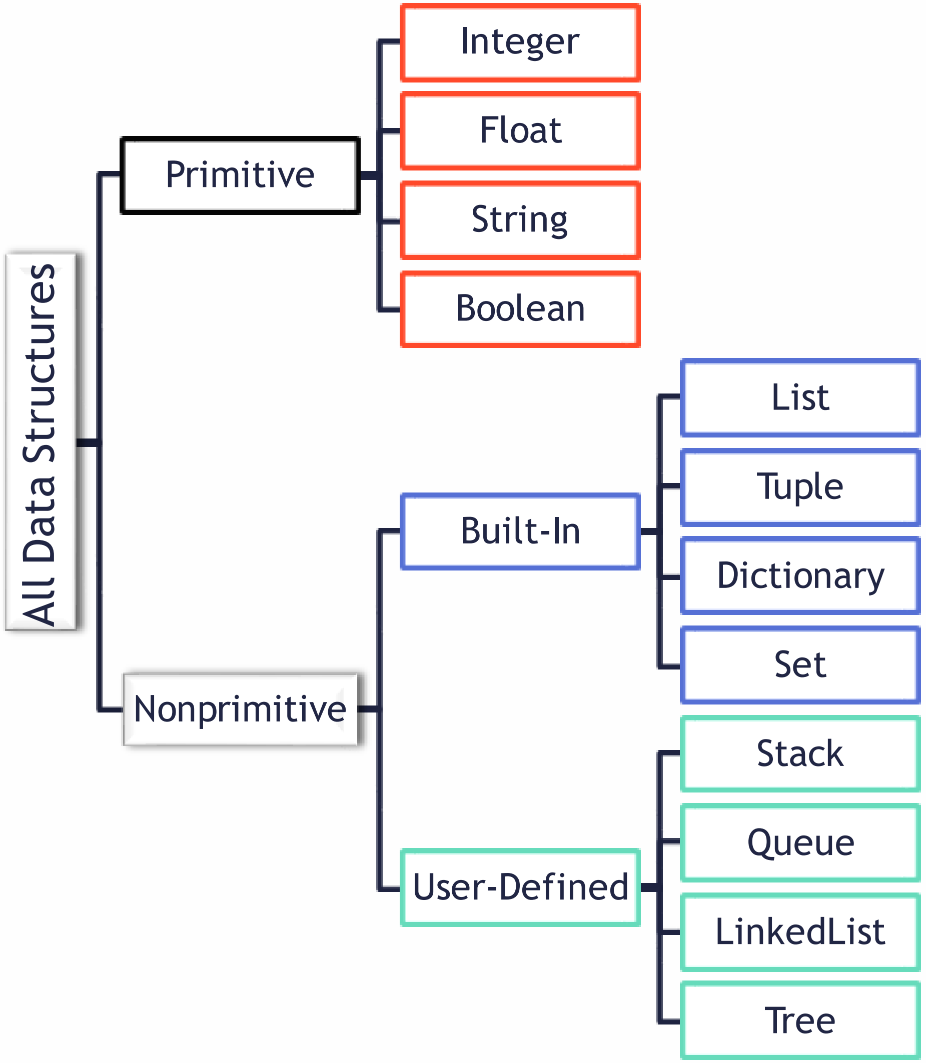
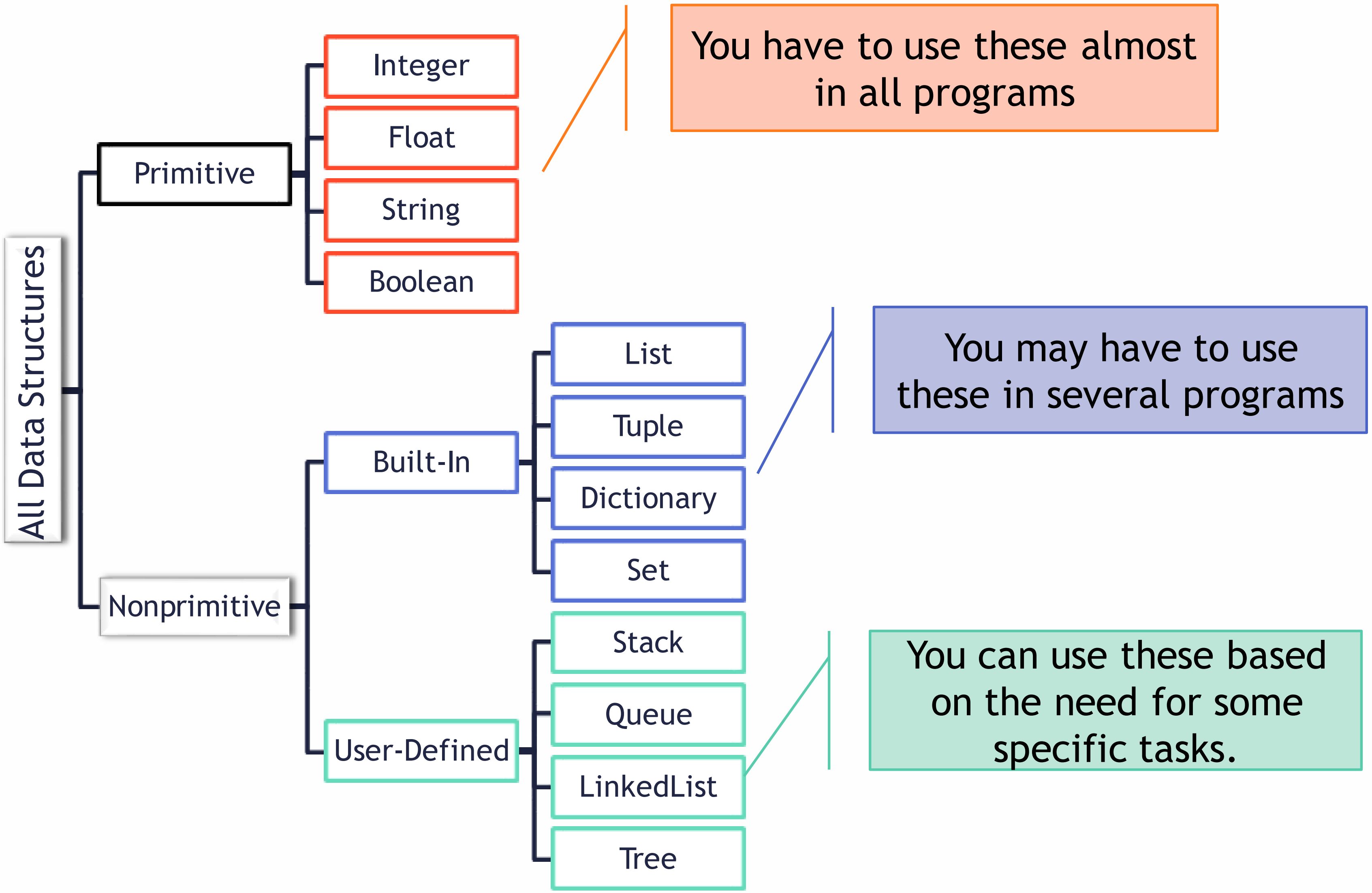
Python Basics – Data Structures

Contents

Python Data Structures



Python Data Structures



Integers and Float

- Numbers: integers & floats

```
age=30
income=1250.567

print(type(age))
print(type(income))

print("Age - datatype", type(age))
print("Income - datatype", type(income))
```

Strings

- Strings are amongst the most popular types in Python.
- There are a number of methods or built-in string functions

Defining Strings

```
name="Sheldon"
print(name)
```

Accessing strings

```
print(name[0])
print(name[0:3])
print(name[3:6])
print(name[3:4])
```

Strings – Slicing

```
message="Python 5 day course - Statinfer"
print(message)
print(message[0:4])
print(len(message))
```

String Concatenation

```
First_name="Sheldon"
Last_name="Cooper"
Name= First_name + Last_name

Name1= First_name +" "+ Last_name
print(Name1)
print(Name1*10)
```

Boolean

- Used to store and return the values True an False.
- Useful for comparison operations

```
a=5>6
print(a)
print(type(a))
```

Boolean

What is the expected output of this code

```
a=5>6
if a:
  print(10)
else:
  print(15)
```

List

- A sequence or a collection of elements.
- A list looks similar to an array, but not array
- Lists, can contain any number of elements. The elements of a list need not be of the same type

List Creation

Creating a list

```
cust=["Sheldon","leonard", "penny"]
Age=[30, 31, 27]

print(cust)
print(Age)
```

Accessing list elements

```
print(cust[0])
print(Age[1])
```

Combining two lists

```
cust_1=["Amy", "Raj"]
Final_cust=cust + cust_1
print(Final_cust)
print(len(Final_cust))
```

Appending an item to the list

```
Final_cust.append("Howard")
print(Final_cust)
```

Delete – based on value (a.k.a – remove)

- The remove() method removes the first matching element (which is passed as an argument) from the list.

```
Final_cust.remove("penny")
print(Final_cust)
```

- What is the expected result for the below code?

```
Age=[30, 31, 27, 30, 48]
Age.remove(30)
print("After removing 30 ==>", Age)
```

Delete – based on index (a.k.a – pop)

- The pop() method removes the item at the given index from the list and returns the removed item.

```
sales=[300, 350, 200, 250, 300]
sales.pop(0)
print("sales after pop ==>", sales)
```

What is the expected result for the below code?

```
sales.pop(0)
print("sales after second pop ==>", sales)
```

Sort the list items

```
Loans=[3, 0, 2, 6, 20]
Loans.sort()
print(Loans)
```

```
Loans.sort(reverse=True)
print(Loans)
```

Negative Index

List	300	350	200	250	400
Usual Index	0	1	2	3	4
Negative Index	-5	-4	-3	-2	-1

```
sales=[300, 350, 200, 250, 400]
print(sales[-1])

print(sales[-1],sales[-2], sales[-3], sales[-4], sales[-5])
```

Slicing a list

What is the expected result for the below code?

```
Age=[15, 44, 38, 19, 36, 25]
print(Age[2:5])
print(Age[2:10])
print(Age[2:])
print(Age[:4])
print(Age[-4:-2])
print(Age[:-4])
print(Age[-4:])
```

Tuples

- Tuple is one of 4 built-in data types in Python
- Collection of items stored in a single variable
- Items inside a tuple can be homogeneous or heterogenous.
- Tuples are sequences, just like lists. There are some major differences between tuples and lists.
- Tuples are immutable. We can NOT update or change the values of tuple elements

Tuples – Defining and Accessing

```
custd_id=("c00194", "c00195", "c00198")
type(custd_id)

rank=(1, 46, "NA", 5, 8)
type(rank[1])
print(type(rank[1]),type(rank[2]))
```

Tuple packing

```
region= "E", "W", "N", "S"
type(region)
```

- The above operation is called tuple packing. "E", "W", "N", "S" are packed together.
- The reverse is also possible.

```
r1, r2, r3, r4= region
print(r1, r4)
print(type(r1))
```

Tuples vs. Lists – Difference

- Tuples are immutable. Seal the data in the tuple, totally prevent it from being modified at any stage of the development.
- Tuples are faster and use less memory than lists.

```
custd_id_t=("c00194", "c00195", "c00198")
custd_id_l=["c00194", "c00195", "c00198"]
custd_id_l[1]="c00176"
custd_id_t[1]="c00176"
```

Tuples utilize less memory

- A list is mutable. Python needs to allocate more memory than needed to the list. This is called over-allocating.
- Meanwhile, a tuple is immutable. When the system knows that there will be no modifications, memory allocations will be very efficient.

```
from sys import getsizeof

custd_id_t=("c00194", "c00195", "c00198")
custd_id_l=["c00194", "c00195", "c00198"]

print("tupple size ", getsizeof(custd_id_t))
print("list size ", getsizeof(custd_id_l))
```

Working with a tuple is faster

- Time that needs to copy a list and a tuple 1 billion times

```
from timeit import timeit
timeit("list(['c00194', 'c00195', 'c00198'])", number=10000000)
```

```
2.224825669999973
```

```
timeit("tuple(('c00194', 'c00195', 'c00198'))", number=10000000)
```

```
0.975565370001023
```

Tuple vs List – Which one to use?

- We have to use both.
- Sometimes we need mutability and modifications to the variables, in some other cases, we need speed and utilize less memory.

Tuple vs List – Which one to use?

- Example-1
  - You are writing a function. You need to take the input parameters from the user and perform the calculations inside the function to return some result.
  - How would you like to store the input parameters ? Inside a tuple? Or inside a list?
- Example-2
  - Store all the columns inside a variable and return the first two columns. Later, rename the first column and pass it on to the next step.
  - How would you like to store the column names? Inside a tuple? Or inside a list?

Dictionaries

- Dictionaries have two major element types key and Value.
- Dictionaries are collection of key value pairs
- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.
- Keys are unique within a dictionary

```
city={0:"L.A", 9:"P.A", 7:"FL"}
type(city)

print(city)
print(city[1])
print(city[0])
```

Dictionaries

- In dictionary, keys are similar to indexes. We define our own preferred indexes in dictionaries

```
cust_profile={"C001":"David", "C002":"Bill", "C003":"Jim"}
print(cust_profile[0])
print(cust_profile[C001])
print(cust_profile["C001"])

#Updating values
cust_profile["C002"]="Tom"
print(cust_profile)
```

Dictionaries

Updating keys in dictionary

Delete the key and value element first and then add new element

```
#Updating Keys
#C001 ---> C009

#Delete + Add

del(cust_profile["C001"])
print(cust_profile)

cust_profile["C009"]="David"
print(cust_profile)
```

Dictionaries

- Fetch all keys and all values separately

```
city.keys()
city.values()
```

Iterating in a Dictionary

```
Employee = {"Name": "Tom", "Emp_id": 198876, "Age": 29, "salary":25000
,"Company":"FB"}
```

- What is the expected output?

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

- What is the expected output?

```
for x in Employee.values():
    print(x)
```

Iterating in a Dictionary

- What is the expected output?

```
for x in Employee.keys():
    print(x,Employee[x])
```

- You can also use items.

```
for x in Employee.items():
    print(x)
```

Sets

- Sets are an **<u>unordered</u>** collection of unique elements.
- Sets are mutable but can hold **<u>only</u>** **<u>unique</u>** values in the dataset.
- Set operations are similar to the ones used in arithmetic.
- There is **<u>no index</u>** attached to the elements of the set
- We cannot directly access any element of the set by the index.

Operations on Sets

- Either use {} or set() function for sets certation

```
products = {"Phone", "T.V", "Tablet", "Laptop", "Fridge", "Camera"}
type(products)

products1 = set(["Phone", "T.V", "Tablet", "Laptop", "Fridge"])
type(products)
```

Careful with empty set created using {}

```
products2={}
type(products2)
```

This is not an empty set

```
dict
```

```
products2=set([])
type(products2)
```

Use set function insted

```
set
```

Operations on Sets

Set values are always unique

```
orders = set(["Phone", "Phone", "toys", "toys", "Camera", "Camera"])
print(orders)
```

```
{'toys', 'Camera', 'Phone'}
```

Operations on Sets

- Sets Union

```
union_set=products|orders
print(union_set)
```

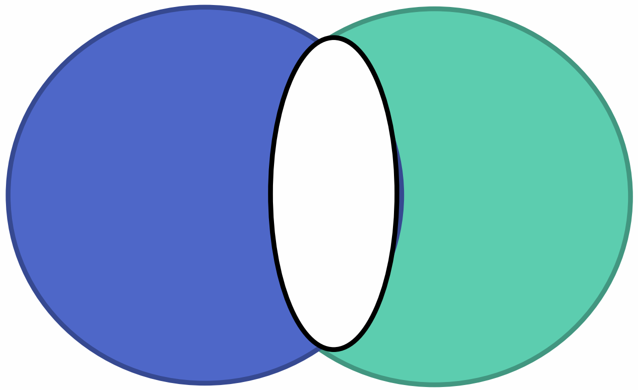
- Sets Intersection

```
intersection_set=products & orders
print(intersection_set)
```

- Symmetric difference

```
Symmetric_diff=products ^ orders
print(Symmetric_diff)
```

Symmetric difference



NumPy Package

Contents

NumPy

- Stands for Numerical Python
- NumPy helps us to create and work with arrays in python
- NumPy is for fast operations on vectors and matrices, including mathematical, logical, shape manipulation, sorting, selecting.
- It is the foundation on which all higher level tools for scientific python packages are built

NumPy

- How to define arrays?
- What is the function name?

NumPy

```
import numpy as np
income = np.array([1200, 1300, 1400, 1500, 1600, 1700])

type(income)

print(income[0])

expenses=income*0.653
print(expenses)

savings=income-expenses
print(savings)
```

List vs Array

- What is the difference ?

Array is different from a List

```
list1=[1,2,3]
list2=[4,5,6]

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

What is the expected output of the below code?

```
list1+list2
```

```
arr1+arr2
```

Elements are Homogenous inside an array

```
c = np.array([[1, 2,3], [4, 'a', 6]])
print(c)
```

- What type of object is c?

N- Dimensional Arrays

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

b=np.array([[1, 2, 3], [4, 5, 6], [9, 9, 9]])
print(b)
```

Shape of array

- How to get the shape of arrays?
- How to extract only the number of dimensions?
- How to extract the total number of elements across all the dimensions?

Shape, Size and ndim

What is the expected output of the below code?

```
print("Shape", a.shape)
print("Size", a.size)
print("ndim", a.ndim)

print("Shape", b.shape)
print("Size", b.size)
print("ndim", b.ndim)
```

Arrays from data frames

```
import pandas as pd
bank=pd.read_csv("https://raw.githubusercontent.com/venkat
areddykonasani/Datasets/master/Bank%20Tele%20Marketing/ban
k_market.csv")

bank.info()

age_var=np.array(bank["age"])
type(age_var)
age_var.shape
```

Arrays from data frames

```
two_vars=np.array(bank[["age","balance"]])
type(two_vars)

two_vars.shape

two_vars.size
```

Indexing & Slicing

- How to access only first 10 elements of an array ?
- How to access only the last element of an array?
- How to access the elements at indexes 1, 9, 10

Indexing & Slicing

```
age_var=np.array(bank["age"])
age_var
```

- What is the expected output of the below code?

```
age_var[0]
age_var[0:10]
age_var[-1]
age_var[[1,9,10]]
```

Indexing & Slicing

- Take a two dimensional array, how to access the first row, first column?
- What is the output for the index notation `[0:2,1]`

Indexing & Slicing

```
two_vars=np.array(bank[["age","balance"]])
two_vars
```

What is the expected output of the below code?

```
two_vars[0,0]
two_vars[0,1]
two_vars[0:2,1]
two_vars[0:2,0]
two_vars[0:2,2]
two_vars[-1]
two_vars[-1, 0:2]
two_vars[-2, 0:2]
two_vars[:, 0]
two_vars[:, 1]
two_vars[0, :]
```

Boolean index

- How to filter select the values of an array based on a condition?

Boolean index

```
age_var=np.array(bank["age"])
age_var

condition=age_var<50
condition

new_age=age_var[condition]

print(age_var.shape)
print(new_age.shape)

#Mark age_var as 1 if condition is met
age_var[condition]=1
age_var
```

Initial Placeholders

- Create a 2dim array with 3rows and 4 columns and fill all the elements with zeros
- Create an array by taking the numbers between 10, 30. Keep the step size as 2.
- Create an array by dividing the space between 10 and 30 into 5 parts.

Initial Placeholders

```
np.zeros((3,4))
np.ones((2,3),dtype=np.int16)

np.arange(10,30,2)
np.arange(10,30,5)
np.arange(10,30,10)

np.linspace(10,30,2)
np.linspace(10,30,10)
np.linspace(10,30,20)
```

Random number generation

```
np.random.random(1)
np.random.random(30)
np.random.uniform(size=30)
np.random.normal(size=30)
np.random.random((2,3))
```

numpy.reshape()

```
a=np.random.uniform(size=30)
```

Can you re-shape the above array as a 2D array with 6 rows and five cols?

numpy.reshape()

```
a=np.random.uniform(size=30)
```

Can you re-shape the above array as a 2D array with 6 rows and five cols?

```
a.reshape(6,5)
```

numpy.reshape()

- reshape() doesn't change the shape - make a note of it.

```
print(a.shape)
print(a.reshape(6,5).shape)
```

numpy.reshape()

- Reshaping as a 3D array

```
a.reshape(3,2,5)
```

numpy.reshape()

- What if, we give wrong dimensions

```
a=np.random.uniform(size=30)
a.reshape(3,1)
```

numpy.reshape()

- What if, we give wrong dimensions

```
a=np.random.uniform(size=30)
a.reshape(3,1)
```

- What if, we want 3 rows and any number of columns.

numpy.reshape()

- You can use negative index for unknown dimension.

```
a.reshape(3,-1)
a.reshape(-1,3)
a.reshape(3,2,-1)
a.reshape(-1,2,3)
a.reshape(-1,2,15)
```

- You can only specify one unknown dimension

```
a.reshape(-1,-1,15)
```

- Flatten the array to one row

```
a.flatten()
```

max and min other functions

```
age_var=np.array(bank["age"])
age_var.max()
age_var.min()
age_var.mean()
age_var.std()
```

Index of max

- Consider this example
- Let this be output probabilities for multiclass classification output for 15 datapoints.
- We need to give only one class as output, the class with max probability.
- How to get the index of the max element?

```
rray([[0.58178245, 0.00234469, 0.97036937, 0.64034516],
       [0.44504406, 0.07178624, 0.50511309, 0.98527334],
       [0.40490749, 0.21520268, 0.66671445, 0.18926015],
       [0.99818906, 0.3702341 , 0.32152925, 0.33452479],
       [0.41693608, 0.99710111, 0.54760253, 0.98896868],
       [0.43080255, 0.6232379 , 0.60616554, 0.41871962],
       [0.57980182, 0.30218979, 0.48831486, 0.17218716],
       [0.38477543, 0.40937626, 0.60831249, 0.23314077],
       [0.24803288, 0.13615116, 0.38076504, 0.80648948],
       [0.64015809, 0.11270068, 0.67419178, 0.63834555],
       [0.0711749 , 0.72234198, 0.83176517, 0.26625898],
       [0.92572131, 0.31060026, 0.39069662, 0.72056121],
       [0.76615175, 0.75503287, 0.57738505, 0.3122232 ],
       [0.67315251, 0.4502434 , 0.64605349, 0.47127994],
       [0.08272156, 0.53467679, 0.29487162, 0.16681734]])
```

numpy.argmax()

- The numpy.argmax() function returns indices of the max element of the array in a particular axis.

```
output_prob.argmax(axis=1)
```

```
rray([[0.58178245, 0.00234469, 0.97036937, 0.64034516],
      [0.44504406, 0.07178624, 0.50511309, 0.98527334],
      [0.40490749, 0.21520268, 0.66671445, 0.18926015],
      [0.99818906, 0.3702341 , 0.32152925, 0.33452479],
      [0.41693608, 0.99710111, 0.54760253, 0.98896868],
      [0.43080255, 0.6232379 , 0.60616554, 0.41871962],
      [0.57980182, 0.30218979, 0.48831486, 0.17218716],
      [0.38477543, 0.40937626, 0.60831249, 0.23314077],
      [0.24803288, 0.13615116, 0.38076504, 0.80648948],
      [0.64015809, 0.11270068, 0.67419178, 0.63834555],
      [0.0711749 , 0.72234198, 0.83176517, 0.26625898],
      [0.92572131, 0.31060026, 0.39069662, 0.72056121],
      [0.76615175, 0.75503287, 0.57738505, 0.3122232 ],
      [0.67315251, 0.4502434 , 0.64605349, 0.47127994],
      [0.08272156, 0.53467679, 0.29487162, 0.16681734]])
```

Conclusion

- Here we have discussed some of the most widely used functions and commands.
- There are many more functions and operations available in NumPy

Conditionals, Loops and functions

Contents

Contents

- Conditionals
- Loops
- Functions

If-Then-Else statement

If Condition

```python
age=60
if age<50:
  print("Group1")
print("Done with if")
```

```python
age=40
if age<50:
  print("Group1")
print("Done with if")
```

If-else statement

```
age=60
if age<50:
  print("Group1")
  print("Done with if")
```

If-else statement

```
age=60

if age<50:
  print("Group1")
else:
  print("Group2")

print("Done with if")
```

Multiple else conditions in if

If condition for checking whether a candidate secured First class/ second class or failed in an exam.

```
marks=86

if(marks<30):
  print("fail")
elif(marks<60):
  print("Second Class")
elif(marks<80):
  print("First Class")
elif(marks<100):
  print("Distinction")
else:
  print("Error in marks")
```

For loop

For loop

Print first 20 values

```
#Example-1

for i in range(1,20):
  print("Number is", i)
```

Break Statement in for loop

- To stop execution of a loop
- Stopping the loop in midway using a condition

- Print cumulative sum and stop when sum reaches 100

```
sumx=0
for i in range(1,200):
  sumx=sumx+i
  if(sumx>100):
    break
  print("Cumulative sum is", sumx)
```

List Comprehension

```
#for i in range(1,20):
#  print("Number is", i)

[i for i in range(1,20)]


result=[i for i in range(1,20)]
print(result)
```

List Comprehension

```
result=[i for i in range(1,20) if i<15]
print(result)

result=[i for i in range(1,20) if i<15 if i%2 == 1 ]
print(result)
```

Writing Function

```
def my_function_name(param1, param2, param3):
    code lines
    code lines
    code lines
    return;
```

Distance Calculation function

- Distance Calculation function

```
def mydistance(x1,y1,x2,y2):
    import math
    dist=math.sqrt(pow((x1-x2),2)+pow((y1-y2),2))
    return(dist)

mydistance(0,0,2,2)
mydistance(4,6,1,2)
```

Data Handling with Pandas

Contents

Contents

- Importing data
- Accessing
- Sub setting
- Sorting
- Missing values
- Duplicates
- Merging
- .apply()
- Groupby
- Cross tabs and pivots
- Dask for large datasets

Data import from CSV files

- Need to use the function read.csv
- Need to use “/” or “\\” in the path. The windows style of path “\” doesn’t work

Importing from CSV files

```
import pandas as pd
Sales = pd.read_csv("https://raw.githubusercontent.com/venkatareddykonasani/Datasets/master/Superstore%20Sales%20Data/Sales_by_country_v1.csv")
print(Sales)
```

Data import from Excel files

- Need to use pandas again

Data import from Excel files

```
wb_data = pd.read_excel("https://raw.githubusercontent.com/venkataredd
ykonasani/Datasets/master/World%20Bank%20Data/GDP.xlsx" , sheet_name="
Sheet1")
print(wb_data)
```

Tip – You can also import zipped files

```
air_bnb_ny=pd.read_csv("https://raw.githubusercontent.com/
venkatareddykonasani/Datasets/master/AirBnB_NY/AB_NYC_2019
.zip", compression="zip")

print(air_bnb_ny.shape)
```

Importing from SAS files

Using pandas

```
gnp = pd.read_sas(r'D:\Google Drive\Training\Datasets\SAS datasets\gnp.sas7bdat')
```

Using pyreadstat package

```
#!pip install pyreadstat
import pyreadstat
gnp_data, meta = pyreadstat.read_sas7bdat(r'D:\Google Drive\Training\Datasets\SAS datasets\gnp.sas7bdat')
```

Basic Commands on Datasets

- Is the data imported correctly? Are the variables imported in right format? Did we import all the rows?
- Once the dataset is inside Python, we would like to do some basic checks to get an idea on the dataset.
- Just printing the data is not a good option, always.
- Is a good practice to check the number of rows, columns, quick look at the variable structures, a summary and data snapshot

Check list after Import

```
Sales.shape
Sales.info()
Sales.columns
Sales.head()
Sales.tail()
Sales.sample(5)
Sales.describe()
Sales["unitsSold"].describe()
Sales["salesChannel"].value_counts()
```

Pandas GUI

```
!pip install pandasgui
```

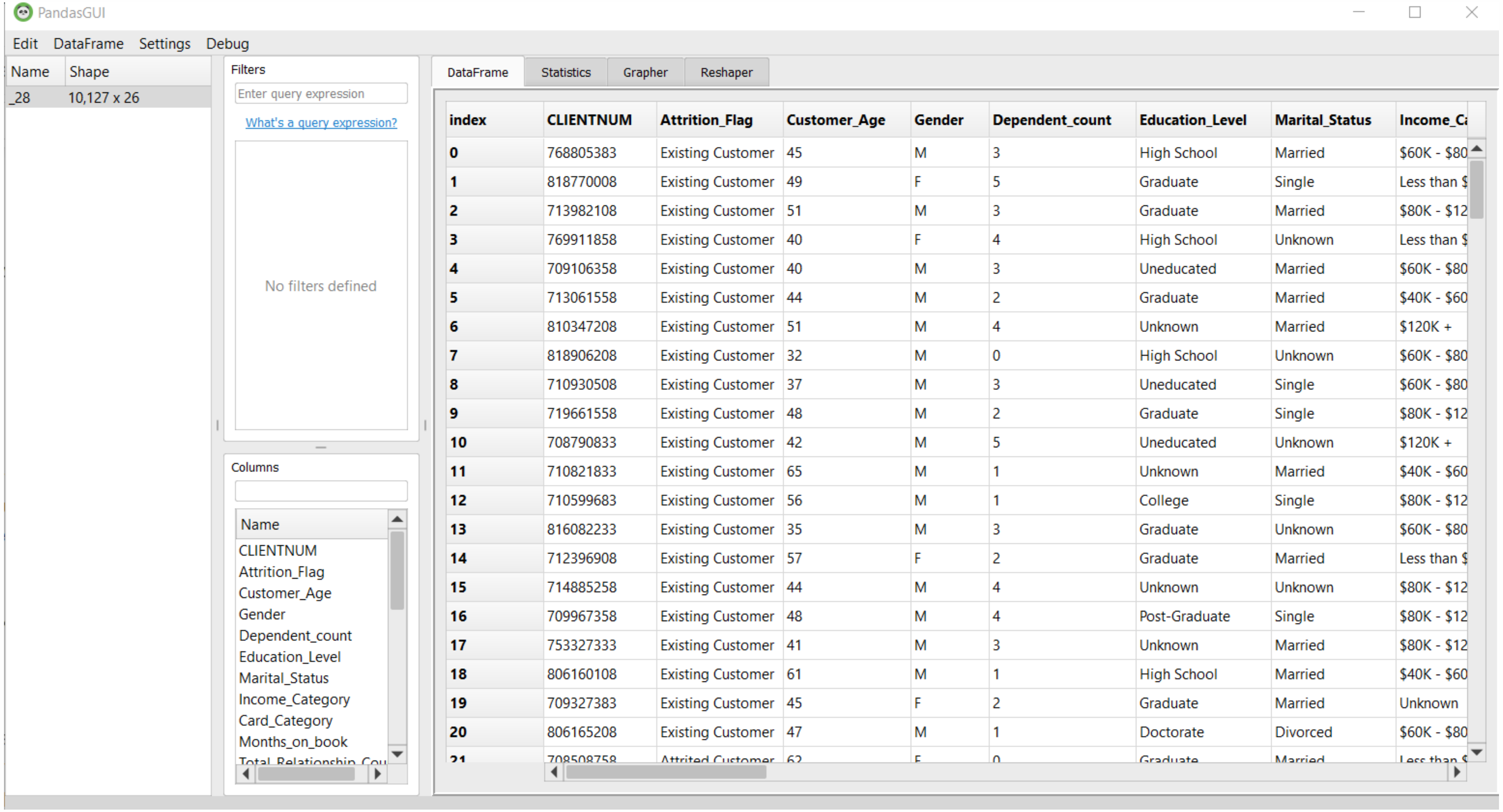
```
from pandasgui import show

show(cc_usage)
```

```
PandasGUI INFO — pandasgui.gui — Opening PandasGUI
INFO:pandasgui.gui:Opening PandasGUI
```

```
<pandasgui.gui.PandasGui at 0x27de9efa160>
```

Pandas GUI



index	CLIENTNUM	Attrition_Flag	Customer_Age	Gender	Dependent_count	Education_Level	Marital_Status	Income_Ca
0	768805383	Existing Customer	45	M	3	High School	Married	$60K - $80
1	818770008	Existing Customer	49	F	5	Graduate	Single	Less than $
2	713982108	Existing Customer	51	M	3	Graduate	Married	$80K - $12
3	769911858	Existing Customer	40	F	4	High School	Unknown	Less than $
4	709106358	Existing Customer	40	M	3	Uneducated	Married	$60K - $80
5	713061558	Existing Customer	44	M	2	Graduate	Married	$40K - $60
6	810347208	Existing Customer	51	M	4	Unknown	Married	$120K +
7	818906208	Existing Customer	32	M	0	High School	Unknown	$60K - $80
8	710930508	Existing Customer	37	M	3	Uneducated	Single	$60K - $80
9	719661558	Existing Customer	48	M	2	Graduate	Single	$80K - $12
10	708790833	Existing Customer	42	M	5	Uneducated	Unknown	$120K +
11	710821833	Existing Customer	65	M	1	Unknown	Married	$40K - $60
12	710599683	Existing Customer	56	M	1	College	Single	$80K - $12
13	816082233	Existing Customer	35	M	3	Graduate	Unknown	$60K - $80
14	712396908	Existing Customer	57	F	2	Graduate	Married	Less than $
15	714885258	Existing Customer	44	M	4	Unknown	Unknown	$80K - $12
16	709967358	Existing Customer	48	M	4	Post-Graduate	Single	$80K - $12
17	753327333	Existing Customer	41	M	3	Unknown	Married	$80K - $12
18	806160108	Existing Customer	61	M	1	High School	Married	$40K - $60
19	709327383	Existing Customer	45	F	2	Graduate	Married	Unknown
20	806165208	Existing Customer	47	M	1	Doctorate	Divorced	$60K - $80

Access rows

```
Sales.iloc[0:10]
Sales.iloc[[1,9,10]]
```

Access Columns

```
column_names=['custId', 'custName', 'custCountry']
Sales[column_names]
Sales.iloc[:,0:4]
Sales.iloc[0:5:,0:4]
```

.iloc vs .loc

```
Sales1=Sales.iloc[20:30]
```

What is the difference?

```
Sales1.iloc[0:5]
Sales1.loc[0:5]
```

.iloc vs .loc

```
Sales1=Sales.iloc[20:30]
```

What is the difference?

```
Sales1.iloc[0:5] #index location
Sales1.loc[0:5] #index values or names
```

What is the difference?

```
Sales.loc[0:5, 0:2] #This works in iloc
Sales.loc[0:5, column_names]
```

Accessing Specific type of Columns only

```
Sales_numerics = Sales.select_dtypes(include=["int64","float64"])
Sales_numerics.info()

Sales_objects = Sales.select_dtypes(include=["object"])
Sales_objects.info()

Sales_non_objects = Sales.select_dtypes(exclude=["object"])
Sales_non_objects.info()
```

Drop

```
wb_data.drop(range(0,10))
wb_data.drop(["Country_code"], axis=1)
```

Subset with variable filter conditions

- How to filter the data based on a variable?
- For example select all the customers with age >40 in bank market data
- Subset all the customers with age>40 and loan="no"

Subset with variable filter conditions

```
bank_subset=bank_data[bank_data['age']>40]
bank_subset

#And condition & filters
bank_subset1=bank_data[(bank_data['age']>40) &  (bank_data['loan']=="no")]
bank_subset1

#OR condition & filters
bank_subset2=bank_data[(bank_data['age']>40) |  (bank_data['loan']=="no")]
bank_subset2
```

Lab: Subset with variable filter conditions

- Data : "./Automobile Data Set/AutoDataset.csv"
- Create a new dataset for exclusively Toyota cars
- Create a new dataset for all cars with city.mpg greater than 30 and engine size is less than 120.
- Create a new dataset by taking only sedan cars. Keep only four variables(Make, body style, fuel type, price) in the final dataset.
- Create a new dataset by taking Audi, BMW or Porsche company makes.

Working with index

```
bank_data.index
bank_subset1.index
```

Reset the index, if required.

```
bank_subset1=bank_subset1.reset_index()
bank_subset1.index
```

The above code creates a new column called index, you can drop it while creating

```
bank_subset1_1=bank_subset1.reset_index(drop=True)
bank_subset1_1.index
```

rename

```
Sales.columns
```

```
Index(['custId', 'custName', 'custCountry', 'productSold', 'salesChannel',
       'unitsSold', 'dateSold'],
      dtype='object')
```

```
Sales=Sales.rename(columns={"dateSold":"DateSold_new"})
Sales.columns
```

```
Index(['custId', 'custName', 'custCountry', 'productSold', 'salesChannel',
       'unitsSold', 'DateSold_new'],
      dtype='object')
```

```
Sales=Sales.rename(columns={"dateSold":"DateSold_new", "custCountry": "Country"})
Sales.columns
```

```
Index(['custId', 'custName', 'Country', 'productSold', 'salesChannel',
       'unitsSold', 'DateSold_new'],
      dtype='object')
```

Calculated Fields

Calculate and Assign it to new variable

```
auto_data['area']=(auto_data[' length'])*(auto_data[' width'])*(auto_data
[' height'])
auto_data['area']
```

Sorting the data

- Its ascending by default

```
Online_Retail_sort=Online_Retail.sort_values('UnitPrice')
Online_Retail_sort.head(20)
```

- Use ascending=False for descending sort

```
Online_Retail_sort=Online_Retail.sort_values('UnitPrice',ascending=False)
Online_Retail_sort.head(20)
```

- Sorting with two cols

```
Online_Retail_sort2=Online_Retail.sort_values(['Country','UnitPrice'], ascen
ding=[True, False])
Online_Retail_sort2.head(5)
```

LAB: Sorting the data

- AutoDataset
- Sort the dataset based on price
- Sort the dataset based on price descending

Identifying & Removing Duplicates

Datasets: Telecom Data Analysis\Bill.csv

```
#Identify duplicates records in the data
dupes=bill_data.duplicated()
dupes
sum(dupes)


#Removing Duplicates
bill_data.shape
bill_data_uniq=bill_data.drop_duplicates()
bill_data_uniq.shape
```

Identifying & Duplicates based on Key

- What if we are not interested in overall level records
- Sometimes we may name the records as duplicates even if a key variable is repeated.
- Instead of using duplicated function on full data, we use it on one variable

```
dupe_id=bill_data["cust_id"].duplicated()
dupe_id
bill_data.shape
bill_data_cust_uniq=bill_data.drop_duplicates(['cust_id'])
bill_data_cust_uniq.shape
```

Data sets merging and Joining

- Datasets: TV Commercial Slots Analysis/orders.csv & TV Commercial Slots Analysis/slots.csv

```
orders1=orders.drop_duplicates(['Unique_id'])
slots1=slots.drop_duplicates(['Unique_id'])

###Inner Join
inner_data=pd.merge(orders1, slots1, on='Unique_id', how='inner')
inner_data.shape
###Outer Join
outer_data=pd.merge(orders1, slots1, on='Unique_id', how='outer')
outer_data.shape
##Left outer Join
L_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='left')
L_outer_data.shape
###Righ outer Join
R_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='right')
R_outer_data.shape
```

Data sets merging and Joining

- Datasets: TV Commercial Slots Analysis/orders.csv & TV Commercial Slots Analysis/slots.csv

```
orders1=orders.drop_duplicates(['Unique_id'])
slots1=slots.drop_duplicates(['Unique_id'])

###Inner Join
inner_data=pd.merge(orders1, slots1, on='Unique_id', how='inner')
inner_data.shape
###Outer Join
outer_data=pd.merge(orders1, slots1, on='Unique_id', how='outer')
outer_data.shape
##Left outer Join
L_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='left')
L_outer_data.shape
###Righ outer Join
R_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='right')
R_outer_data.shape
```

Data sets merging and Joining

```
####Other options
left_on : a column or a list of columns
right_on : a column or a list of columns


other_data=pd.merge(orders1, slots1, left_on=['AD_ID','Product ID'],ri
ght_on=['AD_ID','Product ID'], how='outer’)

other_data.shape
```

Functions on rows and columns

- Consider Credit Card usage data.
- Create a new variable credit_limit_segment by using this below condition
  - If Credit_Limit < 3000- Low
  - If Credit_Limit < 6000- medium
  - Else - high

Using a for loop

```
cc_usage["Credit_Limit_Segment"]=0

for i in range(0, len(cc_usage)):
  if cc_usage["Credit_Limit"][i] < 3000:
    cc_usage["Credit_Limit_Segment"][i]  ="Low"
  elif cc_usage["Credit_Limit"][i] < 6000:
    cc_usage["Credit_Limit_Segment"][i]  ="Medium"
  else:
    cc_usage["Credit_Limit_Segment"][i]  ="high"
```

tip- never use a for loop

- For loop code and iteration over each row works, but never use it.
- It is very basic and time consuming
- There are better ways
  - Vectorizing
  - apply function

Vectorization

```
cc_usage["Credit_Limit_Segment1"]="High"
cc_usage["Credit_Limit_Segment1"][cc_usage["Credit_Limit"]< 3000]="Low"
cc_usage["Credit_Limit_Segment1"][(cc_usage["Credit_Limit"]> 3000) & (cc_usa
ge["Credit_Limit"]< 6000)]="Medium"
```

For loop vs Vectorization

- The execution time comparison for the same operation

For loop

```
Time taken 232.56957602500916
```

Vectorization

```
Time taken 0.029158592224121094
```

.apply() function

- In the previous examples we worked on two different datasets, and different variables.
- In both the cases, credit limit and UnitPrice, we performed binning with different set of limits.
- Can we write a generalized binning function and then apply it on any column?

Generalized function

```
def binning(x, limit1, limit2):
  result="High"
  if x < limit1:
    result="Low"
  if (x > limit1) & (x < limit2):
    result="Medium"
  return(result)
```

How do we apply the above function on a desired column from a dataset? - Using .apply() function

apply() function

- `cc_usage["Credit_Limit_Segment2"]=cc_usage["Credit_Limit"].apply(lambda x:binning(x, 3000, 6000))`

- Here `lambda` is a temporary anonymous function which has been created in apply() itself
- `lambda` function is used for calling another function.
- `lambda` function takes each row supplies it to the binning function then returns the result of binning function

apply() function

```
cc_usage["Credit_Limit_Segment2"]=cc_usage["Credit_Limit"]
.apply(lambda x:binning(x, 3000, 6000))

Online_Retail["UnitPrice_Segment2"]=Online_Retail["UnitPri
ce"].apply(lambda x:binning(x, 2, 4))
```

More on .apply() function

- We need not define a function always

```
cc_usage['Dependent_count_ind']=cc_usage['Dependent_count'].apply(lambda x: "
Low" if x<5 else "high")
```

.apply() on columns

- How to find the mean of each numeric column in a data frame?
- To work on each column we need to make axis=0. This sounds opposite to drop function
- axis=0 – Apply along the row index (returns for each col)
- axis=1 – Appy along the columns (returns for each row)

```
cc_usage.select_dtypes(exclude="object").apply(lambda y: round(y.mean()),
axis=0)
```

```
cc_usage.select_dtypes(exclude="object").apply(lambda y: [round(y.mean()),
 round(y.median())], axis=0)
```

.apply() on columns

- Applying conditions on multiple columns to make a new column
- Create a new variable based on two conditions

```
Total_Relationship_Count > 4 and Credit_Limit < 5000
```

```
def cli_flag(x):
  if x[0] > 4 and x[1]<5000:
    return(1)
  else:
    return(0)

cc_usage["Credit_Line_increase_flag"]=cc_usage[["Total_Relationship_Count","Credit_Limit"]].apply(lambda x:cli_flag(x), axis=1)
```

Group-by

- Import House Sales in King County data.
  https://www.kaggle.com/harlfoxem/housesalesprediction
- group by “condition” of the house and find the below details
  - The number of items in each group
  - The average house price in each group

Group-by

```
kc_house_price.groupby("condition").count()
```

This code gives us the count by each column

```
#Restrict to one column
kc_house_price.groupby("condition")["id"].count()

round(kc_house_price.groupby("condition")["price"].mean())
```

tip – use agg() function

```
kc_house_price.groupby("condition")["id"].count()
```

The above code works, but its better use the aggregate function. agg() function gives us many options

```
kc_house_price.groupby("condition").agg({'id':['count’]})
```

```
kc_house_price.groupby("condition").agg({'price':['mean']})
```

One group-by but multiple aggregated values

```
kc_house_price_grp_agg=kc_house_price.groupby("condition")
.agg({'id':['count'],'price':['mean','min', 'max'] })
```

Group-by more than one column

```
kc_house_price_grp_agg1=kc_house_price.groupby(["condition","floo
rs"]).agg({'id':['count'],'price':['mean','min', 'max'] })
```

tip -Reset index after group by

- The group-by output gives you multi-indexed columns and rows.
- There is a tendency to use the resultant dataset directly.
- Multi-Index is confusing, better to reset the row index and rename the columns.

```
kc_house_price_grp_agg1.columns
kc_house_price_grp_agg1.index

#Updated index
kc_house_price_grp_agg1.columns=['count','avg_price','min_price',
'max_price']
kc_house_price_grp_agg1=kc_house_price_grp_agg1.reset_index()
```

Cross-tabs and Pivot tables

- Dataset bank marketing data
- Calculate the response rate in each category of education

```
pd.crosstab(index=bank_data['education'], columns=bank_data['y’])
```

The above code gives the cross tab with counts. For calculation of percentage we need to use apply function on reach row.

```
pd.crosstab(index=bank_data['education'], columns=bank_data['y']).apply(lam
bda x: x*100/x.sum(), axis=1)
```

Cross-tabs and Pivot tables

- Calculate the response rate in each category of job type

```
pd.crosstab(index=bank_data['job'], columns=bank_data['y']
).apply(lambda x: x*100/x.sum(), axis=1)
```

Pivot tables

- Alternative to cross tabs. Many more parameters.
- Calculate the response rate in each category of job type

```
pd.pivot_table(data=bank_data, index='job', columns='y', v
alues='Cust_num', aggfunc='count’)
```

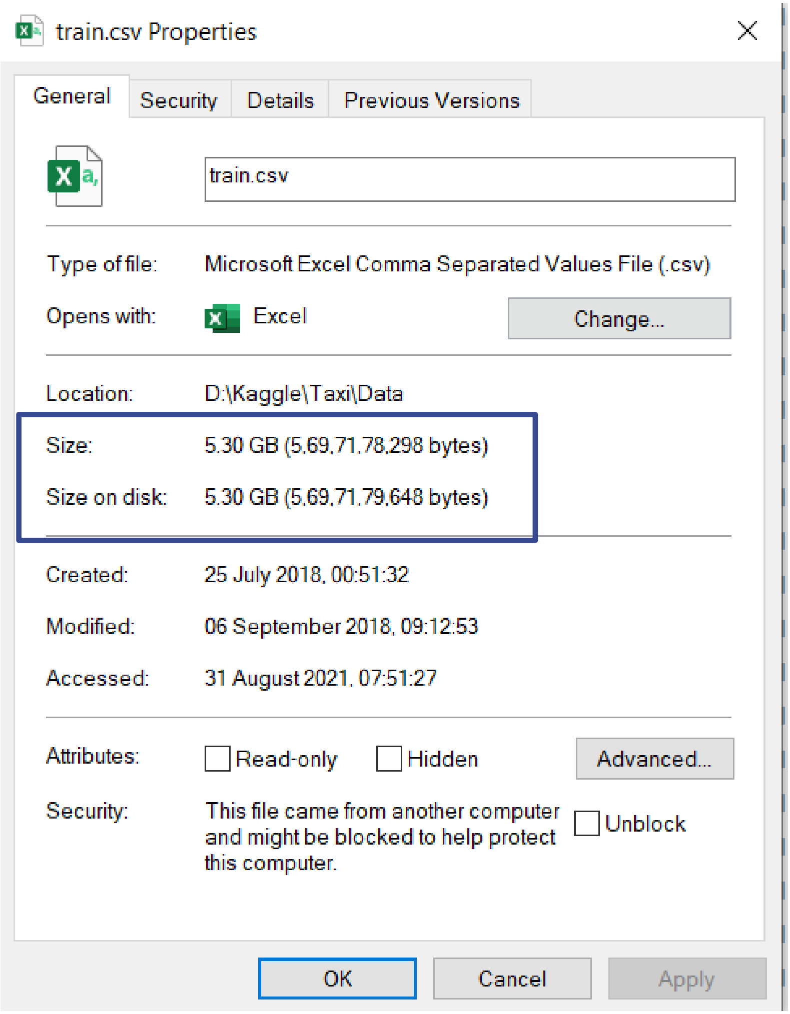
```
pd.pivot_table(data=bank_data, index='job', columns='y', v
alues='Cust_num', aggfunc='count').apply(lambda x: x*100/x
.sum(), axis=1)
```

Pivot tables

- Create a pivot table by that can calculate the average salary in responder and non-responder category inside each of the job type

```
pd.pivot_table(data=bank_data, index='job', columns='y', v
alues='balance', aggfunc='mean')
```

Handling Lage Datasets



Handling Lage Dataset

- Is it possible to import this data as a pandas data frame?
- Is it possible to index and store all this data into python?
- Can you import the above dataset.
- Find out number of rows and columns
- Perform basic descriptive statistics.
- Take a random sample of 1 million records from it.

Dask for large datasets

- Pandas data frames are very slow for large datasets(of size in GBs and TBs)
- Dask is a dedicated package to handle large amounts of data
- Dask can store the data on RAM as well as hard-disk
- Dask can store and access the data from multiple machines in a cluster
- Dask will automatically partition and index the data while reading.

Dask for a large dataset

- Dask doesn't really import the dataset.
- It creates the necessary partitions, pointers and indexes

```
import dask.dataframe as dd
%time taxi_fare_dd = dd.read_csv(r"D:\Kaggle\Taxi\Data\train.csv")
```

```
Wall time: 20.9 ms
```

```
taxi_fare_dd
```

Dask DataFrame Structure:

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
npartitions=90								
	object	float64	object	float64	float64	float64	float64	int64
	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...

Dask Name: read-csv, 90 tasks

Dask for a large dataset

- When we say compute() then the parallel processing will happen to calculate the results.

```
: taxi_fare_dd.shape[0].compute()

: 55423856
```

Check the background processing status

```
from dask.distributed import Client, progress
client = Client()
client
```

Basic Statistics

```
taxi_fare_dd["fare_amount"].mean().compute()
```

Sampling

```
taxi_fare_dd_sample=taxi_fare_dd.sample(frac=0.02)
```

Converting sample into a dataframe

```
taxi_fare_dd_pd_sample=taxi_fare_dd_sample.compute()
```

```
taxi_fare_dd_pd_sample.shape
```

Conclusion

- In this session we started with Data imploring from various sources
- We also learnt manipulating the datasets and creating new variables
- There are many more topics to discuss in data handling, these topics in the session are essential for any data scientist

Dates, Functions, RegEx and more

Contents

- String format
- Handling Dates
- Datetime
- Functions
- args and kwargs
- Regex functions
- map() and filter()
- OOP basics

String Format for custom variables

```
avg_age=cc_usage["Customer_Age"].mean()
print("The average age is {}".format(avg_age))


avg_credit_limit=cc_usage["Credit_Limit"].mean()
print("The average age is {} and the average credit limit
is {}".format(avg_age, avg_credit_limit ))
```

One more way of formatting

```
print(f"The average age is {avg_age} , the average credit limit is {avg_credit_limit}")
```

Reorder the Values

```
avg_Dependent_count=cc_usage["Dependent_count"].mean()
print("The average age is {} , the average credit limit is
 {} , the average Dependent count is {}".format(2.34620321
91172115, 8631.953698034848, 46.32596030413745))

print("The average age is {2} , the average credit limit i
s {1} , the average Dependent count is {0}".format(2.34620
32191172115, 8631.953698034848, 46.32596030413745))
```

Placeholders with name

```
print("The average age is {age} , the average credit limit is {clim} , the average Dependent count is {depen}".format(depen=2.3462032191172115, clim=8631.953698034848, age=46.32596030413745))
```

Customizing the output format

```
print("The average age is {2:0.1f} , the average credit li
mit is {1:0.0f} , the average Dependent count is {0:0.1f}"
.format(2.3462032191172115, 8631.953698034848, 46.32596030
413745))
```

```
print("The average age is {age:0.1f} , the average credit
limit is {clim:0.0f} , the average Dependent count is {dep
en:0.1f}".format(depen=2.3462032191172115, clim=8631.95369
8034848, age=46.32596030413745))
```

Print as percentage

```
avg_util_percent=cc_usage["Avg_Utilization_Ratio"].mean()
print("The Avg_Utilization_Ratio age is {:0.2f} ".format(a
vg_util_percent))
```

```
print("The Avg_Utilization_Ratio age is {:0.2%} ".format(a
vg_util_percent))
```

Working with dates

Prepare the date field

kc_house_price

date
20141013T000000
20140611T000000
20140919T000000
20140804T000000
20150413T000000

Prepare the date field

```
kc_house_price["date"].apply(lambda x: x[0:8])
```

OR

```
kc_house_price["date_split"]=kc_house_price["date"].apply(
lambda x: x.split("T")[0])
kc_house_price["date_split"]
```

Str-p-time : strptime() from string to date

```
from datetime import datetime
kc_house_price["date_cleaned"]=kc_house_price["date_split"
].apply(lambda x: datetime.strptime(x, "%Y%m%d"))
kc_house_price["date_cleaned"]

type(kc_house_price["date_cleaned"][0])
```

Derive Month, Day, Year, weekday

```
kc_house_price['month']=kc_house_price["date_cleaned"].apply
(lambda x: x.month)

kc_house_price['year']=kc_house_price["date_cleaned"].apply(
lambda x: x.year)

kc_house_price['day']=kc_house_price["date_cleaned"].apply(l
ambda x: x.day)

kc_house_price['weekday']=kc_house_price["date_cleaned"].app
ly(lambda x: x.weekday())
```

Extracting strings from dates

- Extract month name from date
- Extract weekday name
- Extract year in a particular format

- Is there any function that can take date time as input and extract the relevant information as strings

```
strftime()
```

Str-f-time : strftime() – From date to string

```
kc_house_price['month_name']=kc_house_price["date_cleaned"].apply(lambda x: x.strftime('%B'))
kc_house_price['month_name']

kc_house_price['weekday_name']=kc_house_price["date_cleaned"].apply(lambda x: x.strftime('%A'))
kc_house_price['weekday_name']

kc_house_price['year_YY']=kc_house_price["date_cleaned"].apply(lambda x: x.strftime('%y'))
kc_house_price['year_YY']
```

Important symbols

```
%d: day of the month, from 1 to 31.

%m: the month as a number, from 01 to 12
%b: first three characters of the month name.
%B: full name of the month

%Y: year in four-digit format
%y: year in two-digit format

%w: weekday as a number, from 0 to 6, with Sunday being 0.
%A: weekday full string.
```

Working with DateTime

```
nifty_stock_price["Datetime1"]=nifty_stock_price["Datetime"].apply(lambda x: x.split('+')[0])
nifty_stock_price["Datetime_cleaned"]=nifty_stock_price["Datetime1"].apply(lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S"))
nifty_stock_price["Datetime_cleaned"]

nifty_stock_price["time"]=nifty_stock_price["Datetime_cleaned"].apply(lambda x: x.time())
nifty_stock_price["time"]

nifty_stock_price["hour"]=nifty_stock_price["Datetime_cleaned"].apply(lambda x: x.hour)
nifty_stock_price[["Datetime_cleaned", "hour"]]
```

Working with DateTime

```
nifty_stock_price["minute"]=nifty_stock_price["Datetime_cleaned"].apply(lambda x: x.minute)
nifty_stock_price[["Datetime_cleaned", "minute"]]

nifty_stock_price["second"]=nifty_stock_price["Datetime_cleaned"].apply(lambda x: x.second)
nifty_stock_price[["Datetime_cleaned", "second"]]
```

Using strftime

```
nifty_stock_price["Datetime_cleaned"].apply(lambda x:
 x.strftime('%H'))
nifty_stock_price["Datetime_cleaned"].apply(lambda x:
 x.strftime('%M'))
nifty_stock_price["Datetime_cleaned"].apply(lambda x:
 x.strftime('%S'))
```

Duration - Time Delta

```
min_date=nifty_stock_price["Datetime_cleaned"].min()
max_date=nifty_stock_price["Datetime_cleaned"].max()
print("min_date {} max date {}".format(min_date,max_date))

#Duration in days
duration=max_date-min_date
duration

#Duration in hours
from datetime import timedelta
duration/timedelta(hours=1)

#Duration in minutes
duration/timedelta(minutes=1)
```

More on Functions

Fixed number of input parameters

Finding the sum of squares of the input values

```
def sum_of_sq(a,b,c,d):
    ss_val=pow(a,2)+pow(b,2)+pow(c,2)+pow(d,2)
    return(ss_val)

sum_of_sq(a=1,b=2,c=3,d=4)
```

args

Finding the sum of squares of the input values of any length

```
def any_sum_of_sq(*args):
    ss_val=0
    for n in args:
        ss_val=ss_val+pow(n,2)
    return(ss_val)

any_sum_of_sq(1,2,3)
```

args and keywords

```
def any_sum_of_sq1(*args, power):
    ss_val=0
    for n in args:
        ss_val=ss_val+pow(n,power)
    return(ss_val)

any_sum_of_sq1(1,2,3, power=1)
any_sum_of_sq1(1,2,3, power=2)
any_sum_of_sq1(1,-2,-3, power=3)
```

args and keywords

```
def any_sum_of_sq2(*args, power, neg_numbers):
    if neg_numbers==True:
        ss_val=0
        for n in args:
            ss_val=ss_val+pow(n,power)
        return(ss_val)
    else:
        return("Negative numbers passed as input")

any_sum_of_sq2(1,-2,-3, power=3, neg_numbers=True)
any_sum_of_sq2(1,-2,-3, power=3, neg_numbers=False)
```

args and keywords

```
def any_sum_of_sq3(*args, power, neg_numbers, neg_power):
    if neg_power==True:
        print("Negative power is accepted")
    if neg_numbers==True:
        ss_val=0
        for n in args:
            ss_val=ss_val+pow(n,power)
        return(ss_val)
    else:
        return("Negative numbers passed as input")


any_sum_of_sq3(1,-2,-3, power=3, neg_numbers=True, neg_power=True)
```

*args and **kwargs

```
def any_sum_of_sq3(*args, **Kwargs):
    power, neg_numbers, neg_power=Kwargs.values()
    if neg_power==True:
        print("Negative power is accepted")
    if neg_numbers==True:
        ss_val=0
        for n in args:
            ss_val=ss_val+pow(n,power)
        return(ss_val)
    else:
        return("Negative numbers passed as input")

kwargs={"power":3, "neg_numbers":True, "neg_power":True}
args=(1,-2,-3)
any_sum_of_sq3(*args, **kwargs)
```

RegEx functions

len()

```
len(air_bnb_ny["name"])
```

startswith() and endswith()

```
mask=air_bnb_ny["name"].str.startswith("Affordable")
air_bnb_ny[mask]
```

```
mask=air_bnb_ny["name"].str.startswith("Affordable", na=False)
air_bnb_ny[mask]
```

```
mask=air_bnb_ny["name"].str.endswith("hotel", na=False)
air_bnb_ny[mask]
```

find()

```
"Entire home/apt".find("apt")

"small room".find("apt")

air_bnb_ny["room_type"].value_counts()
mask=air_bnb_ny["room_type"].str.find("Shared")!= -1
air_bnb_ny[mask]
```

split()

```
"Entire home/apt".split()
"Entire home/apt\tother".split()
"Entire home/apt\tother\nNA".split()
"Entire home/apt\tother\nNA".split("/")

air_bnb_ny["host_name"].str.split("&") # Two Hosts

air_bnb_ny["host_name"].str.split("&")[0] # First Host
air_bnb_ny["First_host"]=air_bnb_ny["host_name"].str.split("&").str.get
(0)
air_bnb_ny["First_host"]

air_bnb_ny["host_name"].str.split("&").str.get(1) # Second Host
```

Contains()

```
mask=air_bnb_ny["name"].str.contains("only for", na=False)
air_bnb_ny[mask]
```

replace()

```
air_bnb_ny["room_type_new"]=air_bnb_ny["room_type"].str.re
place("Entire home/apt", "Guest House")
air_bnb_ny["room_type_new"].sample(20)
```

```
air_bnb_ny["room_type_new"]=air_bnb_ny["room_type"].str.re
place("entire home/apt", "Guest House", case=False)
air_bnb_ny["room_type_new"].sample(20)
```

More on Regex

- Identify email address inside text
- Identify the numbers inside text
- Identify html tags inside text
- Identify specific repeating patterns inside text

Commonly used regular expressions

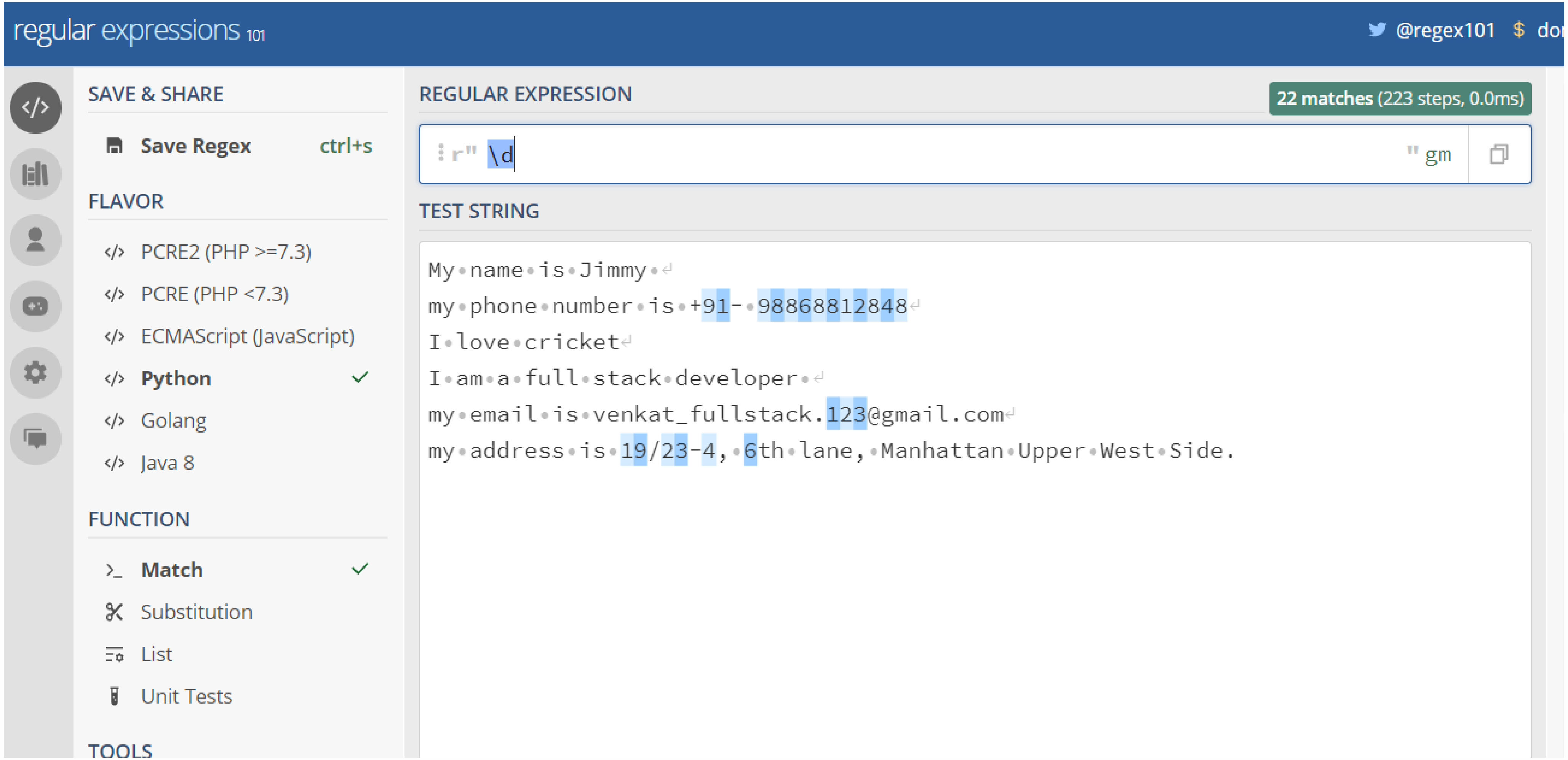
- Digits
  - Whole Numbers – /^\d+$/
  - Decimal Numbers – /^\d*\.\d+$/
  - Whole + Decimal Numbers – /^\d*(\.\d+)?$/
  - Negative, Positive Whole + Decimal Numbers – /^-?\d*(\.\d+)?$/
  - Whole + Decimal + Fractions – /[-]?[0-9]+[,.]?[0-9]*([\/][0-9]+[,.]?[0-9]*)*/
- Alphanumeric Characters
  - Alphanumeric without space – /^[a-zA-Z0-9]*$/
  - Alphanumeric with space – /^[a-zA-Z0-9]*$/
- Email
  - Common email Ids – /^([a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6})*$/
  - Uncommon email ids – /^([a-z0-9_\.\+-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/
- URLs
  - /https?:\/\/(www\.)?[-a-zA-Z0-9@:%._\+~#=]{2,256}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%_\+.~#()?&//=]*)/

RegEx Basic Symbols

Character	Description / What it matches	Example
\	Marks next character as a **special character** or a **literal**.	'**\n**' matches **newline character**. '****' matches "****"
^	**Start of line**/string	
$	**End of line**/string	
.	Matches any **single character** except "\n"	
*	Matches **0 or more** copies (as many as possible) ***of** *preceding character or subexpression*.	**zo*** matches "**z**" and "**zoo**"
+	Matches **1 or more** copies *.	'**zo+**' matches "**zo**" and "**zoo**", but not "**z**"
?	Matches **0 or 1** copies (as few as possible) *.	"**do(es)?**" matches "**do**" or "**does**"
{*n*}	Matches exactly *n* times * or between n AND m) if **{*n*,*m*}** is called	'**o{2}**' matches the two o's in "**food**", but not the o in "**bob**".
[*xyz*]	A **character set** - matches any enclosed characters	'**[abc]**' matches the '**a**' in "**plain**"
[a-z]	Matches a **range of characters**.	'**[a-z]**' matches all lowercase characters
[^*xyz*]	A **negative character set** - matches any character **not** enclosed.	'**[^fo]**' matches the '**d**' in "**food**" '**[^0-9]**' = any non-number char
x|*y*	Either *x* **or** *y*.	"**gray|grey**" matches "**gray**" or "**grey**"
(*pattern*)	A **marked subexpression** - matches *pattern* and captures the match. NOTE: (?:*pattern*), (?=*pattern*) and (?!*pattern*) are variations	"**gr(a|e)y**" matches "**gray**" or "**grey**" (& is more efficient)

regex101.com

https://regex101.com/



Regex

```
import re
twitter_data["raw_removed_digits"]=twitter_data["raw_tweet
"].apply(lambda x: re.sub(r"\d+"," ",str(x))) #Remove digi
ts
twitter_data.iloc[300:350]
```

map() function

map() function

```
mask=air_bnb_ny["name"].str.contains("only for", na=False)
sp_names=set(air_bnb_ny["name"][mask])
sp_names

def clean_tile(x):
    return(x.replace("only for", " "))

result=map(clean_tile, set(sp_names))
print(list(result))
```

map() function

```
result=map(lambda x: clean_tile(x), sp_names)
print(list(result))
```

Filter function

filter()

```
all_names=air_bnb_ny["name"]

def clean_tile_f(x):
    result=str(x).find("only for")!= -1
    return(result)

result1=filter(clean_tile_f, set(all_names))
print(list(result1))
```

filter()

```
result1=filter(lambda x: clean_tile_f(x), all_names)
print(list(result1))
```

Object Oriented Programming

Using the sk-learn package

```
import sklearn
model_1 = sklearn.linear_model.LinearRegression()
model_1.fit(X_train, y_train)
```

This is a model object

The internal code of sk-learn

Linear Regression Class

```
class LinearRegression(MultiOutputMixin, RegressorMixin, LinearModel):

    def __init__(self, *, fit_intercept=True, normalize=False, copy_X=True,
                 n_jobs=None, positive=False):
        self.fit_intercept = fit_intercept
        self.normalize = normalize
        self.copy_X = copy_X
        self.n_jobs = n_jobs
        self.positive = positive


        def fit(self, X, y, sample_weight=None):

        X, y = self._validate_data(X, y, accept_sparse=accept_sparse,
                                   y_numeric=True, multi_output=True)

        if sample_weight is not None:
            sample_weight = _check_sample_weight(sample_weight, X,
                                                 dtype=X.dtype)

        X, y, X_offset, y_offset, X_scale = self._preprocess_data(
            X, y, fit_intercept=self.fit_intercept, normalize=self.normalize,
            copy=self.copy_X, sample_weight=sample_weight,
            return_mean=True)

        if sample_weight is not None:
            # Sample weight can be implemented via a simple rescaling.
            X, y = _rescale_data(X, y, sample_weight)
```

Procedural programming

- If we have to repeat the same task multiple times, we create a function with several parameters.
- We can call the function and pass the relevant parameters to output our results.
- The above method is known as “procedural programming”

Procedural programming

- For example, if we are writing a function performing regression related tasks. We create a large regression function and some sub-functions in it.
  - `def regression(x, y)`
    - `Function-1 : beta coeffects`
    - `Function-2 : R-squared`
    - `Function-3 : VIF`

Object-Oriented Programming (OOP)

- OOP is a different programming paradigm
- Instead of writing a function, we can create classes and we can structure the large programs in a better way

Why OOP is used in development ?

- Easier to organize large programs
- Improves programs readability
- Improves software reusability
- Makes programs easier to develop.

Class and object

- Class
  - A class is like a blueprint or a design
  - In a class we have methods(functions) to perform the calculations
  - int and str are classes
- Object
  - An instance of the class is called an object
  - Object is created using the class blueprint
  - x and msg are objects

```python
x=30
print(type(x))
```

```
<class 'int'>
```

x is an object of integer class

```python
msg="my_message"
print(type(msg))
```

```
<class 'str'>
```

msg is an object of string class

Class and objects

- Class
  - int and str are classes
- Object
  - x and msg are objects

```
x=30
print(type(x))
```

```
<class 'int'>
```

x is an object of integer class

```
msg="my_message"
print(type(msg))
```

```
<class 'str'>
```

msg is an object of string class

Object oriented programming

- For example, if we are writing a function performing regression related tasks. We create a regression class and define methods in it.
  - Class regression:
    - Properties
      - Input data name
      - Data Size
      - Other details
    - Methods
      - Method-1 : beta coefficients
      - Method-2 : R-Squared
      - Method-3 : VIF

Defining a class and object

- `self` – Rerefers to the object itself. We need to pass the object itself as the input every time we are using the object
- `__init__` is a special method. It initializes the attributes of the class.

```
class any_point:
    def __init__(self, x_cord, y_cord):
        self.x=x_cord
        self.y=y_cord
        print("this point = (", self.x, self.y, ")")
```

Define objects

- `point1=any_point(0,0)`
- `point2=any_point(4,5)`

methods inside a class

```python
class any_point:
    def __init__(self, x_cord, y_cord):
        self.x=x_cord
        self.y=y_cord
        print("this point = (", self.x, self.y, ")")

    def eu_dist(self, new_point):
        dist=((self.x-new_point.x)**2+(self.x-new_point.x)**2)**0.5
        return dist

    def man_dist(self, new_point):
        dist1=abs(self.x-new_point.x)+abs(self.x-new_point.x)
        return dist1
```

Using the methods from a class

```
point1=any_point(0,0)
point2=any_point(4,5)

point1.eu_dist(point2)
point1.man_dist(point2)
```

Inheritance

- Inheritance is the ability to derive or inherit the properties from another class.

Inheritance

```
class midpoint:
    def __init__(self, x_cord, y_cord):
        self.x=x_cord
        self.y=y_cord
        print("this point = (", self.x, self.y, ")")

    def mid_point(self, new_point):
        mid_x=(self.x+new_point.x)/2
        mid_y=(self.y+new_point.y)/2
        return (mid_x, mid_y)
```

Inheritance

```
class midpoint1(any_point):
    def mid_point(self, new_point):
        mid_x=(self.x+new_point.x)/2
        mid_y=(self.y+new_point.y)/2
        return (mid_x, mid_y)
m=midpoint1(0,0)
n=midpoint1(4,5)
m.mid_point(n)
```