 'C2'))
df2 = pd.DataFrame({(11,22), (33,44)}, index=[0,1], columns=('C1',
```

```
'C2'))
print('\npd.concat([df1, df2]) :\n', pd.concat([df1, df2]))#concate
over rows
print('\npd.concat([df1, df2], axis=1) :\n', pd.concat([df1, df2],
axis=1)) #concate over columns
print('\npd.concat([df1, df2]) :\n', pd.concat([df1, df2],
keys=('df1', 'df2'))) #specify multindex for Hierarchical Indexing

np.concatenate([x, x], axis=1):
 [[1 2 1 2]
 [3 4 3 4]]

pd.concat([ser1, ser2]):
 1    A
2    B
4    D
5    E
dtype: object

pd.concat([df1, df2]) :
    C1  C2
0   1   2
1   3   4
0  11  22
1  33  44

pd.concat([df1, df2], axis=1) :
    C1  C2  C1  C2
0   1   2  11  22
1   3   4  33  44

pd.concat([df1, df2]) :
        C1  C2
df1 0   1   2
    1   3   4
df2 0  11  22
    1  33  44
```

Concatenation with joins

- we were mainly concatenating DataFrames with shared column names. *In practice, data from different sources might have different sets of column names
- `pd.concat` offers several options in this case.

```
df1 = pd.DataFrame({(1,2), (3,4)}, index=[0,1], columns=('C1', 'C2'))
df2 = pd.DataFrame({(1,2), (3,4)}, index=[0,1], columns=('C1', 'C3'))

print('pd.concat([df1, df2]):\n', pd.concat([df1, df2])) #inner
concatenation
```

```
print("\npd.concat([df1, df2], join='inner'):\n", pd.concat([df1,
df2], join='inner')) #inner concatenation (union)

pd.concat([df1, df2]):
    C1   C2   C3
0   1  2.0  NaN
1   3  4.0  NaN
0   1  NaN  2.0
1   3  NaN  4.0

pd.concat([df1, df2], join='inner'):
    C1
0   1
1   3
0   1
1   3
```

GroupBy: Split, Apply, Combine

- Simple aggregations can give you a flavor of your dataset
- Often we would prefer to aggregate conditionally on some label or index
- This is implemented in the so-called `groupby` operation.

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df

  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5

df.groupby('key').mean()
df.groupby('key')['data'].mean()

key
A    1.5
B    2.5
C    3.5
Name: data, dtype: float64
```

`GroupBy` object supports direct iteration over the groups

```
import pandas
import seaborn as sns
```

```
penguins = sns.load_dataset('penguins')
penguins.head()
```

```
  species     island  bill_length_mm  bill_depth_mm  flipper_length_mm
\
0  Adelie  Torgersen            39.1           18.7              181.0

1  Adelie  Torgersen            39.5           17.4              186.0

2  Adelie  Torgersen            40.3           18.0              195.0

3  Adelie  Torgersen             NaN            NaN                NaN

4  Adelie  Torgersen            36.7           19.3              193.0


   body_mass_g     sex
0       3750.0    Male
1       3800.0  Female
2       3250.0  Female
3          NaN     NaN
4       3450.0  Female
```

```
penguins.groupby('island').mean()
```

```
           bill_length_mm  bill_depth_mm  flipper_length_mm
body_mass_g
island

Biscoe          45.257485      15.874850         209.706587
4716.017964
Dream           44.167742      18.344355         193.072581
3712.903226
Torgersen       38.950980      18.429412         191.196078
3706.372549
```

```
for (method, group) in penguins.groupby('island'):
    print("{0:30s} shape={1}".format(method, group.shape))
```

```
Biscoe                         shape=(168, 7)
Dream                          shape=(124, 7)
Torgersen                      shape=(52, 7)
```

```
penguins.groupby('island')['body_mass_g'].describe()
```

```
           count         mean         std  ...     50%      75%
max
island                                     ...

Biscoe     167.0  4716.017964  782.855743  ...  4775.0  5325.00
6300.0
Dream      124.0  3712.903226  416.644112  ...  3687.5  3956.25
```

```
4800.0
Torgersen   51.0  3706.372549  445.107940  ...  3700.0  4000.00
4700.0

[3 rows x 8 columns]
```

`aggregate()` method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once.

```
penguins.groupby('island')['body_mass_g'].aggregate(['count', min,
np.median, max, np.var])

           count     min  median     max            var
island
Biscoe       167  2850.0  4775.0  6300.0  612863.114133
Dream        124  2700.0  3687.5  4800.0  173592.315762
Torgersen     51  2900.0  3700.0  4700.0  198121.078431

penguins.groupby('island').aggregate({'max_body_mass_g': max,
'min_bill_depth_mm':min })

           body_mass_g  bill_depth_mm
island
Biscoe          6300.0           13.1
Dream           4800.0           15.5
Torgersen       4700.0           15.9
```

transformation can return some transformed version of the full data to recombine.

```
penguins.groupby('island').transform(lambda x: x/ x.mean()).head()

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1:
FutureWarning: Dropping invalid columns in DataFrameGroupBy.transform
is deprecated. In a future version, a TypeError will be raised. Before
calling .transform, select only columns which should be valid for the
transforming function.
  """Entry point for launching an IPython kernel.

   bill_length_mm  bill_depth_mm  flipper_length_mm  body_mass_g
0        1.003826       1.014682           0.946672     1.011771
1        1.014095       0.944143           0.972823     1.025261
2        1.034634       0.976700           1.019895     0.876868
3             NaN            NaN                NaN          NaN
4        0.942210       1.047239           1.009435     0.930829
```

The `apply()` method lets you apply an arbitrary function to the group results.

```
def norm_bodymass_by_sum_billdepth(x):
    # x is a DataFrame of group values
```

```
    x['body_mass_g'] /= x['bill_depth_mm'].sum()**2
    return x
penguins.groupby('island').apply(norm_bodymass_by_sum_billdepth).head(
)
```

```
  species     island  bill_length_mm  bill_depth_mm  flipper_length_mm
\
0  Adelie  Torgersen            39.1           18.7              181.0

1  Adelie  Torgersen            39.5           17.4              186.0

2  Adelie  Torgersen            40.3           18.0              195.0

3  Adelie  Torgersen             NaN            NaN                NaN

4  Adelie  Torgersen            36.7           19.3              193.0


   body_mass_g     sex
0     0.004245    Male
1     0.004302  Female
2     0.003679  Female
3          NaN     NaN
4     0.003905  Female
```

we can perform multidimentional grouping

```
penguins.groupby(['sex', 'island'])['body_mass_g'].mean().unstack()
```

```
island       Biscoe        Dream    Torgersen
sex
Female  4319.375000  3446.311475  3395.833333
Male    5104.518072  3987.096774  4034.782609
```

The same operation can be executed using `pivot_table`

```
penguins.pivot_table('body_mass_g', 'sex', 'island')
```

```
island       Biscoe        Dream    Torgersen
sex
Female  4319.375000  3446.311475  3395.833333
Male    5104.518072  3987.096774  4034.782609
```

we can build even more indices

```
penguins.pivot_table('body_mass_g', ['sex', 'species'], 'island')
```

```
island                Biscoe        Dream    Torgersen
sex    species
Female Adelie    3369.318182  3344.444444  3395.833333
```

```
       Chinstrap          NaN  3527.205882          NaN
       Gentoo     4679.741379          NaN          NaN
Male   Adelie     4050.000000  4045.535714  4034.782609
       Chinstrap          NaN  3938.970588          NaN
       Gentoo     5484.836066          NaN          NaN
```

even more column indices

```
import pandas as pd

mass = pd.qcut(penguins['body_mass_g'], 3)
penguins.pivot_table('body_mass_g', ['sex', 'species'], [mass,
'island'])

body_mass_g      (2699.999, 3700.0]              ... (4550.0, 6300.0]

island                       Biscoe        Dream  ...            Dream
Torgersen
sex    species                                    ...

Female Adelie             3207.8125  3344.444444  ...              NaN
NaN
       Chinstrap                NaN  3446.428571  ...              NaN
NaN
       Gentoo                   NaN          NaN  ...              NaN
NaN
Male   Adelie             3600.0000  3525.000000  ...           4625.0
4687.5
       Chinstrap                NaN  3487.500000  ...           4800.0
NaN
       Gentoo                   NaN          NaN  ...              NaN
NaN

[6 rows x 9 columns]
```

`mean` is the default function for pivoting. However, aggregation method can be specified or even personalized.

```
penguins.pivot_table('body_mass_g', ['sex', 'species'], 'island',
aggfunc={sum})

                       sum
island              Biscoe     Dream Torgersen
sex    species
Female Adelie      74125.0   90300.0   81500.0
       Chinstrap       NaN  119925.0       NaN
       Gentoo     271425.0       NaN       NaN
Male   Adelie      89100.0  113275.0   92800.0
```

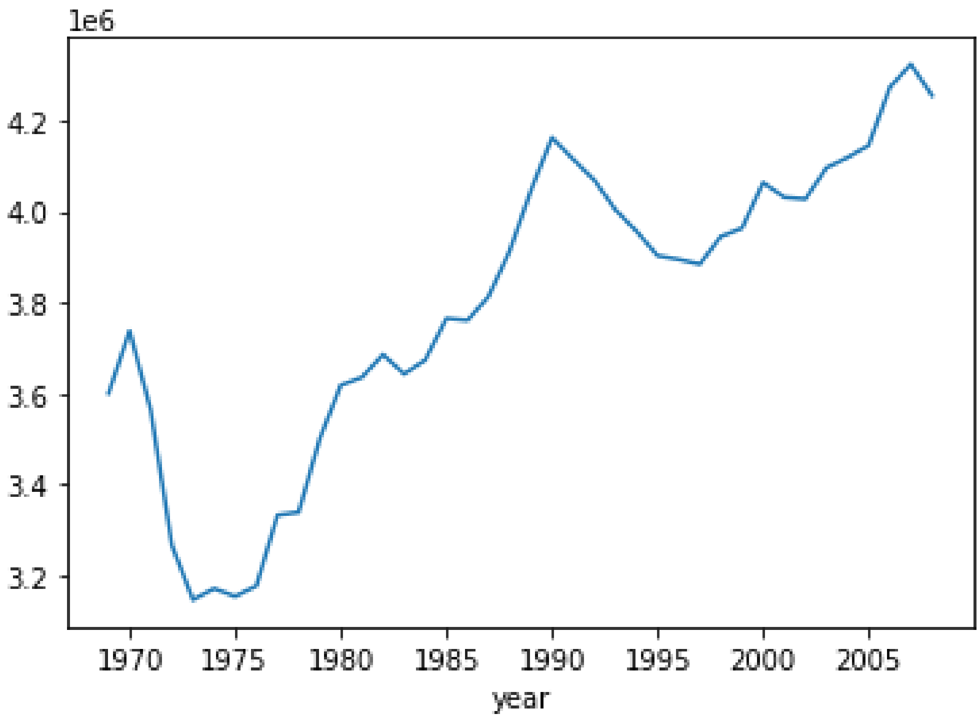
```
       Chinstrap       NaN  133925.0       NaN
       Gentoo     334575.0       NaN       NaN
```

Example: Let's analyse the following data

```
!wget https://raw.githubusercontent.com/jakevdp/data-
CDCbirths/master/births.csv

--2022-11-02 11:34:39--
https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births
.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.110.133, 185.199.111.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|
185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 264648 (258K) [text/plain]
Saving to: ‘births.csv’

births.csv          100%[===================>] 258.45K  --.-KB/s    in
0.03s

2022-11-02 11:34:39 (7.65 MB/s) - ‘births.csv’ saved [264648/264648]


import pandas as pd

data = pd.read_csv('births.csv')
#fill missings
data.fillna(method='ffill')
print("original data shape:", data.shape)

data.head()

original data shape: (15547, 5)

   year  month  day gender  births
0  1969      1  1.0      F    4046
1  1969      1  1.0      M    4440
2  1969      1  2.0      F    4454
3  1969      1  2.0      M    4548
4  1969      1  3.0      F    4548
```

aggragte data based on number of births/year

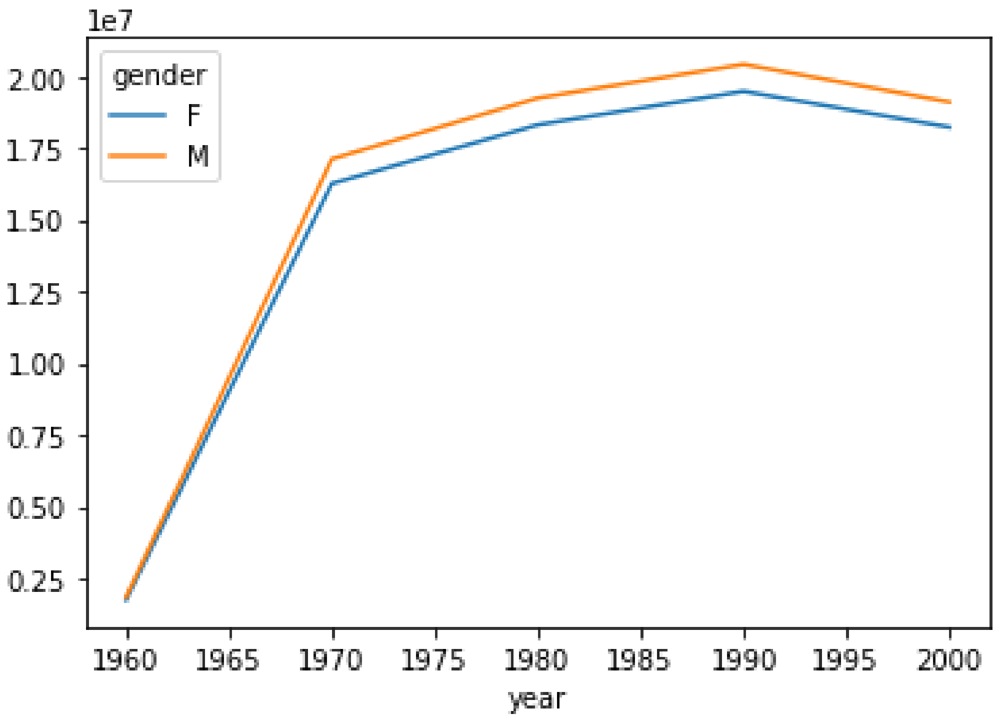
```
sums = data.groupby('year')['births'].sum().plot()
sums

<matplotlib.axes._subplots.AxesSubplot at 0x7f9b85a11790>
```



what are the avrages of births by gender in 5 equal periods?

```
decades = 10*(data['year']//10)
data.pivot_table('births', index=decades, columns='gender',
aggfunc='sum').plot()
data.pivot_table('births', 'gender', decades, aggfunc='sum')
```

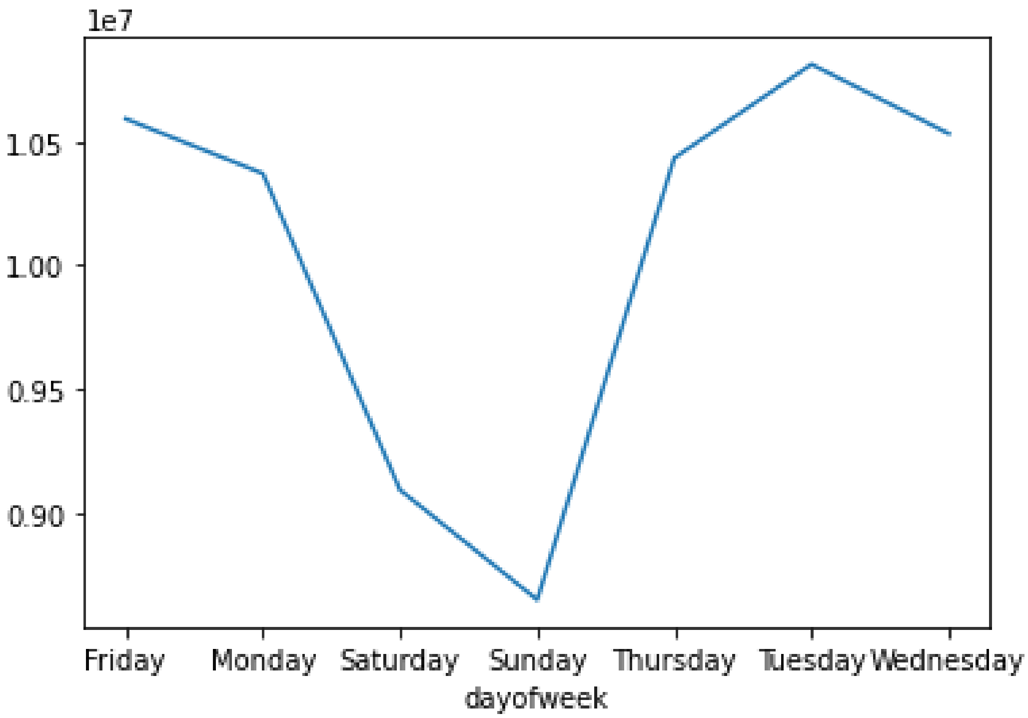
year	1960	1970	1980	1990	2000
gender					
F	1753634	16263075	18310351	19479454	18229309
M	1846572	17121550	19243452	20420553	19106428



Let's see the births quantity per week days

```
import pandas as pd
data['dayofweek'] = pd.to_datetime(10000 * data.year +
                                   100 * data.month +
                                   data.day, format='%Y%m%d',
errors='coerce').dt.day_name()

data.groupby('dayofweek')['births'].sum().plot
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3feab86710>
```


1e7
1.05
1.00
0.95
0.90
Friday
Monday
Saturday
Sunday
Thursday
Tuesday
Wednesday
dayofweek