

# CBCS SCHEME

USN 

--	--	--	--	--	--	--	--	--	--

20MCA31

## Third Semester MCA Degree Examination, Feb./Mar. 2022 Data Analytics using Python

Time: 3 hrs.

Max. Marks:100

Note: Answer FIVE full questions, choosing ONE full question from each module.

### Module-1

- 1 a. Describe arithmetic operators, assignment operators, comparison operators and logical operators in detail with example (08 Marks)
- b. With syntax, explain the finite and infinite looping constructs in python. What is the need for break and continue statements. (07 Marks)
- c. Write a python program to check whether a given number is even or odd. (05 Marks)

### OR

- 2 a. How to declare and call functions in python programs? Illustrate with an example script. (08 Marks)
- b. Illustrate args and kwargs parameters in python programming language with an example. (07 Marks)
- c. Develop a python program to calculate the area of square, rectangle and circle using function. (05 Marks)

### Module-2

- 3 a. Explain any five operations performed on string with an example. (10 Marks)
- b. Demonstrate constructors in inheritance with the help of python program. Take input as student name, subject name, marks of three subjects and calculate the percentage. (10 Marks)

### OR

- 4 a. Differentiate between list tuple, sets and dictionary. (10 Marks)
- b. Create a function product and demonstrate function overloading by accepting required input and print their product. (10 Marks)

### Module-3

- 5 a. Discuss different categories of basic array manipulation with an example. (10 Marks)
- b. Implement the python program to demonstrate the following using numpy array.
  - i) Array searching, sorting and splitting
  - ii) Broad casting. (10 Marks)

### OR

- 6 a. Discuss in detail about pandas data structures. (10 Marks)
- b. Develop a python program to perform arithmetic operations on numpy array. (10 Marks)

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.  
2. Any revealing of identification, appeal to evaluator and/or equations written eg. 42+8 = 50, will be treated as malpractice.

**Module-4**

- 7 a. Explain combining and merging datasets with an example. (10 Marks)  
b. Explain Reshape and pivot operations with an example. (10 Marks)

**OR**

- 8 a. Discuss in detail about data transformation. (10 Marks)  
b. Explain any five built-in string methods with an example. (10 Marks)

**Module-5**

- 9 a. Write short notes on :  
i) Matplot library (10 Marks)  
ii) Seaborn library. (10 Marks)  
b. Implement a python program to demonstrate data visualization using Matplotlib. (10 Marks)

**OR**

- 10 a. Explain the following method with an example graph.  
i) hist() ii) kdeplot() iii) distplot() iv) joinplot(). (10 Marks)  
b. Create a python program to demonstrate data visualization (Line Plot, histogram, Scatter plot) using Seaborn. (10 Marks)

\*\*\*\*\*

1(a) Describe Arithmetic Operators, Assignment operators, Comparison operator, and logical Operators with example

**Solution:**

Arithmetic operators are used to performing mathematical operations like addition, subtraction, multiplication, and division.

Operator	Description	Syntax
+	Addition: adds two operands	$x + y$
-	Subtraction: subtracts two operands	$x - y$
*	Multiplication: multiplies two operands	$x * y$
/	Division (float): divides the first operand by the second	$x / y$
//	Division (floor): divides the first operand by the second	$x // y$
%	Modulus: returns the remainder when the first operand is divided by the second	$x \% y$
**	Power: Returns first raised to power second	$x ** y$

**Assignment Operators**

Assignment operators are used to assigning values to the variables.

Operator	Description	Syntax
=	Assign value of right side of expression to left side operand	$x = y + z$
+=	Add AND: Add right-side operand with left side operand and then assign to left operand	$a += b$ $a = a + b$
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	$a -= b$ $a = a - b$
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	$a *= b$ $a = a * b$
/=	Divide AND: Divide left operand with right operand	$a /= b$

	and then assign to left operand	a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign the result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
=	Performs Bitwise OR on operands and assign value to left operand	a =b a=a b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a<<=b    a= a<<b
<b>Comparison Operators</b>		
<u>Comparison of Relational operators</u> compares the values. It either returns True or False according to the condition.		
Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y

!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y
<=	Less than or equal to True if the left operand is less than or equal to the right	x <= y

1(b) With syntax , Explain the finite and infinite looping constructs in python. What is the need for break and continue statements

Solution :

There are two different types of loop, the finite ones and the infinite ones.

The most common kind of loop is the **finite loop** (i.e., a loop that we explicitly know, in advance, which values the control variables will have during the loop execution)

1. foriin range(0,10):
2.     print(i)

Another example of a finite loop can be done with the while command. The code here produces the same result as the previous finite loop example using for:

1. i=0
2. whilei<10:
3.     print(i)
4.     i+=1

An **infinite loop**, on the other hand, is characterized by not having an explicit end, unlike the finite ones exemplified previously, where the control variable *i* clearly went from 0 to 9 (note that at the end *i* = 10, but that value of *i* wasn't printed). In an infinite loop the control is not explicitly clear, as in the example appearing here:

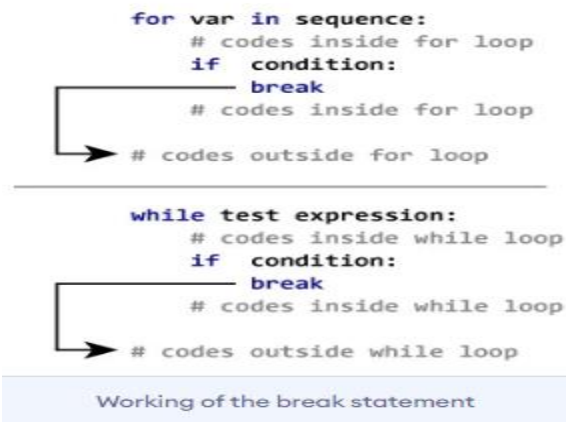
1. i=0
2. whileTrue:
3.     print(i)
4.     ifi==9:
5.         break
6.     else:
7.         i=i+1

### Break and Continue statements

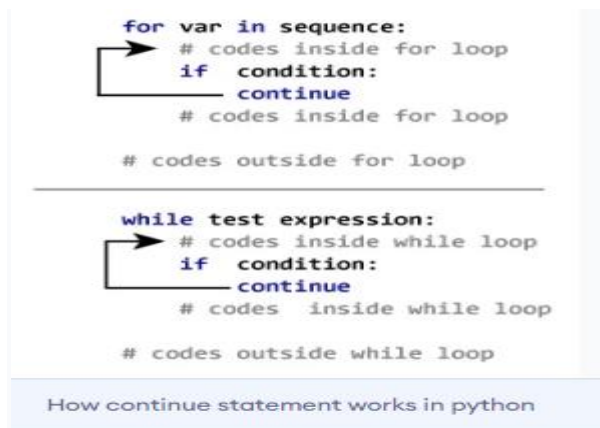
Break and Continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we

wish to terminate the current iteration or even the whole loop without checking test expression.

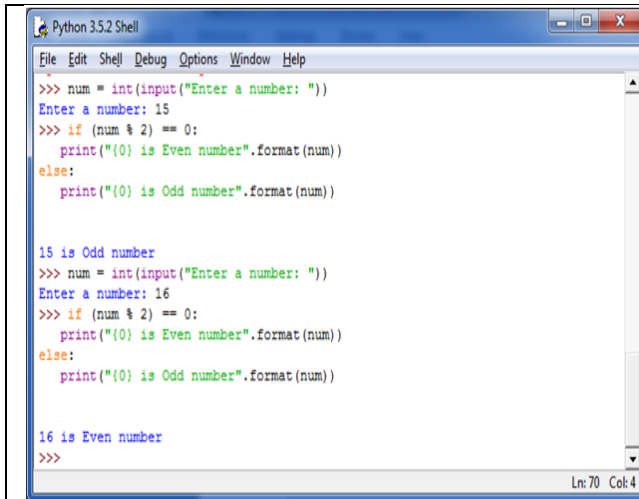


The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.



1(c) Write a Python program to check whether a given number is even or odd

```
num = int(input("Enter a number: "))
if (num % 2) == 0:
    print("{0} is Even number".format(num))
else:
    print("{0} is Odd number".format(num))
```



```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> num = int(input("Enter a number: "))
Enter a number: 15
>>> if (num % 2) == 0:
    print("{} is Even number".format(num))
else:
    print("{} is Odd number".format(num))

15 is Odd number
>>> num = int(input("Enter a number: "))
Enter a number: 16
>>> if (num % 2) == 0:
    print("{} is Even number".format(num))
else:
    print("{} is Odd number".format(num))

16 is Even number
>>>
```

2(a) How to declare and call functions in python program? Illustrate with an example script.

**Solution:**

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result. In Python a function is defined using the def keyword:

```
def my_function():
    print( "Hello from a function" )
```

To call a function, use the function name followed by parenthesis:

```
def my_function():
    print( "Hello from a function" )

my_function()
```

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

```
def my_function(fname):
    print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

2(b) Illustrate args and kwargs parameters in python programming language with an example.

**Solution:**

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function

with 2 arguments, not more, and not less. If you do not know how many arguments that will be passed into your function, add a \* before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: \*\* before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

```
def my_function(**kid):
    print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

**2(c) Develop a python program to calculate the area of square, rectangle, and circle using function**

**Solution:**

**Example:**

**Input:** shape name = "Rectangle"

length = 10

breadth = 15

**Output:** Area: 150

**Input:** shape name = "Square"

side = 10

**Output:** Area: 100

**Approach:**

In this program, We will ask the user to input the shape's name. If it exists in our program then we will proceed to find the entered shape's area according to their respective formulas. If that shape doesn't exist then we will print "Sorry! We cannot find this shape." message on the screen.

```
# define a function for calculating
# the area of a shapes
```

```
def calculate_area(name):
    name = name.lower() # converting all characters into lower cases
    if name == "rectangle":
        l = int(input("Enter rectangle's length: "))
        b = int(input("Enter rectangle's breadth: "))
```



```

    rect_area = l * b
    print(f"The area of rectangle is {rect_area}.")

elif name == "square":          # calculate area of square
    s = int(input("Enter square's side length: "))
    sqt_area = s * s
    print(f"The area of square is {sqt_area}.")

elif name == "triangle":
    h = int(input("Enter triangle's height length: "))
    b = int(input("Enter triangle's breadth length: "))

    tri_area = 0.5 * b * h      # calculate area of triangle
    print(f"The area of triangle is {tri_area}.")

elif name == "circle":
    r = int(input("Enter circle's radius length: "))
    pi = 3.14
    circ_area = pi * r * r      # calculate area of circle
    print(f"The area of triangle is {circ_area}.")

elif name == 'parallelogram':
    b = int(input("Enter parallelogram's base length: "))
    h = int(input("Enter parallelogram's height length: "))

else:
    print("Sorry! This shape is not available")

// Main Code
print("Calculate Shape Area")
shapename = input("Enter the name of shape whose area you want to find: ")
calculate_area(shapename)      # function calling

```

### Output:

Calculate Shape Area

Enter the name of shape whose area you want to find: rectangle

Enter rectangle's length: 10

Enter rectangle's breadth: 15

The area of rectangle is 150.

3(a) Explain any five operations performed on string with an example .

Solution:

#### 1. capitalize()

The capitalize() method returns a string where the first character is upper

case, and the rest is lower case.

Syntax

```
string.capitalize()
```

example:

```
txt = "python is FUN!"  
x = txt.capitalize()  
print(x)
```

2. The **count()** method returns the number of times a specified value appears in the string.

Syntax

```
string.count(value, start, end)
```

example:

```
txt = "python is FUN!"  
x = txt.count()  
print(x)
```

3. The **isalpha()** method returns True if all the characters are alphabet letters (a-z).

Syntax

```
string.isalpha()
```

example:

```
txt = "python is FUN!"  
x = txt.isalpha()  
print(x)
```

4. The **isalnum()** method returns True if all the characters are alphanumeric, meaning alphabet letter (a-z) and numbers (0-9). Example of characters that are not alphanumeric: (space)!#%&? etc.

Syntax

```
string.isalnum()
```

example:

```
txt = "python is FUN*10!"  
x = txt.isalnum()  
print(x)
```

5. The **isdigit()** method returns True if all the characters are digits, otherwise False. Exponents, like <sup>2</sup>, are also considered to be a digit.

Syntax

```
string.isdigit()
```

example:

```
txt = "python is FUN*10!"  
x = txt.isdigit()  
print(x)
```

**3(b) Demonstrate constructors in inheritance with the help of python program.** Take the input as student name, subject name, marks of three subjects and calculate the

percentage.

**Solution :**

```
class Base:
    def __init__(self,sname,subname,m1,m2,m3):
        self.sname=sname
        self.subname=subname
        self.m1=m1
        self.m2=m2
        self.m3=m3
    def display(self):
        print("Student name: ",self.sname)
        print("Student subject name: ",self.subname)
        print("Student marks 1: ",self.m1)
        print("Student marks 2: ",self.m2)
        print("Student marks 3: ",self.m3)
class Derived(Base):
    def __init__(self,sname,subname,m1,m2,m3):
        Base.__init__(self,sname,subname,m1,m2,m3)
    def percentage(self):
        print(((self.m1+self.m2+self.m3)/300)*100,"%")
obj1=Base("Pooja",'Python',98,99,100)
obj2=Derived("Pooja",'Python',98,99,100)
obj2.display()
obj2.percentage()
```

#### 4(a) Differentiate between list tuple sets and Dictionary

A list is a collection of ordered data. A tuple is an ordered collection of data. A set is an unordered collection. A dictionary is an unordered collection of data that stores data in key-value pairs

Tuple	Set	Dictionary
Immutable	Mutable	Mutable
Ordered/Indexed	Unordered	Unordered
Allows duplicate values	Does not allow duplicate values	Does not allow duplicate keys
Empty tuple = ( )	Empty set = set( )	Empty dictionary = { }
Tuple with single item = ("Apple",)	Set with single item = {"Apple"}	Dictionary with single item = {"Hello": 1}
It can store any data type str, list, set, tuple, int and dictionary	It can store data types (int, str, tuple) but not (list, set, dictionary)	Inside of dictionary key can be int, str and tuple only values can be of any data type int, str, list, tuple, set and dictionary

#### 4(b)

**class Compute:**

```

def product(self, x =None, y =None):
    if x !=None and y !=None:
        return x *y
    elif x !=None:
        return x *x
    else:
        return 0

```

```

obj=Compute()
print("productValue:", obj.product())
print("productValue:", obj.product(4))
print("productValue:", obj.product(3, 5))

```

### 5(a)

#### Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape()` method. For example, if you want to put the numbers

1 through 9 in a 3×3 grid, you can do the following:

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
```

```
print(grid)
```

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. You can do this with the `reshape` method, or more easily by making use of the `newaxis` keyword within a slice operation:

```
x = np.array([1, 2, 3])
```

```
x.reshape((1, 3))
```

```
x[np.newaxis, :]
```

#### Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
```

```
y = np.array([3, 2, 1])
```

```
np.concatenate([x, y])
```

```
Out[43]: array([1, 2, 3, 3, 2, 1])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

```
x = np.array([1, 2, 3])
```

```
grid = np.array([[9, 8, 7],
```

```
[6, 5, 4]])
```

```
np.vstack([x, grid])
```

```
Out[48]: array([[1, 2, 3],
```

```
[9, 8, 7],
```

```
[6, 5, 4]])
```

```
y = np.array([[99],
```

```
[99]])
np.hstack([grid, y])
Out[49]: array([[ 9, 8, 7, 99],
 [ 6, 5, 4, 99]])
```

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
grid = np.arange(16).reshape((4, 4))
grid
Out[51]: array([[ 0, 1, 2, 3],
 [ 4, 5, 6, 7],
 [ 8, 9, 10, 11],
 [12, 13, 14, 15]])
```

```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

## 5(b)

```
#sort
import numpy as np
a=[5,4,2,8]
np.sort(a)
np.argsort(a)
print(np.sort_complex(a))
b=np.array([[9,6,3],[7,9,2]])
print("sorted array:",np.sort(b))
print("index sort:",np.argsort(b))
print("complex sort:",np.sort_complex(b))
print("falttened sort:",np.sort(b,axis=0))
```

```
#search
import numpy as np
a=np.array([1,2,3,4,5,6,4,3,4])
x=np.where(a==4)
print(x)
y=np.searchsorted(a,4)
print(y)
z=np.searchsorted(a,[4,6,1])
print(z)
```

```

#splitting
import numpy as np
x=np.arange(9)
print(x)
print(np.split(x,3))
print(np.split(x,[3,5,6,10]))

x=np.arange(9)
print(np.array_split(x,4))

a=np.array([[1,3,5,7,9,11],[2,4,6,8,10,12]])
print("splitting along horizontal axis into 2 parts:\n",np.hsplit(a,2))
print("splitting along vertical axis into 2 parts:\n",np.vsplit(a,2))

```

```

#broadcasting
import numpy as np

x=np.arange(4)
print(x)
y=np.ones(5)
print(y)
print(x.reshape(4,1))
z=np.ones((3,4))
print(np.ones((3,4)))
print(z.shape)

```

6a)

### Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

```
obj = Series([4, 7, -5, 3])
```

```
obj
```

output

```
0 4
```

```
1 7
```

```
2 -5
```

```
3 3
```

The string representation of a Series displayed interactively shows the index on the left

and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1

In [6]: obj.values

```
Out[6]: array([ 4, 7, -5, 3])
```

```
In [7]: obj.index
```

```
Out[7]: Int64Index([0, 1, 2, 3])
```

- Adding index

```
obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
obj2
```

```
Out[9]:
```

```
d 4
```

```
b 7
```

```
a -5
```

```
c 3
```

- Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values:

```
In [11]: obj2['a']
```

```
Out[11]: -5
```

```
In [12]: obj2['d'] = 6
```

```
In [13]: obj2[['c', 'a', 'd']]
```

```
Out[13]:
```

```
c 3
```

```
a -5
```

```
d 6
```

NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [16]: obj2 * 2
```

```
Out[16]:
```

```
d 12
```

```
b 14
```

```
c 6
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be substituted into many functions that expect a

dict:

```
In [18]: 'b' in obj2
```

```
Out[18]: True
```

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

```
In [21]: obj3 = Series(sdata)
```

```
In [22]: obj3
```

```
Out[22]:
```

```
Ohio 35000
```

```
Oregon 16000
```

Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality:

```
In [32]: obj4.name = 'population'In [33]: obj4.index.name = 'state'
```

```
In [34]: obj4
```

```
Out[34]:
```

```
state
```

```
California NaN
```

```
Ohio 35000
```

```
Oregon 16000
```

```
Texas 71000
```

```
Name: population
```

### DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series (one for all sharing the same index). Compared with other such DataFrame-like structures you may have used before (like R's `data.frame`), row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically.

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
```

```
'year': [2000, 2001, 2002, 2001, 2002],
```

```
'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
```

```
frame = DataFrame(data)
```

```
In [38]: frame
```

```
Out[38]:
```

```
pop state year
```

```
0 1.5 Ohio 2000
```

```
1 1.7 Ohio 2001
```

```
2 3.6 Ohio 2002
```

```
3 2.4 Nevada 2001
```

```
4 2.9 Nevada 2002
```

6b)

```
import numpy as np
```

```
a = np.arange(9).reshape(3,3)
```

```
print 'First array:'
```

```
print a
```

```
print '\n'
```

```
print 'Second array:'
```

```
b = np.array([10,10,10])
```

```
print b
```

```
print '\n'
```

```
print 'Add the two arrays:'
```



```
printnp.add(a,b)
print'\n'
```

```
print'Subtract the two arrays:'
printnp.subtract(a,b)
print'\n'
```

```
print'Multiply the two arrays:'
printnp.multiply(a,b)
print'\n'
```

```
print'Divide the two arrays:'
printnp.divide(a,b)
```

### 7a)

The `pd.merge()` function implements a number of types of joins: the one-to-one, many-to-one, and many-to-many joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data.

#### **One-to-one joins**

Perhaps the simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
```

```
print(df1); print(df2)
df3 = pd.merge(df1, df2)
df3
```

The `pd.merge()` function recognizes that each DataFrame has an “employee” column,

and automatically joins using this column as a key. The result of the merge is a new DataFrame that combines the information from the two inputs.

#### **Many-to-one joins**

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate

entries as appropriate. Consider the following example of a many-to-one join:

```
In[4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                          'supervisor': ['Carly', 'Guido', 'Steve']})
print(df3); print(df4); print(pd.merge(df3, df4))
```

#### **Many-to-many joins**

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete

example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group.

By performing a many-to-many join, we can recover the skills associated with any individual person:

```
In[5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',  
'Engineering', 'Engineering', 'HR', 'HR'],  
'skills': ['math', 'spreadsheets', 'coding', 'linux',  
'spreadsheets', 'organization']})  
print(df1); print(df5); print(pd.merge(df1, df5))
```

7b)

### Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are alternately referred to as *reshape* or *pivot* operations.

#### Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame.

**There are two primary actions:**

##### stack

This “rotates” or pivots from the columns in the data to the rows

##### unstack

This pivots from the rows into the columns

##### Example:

```
data = pd.DataFrame(np.arange(6).reshape((2, 3)) index=pd.Index(['Ohio',  
'Colorado'], name='state'), columns=pd.Index(['one', 'two', 'three'], name='number'))  
result = data.stack()
```

A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data. The pivot table takes simple columnwise

data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data. The difference between pivot tables and GroupBy can sometimes cause confusion; it helps me to think of pivot tables as essentially a *multidimensional* version of GroupBy aggregation. That is, you split-apply-

combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

```
import numpy as np
```

```
import pandas as pd
```

```
import seaborn as sns
```

```
titanic = sns.load_dataset('titanic')
```

```
titanic.pivot_table('survived', index='sex', columns='class')
```

the grouping in pivot tables can be specified with multiple levels, and via a number of options

```
age = pd.cut(titanic['age'], [0, 18, 80])
```

```
titanic.pivot_table('survived', ['sex', age], 'class')
```

```
fare = pd.qcut(titanic['fare'], 2)
```

```
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

age as a data dimension. Here cut the age using the pd.cut function.

```
In[6]: age = pd.cut(titanic['age'], [0, 18, 80])
       titanic.pivot_table('survived', ['sex', age], 'class')

Out[6]:
```

	class	First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

We can apply this same strategy when working with the columns as well; let's add info on the fare paid using pd.qcut to automatically compute quantiles:

```
In[7]: fare = pd.qcut(titanic['fare'], 2)
       titanic.pivot_table('survived', ['sex', age], [fare, 'class'])

Out[7]:
```

	fare	[0, 14.454]	First	Second	Third	\\
sex	age					
female	(0, 18]	NaN	1.000000	0.714286		
	(18, 80]	NaN	0.880000	0.444444		
male	(0, 18]	NaN	0.000000	0.260870		
	(18, 80]	0.0	0.098039	0.125000		

	fare	(14.454, 512.329]	First	Second	Third
sex	age				
female	(0, 18]	0.909091	1.000000	0.318182	
	(18, 80]	0.972973	0.914286	0.391304	
male	(0, 18]	0.800000	0.818182	0.178571	
	(18, 80]	0.391304	0.030303	0.192308	

8a)

## Data Transformation

Transforming data such as Filtering, cleaning, and other transformations are important for data analytics.

### → Removing Duplicates

The DataFrame method duplicated returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

**Example:**

```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'], 'k2': [1, 1, 2, 3, 3, 4, 4]})
      k1 k2
0  one  1
1  two  1
2  one  2
3  two  3
4  one  3
5  two  4
6  two  4

data.duplicated()
```

**drop\_duplicates()** method removes duplicate elements and returns a duplicated() false for all the rows. DataFrame column name can be passed as argument to the drop\_duplicates() which will remove duplicate elements based on that column.

### → Transforming Data Using a Function or Mapping

For many datasets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame. For example convert Yes/No to 1/0.

The map() function is used to map values of Series according to input correspondence. Used for substituting each value in a Series with another value, that may be derived from a function, a dict or a Series.

**Example:**

```
import pandas as pd
data = pd.Series(['Yes', 'No', 'Yes', 'No'])
data.map({'Yes': 1, 'No': 0})
```

### →Replacing Values

Pandas **replace()** function is used to replace a string, regex, list, dictionary, series, number etc. from a dataframe/series.

#### Example:

```
data = pd.Series([1., -999., 2., -999., -1000., 3.])
data.replace(-999, np.nan)
```

### →Renaming Axis Indexes

Like values in a Series, axis labels can be similarly transformed by a function or mapping of some form to produce new, differently labeled objects. You can also modify the axes in-place without creating a new data structure.

Pandas **rename()** method is used to rename any index, column or row.

#### Example:

```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),index=['1', 2, 'New York'],
columns=['one', 'two', 'three', 'four'])
data.rename(index=str.title, columns=str.upper)
```

### →Discretization and Binning

Pandas **cut()** function is used to separate the array elements into different bins . The cut function is mainly used to perform statistical analysis on scalar data.

**Example 1:** Let's say we have an array of 10 random numbers from 1 to 100 and we wish to separate data into 5 bins of (1,20] , (20,40] , (40,60] , (60,80] , (80,100] .

```
df= pd.DataFrame({'number': np.random.randint(1, 100, 10)})
df['bins'] = pd.cut(x=df['number'], bins=[1, 20, 40, 60, 80, 100])
print(df['bins'].unique())
```

## 8b)

Method	Description
cat	Concatenate strings element-wise with optional delimiter
contains	Return boolean array if each string contains pattern/regex
count	Count occurrences of pattern
extract	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
endswith	Equivalent to <code>x.endswith(pattern)</code> for each element
startswith	Equivalent to <code>x.startswith(pattern)</code> for each element
findall	Compute list of all occurrences of pattern/regex for each string
get	Index into each element (retrieve <i>i</i> -th element)
isalnum	Equivalent to built-in <code>str.isalnum</code>
isalpha	Equivalent to built-in <code>str.isalpha</code>
isdecimal	Equivalent to built-in <code>str.isdecimal</code>
isdigit	Equivalent to built-in <code>str.isdigit</code>
islower	Equivalent to built-in <code>str.islower</code>
isnumeric	Equivalent to built-in <code>str.isnumeric</code>
isupper	Equivalent to built-in <code>str.isupper</code>
join	Join strings in each element of the Series with passed separator
len	Compute length of each string
lower, upper	Convert cases; equivalent to <code>x.lower()</code> or <code>x.upper()</code> for each element

→lower()

```
import pandas as pd
```

```
import numpy as np
```

```
s = pd.Series(['CMRIT', 'Mysore', 'MCA', 'MCA@cmrit.in', np.nan, '1234', 'CM'])
```

```
s.str.lower()
```

```
print s
```

**Output:**

```
0 cmrit
```

```
1 mysore
```

```
2 mca
```

```
3 mca@cmrit.in
```

```
4 NaN
```

```
5 1234
```

```
6 cm
```

→len()

```
s.str.len()
```

```
0 5.0
```

```
1 6.0
```

```
2 3.0
```

```
3 12.0
```

```
4 NaN
```

```
5 4.0
```

```
6 2.0
```

→count()

```
s.str.count('M')
```

**OUTPUT**

```
0 1.0
```

```
1 1.0
```

```
2 1.0
```

```
3 1.0
```

```
4 NaN
```

```
5 0.0
```

```
6 1.0
```

→startswith()

```
s.str.startswith('M')
```

**Output:**

```
0 False
```

```
1 True
```

```
2 True
```

```
3 True
```

```
4 NaN
```

```
5 False
```

```
6 False
```

→endswith()

```
s.str.endswith('A')
```

**Output:**

```
0 False
```

- 1 False
- 2 True
- 3 False
- 4 NaN
- 5 False
- 6 False

9a)

### i. Matplotlib

Matplotlib is a multiplatform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line. IPython's creator, Fernando Perez, was at the time scrambling to finish his PhD, and let John know he wouldn't have time to review the patch for several months. John took this as a cue to set out on his own, and the Matplotlib package was born, with version 0.1 released in 2003. It received an early boost when it was adopted as the plotting package of choice of the Space Telescope Science Institute (the folks behind the Hubble Telescope), which financially supported Matplotlib's development and greatly expanded its capabilities.

One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large userbase, which in turn has led to an active developer base and Matplotlib 's powerful tools and ubiquity within the scientific Python world.

### ii. Seaborn Visualization

Matplotlib has proven to be an incredibly useful and popular visualization tool, but even avid users will admit it often leaves much to be desired. There are several valid complaints about Matplotlib that often come up:

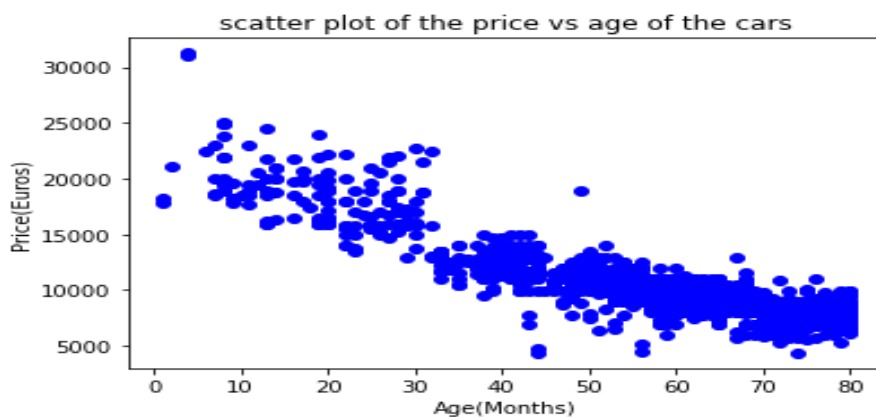
- Prior to version 2.0, Matplotlib's defaults are not exactly the best choices. It was based off of MATLAB circa 1999, and this often shows.
- Matplotlib's API is relatively low level. Doing sophisticated statistical visualization is possible, but often requires a *lot* of boilerplate code.
- Matplotlib predated Pandas by more than a decade, and thus is not designed for use with Pandas DataFrames. In order to visualize data from a Pandas DataFrame, you must extract each Series and often concatenate them together into the right format. It would be nicer to have a plotting library that can intelligently use the DataFrame labels in a plot. Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output.

9b)

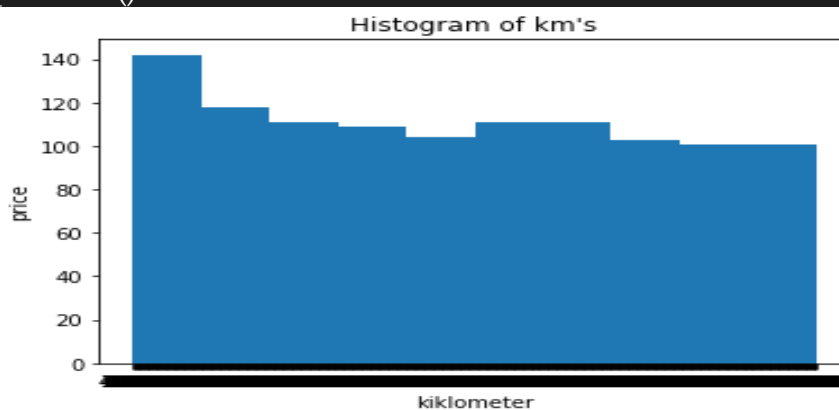
```
df=pd.read_csv("/content/Toyota.csv",index_col=0,na_values=["??","????")
df.head()
```

```
df.dropna(axis=0,inplace=True)
df.head()
```

```
#scatter plot
plt.scatter(df["Age"],df["Price"],color="blue")
plt.title("scatter plot of the price vs age of the cars")
plt.xlabel('Age(Months)')
plt.ylabel('Price(Euros)')
plt.show()
```

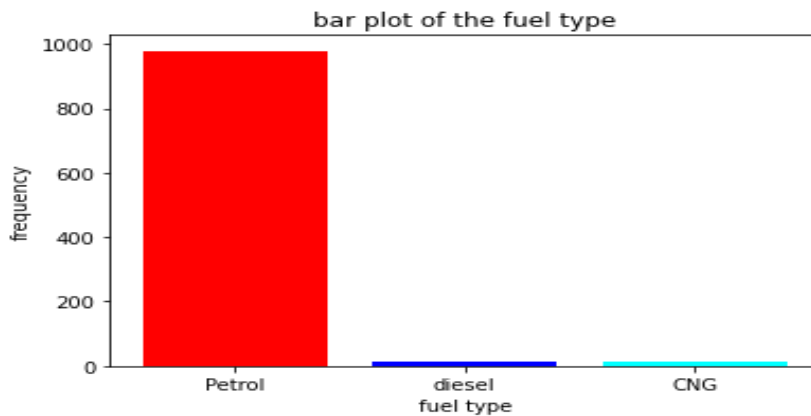


```
#histogarm
plt.hist(df['KM'])
plt.title("Histogram of km's")
plt.xlabel("kiklometer")
plt.ylabel("price")
plt.show()
```



```
#bar graph
plt.bar(index,counts,color=['red','blue','cyan'])
plt.title("bar plot of the fuel type")
plt.xlabel("fuel type")
plt.ylabel("frequency")
```

```
plt.xticks(index,fuelType)
plt.show()
```



10a)

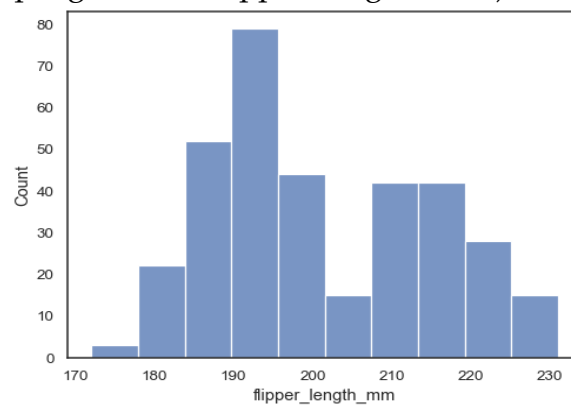
i. **hist()**

A histogram is a classic visualization tool that represents the distribution of one or more variables by counting the number of observations that fall within discrete bins.

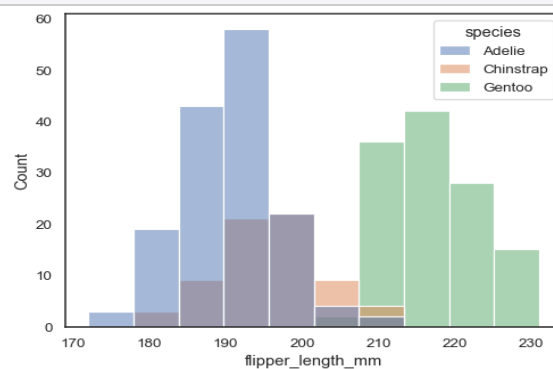
```
seaborn.histplot(data=None, *, x=None, y=None)
```

```
penguins = sns.load_dataset("penguins")
```

```
sns.histplot(data=penguins, x="flipper_length_mm")
```



```
sns.histplot(data=penguins, x="flipper_length_mm", hue="species")
```



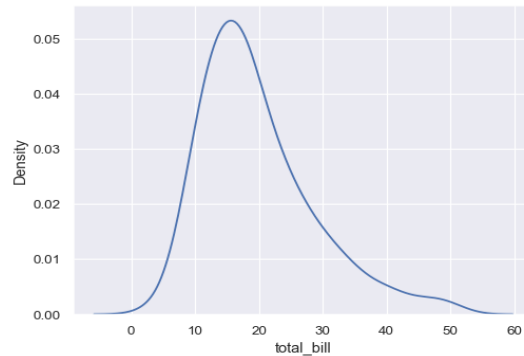
ii. **kdeplot()**



A kernel density estimate (KDE) plot is a method for visualizing the distribution of observations in a dataset, analogous to a histogram. KDE represents the data using a continuous probability density curve in one or more dimensions

**seaborn.kdeplot(x=None, \*, y=None)**

```
tips = sns.load_dataset("tips")
sns.kdeplot(data=tips, x="total_bill")
```



iii. `distplot()`

This function combines the matplotlib hist function (with automatic calculation of a good default bin size) with the seaborn `kdeplot()` and `rugplot()` functions.

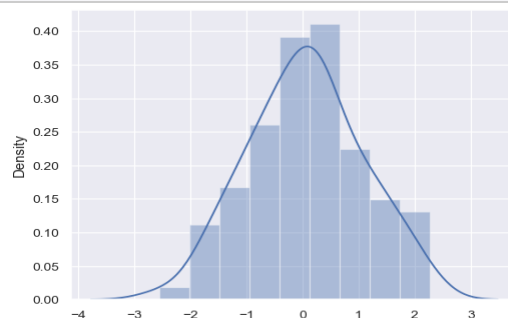
**seaborn.distplot(a=None, bins=None, hist=True, kde=True)**

```
import seaborn as sns, numpy as np
```

```
>>> sns.set_theme(); np.random.seed(0)
```

```
>>> x = np.random.randn(100)
```

```
>>> ax = sns.distplot(x)
```



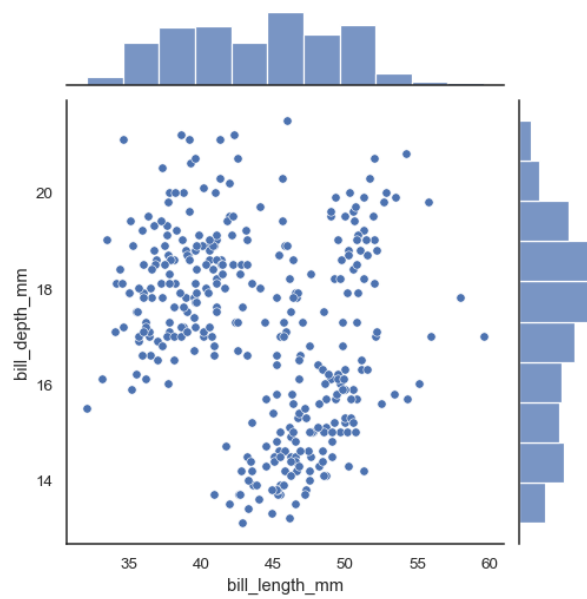
iv. `jointplot()`

Draw a plot of two variables with bivariate and univariate graphs.

**seaborn.jointplot(\*, x=None, y=None, data=None, kind='scatter')**

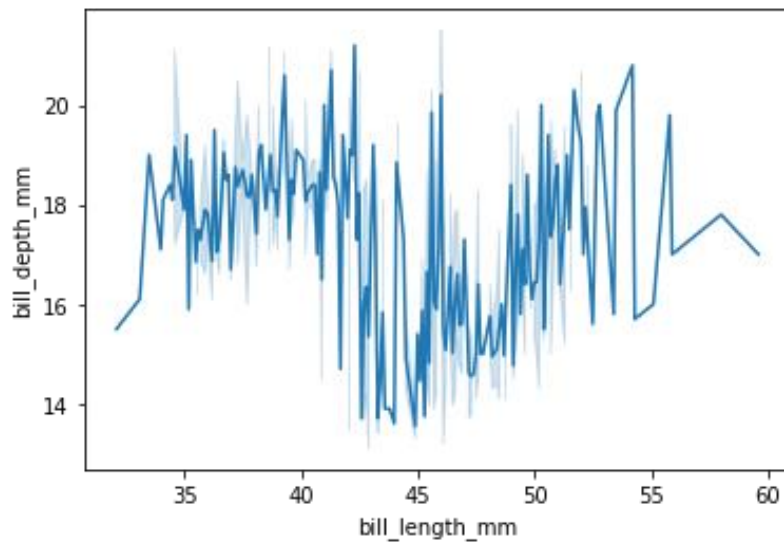
```
penguins = sns.load_dataset("penguins")
```

```
sns.jointplot(data=penguins, x="bill_length_mm", y="bill_depth_mm")
```

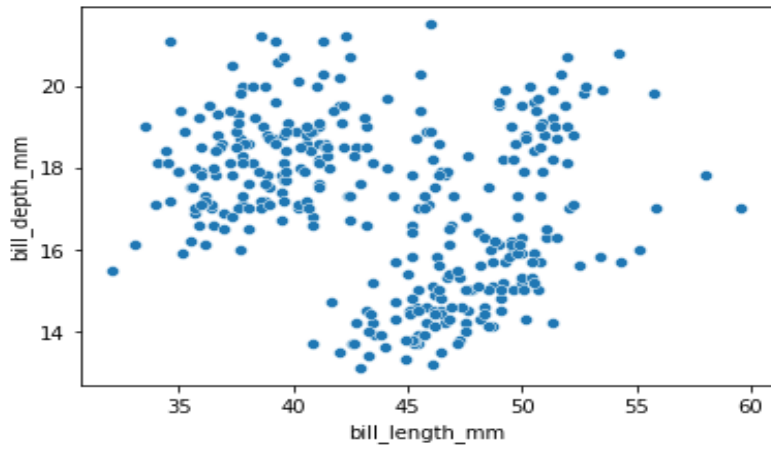


10b)

```
import seaborn as sns
penguins = sns.load_dataset("penguins")
sns.lineplot(data=penguins, x="bill_length_mm", y="bill_depth_mm")
```



```
sns.scatterplot(data=penguins, x="bill_length_mm", y="bill_depth_mm")
```



```
sns.histplot(data=penguins)
```

