Python Cheat Sheet

Functions:

Unpacking list:

```
[ ] #list basics
    List1 = ['hello','great','learning']
    print(*List1) #unpacking
    l=[*List1, 'ok','bye']
    print(l)
    print(*l,sep=', ')

    hello great learning
    ['hello', 'great', 'learning', 'ok', 'bye']
    hello, great, learning, ok, bye
```

```
[ ] a=[1,2,3]
    zipped=zip(a)
    list(zipped)

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

```
[ ] list(zip(range(5),range(100))) #iteration stops when smallest is exhausted

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

Itertools zip_longest:

```
#iteration only stops when longest is exhausted
from itertools import zip_longest
numbers=[1,2,3]
letters=['a','b','c']
longest=range(5)
zipped=zip_longest(numbers,letters,longest,fillvalue='?')
list(zipped)

[(1, 'a', 0), (2, 'b', 1), (3, 'c', 2), ('?', '?', 3), ('?', '?', 4)]
```

```
[ ] for i in zip(l):
        print(i)

    ('hello',)
    ('great',)
    ('learning',)
    ('ok',)
    ('bye',)
```

List comprehensions: Lists that generate themselves within an internal for loop

```
evensquare1=[ num**2  for num in range(10,30)  if num % 2==0]
print(evensquare1)

[100, 144, 196, 256, 324, 400, 484, 576, 676, 784]
```

```
[ ] e=enumerate(mylist)
    print(e)
    print(type(e))
    print(list(e))
    print(list(e))

    <enumerate object at 0x7fcc29aa8c30>
    <class 'enumerate'>
    [(0, 11), (1, 22), (2, 33), (3, 44.4), (4, 55)]
    []
```

Loop with logic:

```
[ ] list=[]
    for num in range(10,20):
        if num%2==0:
            list.append(num**2)
        else:
            list.append(num**3)
    print(list)

    [100, 1331, 144, 2197, 196, 3375, 256, 4913, 324, 6859]
```

It's list comprehension:

```
[ ] list1=[num**2 if num%2==0 else num**3 for num in range(10,20)]
    print(list1)

    [100, 1331, 144, 2197, 196, 3375, 256, 4913, 324, 6859]
```

For list items from 10-19, the numbers will be multiplied by 2 if it is divisible by 2, else the numbers will be multiplied by 3

```
ulist=['u1','u2','u3']
plist=['p1','p2','p3']
uplist=[]
for u in ulist:
    for p in plist:
        uplist.append((u,p))
print(uplist)

[('u1', 'p1'), ('u1', 'p2'), ('u1', 'p3'), ('u2', 'p1'), ('u2', 'p2'), ('u2', 'p3'), ('u3', 'p1'), ('u3', 'p2'), ('u3', 'p3')]
```

```
[ ] uplist=[  (u,p)    for u in ulist   for p in plist]
    print(uplist)

    [('u1', 'p1'), ('u1', 'p2'), ('u1', 'p3'), ('u2', 'p1'), ('u2', 'p2'), ('u2', 'p3'), ('u3', 'p1'), ('u3', 'p2'), ('u3', 'p3')]
```

```
[ ] list=[ (u,p) for u in ulist  if u!='u2'  for p in plist if p!='p2']
    print(list)

    [('u1', 'p1'), ('u1', 'p3'), ('u3', 'p1'), ('u3', 'p3')]
```

```
[ ] d3set={ n%3 for n in range(20)}
    print(d3set)

    {0, 1, 2}
```

Creating key value pairs:

```
[ ] ulist=['u1','u2','u3']
    plist=['p1','p2','p3']
    uplist={u:p    for u in ulist    for p in plist}
    print(uplist)

    {'u1': 'p3', 'u2': 'p3', 'u3': 'p3'}
```

Enmurate: Adds a counter to an iterable.

```
[ ] for index,value in enumerate(mylist):
        print(index,value)
```

```
0 11
1 22
2 33
3 44.4
4 55
```

Lambda functions

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

```
[ ] def mf():
        pass #dummy statement

    mf()
```

```
[ ] #lambda used to create anonymous function objects meaning a function without a name
```

```
iseven=lambda num : num%2==0
r=iseven(20)
print(r)
```

```
True
```

```
e2o3=lambda num: num**2 if num%2==0 else num**3
e2o3(20)
e2o3(9)
```

```
729
```

Reduce function: Used when you need to apply a function to an iterable and reduce it to a single cumulative value.

```
from functools import reduce
import random as rd
```

```
[ ] r=reduce(lambda x,y:x+y , [1,2,3,4,5])
    print(r)
```

```
15
```

```
[ ] rlist=[rd.randint(10,100)  for n in range(10)]
    print(rlist)
```

```
[51, 38, 50, 27, 97, 23, 88, 45, 32, 44]
```

```
[ ] big=reduce(lambda x,y: x if x>y else y, rlist)
    print(big)
```

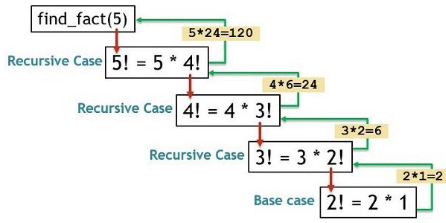
```
97
```

Recursion

```python
def factorial(n):
    if n<=1:
        return 1
    return(n*factorial(n-1))

x=factorial(5)
print(x)
```

```
120
```



A function which calls itself. How this works?

Steps:

1. Returns Number 5*factorial(4)
2. Returns 4*factorial(3)
3. Returns 3* factorial(2)
4. Returns 2* factorial(1)
5. Returns 1
6. Back to Step 4: 2*1=2
7. Back to Step 3: 3*2 =6
8. Back to Step 2: 4*6= 4*3*2=24
9. Back to Step 1: 5*24=120.

Here in step1, factorial 4 is replaced by step 2. Read it as 5*4*factorial(3). In step 2, factorial(3) is read as 3* factorial(2) and so on.

So, ideally this is 5*4*3*2*1

Numpy:

numpy is a core library written in c, c++. the top few layers for user interface is in python the rest is all precompiled libraries. this is how python platform makes up for its speed numpy is the core library for scientific computing in python. Numpy is general purpose array processing package. it provides a high-performance multidimensional array object and tools for working with these arrays

Creation of Matrix

```
[ ] import numpy as np
    mylist=[11,22,33,44,55]
    print(type(mylist))

    <class 'list'>
```

```
[ ] isinstance(1,int)

    True
```

```
[ ] n1=np.array(mylist)
    print(n1)
    print(type(n1))

    [11 22 33 44 55]
    <class 'numpy.ndarray'>
```

A list object has 64 bytes of overhead. For each additional item, its size grows by 8 bytes

```
[ ] n1.dtype #every element of numpy array is taking 8 bytes or 64 bits

    dtype('int64')
```

```
[ ] n1.itemsize # tells that size of each items is 8 bytes as we can see in output

    8
```

```
[ ] mylist=[11,22,33,44.4,55]
    print(type(mylist))

    n1=np.array(mylist)
    print(n1)
    print(type(n1))

    print(n1.ndim)
    print(n1.itemsize)

    <class 'list'>
    [11.  22.  33.  44.4 55. ]
    <class 'numpy.ndarray'>
    1
    8
```

```
[ ] n1.dtype #now we see dtype is float when we changes value in mylist to 44.4

    dtype('float64')
```

```
[ ] mylist1=[11,22,33,44,55]
    print(type(mylist1))

    n1=np.array(mylist1,dtype=np.int8) #we can also force a datatype
    print(n1)
    print(type(n1))

    print(n1.ndim)
    print(n1.itemsize)  #now itemsize is 1 byte or 8 bits

    <class 'list'>
    [11 22 33 44 55]
    <class 'numpy.ndarray'>
    1
    1
```

It is recommended not to change the data type. let numpy decide based on the data.

```python
import numpy as np
#for 2 D
myList2=[[11,12,13,14],[15,16,17,18],[19,20,21,22]] #each sublist represents a row
#this is a 3x4 list
print(myList2)
print(type(myList2))
print()

n34=np.array(myList2) #this is a 3x4 list
print(n34)
print(type(n34))
print()

print(n34.ndim)
print(n34.shape)
print(n34.size)
print(n34.dtype)
print(n34.itemsize)
```

```
[[11, 12, 13, 14], [15, 16, 17, 18], [19, 20, 21, 22]]
<class 'list'>

[[11 12 13 14]
 [15 16 17 18]
 [19 20 21 22]]
<class 'numpy.ndarray'>

2
(3, 4)
12
int64
8
```

```python
n10=np.arange(10) #by default gives 1D numpy matrix
print(n10)
print(n10.shape)
```

```
[0 1 2 3 4 5 6 7 8 9]
(10,)
```

```python
n10=np.arange(10.5,20.3,0.9) #numpy range accepts floating point but python range does not
print(n10)
print(n10.shape)
```

```
[10.5 11.4 12.3 13.2 14.1 15.  15.9 16.8 17.7 18.6 19.5]
(11,)
```

```python
n6=np.arange(6) #make this into 2x3 matrix
print(n6)
print(n6.shape)
```

```
[0 1 2 3 4 5]
(6,)
```

```python
#for 2D
n23=np.arange(6).reshape(2,3)
print(n23)
print(n23.shape)
```

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

If we have number of elements as 6 shape elements have to be factor of that. it can't be 2x1 or 2x4

```python
#for 2D
n23=np.arange(6).reshape(3,-1)
print(n23)
print(n23.shape)
```

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

What happens here is we know total number of elements is 6 and we pass on dimension to be 3 rows, the other dimension We dont want to calculate, numpy will do that for us and give 3x2. we have to pass the other dimension as a negative value

```
#for 2D
n23=np.arange(6).reshape(-1,2)
print(n23)
print(n23.shape)
```

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

What happens here is we know total number of elements is 6 and we pass one dimension to be 3 rows, the other dimension we dont want to calculate, numpy will do that for us and give 3x2. we have to pass the other dimension as a negative value

```
#np.random.seed(23)
n12=np.random.randint(low=10,high=100,size=12) #generate 12 values from 10-100
print(n12)
print(n12.shape)
```

```
[93 68 28 94 25 17 81 25 61 35 46 18]
(12,)
```

These are pseudo random numbers. If we use seed, we'll always get same values.

```
#np.random.seed(23)
n12=np.random.randint(low=10,high=100,size=(3,4)) #generate 12 values from 10-100
print(n12)
print(n12.shape)
#here we cant use -1 since it doesnt know the size only with 3,4 it knows size
```

```
[[54 89 27 17]
 [23 64 29 98]
 [40 78 54 65]]
(3, 4)
```

2D split

```
n68=np.random.randint(low=10,high=100,size=(6,8))
print(n68)
print(n68.shape)
```

```
[[50 53 68 34 98 35 55 69]
 [71 91 85 39 35 16 33 85]
 [23 86 14 34 83 84 73 21]
 [87 39 53 54 87 55 94 18]
 [55 53 43 92 24 30 91 36]
 [23 66 54 98 44 70 54 33]]
(6, 8)
```

hsplit: Split an **array** into multiple sub-**arrays** horizontally (column-wise). Split with axis=1.

```
[ ] #headsplit
    x=np.hsplit(n68,4)
    print(x)

    [array([[50, 53],
           [71, 91],
           [23, 86],
           [87, 39],
           [55, 53],
           [23, 66]]), array([[68, 34],
           [85, 39],
           [14, 34],
           [53, 54],
           [43, 92],
           [54, 98]]), array([[98, 35],
           [35, 16],
           [83, 84],
           [87, 55],
           [24, 30],
           [44, 70]]), array([[55, 69],
           [33, 85],
           [73, 21],
           [94, 18],
           [91, 36],
           [54, 33]])]
```

```
for item in x:
    print(item)
    print()
```

```
[[50 53]
 [71 91]
 [23 86]
 [87 39]
 [55 53]
 [23 66]]

[[68 34]
 [85 39]
 [14 34]
 [53 54]
 [43 92]
 [54 98]]

[[98 35]
 [35 16]
 [83 84]
 [87 55]
 [24 30]
 [44 70]]

[[55 69]
 [33 85]
 [73 21]
 [94 18]
 [91 36]
 [54 33]]
```

Head split splits columns and groups columns together as a numpy array in a list. So, since we did headsplit and 4 as parameter, we get 4 splits. since we have 6x8, 6 rows and 8 columns and we need 4 splits, 8 columns is split 4 times (8/4)=2 so, 2 columns grouped together as one array-> with 6 rows and 2 columns, and we such arrays in a list

Please note: the split must be a factor of the dimension or it throws an error.

```
np.hsplit(n68,3) #the split must be a factor of the dimension or it throws error

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-0a752e898d2e> in <module>()
----> 1 np.hsplit(n68,3) #the split must be a factor of the dimension or it throws error

<__array_function__ internals> in hsplit(*args, **kwargs)

1 frames

<__array_function__ internals> in split(*args, **kwargs)

/usr/local/lib/python3.7/dist-packages/numpy/lib/shape_base.py in split(ary, indices_or_sections, axis)
    871         if N % sections:
    872             raise ValueError(
--> 873                 'array split does not result in an equal division') from None
    874     return array_split(ary, indices_or_sections, axis)
    875

ValueError: array split does not result in an equal division

SEARCH STACK OVERFLOW
```

```
[ ] np.hsplit(n68,1)

[array([[50, 53, 68, 34, 98, 35, 55, 69],
        [71, 91, 85, 39, 35, 16, 33, 85],
        [23, 86, 14, 34, 83, 84, 73, 21],
        [87, 39, 53, 54, 87, 55, 94, 18],
        [55, 53, 43, 92, 24, 30, 91, 36],
        [23, 66, 54, 98, 44, 70, 54, 33]])]
```

```
[ ] np.vsplit(n68,2)

[array([[50, 53, 68, 34, 98, 35, 55, 69],
        [71, 91, 85, 39, 35, 16, 33, 85],
        [23, 86, 14, 34, 83, 84, 73, 21]]),
 array([[87, 39, 53, 54, 87, 55, 94, 18],
        [55, 53, 43, 92, 24, 30, 91, 36],
        [23, 66, 54, 98, 44, 70, 54, 33]])]
```

```
[ ] np.vsplit(n68,3)

[array([[50, 53, 68, 34, 98, 35, 55, 69],
        [71, 91, 85, 39, 35, 16, 33, 85]]),
 array([[23, 86, 14, 34, 83, 84, 73, 21],
        [87, 39, 53, 54, 87, 55, 94, 18]]),
 array([[55, 53, 43, 92, 24, 30, 91, 36],
        [23, 66, 54, 98, 44, 70, 54, 33]])]
```

```
np.hsplit(n68,[3,5,6]) #vectors we have taken are 3 5 6 explained below

[array([[50, 53, 68],
        [71, 91, 85],
        [23, 86, 14],
        [87, 39, 53],
        [55, 53, 43],
        [23, 66, 54]]), array([[34, 98],
        [39, 35],
        [34, 83],
        [54, 87],
        [92, 24],
        [98, 44]]), array([[35],
        [16],
        [84],
        [55],
        [30],
        [70]]), array([[55, 69],
        [33, 85],
        [73, 21],
        [94, 18],
        [91, 36],
        [54, 33]])]
```

Example:

col- 012, 34, 5, 67, get these mentioned columns grouped together

so, if we use a vector those indices will be used for slicing purposes

We have successfully split columns 012, 34, 5, 67 as individual groups as we can see above.

```
[ ] len(np.hsplit(n68,[3,5,6]))

    4
```

so if we use a vector those indices will be used for slicing purposes

```
#col: 01, 2345, 67
np.hsplit(n68,[2,6])

[array([[50, 53],
        [71, 91],
        [23, 86],
        [87, 39],
        [55, 53],
        [23, 66]]), array([[68, 34, 98, 35],
        [85, 39, 35, 16],
        [14, 34, 83, 84],
        [53, 54, 87, 55],
        [43, 92, 24, 30],
        [54, 98, 44, 70]]), array([[55, 69],
        [33, 85],
        [73, 21],
        [94, 18],
        [91, 36],
        [54, 33]])]
```

```
[ ] len(np.hsplit(n68,[2,6])) #:2:6:

    3
```

```
[ ] #rowsplit: 01, 2, 34, 5
    np.vsplit(n68,[2,3,5]) #:2:3:5:

    [array([[50, 53, 68, 34, 98, 35, 55, 69],
            [71, 91, 85, 39, 35, 16, 33, 85]]),
     array([[23, 86, 14, 34, 83, 84, 73, 21]]),
     array([[87, 39, 53, 54, 87, 55, 94, 18],
            [55, 53, 43, 92, 24, 30, 91, 36]]),
     array([[23, 66, 54, 98, 44, 70, 54, 33]])]
```

```
[ ] len(np.vsplit(n68,[2,3,5])) #remember all the split numpy array objects are sitting inside a list

    4
```

2D Stack

In 3D if we are doing head stacking, other 2 dimensions will remain same.
row and column remain same, depth changes, for example, a book with 10 pages and another book with 20 pages is taken. When we stack those two together, row and column remains same, depth becomes 10+20=30. that's hstack

in 2D stacking one-dimension changes, other remains same
in n dimension, nth dim changes, remaining (n-1) remains same

```
[ ] n34=np.arange(12).reshape(3,4)
    print(n34)

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

```
[ ] n36=np.arange(20,38).reshape(3,6)
    print(n36)

    [[20 21 22 23 24 25]
     [26 27 28 29 30 31]
     [32 33 34 35 36 37]]
```

```
[ ] n=np.hstack(tup=(n34,n36)) #hstack takes tuple as input
    print(n.shape)
    print(n)

    (3, 10)
    [[ 0  1  2  3 20 21 22 23 24 25]
     [ 4  5  6  7 26 27 28 29 30 31]
     [ 8  9 10 11 32 33 34 35 36 37]]
```

```
[ ] n64=np.arange(24).reshape(6,4)
```

```
np.vstack((n34,n64))

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

```
[ ] n4a=np.arange(4)
```

```
n4b=np.arange(10,14)
```

```
n4a

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

```
[ ] n4b

    array([10, 11, 12, 13])
```

```
n=np.column_stack(tup=(n4a,n4b))
print(n)

[[ 0 10]
 [ 1 11]
 [ 2 12]
 [ 3 13]]
```

```
[ ] print(n.shape)

    (4, 2)
```

Broadcast

In pandas we use the word concatenation, here we use the word stack

```
n34
```

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

```
n34+20 #scalar broadcasting
```

```
array([[20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
n14=np.arange(20,24).reshape(1,4)
```

```
print(n14) #n14 stretches itself across axis 0
```

```
[[20 21 22 23]]
```

```
n34+n14  # axis 0 stretching based broadcasting
```

```
array([[20, 22, 24, 26],
       [24, 26, 28, 30],
       [28, 30, 32, 34]])
```

```
n31=np.arange(20,23).reshape(3,1)
print(n31)
```

```
[[20]
 [21]
 [22]]
```

```
n34+n31 #axis 1 stretching based broadcasting
```

```
array([[20, 21, 22, 23],
       [25, 26, 27, 28],
       [30, 31, 32, 33]])
```

```
n31
```

```
array([[20],
       [21],
       [22]])
```

```
n14
```

```
array([[20, 21, 22, 23]])
```

```
n31+n14 #both axis stretching based broadcasting
```

```
array([[40, 41, 42, 43],
       [41, 42, 43, 44],
       [42, 43, 44, 45]])
```

Iteration

```
for item in n34: #for sublist in list
    print(item)
```

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

```
for item in np.nditer(n34): #for each of the elements of the matrix
    print(item)
```

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

```
[ ] for item in np.ndenumerate(n34): #elements with position
        print(item)

((0, 0), 0)
((0, 1), 1)
((0, 2), 2)
((0, 3), 3)
((1, 0), 4)
((1, 1), 5)
((1, 2), 6)
((1, 3), 7)
((2, 0), 8)
((2, 1), 9)
((2, 2), 10)
((2, 3), 11)
```

```
for index,value in np.ndenumerate(n34): #elements with position
    print(index,value)

(0, 0) 0
(0, 1) 1
(0, 2) 2
(0, 3) 3
(1, 0) 4
(1, 1) 5
(1, 2) 6
(1, 3) 7
(2, 0) 8
(2, 1) 9
(2, 2) 10
(2, 3) 11
```

3D creation

```
n234=np.random.randint(low=10, high=100, size=(2,3,4))
print(n234)
print(n234.shape)
print(n234.ndim)

[[[51 68 42 27]
  [98 89 43 36]
  [73 20 31 71]]

 [[41 90 94 62]
  [72 91 44 63]
  [70 87 44 10]]]
(2, 3, 4)
3
```

Whenever you see a "*Set of numbers*" closed in double brackets from both ends. Consider it as a "*set*". We have two sets, each set with 3 rows and 4 columns.

Pandas

To read dataset:

```
import pandas as pd
df=pd.read_csv('pokemon_data.csv')
print(df)
```

To view the first five rows:

```
print(df.head(5))
```

```
   #                   Name Type 1  Type 2  HP  Attack  Defense  Sp. Atk  \
0  1              Bulbasaur  Grass  Poison  45      49       49       65
1  2                Ivysaur  Grass  Poison  60      62       63       80
2  3               Venusaur  Grass  Poison  80      82       83      100
3  3  VenusaurMega Venusaur  Grass  Poison  80     100      123      122
4  4             Charmander   Fire     NaN  39      52       43       60

   Sp. Def  Speed  Generation  Legendary
0       65     45           1      False
1       80     60           1      False
2      100     80           1      False
3      120     80           1      False
4       50     65           1      False
```

To view last 3 rows:

```
print(df.tail(3))
```

```
       #                 Name   Type 1 Type 2  HP  Attack  Defense  Sp. Atk  \
797  720  HoopaHoopa Confined  Psychic  Ghost  80     110       60      150   
798  720   HoopaHoopa Unbound  Psychic   Dark  80     160       60      170   
799  721            Volcanion     Fire  Water  80     110      120      130   

     Sp. Def  Speed  Generation  Legendary  
797      130     70           6       True  
798      130     80           6       True  
799       90     70           6       True  
```

To read column headers:

```
#Reading headers
print(df.columns)
```

```
Index(['#', 'Name', 'Type 1', 'Type 2', 'HP', 'Attack', 'Defense', 'Sp. Atk',
       'Sp. Def', 'Speed', 'Generation', 'Legendary'],
      dtype='object')
```

To access only one row of a dataframe:

```
df.Name
```

```
0                  Bulbasaur
1                    Ivysaur
2                   Venusaur
3      VenusaurMega Venusaur
4                 Charmander
               ...          
795                  Diancie
796      DiancieMega Diancie
797      HoopaHoopa Confined
798       HoopaHoopa Unbound
799                Volcanion
Name: Name, Length: 800, dtype: object
```

If we want to access more than one column of a dataframe we can use list for columns and indexing to get rows:

```
print(df[['Name','Type 1','HP']][0:5]) #use list to get more than one columns
```

```
                    Name Type 1  HP
0              Bulbasaur  Grass  45
1                Ivysaur  Grass  60
2               Venusaur  Grass  80
3  VenusaurMega Venusaur  Grass  80
4             Charmander   Fire  39
```

Indexing to access rows:

To get first row:

```
df.iloc[1]
```

```
#                    2
Name           Ivysaur
Type 1           Grass
Type 2          Poison
HP                  60
Attack              62
Defense             63
Sp. Atk             80
Sp. Def             80
Speed               60
Generation           1
Legendary        False
Name: 1, dtype: object
```

To get rows 1, 2 and 3

```
df.iloc[1:4]
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
1	2	Ivysaur	Grass	Poison	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	80	100	123	122	120	80	1	False

To read a value at a specific location: df.iloc[row,col]

```
#Read a specific location(R,C)
print(df.iloc[3,1]) #third row, 1st column
```

```
VenusaurMega Venusaur
```

You can also use iterrows to access each row and get its index:

```
for index,row in df.iterrows():
    print(index,row)
```

```
0 #                      1
Name           Bulbasaur
Type 1             Grass
Type 2            Poison
HP                    45
Attack                49
Defense               49
Sp. Atk               65
Sp. Def               65
Speed                 45
Generation             1
Legendary          False
Name: 0, dtype: object
1 #                    2
Name           Ivysaur
Type 1           Grass
Type 2          Poison
```

Using condition to get rows from data frame accordingly:

```
df.loc[df['Type 1']=='Fire']
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
4	4	Charmander	Fire	NaN	39	52	43	60	50	65	1	False
5	5	Charmeleon	Fire	NaN	58	64	58	80	65	80	1	False
6	6	Charizard	Fire	Flying	78	84	78	109	85	100	1	False
7	6	CharizardMega Charizard X	Fire	Dragon	78	130	111	130	85	100	1	False
8	6	CharizardMega Charizard Y	Fire	Flying	78	104	78	159	115	100	1	False
42	37	Vulpix	Fire	NaN	38	41	40	50	65	65	1	False
43	38	Ninetales	Fire	NaN	73	76	75	81	100	100	1	False

Statistical description of the dataframe columns:

```
df.describe()
```

	#	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation
count	800.000000	800.000000	800.000000	800.000000	800.000000	800.000000	800.000000	800.00000
mean	362.813750	69.258750	79.001250	73.842500	72.820000	71.902500	68.277500	3.32375
std	208.343798	25.534669	32.457366	31.183501	32.722294	27.828916	29.060474	1.66129
min	1.000000	1.000000	5.000000	5.000000	10.000000	20.000000	5.000000	1.00000
25%	184.750000	50.000000	55.000000	50.000000	49.750000	50.000000	45.000000	2.00000
50%	364.500000	65.000000	75.000000	70.000000	65.000000	70.000000	65.000000	3.00000
75%	539.250000	80.000000	100.000000	90.000000	95.000000	90.000000	90.000000	5.00000
max	721.000000	255.000000	190.000000	230.000000	194.000000	230.000000	180.000000	6.00000

Sort values based on column:

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

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
794	718	Zygarde50% Forme	Dragon	Ground	108	100	121	81	95	95	6	True
695	634	Zweilous	Dark	Dragon	72	85	70	65	70	58	5	False
46	41	Zubat	Poison	Flying	40	45	35	30	40	55	1	False
631	570	Zorua	Dark	NaN	40	65	40	80	40	65	5	False
632	571	Zoroark	Dark	NaN	60	105	60	120	60	105	5	False
...	...	...	...	...	...	...	...	...	...	...	...	...
393	359	AbsolMega Absol	Dark	NaN	65	150	60	115	60	115	3	False
392	359	Absol	Dark	NaN	65	130	60	75	60	75	3	False
68	63	Abra	Psychic	NaN	25	20	15	105	55	90	1	False
511	460	AbomasnowMega Abomasnow	Grass	Ice	90	132	105	132	105	30	4	False
510	460	Abomasnow	Grass	Ice	90	92	75	92	85	60	4	False

800 rows × 12 columns

Sort values based on two columns and arrange each column in ascending or descending order.

```python
df.sort_values(['Type 1','HP'], ascending=[1,0]) #first one ascending and second one descending
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
520	469	Yanmega	Bug	Flying	86	76	86	116	56	95	4	False
698	637	Volcarona	Bug	Fire	85	60	65	135	105	100	5	False
231	214	Heracross	Bug	Fighting	80	125	75	40	95	85	2	False
232	214	HeracrossMega Heracross	Bug	Fighting	80	185	115	40	105	75	2	False
678	617	Accelgor	Bug	NaN	80	70	40	100	60	145	5	False
...	...	...	...	...	...	...	...	...	...	...	...	...
106	98	Krabby	Water	NaN	30	105	90	25	25	50	1	False
125	116	Horsea	Water	NaN	30	40	70	70	25	60	1	False
129	120	Staryu	Water	NaN	30	45	55	70	55	85	1	False
139	129	Magikarp	Water	NaN	20	10	55	15	20	80	1	False
381	349	Feebas	Water	NaN	20	15	20	10	55	80	3	False

800 rows × 12 columns

Filtering data:

Filtering the columns based particular conditions.

```python
df.loc[(df['Type 1']=='Grass') & (df['Type 2']=='Poison')]
```

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	94	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	122	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	162	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	180	80	100	123	122	120	80	1	False
48	43	Oddish	Grass	Poison	95	45	50	55	75	65	30	1	False
49	44	Gloom	Grass	Poison	125	60	65	70	85	75	40	1	False
50	45	Vileplume	Grass	Poison	155	75	80	85	110	90	50	1	False
75	69	Bellsprout	Grass	Poison	125	50	75	35	70	30	40	1	False
76	70	Weepinbell	Grass	Poison	155	65	90	50	85	45	55	1	False
77	71	Victreebel	Grass	Poison	185	80	105	65	100	70	70	1	False
344	315	Roselia	Grass	Poison	110	50	60	45	100	80	65	3	False
451	406	Budew	Grass	Poison	70	40	30	35	50	70	55	4	False
452	407	Roserade	Grass	Poison	130	60	70	65	125	105	90	4	False
651	590	Foongus	Grass	Poison	124	69	55	45	55	55	15	5	False
652	591	Amoonguss	Grass	Poison	199	114	85	70	85	80	30	5	False

To retain old index:

```python
new_df=df.loc[(df['Type 1']=='Grass') & (df['Type 2']=='Poison') & (df['HP']>60)]
new_df=new_df.reset_index() #retains old index just in case
new_df
```

	index	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	2	3	Venusaur	Grass	Poison	162	80	82	83	100	100	80	1	False
1	3	3	VenusaurMega Venusaur	Grass	Poison	180	80	100	123	122	120	80	1	False
2	50	45	Vileplume	Grass	Poison	155	75	80	85	110	90	50	1	False
3	76	70	Weepinbell	Grass	Poison	155	65	90	50	85	45	55	1	False
4	77	71	Victreebel	Grass	Poison	185	80	105	65	100	70	70	1	False
5	651	590	Foongus	Grass	Poison	124	69	55	45	55	55	15	5	False
6	652	591	Amoonguss	Grass	Poison	199	114	85	70	85	80	30	5	False

To not retain old index, drop=True.

Use the ~ operator to remove rows based on a certain condition.

```
df.loc[~df['Name'].str.contains('Mega')] #all names that dont contain mega
```

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	94	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	122	60	62	63	80	80	60	1	False

To make changes based on condition:

```
df.loc[df['Type 1']=='Flamer', 'Type 1']='Fire'
df
```

To change columns based on conditions, the columns that needs to be changed is the columns that we specify as a list:

```
df.loc[df['HP']>45, ['Generation','Legendary']]=['Test 1', 'Test 2']
df
```

	#	Name	Type 1	Type 2	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	60	62	63	80	80	60	Test 1	Test 2
2	3	Venusaur	Grass	Poison	80	82	83	100	100	80	Test 1	Test 2
3	3	VenusaurMega Venusaur	Grass	Poison	80	100	123	122	120	80	Test 1	Test 2
4	4	Charmander	Fire	NaN	39	52	43	60	50	65	1	False

To group columns based on a type and get the values in the other columns based on rows that contain this information:

```
df.groupby(['Type 1']).mean().sort_values('Defense', ascending=False)
```

	#	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
Type 1									
Steel	442.851852	65.222222	92.703704	126.370370	67.518519	80.629630	55.259259	3.851852	0.148148
Rock	392.727273	65.363636	92.863636	100.795455	63.340909	75.477273	55.909091	3.454545	0.090909

To concat two dataframes:

Here, we use dictionaries to create dataframes, with keys being the column headers and the values in the list being the column values.

```
df1 = pd.DataFrame ({'stud_roll':[101,102], 'stud_name': ['rk','pk']})
df2 = pd.DataFrame ({'stud_roll':[103,104], 'stud_name': ['ak','mk']})
```

```
df3 = pd.concat ([df1,df2])
```

```
df3
```

	stud_roll	stud_name
0	101	rk
1	102	pk
0	103	ak
1	104	mk

We say, axis=0 to concatenate them across rows and axis=1, to concatenate them across columns. By default, it has axis=0

Apply function to apply the logic of the function to the columns in dataframe:

```
def my_add(value):
    print(value)


df.apply(my_add,axis=1)
```

To apply the function to each element on the dataframe.

```
def my_add(value):
    print(value)

df.applymap(my_add)
```

Pivot table:

```
pd.pivot_table(index='col', datam aggfunc='mean')
```

Based on the index column, it gives you the summary as mean of the other columns for each category in the index column.

For example:

Here, index is brand and based on each brand we get the mean mileage, price and year here.

	mileage	price	year
brand			
acura	120379.666667	7266.666667	2010.333333
audi	118091.000000	13981.250000	2011.250000
bmw	47846.411765	26397.058824	2014.470588
buick	37926.846154	19715.769231	2016.000000
cadillac	40195.900000	24941.000000	2014.900000
chevrolet	65124.461279	18669.952862	2015.616162
chrysler	73004.000000	13686.111111	2014.777778
dodge	44184.863426	17781.988426	2017.291667
ford	52084.304453	21666.888259	2016.762753
gmc	58548.738095	10657.380952	2014.904762

We can also create our own aggfunc and then call it as the parameter value in the pivot_table. Here, the values have the column that we need to apply the sum or average to based on the index column. So, for each value in the index column, we calculate the sum or average of the value column or both based on the aggfunc.

```
def average_vals(price):
    return np.mean(price)
pd.pivot_table(data, index='brand', values='price', aggfunc=[sum,average_vals]).head(10)
```

Source for Pandas: https://www.youtube.com/watch?v=vmEHCJofslg