Numpy

When it comes to handling and processing data and applying mathematical operations on groups of numbers , we need to go beyond simple lists for storage and related processes . Numpy gives us arrays as a data structures to store multiple data points in one place and then apply various operations using direct functions , present in the packages. [and its blazing fast too].

```
A small note here : although you might not use numpy directly a ton in your initial days, you might find packages written on top of numpy [such as Pandas] which do the heavy lifting under the hood; its still good to know functionalities to appreciate what is being done under the hood and at times you might be required to know how numpy functions to extend the functionalities. This will also be really useful when you study deep learning and work with reshaping or subsetting arrays across layers.
```

First thing that we need to learn before we start using various operations , is to actually create a numpy array . lets start there. [Use accompanying notebook to code along and experiment]

Import numpy

we are going to import numpy as alias `np` and will refer to it as `np` for the rest of this unit

```
import numpy as np
```

Creating a numpy array

1-D array from a list

```
np.array([1,22,33,43,98,7,-10])
```

outcome : `array([  1,  22,  33,  43,  98,   7, -10])`

2-D array from a list

we can create higher dimension arrays also , but for all usual data needs 2-D arrays will suffice and we will also limit our discussion to just that . A nested list of list gets converted to a 2-D array. We can convert a flat list also to higher dimension using reshaping , which we will learn about later .

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

outcome :

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

standard arrays

There are standard functions in numpy to create arrays with constants and and in whatever shape we want them in

creating array of ones

```
np.ones((3,4))
```

Outcome :

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

first dimension here `3` applies to , number of nested arrays and second dimension `4` applies to number of elements in each of them

```
np.zeros((2,3))
```

Outcome :

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.full((2,2),7)
```

outcome :

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

Identity matrix is a square diagonal matrix with all elements in the diagonal being 1 and rest of the elements being 0.

```
np.eye(3)
```

Outcome :

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

random arrays

you can also create arrays of any shape/dimensions filled with random values with numpy by using subclass `random` in the package. here are few examples :

arrays filled with random values between 0 to 1 .

```
np.random.random((2,3))
```

Outcome :

```
array([[0.38854667, 0.02783852, 0.95747752],
       [0.52507296, 0.33582279, 0.78034549]])
```

these are random value. when you execute the same code , you will get some different values . if you want these to be reproducible , make use of class `RandomState` . it generates a random number generator with reproducible results .

```
r = np.random.RandomState(1234)
r.random((2,3))
```

Outcome :

```
array([[0.19151945, 0.62210877, 0.43772774],
       [0.78535858, 0.77997581, 0.27259261]])
```

if you also run the above bit of code , you will exact same numbers as you see here . But if you run the generator in a new cell, state will reset and you will get different values .

there are tons of ways in which you can generate random values

- random integers : `np.random.randint`
- random values from a fixed set of values [a list] : `np.random.choice`
- random values from distributions : `np.random.beta` , `np.random.normal` etc

this list is not exhaustive . you can find a more complete reference for generating random data with numpy here : https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.html

Indexing and slicing

Accessing single elements in a 1-D array works just like it did in a list .

indexing starts at the beginning with `0` and you can access elements from the back with starting index as `-1`

```
a=np.array([ 3, 90, 84, 98, 68, 94, 12, 21, 92, 11])
a
```

```
array([ 3, 90, 84, 98, 68, 94, 12, 21, 92, 11])
```

```
a[2]
```

```
84
```

```
a[:4]
```

```
array([ 3, 90, 84, 98])
```

leaving the start empty implies that we start at the very beginning.

```
a[3:]
```

```
array([98, 68, 94, 12, 21, 92, 11])
```

Leaving the end empty implies that we go until the end

```
a[4:-1]
```

```
array([68, 94, 12, 21, 92])
```

-ve indices imply indexing from the end of the array .

```
a[-1:4:-1]
```

```
array([11, 92, 21, 12, 94])
```

the third option here , just like lists, implies the step size . A negative step implies the array traversal is happening backwards , starting from the right side instead of left ,as usually is the case .

here is another example with step `2` traversing from left to right [because of positive step] . If the step size is not mentioned explicitly , default step of `1` from left to right is used .

```
a[2:8:2]
```

```
array([84, 68, 12])
```

takeaway from this so far is that indexing and slicing works just like lists did as far as 1-D arrays are concerned . Nothing out of the blue happens for higher dimensional arrays as such . All that you need to understand for the higher dimensions is that indexing sequence for multiple dimensions starts from the outermost brackets .

Lets understand what i mean with an example .

```
a=np.array([[11,22,33],[-10,1,20],[99,56,4]])
```

`a` here is a 2-D array .

```
a
```

```
array([[ 11,  22,  33],
       [-10,   1,  20],
       [ 99,  56,   4]])
```


we can access elements of a with 2 indices , one for each dimension .

to access for example `56` , we'd need to do this

```
a[2,1]
```

```
56
```

here the first index that we have used `2` , goes one level inside the outermost bracket and picks which individual array i am referring to .

so `0` index refers to `[ 11,  22,  33]`

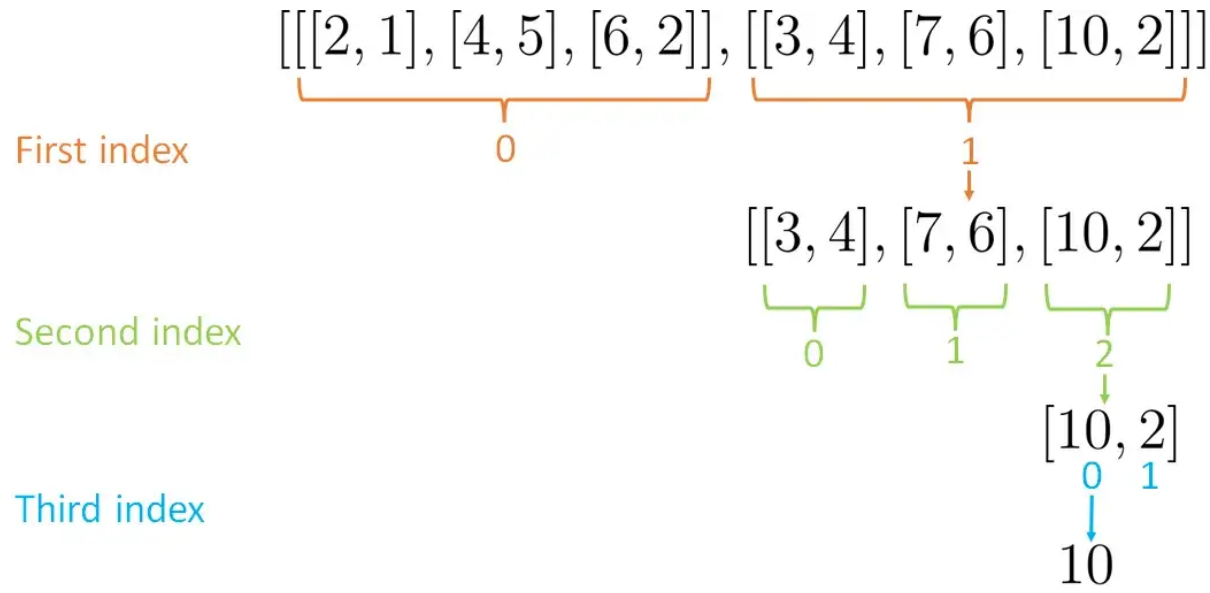
`1` refers to `[-10,   1,  20]`

and `2` refers to `[ 99,  56,   4]`

to go further in these arrays , we'll need to make use of the second index .

we passed second index as `1` , which eventually takes us to index `1` element of `[ 99,  56,   4]` , which happens to be `56` .

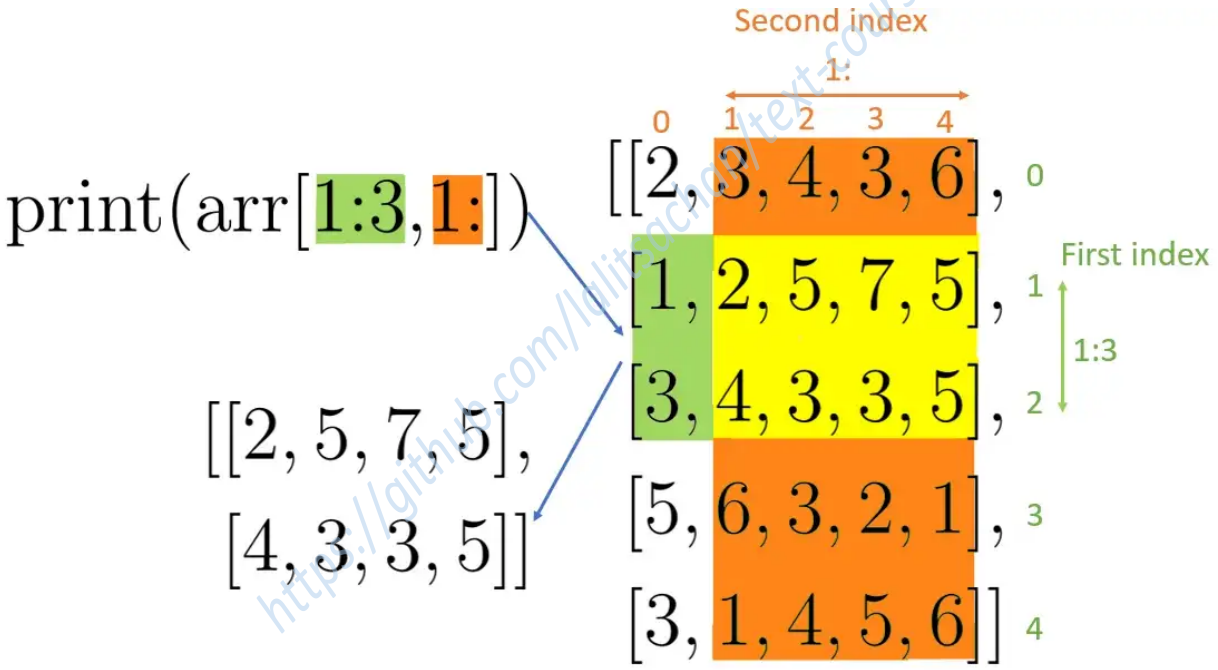
for more clarity , refer to this image of a 3-D array.



source : https://towardsdatascience.com/slicing-numpy-arrays-like-a-ninja-e4910670ceb0

element `10` here will be accessed with 3 indices as we go more and more deeper . `[1,2,0]`

Slicing for 2-D arrays works same as the 1-D arrays , just that now we'll be slicing across all present dimensions . here is another example :



Along the first dimension here there are 5 arrays , and then in the second dimension , each array contains 5 elements . so `1:3` refers to slicing index `1` and `2` arrays . and then second dimension index `1:` , implies getting all the elements in those arrays starting at index `1` and going till the end , as you can see in the figure .

A good way to get used to this is to simply create multi dimension arrays , pass indices to access and slice and try to mak sense of the outcome that you get.

In the next section we will look at how we can create higher dimensions arrays from lower dimensions arrays and vice versa . The exercise essentially will be about reshaping arrays into different shapes than they were originally in . lets see what that is about.

Reshaping

in this section we will start with a base array and reshape this in various ways and learn from the outcome . Remember that if we are changing shapes , number of expected values should be matching , otherwise you will get an error .

```
a=np.array([[11,22,33,44],[-10,1,20,71],[99,56,4,-2]])
a.shape
```

(3,4)

current shape of the array can be accessed through `a.shape` attribute .

```
a.reshape((2,6))
```

```
array([[ 11,  22,  33,  44, -10,   1],
       [ 20,  71,  99,  56,   4,  -2]])
```

from the original shape of `(3,4)` , the elements has been arrange into shape `(2,6)` . note that number elements do not change. Same values get rearranged into a new shape. Note that values are arranged across the outermost dimension here .

you can convert array of any dimension to 1-D array by using function `flatten`

```
b=a.flatten()
b
```

```
array([ 11,  22,  33,  44, -10,   1,  20,  71,  99,  56,   4,  -2])
```

when going from 1-D array to 2-D array , elements are filled along the outermost dimension sequentially .

```
b.reshape((3,4))
```

```
array([[ 11,  22,  33,  44],
       [-10,   1,  20,  71],
       [ 99,  56,   4,  -2]])
```

if you do try to reshape into an array which doesn't have equal number of elements you get an error

```
a.reshape((3,2))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [56], in <cell line: 1>()
----> 1 a.reshape((3,2))

ValueError: cannot reshape array of size 12 into shape (3,2)
```

lets say you dont want to sit down and count the number of elements in the original array so that you can manage equal number of elements.

you can leave the last dimension as `-1` , which implies that it needs to inferred or calculated such that number of elements match. here are a couple examples

```
a.reshape((4,-1))
```

```
array([[ 11,  22,  33],
       [ 44, -10,   1],
       [ 20,  71,  99],
       [ 56,   4,  -2]])
```

```
a.reshape((2,2,-1))
```

```
array([[[ 11,  22,  33],
        [ 44, -10,   1]],

       [[ 20,  71,  99],
        [ 56,   4,  -2]]])
```

Standard Mathematical Operations

One big reason why numpy is popular is its ability to do mathematical operations over multiple values at once . This is not possible with default package `math` in python . lets try and see what happens.

```
import math as m
m.log(3)
```

```
1.0986122886681098
```

so the function `log` from package `math` works alright on scalars or single numbers . However if you try to apply on an array , it throws an error .

```
m.log(b)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [63], in <cell line: 1>()
----> 1 m.log(b)

TypeError: only size-1 arrays can be converted to Python scalars
```

numpy has all the functions available in package math with the property that it can be applied on arrays. without any extra code from the user , it simply applies the function on all the members of the array . [without you having to write a for loop for example]

```
np.log(b)
```

```
array([2.39789527, 3.09104245, 3.49650756, 3.78418963,        nan,
       0.        , 2.99573227, 4.26267988, 4.59511985, 4.02535169,
       1.38629436,        nan])
```

some of the values are `nan` because of negative values in the original array.

you can try this with other mathematical functions also like : `sin` , `cos` , `abs` etc

this behaviour of a function getting applied to all the members extends to exponentiation as well . if you apply some power to an array, result is that power being applied to all the elements in the array .

```
b**2
```

```
array([ 121,  484, 1089, 1936,  100,    1,  400, 5041, 9801, 3136,   16,
        4])
```

when you any kind of mathematical operation between two arrays , result is simply another array, where each element is the result of same operation being done between corresponding elements of the original array. For example :

```
np.array([3,4,5,6])+np.array([2,10,34,-10])
```

```
array([ 5, 14, 39, -4])
```

we tried to add two arrays . what actually happened that individual elements got added and result is an array . this behaviour extends to all mathematical operations .

```
np.array([3,4,5,6])*np.array([2,10,34,-10])
```

```
array([  6,  40, 170, -60])
```

other operations that you can try : `-` , `/` , `**` .

One convenient theme in these examples of operations between arrays is that the dimensions match . So every element in one array has a corresponding element in the other arrays between which the proposed operation can be done .

However , what happens when the dimensions do not match ? lets learn about that in the next section.

Broadcasting

Consider this small example :

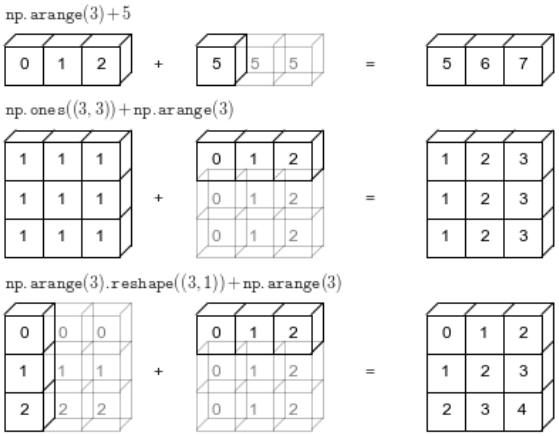
```
a=np.array([3,7,10])
a+5
```

```
array([8,12,15])
```

clearly `a` is a 1-D array with 3 elements in it . 5 is a 1-D array with only one element in it . This is not the case where there is an element corresponding to each element of `a` in another array .

whats going on here is that the smaller array's element , get broadcasted to match dimension of another array so that then each element of the bigger array ends up having a corresponding element with which, then the operation can be carried in the usual manner.

this image is a visual way to look at what broadcasting actually does .



Broadcasting is not a natural mathematical operation , however in terms of data analysis , this behaviour enables some operations which without this facility would have required inefficient for loops. Whereas these internal vectorised operations enabled by broadcasting are much much faster than the same .

One such example can be centring of the data , where we need to subtract mean of the columns from each respective column. In that case we would want to broadcast a 1-D array [containing means of all columns] across a 2-D array [containing data in complete columns] for the operation of subtraction.

There are 3 basic rules that numpy follows for broadcasting :

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

```
M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:

```
M + a
```

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

Broadcasting example 2

Let's take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`
- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Because the result matches, these shapes are compatible. We can see this here:

```
a + b
```

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

Broadcasting example 3

Now let's take a look at an example in which the two arrays are not compatible:

```
M = np.ones((3, 2))
a = np.arange(3)
```

This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
M + a
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-9e16e9f98da6> in <module>()
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding `a`'s shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity.

Linear Algebra Operations

We are assuming in this section , you'd know little about linear algebra and be familiar with things like matrix inverse and eigen values . If you are not , you can skip this section for now and come back to when the discussion on mathematical idea happens else where. Lets get started .

Matrix like operations enabled by numpy went on to become backbone of many python packages implementing various machine learning, deep learning and optimisation algorithms [such as `tensorflow` and `pytorch`] . Lets look at few examples for selected basic linear algebra tasks with numpy .

In our examples here , vector , matrices will be referring to 1-D and 2-D numpy arrays respectively.

dot product between two vectors

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
np.dot(a,b)
```

mathematically dot product is simply sum of element wise multiplication between the vectors . so `32` here is nothing but `1*4+2*5+3*6` .

You can also get cross product with `np.cross` , but that operation is much more common in physics rather than in ML .

matrix product

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

```
((2, 2), (2, 3))
```

Note that for matrix product to be possible , dimensions of both the matrices need to be aligned .

```
np.dot(a,b)
```

```
array([[21, 24, 27],
       [47, 54, 61]])
```

if the dimensions are not aligned , you will get an error .

```
np.dot(b,a)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [70], in <cell line: 1>()
----> 1 np.dot(b,a)

File <__array_function__ internals>:5, in dot(*args, **kwargs)

ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

Transpose , Diagonal, Trace

Transpose of a matrix is where rows become columns and columns become rows . Essentially the dimensions get switched .

```
a
```

```
array([[1, 2],
       [3, 4]])
```

```
a.T
```

```
array([[1, 3],
       [2, 4]])
```

Diagonal of matrix [square] , is array containing elements in the diagonal . For a non-square matrix like `b` , diagonal gives the outcome of primary diagonal starting at left corner of the matrix.

```
print(a)
a.diagonal()
```

```
[[1 2]
[3 4]]
```

```
array([1, 4])
```

```
print(b)
b.diagonal()
```

```
[[ 5  6  7]
[ 8  9 10]]
```

```
array([5, 9])
```

trace of a matrix is simply sum of all the elements in the diagonal .

```
a.trace(),b.trace()
```

```
(5, 14)
```

Det , Rank, Inverse and Identity Matrix

Rank of a matrix is defined as max number of independent rows/cols in the matrix . If any column in the matrix is a perfect linear combination of one or more columns in matrix for example , its determinant will be zero and matrix will not be of full rank [its rank will be less than its square dimension] and it will not be invertible .

Inverse of a matrix , when defined is another matrix with which the matrix multiplication yields an identity matrix. Identity matrix is a diagonal matrix of the same dimension with all the diagonal elements being `1` . If you multiply a matrix with an identity matrix , you get the same matrix back .

consider following matrix which has column as linear combination of other two columns .

```
a = np.arange(1, 10).reshape((3,3))
a
```

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

Here $C3 \rightarrow 2 * C2 - C1$.

```
np.linalg.det(a)
```

```
6.66133814775094e-16
```

This is essentially `0` .

```
np.linalg.matrix_rank(a)
```

```
2
```

if you do try to take an inverse , you will get a matrix with very inflated values and if you multiply it back with `a` it will not yield an identity matrix.

```
b=np.linalg.inv(a)
b
```

```
array([[-4.50359963e+15,  9.00719925e+15, -4.50359963e+15],
       [ 9.00719925e+15, -1.80143985e+16,  9.00719925e+15],
       [-4.50359963e+15,  9.00719925e+15, -4.50359963e+15]])
```

```
np.dot(a,b)
```

```
array([[ 0.,  0.,  0.],
       [-4.,  0., -4.],
       [ 0.,  0.,  0.]])
```

Now lets do the same operation with a matrix of full rank .

```
a=np.array([2,3,4,1,0,1,5,-2,7]).reshape((3,3))
np.linalg.matrix_rank(a)
```

```
3
```

```
b=np.linalg.inv(a)
b
```

```
array([[-0.2,  2.9, -0.3],
       [ 0.2,  0.6, -0.2],
       [ 0.2, -1.9,  0.3]])
```

```
np.dot(a,b)
```

```
array([[ 1.0000000e+00, -8.8817842e-16,  0.0000000e+00],
       [ 0.0000000e+00,  1.0000000e+00,  0.0000000e+00],
       [ 0.0000000e+00,  0.0000000e+00,  1.0000000e+00]])
```

you can see here that the result is an identity matrix with all the diagonal elements being 1 and all the other elements being practically 0 [the non-zero value that you see is very close to zero and is a result of some computation overflow resulting from precision issues]

Eigen Values

A matrix in itself represents a fixed translation and rotation operation on a vector. If a matrix is multiplied with a vector , the outcome is another vector which has now been rotated and elongated or compressed .

Now the eigenvectors are the vectors for any given matrix which do not get rotated when multiplied with that matrix. Mathematically saying x is said to be an eigen vector of matrix A if

$Ax = \lambda x$

where λ is called eigenvalue . It simply implies that if A multiplies with x , the outcome will be a simply multiple of x . Lets see an example with numpy.

```
a
```

```
array([[ 2,  3,  4],
       [ 1,  0,  1],
       [ 5, -2,  7]])
```

```
np.linalg.eig(a)
```

Eigen values : array([9.61961496, -1.37541685, 0.75580189])

Eigen vectors :

```
array([[-0.50310182, -0.8182817 , -0.78151033],
       [-0.14093699,  0.20429773, -0.35756491],
       [-0.8526578 ,  0.53728716,  0.5112621 ]])
```

Lets try if the expression holds true for the results

```
e1,e2,e3=np.array([[-0.50310182, -0.8182817 , -0.78151033],
        [-0.14093699,  0.20429773, -0.35756491],
        [-0.8526578 ,  0.53728716,  0.5112621 ]]).T
```

```
lambda1,lambda2,lambda3=np.array([ 9.61961496, -1.37541685,  0.75580189])
```

Ax >

```
np.dot(a,e1)
```

```
array([-4.83964581, -1.35575962, -8.20223972])
```

λx >

```
lambda1*e1
```

```
array([-4.83964579, -1.35575958, -8.20223973])
```

solving system of linear equations

A system of linear equations can be written in terms of matrices and then solved with linear algebra tools available in numpy . consider this system of equations .

$$x_1 + x_2 - x_3 + 4x_4 = 9$$
$$2x_2 + x_3 - x_4 = 7$$
$$3x_1 + 2x_4 = 8$$
$$2x_1 + 4x_2 + 3x_3 = 15$$

this can be written in matrix format like this .

$$\begin{bmatrix} 1 & 1 & -1 & 4 \\ 0 & 2 & 1 & -1 \\ 3 & 0 & 0 & 2 \\ 2 & 4 & 3 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 9 \\ 7 \\ 8 \\ 15 \end{bmatrix}$$

```
a=np.array([[1,1,-1,4],
            [0,2,1,-1],
            [3,0,0,2],
            [2,4,3,0]])
b=np.array([9,7,8,15])
```

if we consider them equivalent to $Ax = B$, solution will be $x = A^{-1}B$, given that A is invertible . lets solve with numpy . first the longer version.

```
a_inv=np.linalg.inv(a)
x=np.dot(a_inv,b)
x
```

```
array([ 3.17241379,  5.03448276, -3.82758621, -0.75862069])
```

numpy also has a direct solve function for system of linear equation.

```
np.linalg.solve(a,b)
```

```
array([ 3.17241379,  5.03448276, -3.82758621, -0.75862069])
```

check that the solution is correct .

```
np.allclose(np.dot(a, x), b)
```

```
True
```

Another package which builds on top of numpy and lets us process data while not being as verbose as things might have needed to be in numpy; Pandas that we discuss next

Pandas

For this discussion we will be making use of data file `diabetic_data.csv` . The data contains `100000` observation pertaining to patients admitted to around `130` US hospitals where during their hospitalization ; diabetes was entered to the system as a diagnosis.

The data contains such attributes as patient number, race, gender, age, admission type, time in hospital, medical specialty of admitting physician, number of lab test performed, HbA1c test result, diagnosis, number of medication, diabetic medications, number of outpatient, inpatient, and emergency visits in the year before the hospitalization, etc.

source : https://archive.ics.uci.edu/ml/datasets/Diabetes+130-US+hospitals+for+years+1999-2008

Loading Data

You can load csv files into pandas dataframe format using function `read_csv`

```
diab_data=pd.read_csv("diabetic_data.csv")
```

`read_csv` , however is not the only option. On the similar lines there are functions such as `read_excel` , `read_json` and so on which can load many other file formats to pandas dataframe.

if your data is hosted on say sql server , you can load it directly from the database using `read_sql` .

within `read_csv` , the first argument is for the file path, there are other arguments which we did not explicitly use in our example because the nice defaults work most of the time [including our example data here]. But you can explore options like these and some more in the documentation

- `sep` : default is `,` but you can specify any other delimiter also
- `header` : default is infer , the first line data is used as headers . But if that is not the case you can set `header=None` . and separately pass columns as list using argument `names`
- `na_values` : scalar, str, list-like, or dict, optional Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', '', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.
- `nrows` : Number of rows of file to read. Useful for reading pieces of large files. by default entire file is read but using this we can load a small part of the file starting at any row number when combined with option `skiprows` [which is set to zero by default]

- `na_filter` : bool, default True. Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.
- `skip_blank_lines` : bool, default True If True, skip over blank lines rather than interpreting as NaN values.

experiment with these options to understand better their effects

Data Exploration

Once you have the data in memory in form of a pandas dataframe , the next thing is to explore the data. Figuring out what kind of information it contains , how many rows and columns it has , what are the column types and so on. In this section we will get an overview of available functions in pandas for data explorations.

Peripheral Overview

Simply writing the name of the data frame , displays some part of the data with truncation in between. A more popular way of looking at a fraction of the data is to look at top few or bottom few values with functions `head` and `tail` .

```
diab_data.head()
```

This displays top 5 rows of the data by default . you can pass a custom integer also and then it will display those many rows of the data . for example following will display 8 top rows of the data.

```
diab_data.head(8)
```

`tail` function works just like that but it displays bottom rows of the data. If you want to randomly sample some rows from the data. you can use function `sample` .

following bit of code will randomly pick 6 rows from the data every time you execute it .

```
diab_data.sample(6)
```

function `info` gives a good peripheral summary of the data .

```
diab_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 101766 entries, 0 to 101765
Data columns (total 50 columns):
#   Column                    Non-Null Count   Dtype 
---  ------                    --------------   ----- 
0   encounter_id              101766 non-null  int64 
1   patient_nbr               101766 non-null  int64 
2   race                      101766 non-null  object
3   gender                    101766 non-null  object
4   age                       101766 non-null  object
5   weight                    101766 non-null  object
6   admission_type_id         96475 non-null   object
7   discharge_disposition_id  98075 non-null   object
8   admission_source_id       94985 non-null   object
9   time_in_hospital          101766 non-null  int64 
10  payer_code                101766 non-null  object
11  medical_specialty         101766 non-null  object
12  num_lab_procedures        101766 non-null  int64 
13  num_procedures            101766 non-null  int64 
14  num_medications           101766 non-null  int64 
15  number_outpatient         101766 non-null  int64 
16  number_emergency          101766 non-null  int64 
17  number_inpatient          101766 non-null  int64 
18  diag_1                    101766 non-null  object
19  diag_2                    101766 non-null  object
20  diag_3                    101766 non-null  object
21  number_diagnoses          101766 non-null  int64 
22  max_glu_serum             101766 non-null  object
23  A1Cresult                 101766 non-null  object
24  metformin                 101766 non-null  object
25  repaglinide               101766 non-null  object
26  nateglinide               101766 non-null  object
27  chlorpropamide            101766 non-null  object
28  glimepiride               101766 non-null  object
29  acetohexamide             101766 non-null  object
30  glipizide                 101766 non-null  object
31  glyburide                 101766 non-null  object
32  tolbutamide               101766 non-null  object
33  pioglitazone              101766 non-null  object
```

```
34  rosiglitazone             101766 non-null  object
35  acarbose                  101766 non-null  object
36  miglitol                  101766 non-null  object
37  troglitazone              101766 non-null  object
38  tolazamide                101766 non-null  object
39  examide                   101766 non-null  object
40  citoglipton               101766 non-null  object
41  insulin                   101766 non-null  object
42  glyburide-metformin       101766 non-null  object
43  glipizide-metformin       101766 non-null  object
44  glimepiride-pioglitazone  101766 non-null  object
45  metformin-rosiglitazone   101766 non-null  object
46  metformin-pioglitazone    101766 non-null  object
47  change                    101766 non-null  object
48  diabetesMed               101766 non-null  object
49  readmitted                101766 non-null  object
dtypes: int64(10), object(40)
memory usage: 38.8+ MB
```

for some this might be too much information to unpack. you can individual information using some alternate attributes.

Attribute `shape` contains information about number of rows and columns in the data.

```
diab_data.shape
```

(101766, 50)

dataset contains `101766` rows and `50` columns . Attribute `dtypes` can be used to see data types of all the columns , attribute `dtype` with a single column can be used to determine type of that particular column.

```
diab_data.dtypes
```

```
encounter_id               int64
patient_nbr                int64
race                      object
gender                    object
age                       object
weight                    object
admission_type_id         object
discharge_disposition_id  object
admission_source_id       object
time_in_hospital           int64
payer_code                object
medical_specialty         object
num_lab_procedures         int64
num_procedures             int64
num_medications            int64
number_outpatient          int64
number_emergency           int64
number_inpatient           int64
diag_1                    object
diag_2                    object
diag_3                    object
number_diagnoses           int64
max_glu_serum             object
A1Cresult                 object
metformin                 object
repaglinide               object
nateglinide               object
chlorpropamide            object
glimepiride               object
acetohexamide             object
glipizide                 object
glyburide                 object
tolbutamide               object
pioglitazone              object
rosiglitazone             object
acarbose                  object
miglitol                  object
troglitazone              object
tolazamide                object
examide                   object
citoglipton               object
insulin                   object
glyburide-metformin       object
glipizide-metformin       object
glimepiride-pioglitazone  object
```

```
metformin-rosiglitazone     object
metformin-pioglitazone      object
change                      object
diabetesMed                 object
readmitted                  object
```

```
diab_data['age'].dtype
```

```
dtype('O')
```

here `'O'` stands for object

```
diab_data['time_in_hospital'].dtype
```

```
dtype('int64')
```

Note that for the data frame, attribute is `dtypes` [plural] whereas for individual columns it is `dtype` [singular] .

You can get list of just the column names with attribute `columns`

```
diab_data.columns
```

```
Index(['encounter_id', 'patient_nbr', 'race', 'gender', 'age', 'weight',
 'admission_type_id', 'discharge_disposition_id', 'admission_source_id',
 'time_in_hospital', 'payer_code', 'medical_specialty',
 'num_lab_procedures', 'num_procedures', 'num_medications',
 'number_outpatient', 'number_emergency', 'number_inpatient', 'diag_1',
 'diag_2', 'diag_3', 'number_diagnoses', 'max_glu_serum', 'A1Cresult',
 'metformin', 'repaglinide', 'nateglinide', 'chlorpropamide',
 'glimepiride', 'acetohexamide', 'glipizide', 'glyburide','tolbutamide',
 'pioglitazone', 'rosiglitazone', 'acarbose', 'miglitol','troglitazone',
 'tolazamide', 'examide', 'citoglipton', 'insulin',
 'glyburide-metformin', 'glipizide-metformin',
 'glimepiride-pioglitazone', 'metformin-rosiglitazone',
 'metformin-pioglitazone', 'change', 'diabetesMed', 'readmitted'],
dtype='object')
```

Next we would to look at actual values in the data with various subsetting functionalities inbuilt in the pandas dataframes.

Subsetting

Using indices

```
diab_data.iloc[0:2,1:5]
```

	patient_nbr	race	gender	age
0	8222157	Caucasian	Female	[0-10)
1	55629189	Caucasian	Female	[10-20)

using `.iloc` you can pass indices for rows and columns in the square bracket and access the dataframe like a `2-D` array . In the square bracket for indices , left side indices refer to row positions and right side indices refer to columns . Indices behave just like they do in `2-D` arrays. Revise that section if you need a refresher .

Using names

Although you can have row names in terms of having a verbose index, but for all practical purposes , simple numeric sequence starting at `0` is what we will use for rows and traditional columns headers for column names .

you can access a single column with a single square bracket . outcome is a series . [we are going to look at head of the column only in order to avoid printing too many values here]

```
diab_data['num_lab_procedures'].head()
```

```
0    41
1    59
2    11
3    44
4    51
Name: num_lab_procedures, dtype: int64
```

For accessing multiple columns at once , you will need to pass them as a list of column names inside the square bracket [resulting in double square bracket as you see below]

```
diab_data[['num_lab_procedures','admission_type_id']].head()
```

	num_lab_procedures	admission_type_id
0	41	NaN
1	59	Emergency
2	11	Emergency
3	44	Emergency
4	51	Emergency

for using both row names and column names together we need to use `.loc` with the dataframe like this example .

```
diab_data.loc[120:126,['num_lab_procedures','admission_type_id']]
```

	num_lab_procedures	admission_type_id
120	58	Urgent
121	68	NaN
122	47	NaN
123	88	NaN
124	42	NaN
125	62	NaN
126	58	NaN

Conditional Subsetting

On the row name side where we are passing row indices as names , we can also pass conditions for subsetting the data with `.loc` . For example if we want to create a subset with columns `encounter_id` and `patient_nbr` where `race=="AfricanAmerican"` and `gender=="Female"` and `time_in_hospital>10` , we can do it like this :

```
diab_data.loc[(diab_data['race']=="AfricanAmerican") &
              (diab_data['gender']=="Female") &
              (diab_data['time_in_hospital']>=10) ,
['encounter_id','patient_nbr']].head()
```

	encounter_id	patient_nbr
135	2292606	53848278
363	3483090	5877477
369	3547464	4910139
429	3894828	96165
460	4084524	76959585

Summary

The next stage of exploration requires us to look at summaries which gives us much better and quicker handle on whats going on with the data than simply browsing through data rows.

we will start with the default , describe function .

```
diab_data.describe().T
```

	count	mean	std	min	25%	50%	75%	
encounter_id	101766.0	1.652016e+08	1.026403e+08	12522.0	84961194.0	152388987.0	2.302709e+08	4438
patient_nbr	101766.0	5.433040e+07	3.869636e+07	135.0	23413221.0	45505143.0	8.754595e+07	1895
time_in_hospital	101766.0	4.395987e+00	2.985108e+00	1.0	2.0	4.0	6.000000e+00	14.0
num_lab_procedures	101766.0	4.309564e+01	1.967436e+01	1.0	31.0	44.0	5.700000e+01	132.0
num_procedures	101766.0	1.339730e+00	1.705807e+00	0.0	0.0	1.0	2.000000e+00	6.0
num_medications	101766.0	1.602184e+01	8.127566e+00	1.0	10.0	15.0	2.000000e+01	81.0
number_outpatient	101766.0	3.693572e-01	1.267265e+00	0.0	0.0	0.0	0.000000e+00	42.0
number_emergency	101766.0	1.978362e-01	9.304723e-01	0.0	0.0	0.0	0.000000e+00	76.0
number_inpatient	101766.0	6.355659e-01	1.262863e+00	0.0	0.0	0.0	1.000000e+00	21.0
number_diagnoses	101766.0	7.422607e+00	1.933600e+00	1.0	6.0	8.0	9.000000e+00	16.0

This by default generates these 8 numerical summaries for the features which are stored as numbers in the data . we will see functions for summarizing categorical data also in some time, in this section itself.

We don't always have to use describe and wade through all these default results . We can get individual summary stats as well both at data and column levels . Here are some examples.

```
diab_data.median()
```

```
encounter_id          152388987.0
patient_nbr            45505143.0
time_in_hospital              4.0
num_lab_procedures           44.0
num_procedures                1.0
num_medications              15.0
number_outpatient             0.0
number_emergency              0.0
number_inpatient              0.0
number_diagnoses              8.0
```

```
diab_data['num_procedures'].mean()
```

```
1.339730361810428
```

```
diab_data[['num_procedures','number_emergency']].std()
```

```
num_procedures      1.705807
number_emergency    0.930472
```

given below are some of the most used summary functions available in pandas . [some of them might not make sense in context of entire data frame]

Sr.No.	Function	Description
1	count()	Number of non-null observations
2	sum()	Sum of values
3	mean()	Mean of Values
4	median()	Median of Values
5	mode()	Mode of values
6	std()	Standard Deviation of the Values
7	min()	Minimum Value
8	max()	Maximum Value
9	abs()	Absolute Value
10	prod()	Product of Values
11	cumsum()	Cumulative Sum
12	cumprod()	Cumulative Product

We can get unique values and number of unique values [this can be applied on data frame as well] with functions `unique()` and `nunique()` respectively.

```
diab_data.nunique()
```

```
encounter_id                101766
patient_nbr                  71518
race                             6
gender                           3
age                             10
weight                          10
admission_type_id                7
discharge_disposition_id        25
admission_source_id             16
time_in_hospital                14
payer_code                      18
medical_specialty               73
num_lab_procedures             118
num_procedures                   7
num_medications                 75
number_outpatient               39
number_emergency                33
number_inpatient                21
diag_1                         717
diag_2                         749
diag_3                         790
number_diagnoses                16
max_glu_serum                    4
A1Cresult                        4
metformin                        4
repaglinide                      4
nateglinide                      4
chlorpropamide                   4
glimepiride                      4
acetohexamide                    2
glipizide                        4
glyburide                        4
tolbutamide                      2
pioglitazone                     4
rosiglitazone                    4
acarbose                         4
miglitol                         4
troglitazone                     2
tolazamide                       3
examide                          1
citoglipton                      1
insulin                          4
glyburide-metformin              4
glipizide-metformin              2
glimepiride-pioglitazone         2
metformin-rosiglitazone          2
metformin-pioglitazone           2
```

```
change                      2
diabetesMed                 2
readmitted                  3
```

```
diab_data['admission_type_id'].unique()
```

```
array([nan, 'Emergency', 'Urgent', 'Elective', 'Newborn', 'Not Available',
 'Not Mapped', 'Trauma Center'], dtype=object)
```

For categorical data , we need frequencies of individual categories as summaries . There are couple of functions which can help us with that .

```
diab_data['admission_type_id'].value_counts(dropna=False)
```

```
Emergency        53990
Elective         18869
Urgent           18480
NaN               5291
Not Available     4785
Not Mapped         320
Trauma Center       21
Newborn             10
```

this can be converted to percentages also with option `normalize`

```
diab_data['admission_type_id'].value_counts(dropna=False,normalize=True)
```

```
Emergency        0.530531
Elective         0.185416
Urgent           0.181593
NaN              0.051992
Not Available    0.047020
Not Mapped       0.003144
Trauma Center    0.000206
Newborn          0.000098
```

for two way frequency or cross tables we will use function `crosstab` from pandas .

```
pd.crosstab(diab_data['gender'],diab_data['admission_type_id'])
```

gender	Female	Male	Unknown/Invalid
admission_type_id			
Elective	9840	9028	1
Emergency	29448	24540	2
Newborn	3	7	0
Not Available	2609	2176	0
Not Mapped	176	144	0
Trauma Center	9	12	0
Urgent	9894	8586	0

this also can be converted to having row wise or column wise percentages .

```
pd.crosstab(diab_data['admission_type_id'],diab_data['gender'],normalize='index')
```

this gives row percentages.

gender	Female	Male	Unknown/Invalid
admission_type_id			
Elective	0.521490	0.478457	0.000053
Emergency	0.545434	0.454529	0.000037
Newborn	0.300000	0.700000	0.000000
Not Available	0.545246	0.454754	0.000000
Not Mapped	0.550000	0.450000	0.000000
Trauma Center	0.428571	0.571429	0.000000
Urgent	0.535390	0.464610	0.000000

```
pd.crosstab(diab_data['admission_type_id'],diab_data['gender'],normalize='columns')
```

this gives column percentages.

gender	Female	Male	Unknown/Invalid
admission_type_id			
Elective	0.189307	0.202908	0.333333
Emergency	0.566536	0.551547	0.666667
Newborn	0.000058	0.000157	0.000000
Not Available	0.050193	0.048907	0.000000
Not Mapped	0.003386	0.003236	0.000000
Trauma Center	0.000173	0.000270	0.000000
Urgent	0.190346	0.192974	0.000000

Numeric summaries that we saw earlier were at entire data level , we can get those summaries at different levels also using `groupby` .

```
diab_data.groupby(['readmitted'])['num_lab_procedures','num_medications'].mean()
```

	num_lab_procedures	num_medications
readmitted		
<30	45.0	16.0
>30	45.0	15.0
NO	44.0	14.0

In the next section we are going to looking at methods available for making changes in the data .

Data Modification

Deleting Columns

A single column can be permanently deleted with keyword `del` .

```
del data_name['column_name']
```

for deleting multiple columns we can use function `drop` .

```
diab_data.drop(['examide','citoglipton'],axis=1,inplace=True)
```

`axis=1` here refers to dropping columns. with `axis=0` , the same function can be used to drop rows as well, but in practice its used for dropping rows rarely . Usually subsetting on rows is basis some condition which can be done with `.loc` .

`inplace=True` removes the columns in place, whereas not using that argument results in an explicit outcome and no changes in the original dataframe.

Change Column Name

column names can be changed with function `rename` , it takes a dictionary as input where keys are the old column names and values are the new names that we want .

here is an example .

```
diab_data.rename(columns={'admission_type_id':'admission_type',
'discharge_disposition_id':'discharge_disposition',
'admission_source_id':'admission_source'},inplace=True)
```

```
diab_data.columns
```

> Index(['encounter_id', 'patient_nbr', 'race', 'gender', 'age', 'weight',
> **'admission_type, 'discharge_disposition, 'admission_source,**
> 'time_in_hospital', 'payer_code', 'medical_specialty',
> 'num_lab_procedures', 'num_procedures', 'num_medications',
> 'number_outpatient', 'number_emergency', 'number_inpatient', 'diag_1',
> 'diag_2', 'diag_3', 'number_diagnoses', 'max_glu_serum', 'A1Cresult',
> 'metformin', 'repaglinide', 'nateglinide', 'chlorpropamide',
> 'glimepiride', 'acetohexamide', 'glipizide', 'glyburide', 'tolbutamide',
> 'pioglitazone', 'rosiglitazone', 'acarbose', 'miglitol', 'troglitazone',
> 'tolazamide', 'examide', 'citoglipton', 'insulin',
> 'glyburide-metformin', 'glipizide-metformin',
> 'glimepiride-pioglitazone', 'metformin-rosiglitazone',
> 'metformin-pioglitazone', 'change', 'diabetesMed', 'readmitted'],
> dtype='object')

Adding New Columns

```
diab_data['new_column']=np.random.random(diab_data.shape[0])
```

this adds a new column with name `new_column` in the data with random values. We can pass some other list of values also having as many values as there are rows in the data.

```
diab_data['new_column']=100
```

this adds the same name column which is filled with constant 100. we can similarly add any number columns which take values as result of some algebraic operations or conditional statements. Here are few examples .

```
diab_data['new_column']=np.log(diab_data['time_in_hospital'])
```

```
diab_data['new_column']=diab_data['num_medications']/diab_data['num_procedures']
```

```
diab_data['new_column']=np.where(data_diab['time_in_hospital']>4,'high','low')
```

Keep in mind that we don't really have to always add a new column, we can modify existing columns in the data as well by simply reassigning/overwriting as above.

Value Replacement

Lets say we want to replace all "?" appearing in the data anywhere with missing values. We can do that using function replace which can be used both at data and individual column levels. here are two examples .

```
diab_data.replace({"?":np.nan},inplace=True)
```

This replaces `?` with `np.nan` [missing values] across all the columns in the entire data. The dictionary being used by `replace` multiple replacement options as well. We can use replace for specific columns also like this example here:

```
diab_data['age'].replace({'[0-10)':5, '[10-20)':15,
                          '[20-30)':25, '[30-40)':35, '[40-50)':45,
                          '[50-60)':55,'[60-70)':65, '[70-80)':75,
                          '[80-90)':85, '[90-100)':95},inplace=True)
```

I want to add some more examples here to showcase how versatile replace is for pandas DataFrames . Those examples however are on a toy data to show the functionalities easily.

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

```
df
```

	A	B	C
0	0	5	0
1	1	6	0
2	2	7	1
3	3	8	1
4	4	9	2

This will replace all `0`s in the dataframe in all columns with value `5`. First argument is what you want to replace second argument is what you want to replace those values with

```
df.replace(0,5)
```

	A	B	C
0	5	5	5
1	1	6	5
2	2	7	1
3	3	8	1
4	4	9	2


```
df
```

	A	B	C
0	0	5	0
1	1	6	0
2	2	7	1
3	3	8	1
4	4	9	2

this replace all the values in the list in first argument with value specified in the second argument

```
df.replace([0,1,2,3],4)
```

	A	B	C
0	4	5	4
1	4	6	4
2	4	7	4
3	4	8	4
4	4	9	4

```
df
```

	A	B	C
0	0	5	0
1	1	6	0
2	2	7	1
3	3	8	1
4	4	9	2

for this to be replace list and replacement list should have same size however in most practical scenarios we might have different replacements for different columns

```
df.replace([0,1,5,8],[10,11,55,88])
```

	A	B	C
0	10	55	10
1	11	6	10
2	2	7	11
3	3	88	11
4	4	9	2

```
df
```

	A	B	C
0	0	5	0
1	1	6	0
2	2	7	1
3	3	8	1
4	4	9	2

it will replace different values in different columns with specified replacement value

```
df.replace({'A':0,'B':7,'C':1},999)
```

	A	B	C
0	999	5	0
1	1	6	0
2	2	999	999
3	3	8	999
4	4	9	2


```
df
```

	A	B	C
0	0	5	0
1	1	6	0
2	2	7	1
3	3	8	1
4	4	9	2

You can also pass dictionary for each column inside the dictionary input as a dictionary against each column name and specify different replacement pairings for each.

```
df.replace({'A':{0:100,1:99},'B':{5:55},'C':{0:10000,1:11}})
```

	A	B	C
0	100	55	10000
1	99	6	10000
2	2	7	11
3	3	8	11
4	4	9	2

String Operations

String operations can be carried out on the entire column by using `str` attribute. For example the earlier function `replace` looks at complete values . If we want to replace text 'Transfer from'; occurring in the values of the column `admission_source` ; we will have to use string processing. Here is how we can do that .

```
diab_data['admission_source']= diab_data['admission_source'].str.replace('Transfer from','tfr')
```

Note that there is no `inplace` option available for the function being applied on `str` attribute .

Type Casting

Many at times we will need to change data types of the columns . For example categorical data to numeric . Pandas has many functions of the kind `pd.to_datatype` which can be used to change the data types .

here is the standard syntax to make use of for such operations.

```
data['column_name']=pd.to_numeric(data['column_name'],errors='coerce')
```

the option `errors='coerce'` exist here to handle values which can not be converted to a number , this option simply replace such values with `NAN` , which is the expected behavior for such an operation.

similar to `to_numeric` , there are options for `to_datetime` , `to_timedelta` .

Handling duplicates

the example dataset that we are working with doesnt have any duplicate rows , lets add some and then we will learn how to remove them.

```
diab_data=diab_data.append(diab_data.sample(100),ignore_index=True)
```

Now if we count how many duplicate rows are there in the data , we should see 100 duplicate rows.

```
diab_data.duplicated().sum()
```

```
100
```

Note that rows will be counted as duplicate only if values across all the columns match .We can also use argument `subset` for the function `duplicated` to pass a list of columns which should be considered to check duplicates if we dont want all the columns in the data to be used for declaring duplicates.

here is an example .

```
diab_data.duplicated(subset=['race','gender','payer_code']).sum()
```

```
101688
```

you can see that there many more observations which have matching values across `['race','gender','payer_code']` .

we can use function `drop_duplicates` to remove such duplicate records .

```
diab_data.drop_duplicates(keep='first',inplace=True)
```

in this example , when duplicates are removed , among the duplicates; first record is kept in the data. If in a rare case you want to remove all records which have any duplicate in the data entirely , you can use `keep=False` . using `keep='last'` will result in last record among the duplicates being kept in the data.

`drop_duplicates` function also has argument `subset` which can be used as discussed earlier.

Handling Missing Values

Lets see if our data has any missing values

```
diab_data.isna().sum().sort_values(ascending=False)
```

```
weight                      98664
medical_specialty           50004
payer_code                  40294
admission_source             6789
admission_type               5298
discharge_disposition        3693
race                         2274
diag_3                       1424
diag_2                        358
diag_1                         21
tolazamide                      0
citoglipton                     0
examide                         0
encounter_id                    0
troglitazone                    0
miglitol                        0
rosiglitazone                   0
pioglitazone                    0
acarbose                        0
glipizide-metformin             0
insulin                         0
glyburide-metformin             0
glyburide                       0
glimepiride-pioglitazone        0
metformin-rosiglitazone         0
metformin-pioglitazone          0
change                          0
diabetesMed                     0
readmitted                      0
tolbutamide                     0
repaglinide                     0
glipizide                       0
acetohexamide                   0
gender                          0
age                             0
time_in_hospital                0
num_lab_procedures              0
num_procedures                  0
num_medications                 0
number_outpatient               0
number_emergency                0
number_inpatient                0
number_diagnoses                0
max_glu_serum                   0
A1Cresult                       0
metformin                       0
patient_nbr                     0
nateglinide                     0
chlorpropamide                  0
glimepiride                     0
new_column                      0
```


we can use function `fillna` to fill this missing values both at data as well as column level.

say we want to fill missing values across data with `0` , we can use this generic syntax .

```
data.fillna(0,inplace=True)
```

This however is not a smart practice . Most of the time , it makes to treat every column separately. For example , lets say we want to fill missing values in a categorical column with a placeholder `__missing__` . we can do that as follows .

```
data['categorical_column_name'].fillna('__missing__',inplace=True)
```

If we want to fill missing values of a numeric column with its median , we can do .

```
data['num_column_name'].fillna(data['num_column_name'].median(),inplace=True)
```

We can also use dictionary as input to function `fillna` where keys will be the column names and replacement values will be the values . here is the standard syntax .

```
data.fillna({'col1':val1,'col2':val2...},inplace=True)
```

Sorting and Merging DataFrames

Two more important operations in data exploration and data consolidation are sorting and merging. Sorting pertains to reordering your data by some column and merging refers to combining multiple datasets into one with some reference. Lets look at sorting first with a generated dataset first.

Sorting

This is the data frame that we will be sorting , has 4 columns and contains random integers between 2 to 8 in each. Although the example here shows sorting on numbers only , sorting can be done on character data also, it follows dictionary order.

```
df=pd.DataFrame(np.random.randint(2,8,(20,4)),columns=['A','B','C','D'])
```

```
df
```

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

We use function `sort_values` for sorting . By default sorting happens in ascending manner. For the changes to persist in the dataframe , use `inplace=True` or assign the result back to the dataframe using `=`.

```
df.sort_values('A')
```

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

You can reverse the order of sorting by specifying `ascending=False` .

```
df.sort_values('A',ascending=False)

```

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

You can sort by multiple columns as well . Sorting happens in sequence . First all rows are sorted by values of the first column. Then the sorting happens by values of second column within the groups defined by values of the first column. Then the sorting happens by values of the third column within the groups defined by values of first and second column and so on.

```
df.sort_values(['A','B','C'])
```

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

You can control the sorting order here also by using option `ascending` . Pass a list of `True` , `False` in the coerrespording order of the column name list passed as first input to the function .

```
df.sort_values(['A','B','C'],ascending=[True,False,True])

```

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

Combining dataframes by stacking them vertically or horizontally

We will use some small datasets for this.

```
df1=pd.DataFrame([('a',1),('b',2)],columns=['letter','number'])
```

```
df1
```

	letter	number
0	a	1
1	b	2

```
df2=pd.DataFrame([('c',3,'cat'),('d',4,'dog'),('e',7,'parrot')],
                 columns=['letter','number','animal'])
```

```
df2
```

	letter	number	animal
0	c	3	cat
1	d	4	dog
2	e	7	parrot

function used for stacking is `concat` , option `axis=0` makes the input dataframes stack along rows or vertically . Note that both the dataset do not have same number of columns. Whereever the function find same columns in dataframes being stacked it alings values of those . Columns which are present only in one of the dataframes , they end up having missing values for corresponding rows as shown in the example below.

```
pd.concat([df1,df2],axis=0)
```

	letter	number	animal
0	a	1	NaN
1	b	2	NaN
0	c	3	cat
1	d	4	dog
2	e	7	parrot

For stacking dataframes horizontally or side by side, use option `axis=1` which makes the dataset stack along columns. Be careful while doing this because this can lead to having multiple columns in the data having same names. This doesnt throw an error while stacking but can become and issue later on in your process.

```
pd.concat([df1,df2],axis=1)
```

	letter	number	letter	number	animal
0	a	1.0	c	3	cat
1	b	2.0	d	4	dog
2	NaN	NaN	e	7	parrot

you can avoid this by renaming your columns before hand .

```
df3=df2.rename(columns={'letter':'letter_df2','number':'number_df2'})
```

```
df3
```

	letter_df2	number_df2	animal
0	c	3	cat
1	d	4	dog
2	e	7	parrot

```
df_stack_2=pd.concat([df1,df3],axis=1)

```

```
df_stack_2
```

	letter	number	letter_df2	number_df2	animal
0	a	1.0	c	3	cat
1	b	2.0	d	4	dog
2	NaN	NaN	e	7	parrot

merging

Many at times when data is coming from disparate sources, you can combine them simply by stacking vertically or horizontally. A proper reference for information coming from all sources needs to be maintained . For example lets say you have medical history of patients in one data and their billing details in another , you would want to combine them in such a way that medical history and billing details of the same customer are in the same row, otherwise the data will not make sense or it will be downright wrong.

This process of combining data with some common reference between rows is called merging . There are four kind of common merging methods. Lets create the DataFrames for this first.

```
df1=pd.DataFrame({'custid':[1,2,3,4,5],'product':['A','B','C','A','D']})
```

```
df1
```

	custid	product
0	1	A
1	2	B
2	3	C
3	4	A
4	5	D

```
df2=pd.DataFrame({'custid':[3,4,5,6,7,8],'state':['TN','TN','Assam','MH','JK','PN']})
```

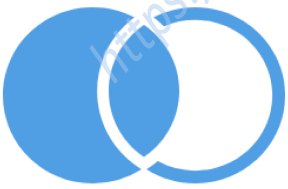
```
df2
```

	custid	state
0	3	TN
1	4	TN
2	5	Assam
3	6	MH
4	7	JK
5	8	PN

4 kind of joins or ways of merging

- inner join : keeps the keys [and the corresponding info] which are common in both
- outer join/ full join : keeps all the keys and fills missing info with NaN
- left join : keeps the keys which are present in left data
- right join : keeps the keys which are present in right data

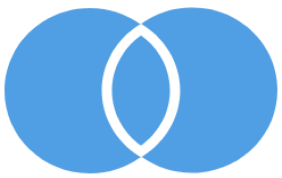
option `on` is where you can pass the columns which are `keys` or common reference between the two dataframes .



Left outer join

Inner join

Right outer join



Full outer join

```
pd.merge(df1,df2,on=['custid'],how='inner')
```

	custid	product	state
0	3	C	TN
1	4	A	TN
2	5	D	Assam

```
pd.merge(df1,df2,on=['custid'],how='outer')
```

	custid	product	state
0	1	A	NaN
1	2	B	NaN
2	3	C	TN
3	4	A	TN
4	5	D	Assam
5	6	NaN	MH
6	7	NaN	JK
7	8	NaN	PN

```
pd.merge(df1,df2,on=['custid'],how='left')
```

	custid	product	state
0	1	A	NaN
1	2	B	NaN
2	3	C	TN
3	4	A	TN
4	5	D	Assam

This concludes our discussion on pandas.