In [5]:
```
%%html

<h1>Code from Python For Data Analysis book by Wes McKinney</h1>
```

Code from Python For Data Analysis book by Wes McKinney

In [4]:
```
%%html

<h1>List of chapters in this book:<h1>
<h2>Chapter 2  - Python language basics</h2>
<h2>Chapter 3 - Built in data structures, functions, and files</h2>
<h2>Chapter 4  - Numpy basics: Arrays and Vectorized Computation</h2>
```

List of chapters in this book:

Chapter 2 - Python language basics

Chapter 3 - Built in data structures, functions, and files

Chapter 4 - Numpy basics: Arrays and Vectorized Computation

In [2]:
```
%%html

<h2>Chapter 2  - Python language basics</h2>
```

Chapter 2 - Python language basics

In [31]:
```
# Python statement doesnot need to use semicolon to terminate. However, semicolon can be
# used to seperate multiple statements on a single line.
```

```
a=1;c=3;b=1
print(c)
```

```
3
```

In [32]:

```
# EVerything in python is a python object - string, data structures, function, class etc.
```

In [35]:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have the same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison returns True because x is equal to y
```

```
True
False
True
```

In [4]:

```
# Type casting
val = "3.14"
print(type(val))
val = float(val)
print(type(val))
val = bool(val) # most non zero value will be true for boolean casting and false for zero.
print(val)
```

```
<class 'str'>
<class 'float'>
True
```

In [8]:

```
# Date and times

from datetime import datetime, date, time
```

```
dt = datetime(2011,10,29,20,30,21)

print(dt.date())
print(dt.day)
print(dt.minute)
```

```
2011-10-29
29
30
```

In [9]:
```
%%html

<h2>Chapter 3  - Built in data structures, functions, and files</h2>
```

Chapter 3 - Built in data structures, functions, and files

In [14]:
```
%%html

<h3>Tuple</h3>
```

Tuple

In [12]:
```
tup = (1,2,3)

#Immuatable
print(tup[0])
tup[0] =4
```

```
1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 8
      6 #Immuatable
      7 print(tup[0])
----> 8 tup[0] =4

TypeError: 'tuple' object does not support item assignment
```

In [16]:
```
# In many context, paranenthesis can be omitted
tup = 1,2,3

# Tuple casting
tup = tuple('string')
print(tup)
```

```
('s', 't', 'r', 'i', 'n', 'g')
```

In [18]:
```
# Nested tuple
tup = (4,5,6),(7,8)
print(tup)
print(tup[0])
```

```
((4, 5, 6), (7, 8))
(4, 5, 6)
```

In [19]:
```
# tuple can hold mixed data formats
tup = (4,5,6),"hello",(5,6)
```

In [20]:
```
print(tup)
```

```
((4, 5, 6), 'hello', (5, 6))
```

In [24]:
```
# Unpacking tuples
tup = (1,2,3)
a,b,c = tup
print(a)

tup = (4,5,(6,7))
a,b,(c,d) = tup
print(c)
```

```
1
6
```

In [28]:
```
# special unpacking
values = (1,2,3,4,5)

a,b,*rest = values
print(rest)
```

```
# Tuple has a count method that can count the occurrence of a value.
print(values.count(1))
```

```
[3, 4, 5]
1
```

In [29]:

```
%%html

<h3>List</h3>
```

List

In [31]:

```
# List is a mutable object

li = [1,2,3]
print(type(li))
```

```
<class 'list'>
```

In [45]:

```
# we can append a new value to the end of list using append function
# we can insert a new value to a specific location
li = ['f', 'g', 'h']
li.append('i')
print(li)
li.insert(0,'e')
print(li)

# Inverse operation of the insert is pop - f is gone
li.pop(1)
print(li)

# We can remove by value
li.remove('i')
print(li)

# Checking a value contain in list
print('g' in li)
print('f' in li)
```

```
['f', 'g', 'h', 'i']
['e', 'f', 'g', 'h', 'i']
['e', 'g', 'h', 'i']
['e', 'g', 'h']
True
False
```

In [46]:

```
# We can concatenate two list using + operator. Also, we can extend method.
li2=[1,2]
print(li+li2)
li.extend([3,4])
print(li)
```

```
['e', 'g', 'h', 1, 2]
['e', 'g', 'h', 3, 4]
```

In [48]:

```
# Sorting
a =[4,5,2,3,1]
a.sort()
print(a)
```

```
[1, 2, 3, 4, 5]
```

In [57]:

```
# Slicing

str = "hello!"
li = list(str)
print(li)
# 0   1   2   3   4   5 => +ive indices
# h   e   l   l   o   !
#-6  -5  -4  -3  -2  -1 => -ve indices
# An important point to remember is that the start index is included and the stop index is not. Another point negat
print(li[1:2])
print(li[-6:-4])

# Third option in slicing - Take every other element, -v for revering the list
print(li[::2])
print(li[::-1])
```

```
['h', 'e', 'l', 'l', 'o', '!']
['e']
['h', 'e']
['h', 'l', 'o']
['!', 'o', 'l', 'l', 'e', 'h']
```

In [58]:
```
%%html

<h3>Dictionary</h3>
```

Dictionary

In [59]:
```
dict = {}
```

In [61]:
```
# It has key value pair and it is mutable object
dict ={"a":1,"b":4}
print(dict)
# It can have mixed data type values
```

```
{'a': 1, 'b': 4}
```

In [63]:
```
# Keys & value
print(dict.keys())
print(dict.values())
```

```
dict_keys(['a', 'b'])
dict_values([1, 4])
```

In [69]:
```
#defaultdict

words = ["apple", "bat", "bar", "atom", "book"]
by_letter = {}
for word in words:
    letter = word[0]
    if letter not in by_letter:
       by_letter[letter] = [word]
    else:
       by_letter[letter].append(word)

print(by_letter)
```

```python
# We can replace this by defaultdict which add default value

from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    letter = word[0]
    by_letter[letter].append(word)
print(by_letter)
```

```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
defaultdict(<class 'list'>, {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']})
```

In [4]:
```
%%html

<h3>Set</h3>
```

Set

In [8]:
```python
# It is an unorder collection of unique elements
a = {1,2,2,3,4,3}
print(a)

# We can do set operations: uniio, intersection, difference , & symmetric_difference
a = {1,2,3,4,5}
b = {3,4,5,6,7,8}
a.union(b)

a = {1,2,3,4,5}
b = {3,4,5,6,7,8}
a.intersection(b)

a = {1,2,3,4,5}
b = {3,4,5,6,7,8}
a.difference(b)

a = {1,2,3,4,5}
b = {3,4,5,6,7,8}
a.symmetric_difference(b)
```

```
{1, 2, 3, 4}
```

Out[8]:
```
{1, 2, 6, 7, 8}
```

In [9]:
```
%%html

<h3>Built-in Sequence Functions</h3>
```

Built-in Sequence Functions

In [11]:
```
# Enumerate will return (i, values) of a sequence
seq = [5,3,7,9,8]
for i, value in enumerate(seq):
    print("{}:{}".format(i, value))
```

```
0:5
1:3
2:7
3:9
4:8
```

In [14]:
```
# Zip   pairs up the elements of a number of lists, tupes, or other sequences to create a list of tuples

seq1 = ['foo', 'bar', 'baz']
seq2 = ['one','two', 'three']

zipped = zip(seq1, seq2)
print(list(zipped))
```

```
[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

In [20]:
```
seq3 = [False, True]
# As seq3 has only two elements other sequences have three elements.nuber of elements in the zip result is determin
zipped = zip(seq1, seq2, seq3)
print(list(zipped))

# Use of Enumerate with zip

for index, (a,b) in enumerate(zip(seq1, seq2)):
    print(f"{index}: {a},{b}")
```

```
[('foo', 'one', False), ('bar', 'two', True)]
0: foo,one
1: bar,two
2: baz,three
```

In [21]:
```
%%html

<h3>List, set, dictionary comprehentions</h3>
```

List, set, dictionary comprehentions

In [26]:
```
# List comprehension
#li = [expr for value in seq if condition]

strings= ["a", "as", "bat", "car", "dove", "python"]
li = [x.upper() for x in strings]
print(li)

li = [x.upper() for x in strings if len(x) > 2]
print(li)
```

```
['A', 'AS', 'BAT', 'CAR', 'DOVE', 'PYTHON']
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

In [30]:
```
# Dictionary comprehension
# dict ={key-expr: value-expr for value in collection if condition}
strings= ["a", "as", "bat", "car", "dove", "python"]
dict  = {index:value for index, value in enumerate(strings)}
print(dict)
```

```
{0: 'a', 1: 'as', 2: 'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

In [31]:
```
# Set comprehension
#set = {expr for value in collection if condition}
```

In [33]:
```
# Nested list comprehension
some_tuples = [(1,2,3), (4,5,6),(7,8,9)]
# We can flatten this by nested list comprehension
flattened = [x for tup in some_tuples for x in tup]
print(flattened)
```

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

In [36]:
```
%%html

<h3>Functions</h3>
```

Functions

In [2]:
```
# python function can return multiple values
# return a,b,c

# Anonymus lambda function
short_func = lambda x:x*2
print(short_func(2))
```

```
4
```

In [10]:
```
#A generator is a convenient way similar to writing a normal function, to construct a new iterable objects.
#Wheras normal functions execute and return a single result at a time,
#generator can return a sequence of multiple values of pausing and resuming execution each time the generatoris use

def squares(n=10):
    print("Generating")
    for i in range(1, n+1):
        yield i**2

gen = squares()
gen
for x in gen:
    print(x)
```

```
Generating
1
4
9
16
25
36
49
64
81
100
```

In [ ]:
```
# need to learn more about Generator
```

In [11]:
```
%%html

<h2>Chapter 4  - Numpy basics: Arrays and Vectorized Computation</h2>
```

Chapter 4 - Numpy basics: Arrays and Vectorized Computation

In [89]:
```
import numpy as np
my_arr = np.arange(10)

# It takes only less memory comapred to built-in python sequence
# Numpy is faster than regular python code because it's C-based algorithm -> numpy-based algorithm are generally 10
# Numpy operation perform complex computation on entire array without the need for python for loops, which can be s
my_arr * 2
```

Out[89]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [18]:
```
%%html

<h3>The numpy ndarray: a multidimensional array object</h3>
```

The numpy ndarray: a multi dimennsional array object

In [19]:
```
# List to np array using np.array()
```

```
data = np.array([[1.5,-0.1,3],[0,-3,6.5]])
```

In [20]:
```
data
```

Out[20]:
```
array([[ 1.5, -0.1,  3. ],
       [ 0. , -3. ,  6.5]])
```

In [36]:
```
# Mathematical operations
data*10
data+data
1/data
data**2
data.shape
data.dtype

# logical operation
data > data
data == data
```

```
/tmp/ipykernel_140/1500792984.py:4: RuntimeWarning: divide by zero encountered in divide
  1/data
```

Out[36]:
```
array([[ True,  True,  True],
       [ True,  True,  True]])
```

In [28]:
```
# Creating an empty array with zeros and Ones
# one dimension
np.zeros(10)
```

Out[28]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [30]:
```
# Higher dimension
np.zeros((3,2))
```

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

In [31]:
```
np.ones(10)
```

Out[31]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [32]:
```
#There is an empty function, but the value can contain non zero garbage values- np.empty()
# np.arange is similar to python range function
np.arange(10)
```

Out[32]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [ ]:
```
#Numpy data types: int, float, complex, bool, object(python object type- value can be python object), string_, unic
```

In [37]:
```
%%html

<h3>Basic indexing & slicing</h3>
```

Basic indexing & slicing

In [42]:
```
arr = np.arange(10)
arr
```

Out[42]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [57]:
```
# The key distinction from python list is that array slices are views on the original array. so modification on the
arr[5:8] = 10
arr
arr = np.arange(10)
# same as list, for the +ve & -ve indexing, start index is inclusive whereas end index is not inclusive:
arr[5:8]

arr[-6:-3]
```

Out[57]:
```
array([4, 5, 6])
```

In [79]:
```
# For tow dimenaional array

#np_array[row, column]
arr = np.arange(9)
```

```
arr = np.reshape(arr, (3,-1))
arr
```

Out[79]:

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

In [84]:

```
arr[2,2] # 8
arr[0:2,0:1]
```

Out[84]:

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

In [88]:

```
arr[:,:2]
```

Out[88]:

```
array([[0, 1],
       [3, 4],
       [6, 7]])
```

In [91]:

```
%%html

<img src='images/slicing.png', width=300/>
```

No description has been provided for this image

In [92]:

```
%%html
<h3>Fancy indexing</h3>
```

Fancy indexing

In [94]:

```
arr = np.zeros((8,4))
arr
```

Out[94]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [95]:
```
for i in range(8):
    arr[i] = i
```

In [96]:
```
arr
```

Out[96]:
```
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

In [97]:
```
arr[[4,3,0,6]] # this gives the rows in the specific order mentioned
```

Out[97]:
```
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

In [99]:
```
# We can use -ve indices to select the rows from the end
arr[[-1,-3,-5]]
```

Out[99]:
```
array([[7., 7., 7., 7.],
       [5., 5., 5., 5.],
       [3., 3., 3., 3.]])
```

In [102…
```
# We can use rows and columns
arr[[1,2,3],[0,3,1,]]
```

Out[102… array([1., 2., 3.])

In [104…
```
#Transposing the array
arr.T
```

Out[104…
```
array([[0., 1., 2., 3., 4., 5., 6., 7.],
       [0., 1., 2., 3., 4., 5., 6., 7.],
       [0., 1., 2., 3., 4., 5., 6., 7.],
       [0., 1., 2., 3., 4., 5., 6., 7.]])
```

In [105…
```
# inner matrix produc is using np.dot and other way to do matrix multiplcation is using @ -> arr.T @ arr
```

In [107…
```
# Pseudo random number generation

samples = np.random.standard_normal(size =(4,4))
samples
```

Out[107…
```
array([[-0.56151294, -0.7172941 , -0.17567785,  0.87164726],
       [-1.80189301, -0.02615516,  0.39550439, -1.87444707],
       [ 1.44679248, -1.40390773,  2.30413065,  0.10003334],
       [ 1.32845772,  0.45019542,  1.4233421 , -0.78065266]])
```

In [108…
```
%%html
<h3>Universal functions: Fast Element-wise Array functions</h3>
```

Universal functions: Fast Element-wise Array functions

In [112…
```
print(arr)
np.add(arr, 1)
```

```
[[0. 0. 0. 0.]
 [1. 1. 1. 1.]
 [2. 2. 2. 2.]
 [3. 3. 3. 3.]
 [4. 4. 4. 4.]
 [5. 5. 5. 5.]
 [6. 6. 6. 6.]
 [7. 7. 7. 7.]]
```

Out[112…

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

In [14]:

```
%%html
Some universal functions
<img src='images/ufunc.png', width=500/>
```

Some universal functions

No description has been provided for this image

In [1]:

```
%%html
<h3>Expressing conditional logic as array operations</h3>
```

Expressing conditional logic as array operations

In [7]:

```
import numpy as np
rng = np.random.default_rng(seed=12345)
arr = rng.standard_normal((4,4))
arr
```

Out[7]:

```
array([[-1.42382504,  1.26372846, -0.87066174, -0.25917323],
       [-0.07534331, -0.74088465, -1.3677927 ,  0.6488928 ],
       [ 0.36105811, -1.95286306,  2.34740965,  0.96849691],
       [-0.75938718,  0.90219827, -0.46695317, -0.06068952]])
```

In [8]:

```
arr > 0
```

Out[8]:

```
array([[False,  True, False, False],
       [False, False, False,  True],
       [ True, False,  True,  True],
       [False,  True, False, False]])
```

In [12]:
```python
# The second and third term is scalar values which replaces the original array with scalar values depends on the co
np.where(arr>0, 2, -2)
```

Out[12]:
```
array([[-2,  2, -2, -2],
       [-2, -2, -2,  2],
       [ 2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

In [13]:
```python
# We can also uses np.where to get the values of any of the two arrays depends on the condition

xarr = np.array([1.1,1.2,1.3,1.4,1.5])
yarr = np.array([2.1,2.2,2.3,2.4,2.5])
cond = np.array([True, True, False, False, True])
np.where(cond, xarr,yarr)
```

Out[13]:
```
array([1.1, 1.2, 2.3, 2.4, 1.5])
```

In [15]:
```
%%html
<h3>Mathematical & statistical methods</h3>
```

Mathematical & statistical methods

In [30]:
```python
arr = rng.standard_normal((5,4))
```

In [18]:
```python
arr.mean()
#or using universal function
np.mean(arr)
```

Out[18]:
```
0.17933634979615845
```

In [19]:
```python
arr.sum()
```

Out[19]:
```
3.586726995923169
```

In [20]:
```python
# sum along the rows
arr.sum(axis=0)
```

```
Out[20]: array([ 2.71870476,  0.30188842, -0.49975489,  1.0658887 ])
```

```
In [21]: # sum slong the columns
         arr.sum(axis=1)
```

```
Out[21]: array([ 1.50701272,  0.30393615, -1.13399399,  5.9488939 , -3.03912178])
```

```
In [22]: #There are other array statistical methods
         #std, var, min, max,argmin,argmax,cumsum,cumprod
```

```
In [32]: # axis =0  means operation along the rows. It means that if we have 4 columns & n rows. we do the operation along t
         # axis = 1 means operation along the columns. It means that if we have 4 colunms & n rows. we do the operation alon
```

```
In [33]: # Sorting
         arr = rng.standard_normal((5,4))
         print(arr)
         arr.sort(axis=0) # soring along the rows
         arr
         # We can also use universal function np.sort(arr)
```

```
[[-0.35947965 -0.74864398 -0.96547891  0.36003466]
 [-0.24455253 -1.99585661 -0.15524762  1.06383087]
 [-0.27517157 -1.85333593 -0.12434193  0.78497452]
 [ 0.2019986  -0.42807444  1.8482889   1.89995289]
 [-0.09842503  0.81344544  0.39249439  0.7814429 ]]
```

```
Out[33]: array([[-0.35947965, -1.99585661, -0.96547891,  0.36003466],
                [-0.27517157, -1.85333593, -0.15524762,  0.7814429 ],
                [-0.24455253, -0.74864398, -0.12434193,  0.78497452],
                [-0.09842503, -0.42807444,  0.39249439,  1.06383087],
                [ 0.2019986 ,  0.81344544,  1.8482889 ,  1.89995289]])
```

```
In [34]: %%html
         <h3>File input and output with arrays</h3>
```

File input and output with arrays

```
In [37]: #np.load(), np.save("name", arr) => saved in .npy format
         # uncompresses saving np.savez("name.npz", arr), you can also save multiple arrays np.savez("name.npz", a=arr1, b=a
```

```
# arr['a']
```

In [47]:
```
%%html
<h3>Linear Algebra</h3>
Matrix multiplication - dot product & element-wise multiplication
<br/>
<br/>
<img src='images/dot_element_wise.png', width=500/>
```

Linear Algebra

Matrix multiplication - dot product & element-wise multiplication

No description has been provided for this image

In [53]:
```
# dot product
x = np.array([[1.,2.,3.],[4.,5.,6.]])
y = np.array([[6.,23.],[-1,7],[8,9]])
np.dot(x,y)
```

Out[53]:
```
array([[ 28.,  64.],
       [ 67., 181.]])
```

In [54]:
```
# np.matml & @ are same -> element-wise matrix multiplication
np.matmul(x,y)
```

Out[54]:
```
array([[ 28.,  64.],
       [ 67., 181.]])
```

In [55]:
```
x@y
```

Out[55]:
```
array([[ 28.,  64.],
       [ 67., 181.]])
```