MY PYTHON DOCUMENTATION

INTRODUCTION

PYTHON: **Python** is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language.

->Python is an easy to learn,readable

🡪Python is dynamically-typed

🡪it has vast amount of libraries and modules( numpy ,pandas ,math ,time ,date)

🡪we can build our logic with built in type functions or with our own logic

Ex: program to convert a number decimal <-> binary

Using built in function:

# input a decimal num

decimal\_number = int(input(“enter a decimal number:”))

#convert to binary using bin() function

binary\_num = bin(decimal\_number) #4 =100(binary)

Print(“binary format”,binary\_num)

Explanation : bin() converts decimal to binary format

Without built in functions:

def deci\_to\_binary(num):

binary\_num = “ “

if num == 0:

return “0”

while num > 0:

remainder = num % 2

binary\_num = str(remainder) + binary\_num

num = num // 2

return binary\_num

# input a decimal num

num = int(input(“enter a decimal number:”))

binary\_num = print(deci\_to\_binary(num)) # calling function

Print(“binary format”,binary\_num)

Explanation: here we use our own logic by dividing the number with 2 and storing the remainder in “decimal number”

Variables and Data types:

In Python, you can store data in variables. Data types include:

1.Integers: Whole numbers, e.g., 23

2.Floats: Decimal numbers, e.g., 3.1235

3.Strings: Text, e.g., "Hello, ACCELYZIE"

4.Booleans: True or False values, e.g., True

5.List : list represents ordered , mutable sequence of values (e.g [1,2,3,4,”sis”])

6.Tuple : Tuple data type represents ordered,immutable sequences of values (e.g.,(1,2,3,8,6))

7.Dictionaries : Dictionary data type represents collection of key – value pairs(e.g {‘name” : “keerthi”, “age” :20})

8.None: None type represents the absence of value e.g: None

9.set :it is unordered collection of unique elements e.g {1,3,5,6} (no duplicate values)

Basic Operations:

You can perform arithmetic operations like addition, subtraction, multiplication, and division using symbols +, -, \*, /.

Ex:  PERFORM ALL ARTHMETIC OPERATIONS ON 3 NUMBERS

a = int(input("enter 1 st number:"))

b = int(input("enter 2 nd number:"))

c = int(input("enter 3 rd number:"))

print("arthmatic operations:")

print("addition of  numbers:",str(a+b+c))

print("subtraction of 3 numbers:",str(a-b-c))

print("multiplication of 3 numbers:",str(a\*b\*c))

print("dividion of 3 numbers",str(a / b / c))

print("modulus of 3 numbers:",str(a % b % c))

Print Statements:

We can display information using the print() function. For example:

print("Accelyzei")

Input from Users;

We can get input from users using the input() function:

Num = int(input(“enter a number:”))

name = input("What's your name? ")

print("Hello,", name)

Conditional Statements:

These help you make decisions in your code using if, elif (else if), and else. For example:

 vote eligibility

'''age = int(input("enter your age:"))

if(age >= 18):

    print(" you are eligible to vote:",age)

else:

    print("you have ",18-age ,"years to get eligibilty") '''

# checking credintials:

print(" printing person details:")

name = input("enter your name:")

age = int(input("enter your age:"))

if (name == "keerthi chavla" or "Prasanna"):

    print("welcome ",name ,"your age is",age)

else:

    print("invalid credintials: please enter again")

if: if checks the condition and execute the statements if condition TRUE other wise not execute

else:if condition not True then else statement execute

Loops:

let you repeat actions. for loops are used to iterate over sequences

while loops run as long as a certain condition is true.

EX:

n = int(input("no of sets to ombine"))

for i in range(n):

    n\_el = int(input('enter no of elements in set:'))

    s = set()

    for j in range(n\_el):

        ele = int(input("enter your set element:"))

        s.add(ele)

    s=s.union(s)

above example for union of sets

for loop to iterate no of required sets to combine

Dictionary operations:

Creation of dictionary

d1 = {"company" :"Apple" , "model":"11 pro" , "year":2023}

# creating dictionary using from keys() 🡪creates with keys and values

a=["name","food","movie"]

b = 3

# create a dict with new keys and values

d2 = dict.fromkeys(a,b)

print(d2)

op: {“name”:3, ”food”:3 ,”movie” :3 }

nested dictionaries

ex:

details = {'person1': {"name":"aswini","age":20,

                        "address":{'dist':'sklm','vill':'dosari'}},

           'person2':{"name":

    "prassu","age": 19,

    "address":{'dist':'vizayanagaram','vill':'rajam'}

    }}

print(details)

print("persion 1 name:",details['person1']['name'])

print("person 2 address:",details['person2']['address']['vill'])

Functions:

->Functions are reusable blocks of code which perform particular task

-> helps to organize your code and make it more readable.

Ex: str = "ALPHA NUMARO"

def change\_name(str):  #function definition

    print("our company name is:",str)

    return

# new name= "Accelezie"

change\_name("Accelyzie")  #calling function

print("company old name is :Alpha numaro")

change\_name(“Accelyzei”) # we send the value to function definition.

Op: our company name is: Accelyzei

Company old name is : Alpha numaro

Comments

We can add comments to explain your code.& for future reference, Comments are not executed and start with #:

# This is a comment explaining the following line

age = 25 #intialize age as 25

“ “ “

These are multi line comments

Hello

Hi

“ “ “

Libraries and Modules:

->Python has a vast ecosystem of libraries and modules that provide additional functionality.

->We can import and use them in your code:

Time and Datetime :

import time

import calendar

import pandas as pd

# we have to import these modules

now = time.time()

print("time:" ,now)

localtime = time.localtime(time.time())

print(localtime)

time.localtime() 🡪 gives present time in readable format

Extract year,month from data frame

data = {'Date': ['2023-09-05', '2023-08-15', '2022-07-25', '2022-06-10']}

df = pd.DataFrame(data)

df["dob"] = pd.to\_datetime(df['Date'],format = '%Y-%m-%d')

#extracting year month from date

df['year']  = df['dob'].dt.year

df['month'] = df['dob'].dt.month

df= dob year month

03-10-2023 2023 10

15-08-2023 2023 08

25-07-2022 2022 07

Astime():

print("normal time readable format:",time.asctime(time.localtime(time.time())))

print("calender functions")

cal = calendar.month(2002,5)

print(cal)

op:calendat.moth() give particular month of given year

Date: 13-09-2023

FILE HANDLING OPERATION:

file operations:

The open() function takes two parameters; filename, and mode.

import os

f1 = open("demo.txt","w")

f1.write("documentation should be clean and simple")

f1.close()  # close the file after your work

print("reading data")

f1 = open("demo.txt","r")

my\_data = f1.read()   # store read data in variable

print("demo file data:",my\_data)

f1.close()

#

print("appending data")

f1 = open("demo.txt","a")

f1.write("you learn new things every day")

f1.close()

print(" data appended & file closed")

#f1.append("you learn everyday")

# delete files

print("delete files fileobj.remove(file\_name)")

os.remove("file2.txt")

# to avoid error first check whether the file exist or not

if os.path.exists("none.txt"):

    os.remove("none.txt")

else:

    print("such a file not exist in your system")

print("we can remove dir --> os.rmdir(folder)") # can remove empty folders only

append -> add data at the end of file content

write mode : over write the previous data

delete -> to delete file we use os.remove() method

mport os

# file operations

#strip() removes given char from both ends

f= open("file2.txt",'r')

line = f.readline()

print("before strip:",line)

line2=line.strip("A")

print("after strip:",line2)

line3=line.strip(" ")

print("after strip only spaces:",line3)

# right strip

line4 = line.rstrip("e")

print("removeing right char:",line4)

desc: to operate on file first we have to open particular file in required mode

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

‘ rb’ – open file for reading binary format

after our file operatons we should close our file

file\_obj.close()

ex:

mport os

def create\_file(filename):

    try:

        with open('filename','w') as f:

            f.write("hello accelyzie \n")

        print("file" + filename + "created with data successfully" )

    except IOError:

          # if we want to specify error

          print("error : file was not created" +filename)

def read\_file(filename):

     try:

          with open(filename,'r') as f:

               content = f.read()

               print(content)

     except IOError:

          print("successfully file content readed" +filename)

def append\_data(filename,text):

     try:

          with open(filename,'a') as f:

               f.write(text)

          print("text appended to the file" +filename)

     except IOError:

          print("could not  appended to the file" +filename)

def rename\_file(filename,newfile):

     try:

          os.rename(filename,new\_file)

          print("file:" + filename + "renamed to"  +new\_file)

     except IOError:

          print("could not rename the file" +filename)

def delete\_file(new\_file):

     try:

          os.remove(new\_file)

          print("file" +filename + "deleed successfully")

     except IOError:

          print("error : can not delelted" +new\_file)

filename = "ex.txt"

new\_file = "changed\_ex.txt"

text = "i am add some data"

create\_file(filename)

read\_file(filename)

append\_data(filename,text)

read\_file(filename)

rename\_file(filename,new\_file)

delete\_file(new\_file)

ex: file programs

1. Count no of words

Step 1: open the file in “r” mode

Step 2: read the file data in variable

Step 3: split the content at space using .split() method

Step 4:close the file

#no of words in the file

f2 = open("demo.txt","r")

f2\_content = f2.read()

words = f2\_content.split()

print("no of words:",len(words))

print("words",words)

f2.close()

2.merge files and copy the data into new file

print(" mergeing  files data")

file1 = open("demo.txt",'r')

file2 = open("prog.txt",'r')

new\_file = open("p\_lang.txt",'w')

a = file1.read()

b = file2.read()

c = a + "\n" + b  #merge files data

new\_file.write(c)

file1.close()

file2.close()

new\_file.close()

print("data successfully merged")

c = a + “\n” b # merge two files data into one variable

new\_file.write(c) # write the data into another file using write()

after we should close all the files to avoid errors

PYTHON DATA STRUCTURES:

PYTHON STRINGS:

1.String : sequence of characters

Ex: s= “ Accelyzei – (accelerate , innovative , analyse) “

Sliceing -> we can get particular character or sub string using sliceing by index

S[1] -> c

S[1:7] -> ccelyz #( returns1 to 6 index chars)

Negative indexing:

S[-3] -> s # gives last to 3 rd character

S[-5 : -2] -> yze # 5,4,3 rd charcters from last

2. ->The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are

-> we can insert values in string using format method

Ex:

String formatting: # str operations

s = "Accelyzei - ( Accelerate , Innovative, Analyze) "

print("given string:",s)

print("sliceing:", s[1:4])

print("negative indexing s[-5:-2]:",s[-5:-2])

# string formating

#we can insert numbers into stings using format()

s1 = "we got independence in {} and we celebrate  {} -15 as independence day "

year =1947

print(s1.format(year,"August"))

s2 = "today is better than {0} we never forget {1}"

print(s2.format("Yestarday","hope")

Ex2: print palindrome strings

Palindrome menas string should be same after reversing : sis <-> sis

li = ["mom","you","sis","mean","dad"]

pal\_str = []

def is\_pal(st):

    if st == st[ : :-1]:

        pal\_str.append(st)

    return

for st in li:

    is\_pal(st)

print("original strings:",li)

print("palindrom strings:",pal\_str)

Op: original strings: ['mom', 'you', 'sis', 'mean', 'dad']

palindrom strings: ['mom', 'sis', 'dad']

ex:3 # count no of vowels consonets using string methods

# counting no of vowels consonents

vowels = {"a","e","i","o","u","A","E","I","O","U"}

nv = 0; nc = 0 ; n\_upper = 0 ; n\_lower = 0

alph =0

for i in range(len(s)):

    if s[i].isalpha():

        alph = alph + 1

        if s[i] in vowels:

            nv = nv + 1

        else:

            nc = nc + 1

        # count no of lower case ,upper case characters

        if s[i].islower():

            n\_lower = n\_lower + 1

        if s[i].isupper():

            n\_upper = n\_upper + 1

print("no of alphabets in string:",alph)

print(" no of vowels in string:",nv)

print(" no of consonents in string:",nc)

print(" no of n\_lowercase charcters in string",n\_lower)

print(" no of n\_uppercase charcters in string",n\_upper)

2.set : set is a unordered collection of elements

-> mutable

->not allow duplicate elements

S = set() or set ={ 3,4,5} #declaration of set

3.Frozen Set: it is one type of set

-> not changeable and does not allow duplicate values

-> used as dictionary keys and elements of set

1 = frozenset([0,1,2,3,4,5,"mom"])

f2 = frozenset([0,3,8,4,5,6,2,"sis"])

print(f1)

print(f2)

union = f1.union(f2)

intersection = f1.intersection(f2)

difference = f1.difference(f2)

print("union:",union)

print("intersection:",union)

print("difference:",union)

# used in dictionaries

dic\_with\_frozenset = {f1:"val 1",f2:"val 2"}

print("dic\_with\_frozenset:",dic\_with\_frozenset)

# set of frozen sets

print(" set of frozen sets:",{f1,f2})

4.Stack: it is one type of data structure which follows (Lifo)

-> last in first out

-push() # add elements into stack

Pop() # remove last element

Pee() # top most elemet(last inserted)

EX:

from collections import deque

print("stack using deque")

stack = deque()  # dynamic and efficient

stack.append("home page")

stack.append("gallary")

stack.append('contact page')

stack.append("catalogue page")

print(stack)

print(stack.pop())

print(stack)

deque() it provide stack structure dynamically

->easy to implement stack operations

Note: we can implement stack using list ,but linst costly to copy elements some stack operations

Ex 2:

Implementation of stack using class:

#stack using class

class stack1:

    def \_\_init\_\_(self):

        return self.container = deque()

    def push(self,val):

        return self.container.append(val)

    def pop(self):

        return self.container.pop()

    def peek(self):

        return self.container[-1]  # top most element

    def is\_empty(self):

        return len(self.container) == 0

    def size(self):

        return len(self.container)

after that we create class object s= stack1()

and call the class methods:

s=stack1()  # stack1 object creation

s.push(5)

s.push(6)

s.push(9)

s.peek()

s.pop()

Application: moving to previous window in browser

Ex:

Op: deque(['home page', 'galary', 'contact page', 'catalogue page'])

Op: Pop() -> catalogue page # pop() means remove last element

DATE:14-09-2023

NUMPY:

Why numpy and advantages:

->numpy is a python library, used to working with arrays

->much faster and efficient than list

->stores homogenous objects

->used for scientic computations on large no of data

->multi dimentional and provides element wise operation

Import numpy library

Import numpy as np

A1 =np.array([4,8,9,7])

creation of numpy array

import numpy as np

a1 = np.array([4,2,23,3,16])

print("a1",a1)

print(type(a1))    # version of np

print("version of ",np.\_\_version\_\_)

print("nested arrays and multi dimentional arrays:")

a2 = np.array([43])

print(a2,type(a2))  # o -dimentional array

a3 = np.array([1,2,3,4,5])

a4 =np.array([[4,5,6,3],[8,9,7,19]])

mu\_a5 = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])

print(mu\_a5)

print(type(mu\_a5))

print("dimentionality of mu\_a5",mu\_a5.ndim)

print("dimentionality pf a4:",a4.ndim)

print("Accesing array elements")

print("1 st row 2nd col elenemnt:",a4[0,1])

print("2 nd row 4 th col element:",a4[1,3])

print(" -ve indexing:a[0,-1]",a4[0,-1])

print("2 nd table 1 st row 1 st col element:",mu\_a5[1,0,0])

print("a3",a3[: :2])

a6 = np.array(["jaga","prassu","mom","siddu"])

arr2 = np.array([3.4,1.2,5.6,7.0])

print(a6)

print("a6 data type:",a6.dtype)

#print("convert data type using astype:",a6.astype(int))

print("convert data type using astype:",arr2.astype(int))

Array multi dimentional

To know the dimentionality ->arr.ndim

🡪We can create array using random module or

->arr.ones()

->arr.zeros()

->arr.shape() # tells that no of rows and colums in array

->Arr.reshape() #converting the array into required shape

Accesing array elemnts ->retrieve particular value

Op: 2nd element on 1 row

2

5 th element on 2nd row

10

negative indexing

10

2nd row

[ 6 7 8 9 10]

2.Copy() and view()

#changes on view effect the original array

arr = np.array([1,2,4,9])

x=arr.view()

x[0] = 34

print("arr",arr)

print("",x)

y = arr.copy()

arr[1] = 12

print("arr not effect if there is any changes in cope",arr)

print("y - copy",y)

op:

convert data type using astype: [3 1 5 7]

arr [34 2 4 9]

[34 2 4 9]

arr not effect if there is any changes in cope [34 12 4 9]

y - copy [34 2 4 9]

3.astype() ->change array datatype into another

4. cheking non zeros

y=np.array([1,0,0,0])

print("original array:")

print(y)

print("test whether anyone of the element of given array is non-zeros:")

print(np.any(y))

y=np.array([0,0,0,0])

print("original array:")

print("test any one whether any of the element is non-zeros")

print(np.any(y))

5. a=np.array([2,0,np.nan,np.inf])

print("original array")

print(a)

print("test element wiese for positive or negitive infinity:")

print(np.isinf(a))

b=np.array([-4,np.nan,np.inf,np.inf])

print("array b:")

print(b)

print(np.isinf(b))

op:original array

[ 2. 0. nan inf]

test element wiese for positive or negitive infinity:

[False False False True]

array b:

[-4. nan inf inf]

[False False True True]

6. m=np.array([2+5j,4+0j,5,43,6+7j])

print("original array:")

print(m)

print("checking for complex number:")

print(np.iscomplex(m))

print("cheking for real number:")

print(np.isreal(m))

print("cheking for scalar:")

print(np.isscalar(3.1))

print(np.isscalar([3.1])

op:

original array:

[ 2.+5.j 4.+0.j 5.+0.j 43.+0.j 6.+7.j]

checking for complex number:

[ True False False False True]

cheking for real number:

[False True True True False]

cheking for scalar:

True

False

# isreal(),isscalar(), iscomplex() returns TRUE OR FALSE based on value

7. #lcm

n=18 ; n2=26

print("lcm of 18,26",np.lcm(n,n2))

#reduce -> reduce array by 1d

d\_2 = np.random.arange(1,11)

y= np.lcm.reduce(d\_2)

print(y)

8. import numpy as np

x=np.random.randint(0,11,5)

print("vector of length 5 filled witharbitary from 0 to10")

print(x)

op: vector of length 5 filled witharbitary from 0 to10

[ 1 5 8 10 2]

Random() module ->returns random values withing required range

original array:

[1 0 0 0]

test whether anyone of the element of given array is non-zeros:

True

original array:

test any one whether any of the element is non-zeros

False

9.Matrix operations using numpy

print("matrix operation")

mat1 = np.array([[1,2,3],[4,5,6]])

mat2 = np.array([[7,8,3],[0,9,10]])

print("matrix 1",mat1)

print("matrix 2",mat2)

print("Transpose of matrix",mat1.T)  #np.transpose(mat1)

print("addition of matrix:",mat1 + mat2)

print("subtraction of matrix:",mat1 -  mat2)

print("dot product pf matrix",np.dot(mat1,mat2))  #mat1 @ mat2 = dot product

p

\*using numpy we can add subtract and multiply the arrays

\*using numpu we can also get eigen values of matrix

Ex:

#inverse of matrix

mat3 = np.array([[1,2],[3,4]])

y = np.linalg.inv(mat3)

print("inverse of mat1",y)

det = np.linalg.det(mat3)

print("det of mat1",det)

z=np.linalg.eig(mat3)

print("eigen vlaues of mat3",z)

op:

eigen vlaues of mat3 (array([-0.37228132, 5.37228132]), array([[-0.82456484, -0.41597356],

[ 0.56576746, -0.90937671]]))

det of mat1 -2.0000000000000004

inverse of mat1 [[-2. 1. ]

[ 1.5 -0.5]]

DATA SCIENCE:

Data science is a brach of cse which deals with data and analyse the data

->data science uses statistical &computational methods

->it provides summarization and visualization of data

->from data science we can get better decision making

->it includes:

Data preprocessing

Data Normalization

Data summarization

Trend analysis

Data visualization

Data analysis & Decision making

Data preprocessing:

It includes cheking null or nan values,and cleaning the data

Means after identification we can drop the null value columns or fill ,replace with mean or required values

Methods:

Isnull() ,isnan() # for checking null values

Dropna() # drop nan value row or colums

Fillna().mean ,replace(nan,”new value”) # replace nan columns with proper value

Data Normalization:

The primary purpose of normalization is to remove or reduce the effects of variations or differences in the scales of different variables,

making it easier to compare and analyze the data.

Normalization techniques include Min-Max scaling (scaling data to a specific range, often [0, 1] or [-1, 1])

Ex:

import pandas as pd

import numpy as np

person={

'age':[20,30,50],

'salary':[10000,2000,50000],

}

dt=pd.DataFrame(person)

dt

op:

age salary

20 10000

13 2000

25 050000

#above comparision difficult after normalization(Scaling the values) easy to compare ,

#normalization

dt['age']=dt['age']/dt['age'].max()

dt['salary']=dt['salary']/dt['salary'].max()

df

op:

age salary

0.4 0.20

0.6 0.04

1.0 1.00

Ex2:

import pandas as pd

import numpy as np

person={

'age':[20,30,50],

'salary':[10000,2000,50000],

}

dt=pd.DataFrame(person)

dt

dt['age']=(dt['age']-dt['age'].min())/(dt['age'].max()-dt['age'].min())

dt['salary']=(dt['salary']-dt['salary'].min())/(dt['salary'].max()-dt['salary'].min())

op:

age salary

0.000000 0.166667

0.333333 0.0000002

1.000000 1.000000

Data Summarization:

Summarization involves generating concise and meaningful statistics or information from a dataset.

You can use the **describe()** -> method to generate basic statistics for each numeric column in your DataFrame, such as count, mean, standard deviation, minimum, and maximum

Importing pandas:

Ex:

import pandas as pd

dp=pd.read\_csv("employees.csv")

dp

| **First Name** | **Gender** | **Start Date** | **Last Login Time** | **Salary** | **Bonus %** | **Senior Management** | **Team** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Douglas | Male | 08-06-1993 | 12:42 PM | 97308 | 6.945 | True | Marketing |
| **1** | Thomas | Male | 3/31/1996 | 6:53 AM | 61933 | 4.170 | True | NaN |
| **2** | Maria | Female | 4/23/1993 | 11:17 AM | 130590 | 11.858 | False | Finance |
| **3** | Jerry | Male | 03-04-2005 | 1:00 PM | 138705 | 9.340 | True | Finance |
| **4** | Larry | Male | 1/24/1998 | 4:47 PM | 101004 | 1.389 | True | Client Services |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **995** | Henry | NaN | 11/23/2014 | 6:09 AM | 132483 | 16.655 | False | Distribution |
| **996** | Phillip | Male | 1/31/1984 | 6:30 AM | 42392 | 19.675 | False | Finance |
| **997** | Russell | Male | 5/20/2013 | 12:39 PM | 96914 | 1.421 | False | Product |
| **998** | Larry | Male | 4/20/2013 | 4:45 PM | 60500 | 11.985 | False | Business Development |
| **999** | Albert | Male | 5/15/2012 | 6:24 PM | 129949 | 10.169 | True | Sales |

1000 rows × 8 columns

Some pandas operations

Ex3:

import pandas as pd

ecom=pd.read\_csv("ecom.csv")

ecom

ecom.head() # first 5 rows data

op:

|  |
| --- |
|  |
| **redit Card** | **CC Exp Date** | **CC Security Code** | **CC Provider** | **Email** | **Job** | **IP Address** | **Language** | **Purchase Price** |
| **0** | 16629 Pace Camp Apt. 448\nAlexisborough, NE 77... | 46 in | PM | Opera/9.56.(X11; Linux x86\_64; sl-SI) Presto/2... | Martinez-Herman | 6011929061123406 | 02/20 | 900 | JCB 16 digit | pdunlap@yahoo.com | Scientist, product/process development | 149.146.147.205 | el | 98.14 |
| **1** | 9374 Jasmine Spurs Suite 508\nSouth John, TN 8..  . | 28 rn | PM | Opera/8.93.(Windows 98; Win 9x 4.90; en-US) Pr... | Fletcher, Richards and Whita | 3337758169645356 | 11/18 | 561 | Mastercard | anthony41@reed.com | Drilling engineer | 15.160.41.51 | fr | 70.73 |
| **2** | Unit 0065 Box 5052\nDPO AP 27450 | 94 vE | PM | Mozilla/5.0 (compatible; MSIE 9.0; Windows NT ... | Simpson, Williams and Pham | 675957666125 | 08/19 | 699 | JCB 16 digit | amymiller@morales-harrison.com | Customer service manager | 132.207.160.22 | de | 0.95 |
| **3** | 7780 Julia Fords\nNew Stacy, WA 45798 | 36 vm | PM | Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_8\_0 ... | Williams, Marshall and Buchanan | 6011578504430710 | 02/24 | 384 | Discover | brent16@olson-robinson.info | Drilling engineer | 30.250.74.19 | es |  |

\*\* 1.How many rows and columns are there? \*\*

ecom.info() -> DataFrame's metadata and basic information no of rows ,col,data types, memory etc..

1. ecom['Purchase Price'].mean() # average purchase price

op: 50.34730200000025

1. \*\* What were the highest and lowest purchase prices? \*\*
2. ecom['Purchase Price'].max()

ecom.[‘Purchase Price’].min()

op

99.0

00.0

5. \*\* How many people have the job title of "Lawyer" ? \*\*

ecom[ecom['Job']**==**'Lawyer'].count()

1. ecom["AM or PM"].value\_counts()

value\_counts() method returns no of different catagories

PM 5068

AM 4932

Name: AM or PM, dtype: int64

\* What is the email of the person with the following Credit Card Number: 4926535242672853 \*\*

Ecom[ecom[“credir\_card’]==49327516][“Email”]

Op:

1234 bondellen@williams-garza.com

Name: Email, dtype: object

Visualization:

visualization also provides summarization, outliers identification data distbution

We can visualize the data usig bar plot,scatter plot,box plot,pie chart,etc…

Ex:

Plot

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt # for visualization

weights=[3.1,2.9,3.0,3.2,3.5,3.6,3.5,3.8]

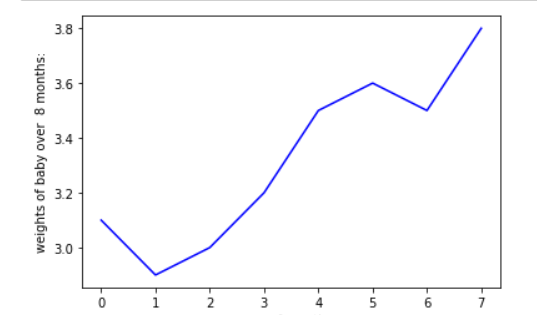
plt.plot(weights,'b') # b,r,bo,ro,g\*,g^ color and point attributes

plt.ylabel("weights of baby over 8 months:")

plt.xlabel("no of months")

plt.show()

op:



print("ploting with different points")

weights=[3.1,2.9,3.0,3.2,3.5,3.6,3.5,3.8]

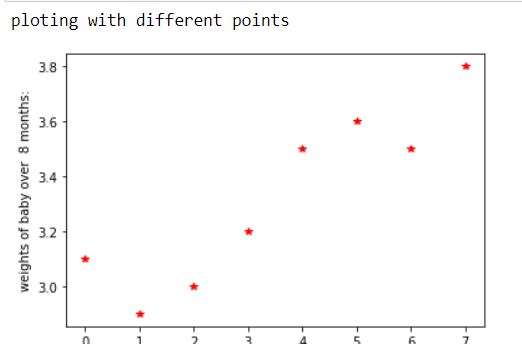
plt.plot(weights,'r\*') # b,r,bo,ro,g\*,g^ color and point attributes

plt.ylabel("weights of baby over 8 months:")

plt.xlabel("no of months")

plt.show()

op:



3.bar plot

print("bar")

country = ["ind","pak","usa","uk","bangla"]

revenue = [123.45,112,149.43,130.3,117.77]

plt.figure(figsize=(10,6)) #w=10,h=6

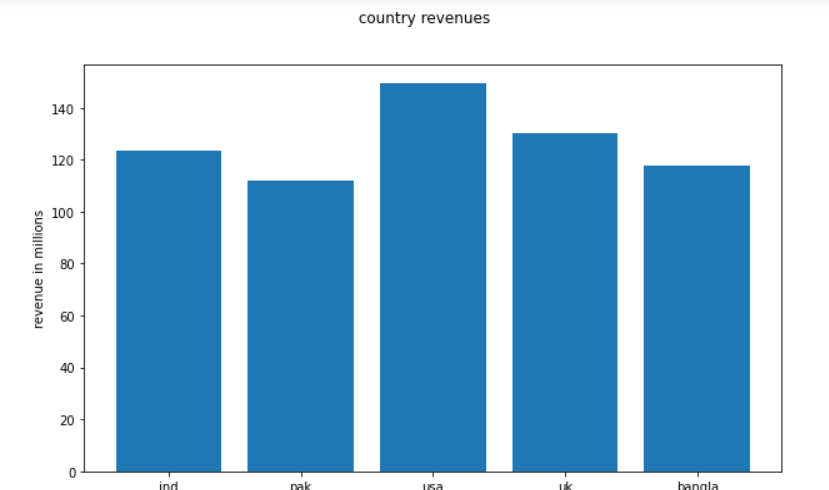
plt.bar(country,revenue)

plt.ylabel("revenue in millions")

plt.xlabel('country')

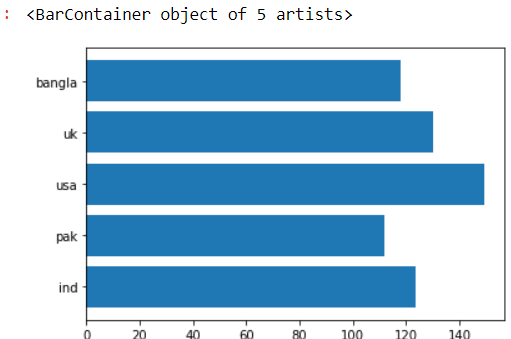
plt.suptitle(" country revenues")

plt.show()



Horizontal bar:

plt.barh(country,revenue)



Pie chart:

response = ['research','job','degree','status','passion']

counts = [22,76,55,17,10]

print("ploting 2 plots on same figure ")

plt.figure(figsize = (9,9))

plt.subplot(211)

plt.bar(response,counts,color='gray')

# vrtcle ploting

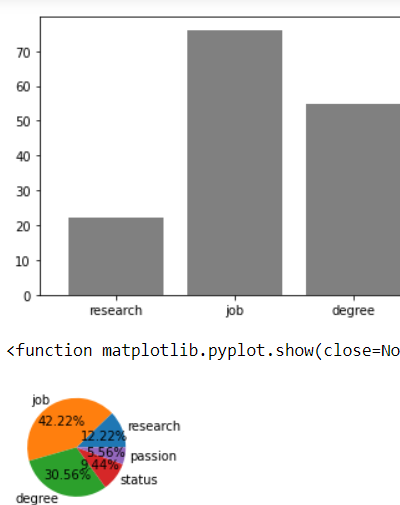
plt.show()

plt.subplot(212)

plt.pie(counts,labels=response,autopct ='%1.2f%%')

plt.show

op:



4.bubble polt with different attribure

print("bubble plot with dif colors")

#different colrs for dif points,dif size

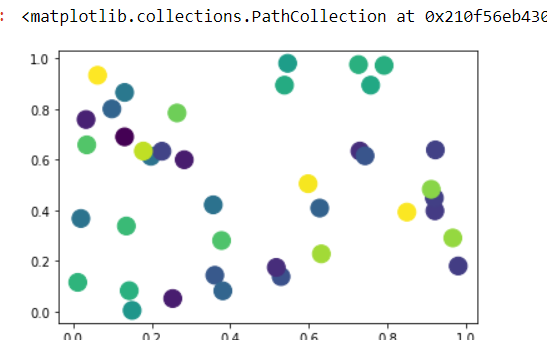
x=np.random.rand(40)

y=np.random.rand(40)

colors=np.random.rand(40)

plt.scatter(x,y,s=2\*100,c=colors)

op:



With different size ,random colors

#also we can use transparency in color: alpha=0.4/0.3/0.5

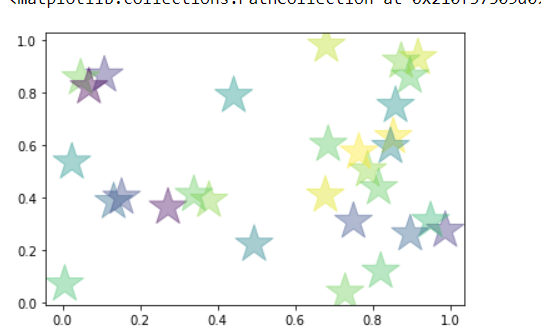
x=np.random.rand(30)

y=np.random.rand(30)

color=np.random.rand(30)

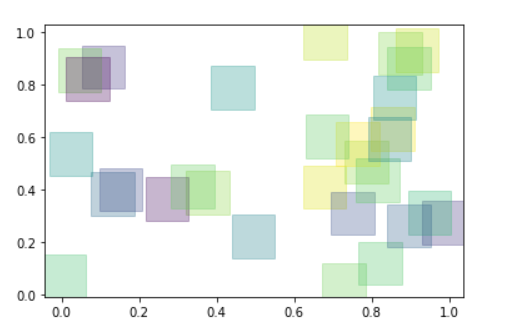
plt.scatter(x,y,c=color,s=2\*500,marker='\*',alpha=0.4) #alpha means transparency

op:



plt.scatter(x,y,c=color,s=2\*600,marker='s',alpha=0.3)

#marker attribute changes the shape of points(dot,bubble,triangle,square)



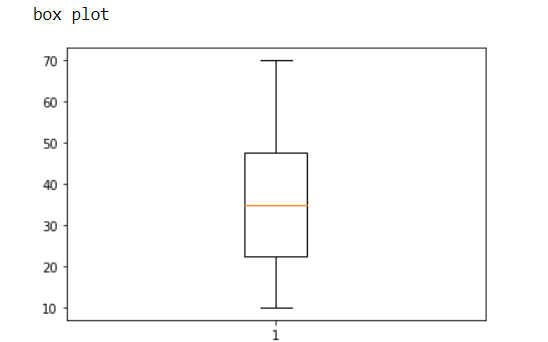
print("box plot") #used to show meand and outliers

data=[10,20,30,40,50,70]

plt.boxplot(data)

#with 1 parameter

plt.show()



#box plot provides identification of outliers,distribution of variables,mean

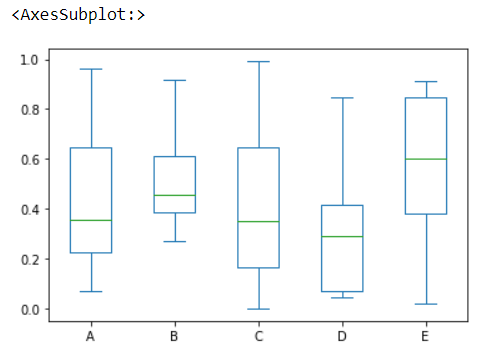
#scater plot used to visualize the clusters and outliers

Ex7:

df = pd.DataFrame(np.random.rand(10,5),columns=['A','B','C','D','E'])

df.plot.box()

op:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-date:15-09-2023

Generators:it retruns generator object

It gives iterator,

Generators are created using functions with yield keyword

Next() -> execution resumes where it is left over

If we want to one value at a time use generator

Generator s uses memory efficient

Ex:1

print("generator gives iterators:")

# genrator created with yeild keyword

def topten():

    n=1

    while n<=10:

        sq = n\*n

        yield sq

        n =  n+1

values = topten()

for i in values:

    print(i)

ex2:

# fibanoci series using generator

def fib():

    a,b =0,1

    while True:

        yield a

        a,b =b,a+b

fib\_generator = fib()

for i in range(10):

    print(next(fib\_generator))

REGULAR EXPRESSIONS:

Used to find and search required pattern

For this we have to import re module

The re module offers a set of functions that allows us to search a string for a match:

|  |  |
| --- | --- |
| **Function** | **Description** |
| [findall](https://www.w3schools.com/python/python_regex.asp#findall) | Returns a list containing all matches |
| [search](https://www.w3schools.com/python/python_regex.asp#search) | Returns a [Match object](https://www.w3schools.com/python/python_regex.asp#matchobject) if there is a match anywhere in the string |
| [split](https://www.w3schools.com/python/python_regex.asp#split) | Returns a list where the string has been split at each match |
| [sub](https://www.w3schools.com/python/python_regex.asp#sub) | Replaces one or many matches with a string |

Ex:

import re

print("\* means search for 0 or more occurence")

print(" .  means search for 1 or more occurence")

print("? means serch for 0 or 1 occurence")

print(" {} means exactly specified no of occurence")

text = "Accelyzei is an innovative , accelarate "

x=re.split("\s",text)

print(x)

y=re.split("\s",text,1) # split onl 2 occurence

print(y)

def check\_password(password):

    if 8 <= len(password) <= 20:

        if re.match('^(?=.\*\d)(?=.\*[a-zA-Z](?=.\*[@$%^&+!=].\*$)',password):

            return True

    return False

password = input("enter your password")

if check\_password(password):

    print(" you used valid password pattern")

else:

    print("not valid password pattern")

op:

['Accelyzei', 'is', 'an', 'innovative', ',', 'accelarate', '']

['Accelyzei', 'is an innovative , accelarate ']

Password:Prasanna#123

You used valid password patern

->{} means exactly specified no of occurrence

->\* means search for 0 or more occurence

-> . means search for 1 or more occurence

->? means serch for 0 or 1 occurrence

-> $ indicates end with particular string

Lambda :

It is a single line function and anonimou(name less)function

function defined using the **lambda** keyword.

Syntax: lambda arguments:expression

**nu= [1,2,3,4,5]**

**sq =list(map(lambda x:x\*\*2 ,nu)) #single line expression reduce no of line code**

**print(sq)**

**lambd function comes with different functions(map,filter,reduce etc..)**

**1.filter():**

**->**When you want to select specific elements from an iterable based on a condition we can use filter()

->It returns a new iterable containing only the elements that satisfy the given condition.

Syntax:filter(func,iterabl)

Ex: such as extracting even numbers, filtering out invalid data, or selecting items that meet certain criteria.

Ex1:

print("filter function")

nums = [2,3,6,8,4,6,2,9]

even = list(filter(lambda n:n%2==0 , nums)) #if fuction return true or flase use FILTER()

**2.map():**

**->**The **map** function is used to apply a specified function to each element of an iterable

and return an iterable containing the results.

->used to transform data by applying a function to each item in a list.

It maps to list,set,tuple.

3.reduce():

->The **reduce** function is used to repeatedly apply a specified function to the elements of an iterable (e.g., a list) and accumulate the results.

->It is often used for tasks like summing all elements in a list or finding the maximum/minimum value in a list.

Ex:

print("filter function")

nums = [2,3,6,8,4,6,2,9]

even = list(filter(lambda n:n%2==0 , nums)) #if fuction return true or flase use FILTER()

double = list(map(lambda n:n+2,even))   # map function perform operations and map to list or set

print("double of nums using map",double)

sum = reduce(lambda a,b : a+b ,double)   # from all the values gives single value from the all operations

print("sum using reduce function:",sum)

com = ['a','c','c','e','l','y','z','e','i']

name = str(reduce(lambda a,b:a+b,com))

print("company name using reduuce",name)

op:

company name using reduce: accelyzei #accumulate the list values

Serialization:

->Process of converting complex data such as objects or dictionaries into a format to store,transmit easily

->it is used to save the state of an object to file and transmit over network safely

->store data in bytes

Use cases:

1.inter process communication

2. to improve perfomence

3.save state of object

\*We use pickle librarie or json ,marshal libraries

Ex:

print("serialization : conveting complex data structure ,such as objects or dictionaries \n into a format that can be easily stored ,transmitter")

import pickle

data = {'name':'Keerthi','company':'Accelyzei','job':'developer','age':20}

with open('data.pkl','wb') as file:

    pickle.dump(data,file)    #serialization

with open('data.pkl','rb') as file:

    loaded\_data = pickle.load(file)    # deserialization

print(loaded\_data)

op:{‘name’:’keerthi’,’company’:’accelyzei’}

3.Decorators:

decorators are a powerful and flexible way to modify or extend the behavior of functions or methods without changing their code

methods to add some additional functionality or behavior.

Decorators are essentially functions that wrap other functions

Python provides several built-in decorators, such as **@staticmethod**, **@classmethod**, and **@property**, which are commonly used in object-oriented programming

Ex:

import functools

def div(a,b):

    print(a/b)  # if a>b

def smart\_div(func):

    def inner(a,b):

        if(a<b):

            a,b = b,a

        return func(a,b)

    return inner

div =smart\_div(div)

div(2,4)

we can create user defined decorators

in the above div () indirectly called using smart\_div() decorator

smart\_div provides addition features to div() function in division

uses of decorators:

\*logging

\*Authentication

\*Aspect oriented programming

\*Add extra features to existing functions

\*ApI end

\*Validations (to validate i/p params)

4.partial functions:

Partial functions in Python are functions that are derived from existing functions by fixing one or more arguments to specific values,

You can create partial functions in Python using the **functools.partial** function from the **functools** module.

Ex:

# partial functions

def multiply(x,y):

    return x\*y

double = functools.partial(multiply,2)  # create a special argument

result = double(5)  #equvalent to multiple(x\*y)

print("partial fuctions:",result)

here double(5) # is equivalent to multiple()

5.closure:

Closure is a function object that remembers values in enclosing scopes even if they are not present in memory

->flexible

->modular code design

Uses:

\*Funtion Factories

\*Data hiding and encapsulation

\*Callback functions (passing functions as arguments to other functions)

\*event handling

Ex:

print("closure remembers values enclosing scopes evnn they not present in memory")

def outer\_fun(x):

    def inner\_fun(y):

        return x+y

    return inner\_fun

closure = outer\_fun(10) # = innerfunction

print("closure : outer retuns innerfunc->",closure)

result = closure(23)  #thi add 10(from outer function) to 23

print(result)

Op:

closure : outer retuns innerfunc-> <function outer\_fun.<locals>.inner\_fun at 0x000001C0C0B88F70>

33

6.code Intraspection:

It refers to the ability to examine & analyse code at runtime.

Code inspection for debugging to understand code behaviour

Various built in functions are used

\*dir(obj) #list all the attributes(methods) of an object

\*type(obj) #determine the type of object

\*help() #information about an obj or function ,

\* import inspect #inspect module more advance access,manipulate attributes,source code,other info

Inspect.getmember(),inspect.getsource()

\*\_\_doc\_\_ # to understand the purpose of data types associated with functions & classes

\*callable() #checks if an obj callable or not

Ex:

print("code inspection")

import inspect

k = "name"

num = 4.8

def employee(a,b):

    print("names",a,b)

obj = employee('keerthi','tanuja')

print(dir(k))

op: etitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

print("type() of num=4.8 ",type(num))

print("type() of obj",type(obj))

op: type() of num=4.8 <class 'float'>

type() of obj <class 'NoneType'>

a= inspect.getmodule(obj)

print("a",a)

op:none # no module defined in the employee function

Date:20-09-2023

FastAPI features

FastAPI work flow

Path Parameters

Query Parameters

Request Body

Query Parameters and String Validations

Path Parameters and Numeric Validation

What is API:

Application programming interface . It is a set of rules and protocols that allows different software applications to communicate with each other.

It provides Data exchange ,security ,abstraction ,documentation.

FAST API:

Fast API is a Web Frame work for building RESTFUL API in python.

->when we requires high perfomence and scalable API , FAST API is better

MY APP < ----------> OTHER APP

(api) works as interface

1.Features of FAST API:

1.open API: or API creation, including declarations of path operations, parameters, body requests, security, etc.

\*Automatic data model documentation with [**JSON Schema**](https://json-schema.org/)

\*also allows using automatic **client code generation** in many languages.

2.Fast Perfomence:

As API is built on top of Starlette and Pydantic, making it one of the fastest Python web frameworks available

It provides asynchronous programming features in Python for maximum performance.

3. **Automatic Validation:**

It automatically validates request and response data using Python type hints and Pydantic models, which helps catch errors early in development.

4. **Automatic Documentation:**

FastAPI generates interactive API documentation automatically using tools like Swagger and ReDoc.

**5.Built-in Authentication:**

It provides built-in support for OAuth2, JWT, and other authentication mechanisms to secure your APIs.

6.Dependency injection

FastAPI allows you to define and inject dependencies into your route handlers,

And making it easy to manage database connections, authentication, and other resources.

**7.WebSocket Support:**

You can create WebSocket endpoints alongside traditional RESTful APIs.

Web socket: web socket refers to the ability of a web server and web application to establish multi duplex communication over channels(user-server communication)

\*browser support

8. **Background Tasks:**

**->**FastAPI supports background tasks and periodic tasks using Python's built-in asyncio features.

9. **Middleware Support:**

we can add middleware to customize the request/response pipeline.

Work Flow of FAST API:

->Controllers

->database

->model

->Schema

->Service

1. **Installation:** we start by installing FastAPI and any additional dependencies you might need

Command: Pip install fastapi uvicorn

2. **Creating a FastAPI App:**

We create a FastAPI application instance and define your API routes using Python functions.

We can also define models using Pydantic classes to validate input and output data.

3. **Defining Routes:**

we define API routes by creating Python functions with decorators like

**@app.get("/")**, **@app.post("/items/")**.

these decorators specify the HTTP method and URL path for each route.

4. **Request Handling:**

Inside your route functions, you define the logic to handle incoming requests, process data, and return responses.

**5.Running the App:**

You run your FastAPI application using a web server like Uvicorn:

Ex:uvicorn myapp:app –host—address

Here, **myapp** is the Python file containing our FastAPI application (**app** is the name of our FastAPI instance)

**6.Automatic Documentation:**

we can access the automatically generated API documentation at **http://localhost:8000/docs** or **http://localhost:8000/redoc** in your browser.

7.Testing:

We can test our application using testing tool like pytest

Pydantic:

Pydantic used in conjunction with Fast API for data validation,serialization(transmission of data)

\*Automatically convert data between data types and JSON

\*Type hinting :we can specify the expected tpes of fields

\*custom validations

Ex:

From fastapi import FastAPI

From pydantic import BaseModel

app = Myapp()

class Item(BaseModel):

name:str

price:float

is\_offer:bool = name

@app.post(“/item/”)

Async def create\_item(item:item): -> no comparision b/w db

Asynchronization: tasks can be started and continue running

Efficient

Synchronization: task executed one after other in sequential manner

Some times slower the process

Date:21-09-2023

Get() vs post()

Get

* GET requests can be cached
* GET requests remain in the browser history
* GET requests can be bookmarked
* GET requests should never be used when dealing with sensitive data
* GET requests have length restrictions
* GET requests are only used to request data (not modify)

Post:

POST is used to send data to a server to create/update a resource.

* POST requests are never cached
* POST requests do not remain in the browser history
* POST requests cannot be bookmarked
* POST requests have no restrictions on data length

HTTP methods:

GET

POST

PUT

HEAD

DELEAT

PATCH

OPTIONS

CONNECT

TRACE

3. Path Parameters:

Setp1:create a file with .py -> myapp.py()

Step2: create router with @app\_name.get(‘/’) or with your required method

Stip3:path parameters are defined within router url()

Ex:

from fastapi import FastAPI

app = FastAPI()     #object creation , app as obj name

@app.get('/')       #router creation

def home():

    return {'name':'prasanna','city':'rajam'}

# to create another function

#create another url

@app.get('/items/')  #it is one url with itmes as name

def list\_items():

    return {'company':["accelyzei","tcs","apple"],'type':'MNCs & STARTUPS'}

   # return ["prasanna","sada","aswini,"dinesh","home"]

@app.get('/items/{item\_id}')   #sending path parameters

def item(item\_id):   #path param(item\_id) to function

    return {'id':item\_id,'company':'accelyzei','location':'hyd'}

print("specifiying data type to path parameter")

@app.get('/item/{item\_id}')

def items(item\_id:int):

    return {'id\_count':item\_id,'name':'Durga Prasanna','goal':'IPS'}

send request: executipn

uvicorn main:app –reload

after pasting url

ex: https://120.0.0.1800/item/

ex: [https://120.0.0.1800/item/{item\_id}](https://120.0.0.1800/item/%7bitem_id%7d)

3.Query Parameters:

Query parameters are parameters that are included in the URL of an HTTP request.

They are typically used for optional or additional data that can modify the behavior of an API endpoint.

Query parameters are usually preceded by a question mark **?** in the URL and are in the form of key-value pairs separated by ampersands **&**

* **Query parameter defined at function parameters in router handler.**

**Ex:**

from fastapi import FastAPI

app = FastAPI()

# create router

@app.get('/items/{item\_id}')  #path parameter

def item(item\_id):

    return {'id':item\_id,'media\_name':['youtube','insta','twitter'],'fav':'Durga prasanna'}

#creating another router url

@app.get('/client/')

#query\_param:type=value\_of\_param

def client\_details(limit:int=10,active:bool=True,):

    return {'client':'{} client details'.format(limit),'active':active}

#emegency:optional[str] = None,skip:int=2

#'emergency':emergency,'skiped':skip

print("if we want query our own parameters:declare at starting of para")

#clent\_deatils()

**# we can define optional parameters like limit,active,emergency,skip**

**# by using query parameters we can change the default values of params dynamically**

**Ex:**

[**https://127.0.0.1.1800/clients/?limit=100**](https://127.0.0.1.1800/clients/?limit=100) **&active=False&skip=1**

**\*we can use user defined query perameters at stating of function parameters**

**5.Request Body**

When you need to send data from a client (let's say, a browser) to your API, you send it as a **request body**.

The request body typically contains data that is sent from the client to the server, often in JSON format, form data, or other content types

FastAPI provides an easy way to extract and parse the request body for further processing.

**1.We can Import the Request Class**:

2. **Parsing JSON Data**: with the help of pydantic class

Ex:

from fastapi import FastAPI ,Request

from pydantic import BaseModel

app=FastAPI()

#using pydantic

class New\_Employee(BaseModel):

    emp\_id:int

    name:str

    age:int

    teams:list

@app.post("/add\_employee")

def add\_employee(employee:New\_Employee) :  #sending base class

    new\_em = employee(emp\_id = employee.emp\_id,name=employee.name,age=employee.age)

    #new\_em is an instance of employee

    New\_Employee.save()

    return {'message':'employee is successfully added'}

#OR using request

@app.post("/items/")

async def create\_item(request:Request): #sending request

    #access reques body

    body = await request .body

    return{'request body':body.decode()}

1. Query parameters and string validations:

**FastAPI** allows you to declare additional information and validation for your parameters.

**Ex:**

from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")

async def read\_items(q: str | None = None):

results = {"items": [{"item\_id": "Foo"}, {"item\_id": "Bar"}]}

if q:

results.update({"q": q})

return results

**ex2:**

from fastapi import FastAPI ,Request

from pydantic import BaseModel

app=FastAPI()

#using pydantic

class New\_Employee(BaseModel):

    emp\_id:int

    name:str

    age:int

    teams:list

@app.post("/add\_employee")

def add\_employee(employee:New\_Employee) :  #sending base class

    new\_em = employee(emp\_id = employee.emp\_id,name=employee.name,age=employee.age)

    #new\_em is an instance of employee

    New\_Employee.save()

    return {'message':'employee is successfully added'}

#OR using request

@app.post("/items/")

async def create\_item(request:Request): #sending request

    #access reques body

    body = await request .body

    return{'request body':body.decode()}

In this example, we define a query parameter **search\_query** with the following validation rules:

* It must be a non-empty string (**...**).
* It has a title "Search Query" (used in documentation).
* It must have a minimum length of 3 characters.
* It must have a maximum length of 50 characters.
* It must match the regex pattern **^[a-zA-Z0-9\_]+$**

By using Pydantic models and the **Query** class from FastAPI, you can easily define and validate query parameters with various constraints, ensuring that the input data meets your requirements while keeping your code clean and readable.

7.Path Parameters and Numaric Validations:

we can define path parameters and apply numeric validations using Pydantic models.

Path parameters are used to capture values from the URL path, and you can specify numeric validations for these parameters using Pydantic's field validation features

First, import Path from fastapi, and import Annotated:

Date:22-09-2023

3.Extra Data types:

Up to now, you have been using common data types, like:

* int
* float
* str
* bool

Here are some of the additional data types you can use:

* UUID:
  + A standard "Universally Unique Identifier", common as an ID in many databases and systems.
  + In requests and responses will be represented as a str.
* datetime.datetime:
  + A Python datetime.datetime.
  + In requests and responses will be represented as a str in ISO 8601 format, like: 2008-09-15T15:53:00+05:00.
* datetime.date:
  + Python datetime.date.
  + In requests and responses will be represented as a str in ISO 8601 format, like: 2008-09-15.
* datetime.time:
  + A Python datetime.time.
  + In requests and responses will be represented as a str in ISO 8601 format, like: 14:23:55.003.
* datetime.timedelta:
  + A Python datetime.timedelta.
  + In requests and responses will be represented as a float of total seconds.
  + Pydantic also allows representing it as a "ISO 8601 time diff encoding", [see the docs for more info](https://pydantic-docs.helpmanual.io/usage/exporting_models/#json_encoders).
* frozenset:
  + In requests and responses, treated the same as a set:
    - In requests, a list will be read, eliminating duplicates and converting it to a set.
    - In responses, the set will be converted to a list.
    - The generated schema will specify that the set values are unique (using JSON Schema's uniqueItems).
* bytes:
  + Standard Python bytes.
  + In requests and responses will be treated as str.
  + The generated schema will specify that it's a str with binary "format".
* Decimal:
  + Standard Python Decimal.
  + In requests and responses, handled the same as a float.
* You can check all the valid pydantic data types here: [Pydantic data types](https://docs.pydantic.dev/latest/usage/types/types/).

**EmailStr**: This data type validates that a string is a valid email address.

**UrlStr**: It validates that a string is a valid URL.

**PaymentCardNumber**: This data type validates that a string is a valid payment card number (credit card or debit card).

**AnyUrl**: This data type validates that a string is either a relative URL or a full URL.

**SecretStr**: It's used to handle sensitive data like passwords. It ensures that the data is treated as a secret, and its value is not exposed in logs or documentation.

Ex: from pydantic import SecretStr

password: SecretStr

ex:

from fastapi import Body, FastAPI

from datetime import datetime,time,timedelta

from uuid import UUID

from typing import Annotated

app = FastAPI()

@app.post("/items/{item\_id}")

async def read\_items(

    item\_id:UUID

    start\_datetime:Annotated[datetime|None,Body(default=None)],

    end\_datetime :Annotated[datetime | None,Body(default=None)],

    repeat\_at:Annotated[time|None,Body(default=None)],

    process\_after : Annotated[timedelta|None,Body(default=None)]=None,

):

start\_process = start\_datetime + process\_after

duration = end\_datetime - start\_process

return {

    "itemid":item\_id,

    "start\_datetime":start\_datetime,

    "end\_datetime" : end\_datetime,

    "repeat\_at" :repeate\_at

}

Cookies:

Cookies are small pieces of data that are stored on the client's browser and can be used to maintain session state or store user-specific information

**fastapi.responses** module to set and get cookies in HTTP responses and requests

* 1. **Setting Cookies in FastAPI**: #**JSONResponse** class from the **fastapi.responses** module
  2. **Getting Cookies in FastAPI**: #To get cookies sent by the client in a FastAPI request,

Ex:

#cookies

@app.get("/set-cookie/")

async def set\_cookie():

    response = JSONResponse(content={"message": "Cookie set!"})

    response.set\_cookie(key="user\_id", value="12345", max\_age=3600, httponly=True)

    return response

from fastapi import FastAPI, Request

app = FastAPI()

@app.get("/get-cookie/")

async def get\_cookie(request: Request):

    user\_id = request.cookies.get("user\_id")

    return {"user\_id": user\_id}

usecases:

1. **User Authentication**: Storing a session ID or token in a cookie to authenticate users and maintain their session state.
2. **Remember Me Functionality**: Allowing users to stay logged in by setting a "remember me" cookie with a longer expiration time.
3. **User Preferences**: Storing user preferences or settings in cookies, such as theme choices or language preferences.
4. **Tracking User Behavior**: Collecting data about user behavior, such as which pages they've visited or their shopping cart contents.
5. **Cross-Site Request Forgery (CSRF) Protection**: Using anti-CSRF tokens stored in cookies to protect against CSRF attacks.
6. **Web Analytics**: Tracking user interactions and behavior on a website for analytics purposes.

haders:

work with header parameters to access and manipulate data sent in the HTTP headers of an incoming request.

HTTP headers are used to transmit metadata and configuration information between the client and the server

Used with header()

Ex:

@app.get("/process/")

async def process\_headers(

    user\_agent: str = Header(None, description="User-Agent header"),

    api\_key: str = Header(..., description="API Key header")

):

    return {"User-Agent": user\_agent, "API Key": api\_key}

Form Data :

we can handle form data (also known as HTML form data or form-urlencoded data) in your API endpoints by using the **Form** class from the **fastapi** module.

This allows you to parse and validate data submitted via HTML forms. Here's how to work with form data in FastAPI:

Ex:

from fastapi import FastAPI, Form,Header,Cookie

app = FastAPI()

@app.post("/submit-form/")

async def submit\_form(

    username: str = Form(..., min\_length=3, max\_length=20),

    email: str = Form(..., regex=r"^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$"),

    age: int = Form(..., gt=0)

):

    return {"username": username, "email": email, "age": age}

import requests

data = {

    "username": "keerthi",

    "email": "keerthichavla@gmail.com",

    "age": 20

}

response = requests.post("http://localhost:8000/submit-form/", data=data)

print(response.json())

Date: 25-09-2023

# Request Forms and Files:

# You can define files and form fields at the same time using File and Form

# Install: pip install python-multipart # for forms and file

# Ex:

from typing import Annotated

from fastapi import FastAPI, File, Form,UploadFile,Body

app = FastAPI()

@app.post("/files/") #upload file with uploadFile attribute

async def createfile(

    file:bytes = File(...),

    fileb:UploadFile=File(...),

    token:str=Form(...),

    hello:str=Body(...)

):

    return {'filesize':len(file),

            'token':token,

            'file\_content':fileb.content\_type}

# Here we use File() , Form () feilds

# # we can upload a file using UploadFile attribute

# And you can declare some of the files as bytes and some as UploadFile.

# 3.handling errors:

# Handling errors in FastAPI is an essential aspect of building robust and user-friendly web APIs.

# ->**Exception Handling:** HTTPException and RequestValidationError that you can use,

# ->**Custom Exception Handlers**: app.exception\_handler decorator

# ->**Middleware**: handle errors and perform additional error processing at a global level.

# Ex:

from fastapi import FastAPI,HTTPException,Request

from fastapi.responses import JSONResponse

from fastapi.middleware import Middleware

print("There are many situations in which you need to notify an error to a client that is using our API.")

#HTTP status code in the range of 400 (from 400 to 499).

#This is similar to the 200 HTTP status codes (from 200 to 299). Those "200" status codes mean that somehow there was a "success" in the request.

#The status codes in the 400 range mean that there was an error from the client.

#Remember all those "404 Not Found" errors (and jokes)?

app = FastAPI()

items = {"foo": "The Foo Wreslers"}

@app.exception\_handler()  #one type of decorator

@app.get("/items/{item\_id}")

async def read\_item(item\_id:str):

    if item\_id not in items:

        raise HTTPException(status\_code=404,detail="Item not found")

    return {"item":items[item\_id]}

#custom error handling

class UnicornException(Exception):

    #inherits the Exception class

    def \_\_int\_\_(self,name:str):

        self.name = name

@app.exception\_handler(UnicornException)

async def uvicorn\_exc\_handler(request:Request,exc:UnicornException):

    return JSONResponse(status\_code=418,

        content={"message": f"Oops! {exc.name} did something. There goes a rainbow..."},)

@app.get("/unicorns/{name}")

async def read\_unicorn(name: str):

    if name =='yaho':

        raise UnicornException(name=name)

    return {"unicorn\_name":name}

#/unicorns/yolo, ->the path operation will raise a UnicornException.

# Middle ware error -> perform additional error processig ar global

async def custom\_err\_middleware(request:Request,call\_next):

        try:

             response = await call\_next(request)

             return response

        except HTTPException as exc:

             return JSONResponse(content={"error": exc.detail}, status\_code=exc.status\_code)

app.middleware.append(custom\_err\_middleware)

# 4.Path Operation Configuration:

# We use several configuration to customize various aspecs of api

# Path()

# Query()

# Tags: -> set of unique tags strings for itmes

# Ex:

# to cusomize various aspects of api routes

# path(), query() ,tags = []

# tags set of unquie tag strings for this item we can add tags to parameters

from enum import Enum

from fastapi import FastAPI

app = FastAPI()

class Tags(Enum) : #tags set of unquie tag strings for this item

    items = "items"

    user = "user"

@app.get("/items/",tags = [Tags.items])

async def get\_items():

    return ['apple','apple products']

@app.get("/users/", tags=[Tags.users])

async def read\_users():

    return ["steev", "modi",

            "prasanna"]

# Import Enum

# We can add tags to your path operations,pass the parameters tags with list

# Enum: it allows fixed set of values for specific field

# 5.Json compatable encoder & Body –Updates:

# Json-javascript object notation

# It is format to sent data from clent to server ,server to client

# ->some times we might convert a data type (pydantic,datetime) to compatable JSON

# -> to store in database

# 🡪To update data in databse.

# You can use jsonable\_encoder for that.

# For Body() updates

# 1.put() ->to replace existing data

# 2.patch() ->for partial updates

# Steps:

### **Partial updates recap**[**¶**](https://fastapi.tiangolo.com/tutorial/body-updates/#partial-updates-recap)

In summary, to apply partial updates you would:

* (Optionally) use PATCH instead of PUT.
* Retrieve the stored data.
* Put that data in a Pydantic model.
* Generate a dict without default values from the input model (using exclude\_unset).
  + This way you can update only the values actually set by the user, instead of overriding values already stored with default values in your model.
* Create a copy of the stored model, updating it's attributes with the received partial updates (using the update parameter).
* Convert the copied model to something that can be stored in your DB (for example, using the jsonable\_encoder).
  + This is comparable to using the model's .dict() method again, but it makes sure (and converts) the values to data types that can be converted to JSON, for example, datetime to str.
* Save the data to your DB.
* Return the updated model.

# Ex:

#some cases where you might need to convert a data type (like a Pydantic model)

# to something compatible with JSON (like a dict, list, etc).

from datetime import datetime

from fastapi import FastAPI

from fastapi.encoders import jsonable\_encoder

from pydantic import BaseModel

fake\_db = {}

class Item(BaseModel):

    title: str

    timestamp: datetime

    description: str | None = None

app = FastAPI()

@app.put("/items/{id}")

def update\_item(id: str, item: Item):

    json\_compatible\_item\_data = jsonable\_encoder(Item)

    fake\_db[id] = json\_compatible\_item\_data

#Body updates

#to update an item you can use the HTTP PUT

# Partial updates with PATCH

from fastapi import FastAPI

from fastapi.encoders import jsonable\_encoder

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

    name: str | None = None

    description: str | None = None

    price: float | None = None

    tax: float = 10.5

    tags: list[str] = []

items = {

    "foo": {"name": "Foo", "price": 50.2},

    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},

    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},

}

@app.get("/items/{item\_id}", response\_model=Item)

async def read\_item(item\_id: str):

    return items[item\_id]

@app.patch("/items/{item\_id}", response\_model=Item)

async def update\_item(item\_id: str, item: Item):

    stored\_item\_data = items[item\_id]

    stored\_item\_model = Item(\*\*stored\_item\_data)

    update\_data = item.dict(exclude\_unset=True)   # excludes default values

    updated\_item = stored\_item\_model.copy(update=update\_data)

    items[item\_id] = jsonable\_encoder(updated\_item)

    return updated\_item

# # most of time patch() is useful than put() by using exclude\_unset=True

# Date:26-09-2023

# DEPENDENCIES

# 1.Dependency:

# What is dependency:

# dependencies are reusable components or functions that can be used to manage and share common logic across different route handlers (endpoints).

# Dependency injection in FastAPI is a powerful feature that allows you to provide dependencies

# such as database connections or configuration settings, to your route handlers in a clean and organized way.

# Depends class to inject these dependencies into your route handlers.

# Use cases:

* **Authenticate Users**: Verify user credentials, check API keys, or perform token-based authentication.
* **Retrieve Data**: Fetch data from databases, external services, or other sources before processing requests.
* **Validate Inputs**: Check and validate incoming data, such as query parameters, headers, or request bodies.
* **Manage Resources**: Establish and manage database connections, open and close files, or handle other resource-related tasks.
* **Inject Configuration**: Provide configuration settings or context-specific information to route handlers.

# reuse of common logic across multiple routes in your FastAPI application.

# Ex:dependency

from fastapi import Depends , FastAPI,Body,HTTPException,Header

from pydantic import BaseModel

from fastapi.security import OAuth2PasswordBearer #  it is class to secure password

# dependencies :Depends class to inject these dependencies into your route handlers

#a powerful feature that allows you to provide dependencies, such as

#database connections or configuration settings,

app = FastAPI()

fakedb = [{"itme1" : "smart phone"},{"item2":"laptop"},{"item3":"desktop"}]

async def hello():

    return "hello sir"

async def common\_parameters(q:str | None = None, skip:int = 0,limit:int =100,b:str=Depends(hello)):

    return {"q":q, "skip":skip, "limit":limit}

# here we derive hello dependency

# For dependency: we have to import Depends from fastapi

# Here: common\_parameter () uses hello() as dependency

# 2.classes – depends:

# using custom dependency classes allows you to encapsulate complex logic, authentication, and data retrieval in a modular way.

# Ex:

print("classes - dependencies ")

#2. use class as dependency

class commonQueryParams:

    def \_\_init\_\_(self,q:str|None,skip:int=0 , limit:int=100):

        self.q =q

        self.skip =skip

        self.limit = limit

@app.get("/items2/")

async def read\_items(common:commonQueryParams = Depends(commonQueryParams)):

    response ={} #empyt dic

    if common.q:

        response.update({'q': common.q})

    items = fakedb[common.skip:common.skip + common.limit] #assign  items to item

    response.update({'itmes':items})  # add items to response dict

    return response

# . here Commonqueryparameters class as a dependency by specifying it as a parameter.

# # so we can use whole llogic of the class in our router

# 3.sub Dependencies:

Ex: def query\_extractor(q:str | None = None):

    return q

def query\_or\_body\_extract(q:str = Depends(query\_extractor),last\_query:str | None=Body(None)):

    if not q:

        return last\_query

    return q

@app.post("/items/")

async def try\_query(query\_or\_body:str = Depends(query\_or\_body\_extract)):

    return {"query\_or\_body":query\_or\_body}

# when we call query\_or\_body ->query\_extractor  sub dependency

# Here we call query\_or\_body(dependency) -> query\_extractor

# 4.Dependencies in path operation decorators:

# In some cases we don't really need the return value of a dependency inside your path operation function.

# We can put dependencies in your path operation decorator ex:@app.get(dependencies=[Depends(verify\_item), Depends(verify\_key)])

# If we want return value of depends put them in path operation definition

# Ex:

#4.dependencies in path operators and gloabal

#put dependencies at routers

async def verify\_token(x\_token: str= Header()):

    if x\_token != "fake-super-secret-token":

        raise HTTPException(status\_code=400, detail="X-Token header invalid")

async def verify\_key(x\_key: str= Header()):

    if x\_key != "fake-super-secret-key":

        raise HTTPException(status\_code=400, detail="X-Key header invalid")

    return x\_key

@app.get("/items/", dependencies=[Depends(verify\_token), Depends(verify\_key)])

async def read\_items():

    return [{"item1": "laptop"}, {"item2": "desktop"}]

# 5. Global dependency:

# When we use dependencies for whole application then we can define globally

# Ex:

app= FastAPI(dependencies=[Depends(verify\_token),Depends(verify\_k

# uses:

# ->understanding easy

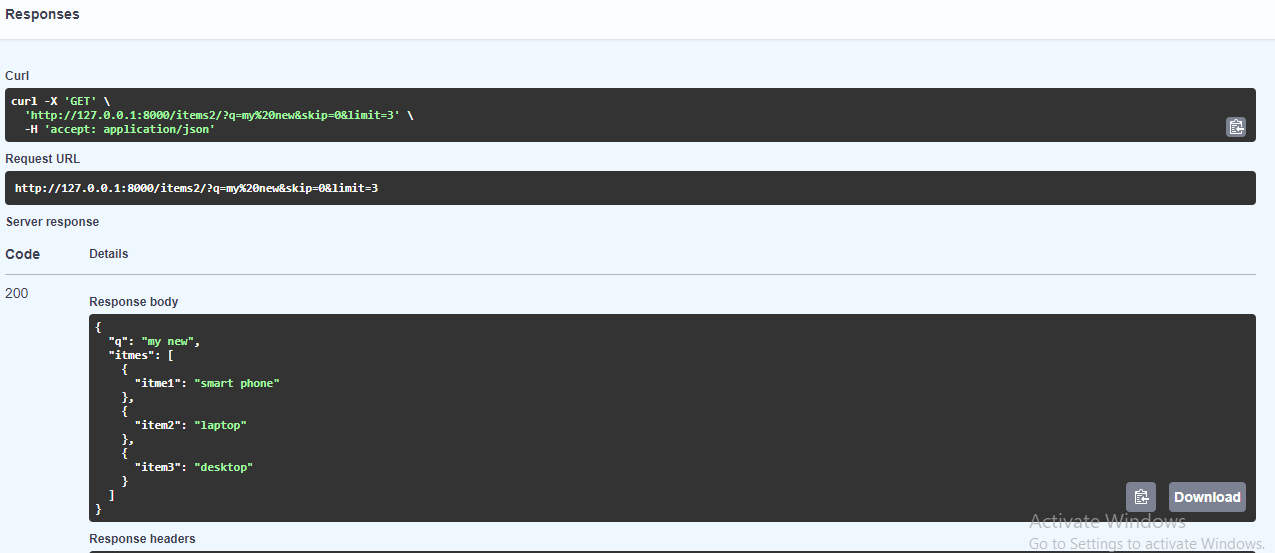
# ->clean code

# ->reusability

# SECURITY:

# 

# 



Date:27-09-2023

SECURITY:

you want to have a way for the frontend to authenticate with the backend, using a **username** and **password**.

We can use **OAuth2** to build that with **FastAPI**

First install [python-multipart](https://andrew-d.github.io/python-multipart/).

E.g. pip install python-multipart.

This is because **OAuth2** uses "form data" for sending the username and password.

Some of considerations to use fast api

1. **Input Validation and Data Serialization**
2. **Authentication**

**->**You can integrate OAuth2, JWT (JSON Web Tokens), API keys, or other custom authentication methods to protect your endpoints.

3. **Password Hashing: we can use hashed\_password using bcrypt** or **passlib** to securely hash and store user passwords.

4. **Cross-Site Scripting (XSS) Prevention**:

5. **Cross-Site Request Forgery (CSRF) Protection**:to protect that import **Depends** function to include CSRF protection in your routes.

6. **HTTPS**: Always deploy your FastAPI application over HTTPS to encrypt data transmitted between the client and server.

7. **Regular Updates**:

8. **Content Validation**:

Ex:

# for security purpose import oAuth2

from fastapi import FastAPI,dict, HTTPException, status

from fastapi.security import OAuth2PasswordBearer ,OAuth2PasswordRequestForm

#  it is class to secure password

from fastapi import Depends

from pydantic import BaseModel

app = FastAPI()

oauth2\_scheme = OAuth2PasswordBearer(tockenUrl = "token")

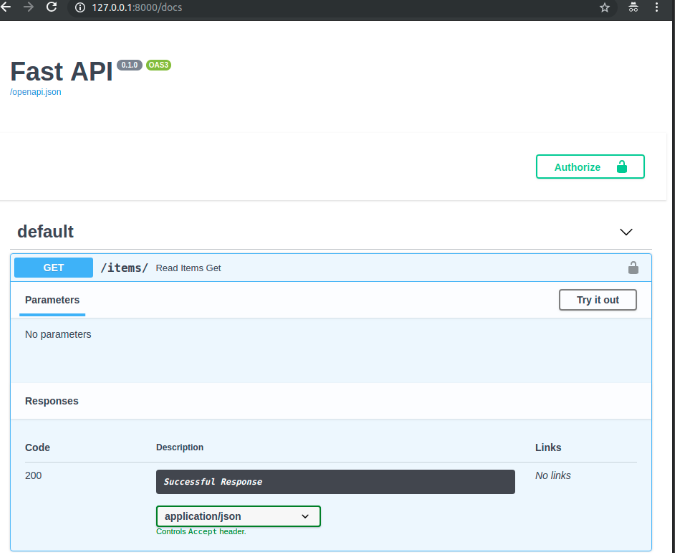
@app.get("/items/")  #decorator

async def read\_items(token: str = Depends(oauth2\_scheme)) : #using dependencies

    return {"token":token}

op:

when we import oAuth2 then it will show Authorize box



**Ex 2:**

**Get username and password**

**->create a fake database with some user data ,and sending username passwords securely**

# create a database

fake\_database = {

    "keerthi": {

        "username":"KeerthiChavla" ,

        "email":"vkeerthichavla@gmail.com",

        "fullname":"Keerthi chavla vandana",

        "hashed\_password":"mysismyprinces",  # it checks plain password and hashed password

        "disable":False

     } ,

    "prasanna" : {

        "username":"prasanna",

        "email" : "durgaprasanna@gmail.com",

        "fullname":"v.DurgaPrasanna",

        "hashed\_password":"mysismyworld",

        "disable": True

    }

}

def fake\_hash\_password(password: str):

    return f"fakehashed password:{password}"

def userInDb(user):

    hashed\_password = str

class user(BaseModel):

    username:str

    email:str|None = None

    full\_name:str |None=None

    disabled:bool |None = None

def get\_user(db,username:str):  #username=token

    if username in db:

        user\_dict = db[username]

        return userInDb(\*\*user\_dict)

def fake\_decode\_token(token): #definition

    return get\_user(fake\_database,token) #return these to get\_user method

    #return user(

       # username = token + "fakedecoded" , email = "keerthi@gmail.com" ,full\_name ="keerthi chavla"

   # )

async def get\_current\_user(token:str= Depends(oauth2\_scheme)): #dervies dependency

                        # name:type=field()

    user = fake\_decode\_token(token)

    return user

@app.get("/items/")  #decorator

async def read\_items(token: str = Depends(oauth2\_scheme)) : #using dependencies

    return {"token":token}

@app.get("/users/me")

async def get\_me(current\_user:user=Depends(get\_current\_user)):

   return current\_user

**Token: Token piece of data ,often a string to prove identity of user**

**->access token ,authentication token**

**->user receive an access to verify permission or grant then user must include in their request**

**->**Normally, a token is set to expire after some time.

Token URL:

Specific endpoint URL where client sends that token

When we create an instance of the OAuth2PasswordBearer class we pass in the tokenUrl parameter.

This parameter contains the URL that the client (the frontend running in the user's browser) will use to send the username and password in order to get a token.

security with json web tocken and uses:

JWT is a compact, self-contained way to represent information between two parties securely as a JSON object.

Uses cases:

1. **Authentication**: When a user logs in, the server generates a JWT containing user information (such as user ID or username) and signs it with a secret key
2. **Stateless Authentication**: server doesn't need to store session information on the server-side
3. **Single Sign-On (SSO)**:
4. **Secure Data Exchange**

import the tools we need from passlib.

Create a PassLib "context". This is what will be used to hash and verify passwords.

**Date:28-09-2023**

**MIDDLEWARE:**

Middleware in FastAPI is a powerful feature that allows you to perform actions before and after handling requests and responses.

Middleware functions are executed in a specific order for each incoming HTTP request,

and they can be used to add custom logic, modify request data, or perform various tasks like authentication, logging, error handling, and more.

We add middileware to our application : **app.middleware** attribute

->it should be a coroutine that takes two arguments: **request** and **call\_next**

**Request:** contains information about the incoming HTTP request, including details like the HTTP method, URL, headers, query parameters, and request body. Access to request content

**Call\_next:** represents the next middleware in the middleware stack or the main route handler.

Ex:

from fastapi import FastAPI,Request

from fastapi.middleware.cors import CORSMiddleware

from starlette.middleware.base import BaseHTTPMiddleware

#custom middleware

# Define a middleware function

app=FastAPI()

class custom\_middleware(BaseHTTPMiddleware):

    async def custom\_middleware(self,request:Request,call\_next):

        # this code runs before handlng the request

        print("Before request")

        response = await call\_next(request) # continue to next route

        print("After request")

        return response

app.add\_middleware(custom\_middleware) #allow http request

@app.get("/blah")  # router will execite after the middleware

async def read():

    return {"message":"Hello mom and sis"}

Here we define a simple route handler using **@app.get("/")**. This route handler will be executed after the middleware.

**await call\_next(request)** to pass control to the next middleware or the main route handler.

Op:{“message”:”hello mom and sis”}

A function call\_next that will receive the request as a parameter

Then it returns the response generated by the corresponding *path operation*.

Middleware takes each **request** that comes to your application.

* Then it passes the **request** to be processed by the rest of the application (by some *path operation*).

Uses of middleware:

enhancing the capabilities of your web application by providing a way to intercept and process requests and responses.

1.Authentication

Ex: For example, you can create a middleware that checks for authentication tokens in request headers, validates user sessions, or verifies API keys.

**2.Authorization:**

We can check roles, permissions, or ownership before allowing access.

**3.Logging**: This is valuable for debugging and auditing purposes.

4. **Request and Response Transformation**: Middleware can modify the request or response before they reach the route handler.

For instance, you can preprocess request data, sanitize inputs, or format responses consistently.

**5.CORS Handling**: Middleware can handle Cross-Origin Resource Sharing (CORS) headers, allowing or denying access to your API from specific domains or origins.

6. **Error Handling**: Middleware can catch and handle errors that occur during request processing.

It can customize error responses and provide meaningful error messages to clients.

**7.Request Timing and Metrics**: Middleware can record request/response timings and metrics for monitoring and performance analysis.

8. **Content Type Handling**: Middleware can ensure that request and response content types are appropriate, converting data formats if necessary (e.g., JSON to XML).

9.security:

**Date:29-09-2023**

**Get(),put(),post(),delete() methods:**

FastAPI provides easy-to-use decorators for handling various HTTP methods such as GET, POST, PUT, and DELETE.

**Get():**To handle a GET request in FastAPI, you can use the **@app.get()** decorator.

**Purpose:** Used to request data from a specified resource.

**Use Cases:**

* Fetching data from a server/database.
* Retrieving information without changing the state of the server.
* Idempotent operations (multiple identical requests will produce the same result).

2.post():

**Purpose:** Used to submit data to be processed to a specified resource.

->send data via headers only

**Use Cases:**

* Submitting form data.
* Uploading a file.
* Creating a new resource on the server.
* Non-idempotent operations (each request can result in a different state on the server).

### 3. **PUT Request:**

**Purpose:** Used to update a resource or create it if it does not exist.

**Use Cases:**

* Updating existing data.
* Creating a resource if it doesn't exist (when the resource can be created at a specific URL).

### **4.DELETE Request:**

Used to request the removal of a resource.(Delete the data (ex:username))

Send data via header/url

**Use Cases:**

* Deleting a specific resource on the server.
* Irreversible actions where data needs to be removed permanently.

Ex:

from fastapi import  FastAPI ,Form,Body,Request

from fastapi.templating import Jinja2Template

from fastapi import HTMLResponse

from pydantic import BaseModel

app = FastAPI()

#Basemodel

class read(BaseModel):

    name:str

    country:str

    age:int

    salary:float = None #not much required

@app.post("/language/") #router

async def language(language\_name:str=Form(...),country:str=Form(...)):

    return {"name":language\_name,"country":country}

# sending data using get,post,put,delete methods

list\_of\_users = list()

@app.get("/home/{username}") #router

def write\_home(username:str,age:int):

    return{

        "username":username,

        'age':age

    }

@app.post("/postdata/") # we can send data via header only

def put\_data(username:str):

    list\_of\_users.append(username)

    return{

        'username':list\_of\_users

    }

#send data via put request

@app.put('/username/{user\_name}')

def put\_data(username:str):

    print(username)

    list\_of\_users.append(username)

    return {'usename':username}

# delete data via DELETE()

@app.delete("/deleteData/{username}")

def delete\_Data(username:str):

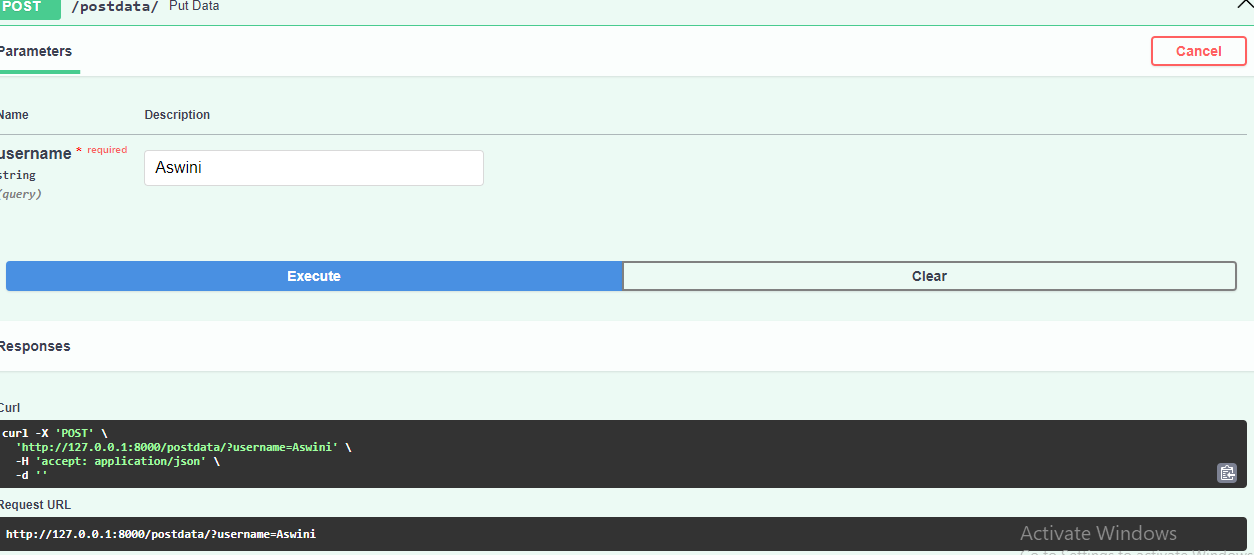
    list\_of\_users.remove(username)

    return list\_of\_users

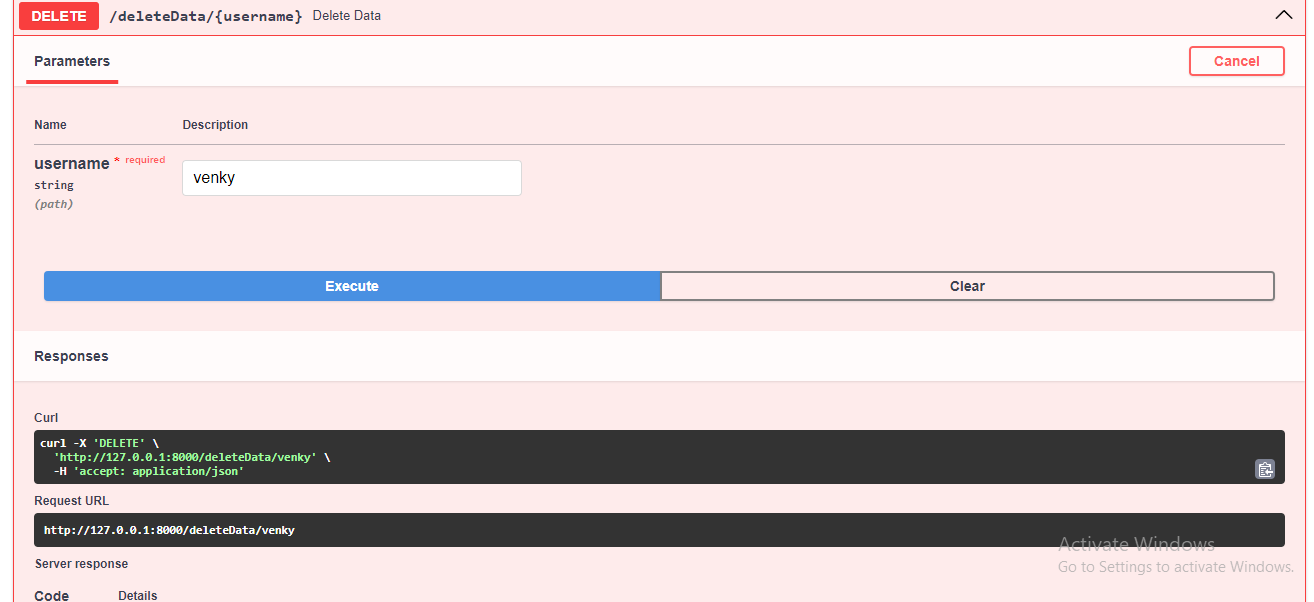
For example, in a To-Do list application:

* **GET** might retrieve all tasks or a specific task.
* **POST** could be used to add a new task.
* **PUT** could be used to update an existing task.
* **DELETE** could be used to remove a task from the list.

**Delete(): delete the given data**

**Op:adding usernames** ****

Delete particular user:

****

**After deletion of user ->users\_list:**

****

**Here we delete user venky from list using delete()**

**We show Remaining users list**

**Method 2:**

**We can specify our requires request methods in single router using :@api.router(method=[“GET”,”POST”,”DELETE”,”PUT”])**

**Ex:**

print(" defineing all request methods in single route")

@app.api\_route("/homedata/", methods = ["GET","POST","PUT","DELETE"])

def handle(username:str):

    return {"username": list\_of\_users}

****

**Sending data Dynamically to html page**

**Import jinja2**

**We have to give path of html page at templte**

**Ex:**

**Template= Jinja2Templates(directory=”head.html”)**

**And also import HTMLResponse**

**Ex:** templates=Jinja2Templates(directory="html.html")

#sending data dynamically

@app.get("/home/{username}",response\_class=HTMLResponse) #send data to html page

def write\_home(request:Request,username:str):

    return templates.TemplateResponse('html.html',{request:request,"username":username})

#what is the request coming from user that use as context.

# we send username to html page

**Date:03-10-2023**

**CORE(cros origin resource sharing):**

**Cors:it refers to** the situations when a frontend running in a browser has JavaScript code that communicates with a backend, and the backend is in a different "origin" than the frontend.

Origin : An origin is the combination of protocol (http, https), domain (myapp.com, localhost, localhost.tiangolo.com), and port (80, 443, 8080).

Ex:

* http://localhost//gmail.com
* https://localhost:8000
* http://localhost:8080

let's say you have a frontend running in your browser at http://localhost:8080, and its JavaScript is trying to communicate with a backend running at http://localhost

the browser will send an HTTP OPTIONS request to the backend, and if the backend sends the appropriate headers authorizing the communication from this different origin (http://localhost:8080) then the browser will let the JavaScript in the frontend send its request to the backend.

To achieve this, the backend must have a list of "allowed origins".

Wildcards: list as "\*" (a "wildcard") to say that all are allowed.

But that will only allow certain types of communication, excluding everything that involves credentials: Cookies, Authorization headers like those used with Bearer Tokens, etc.

## **Use CORSMiddleware:**

Steps:

->import CORSMiddleware

-> create list of allowed origins

->add it as middleware to our application

We can specify at backend:

* Credentials (Authorization headers, Cookies, etc).
* Specific HTTP methods (POST, PUT) or all of them with the wildcard "\*".
* Specific HTTP headers or all of them with the wildcard  "\*".

Ex:

from fastapi import FastAPI

from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

origins = [

    "http://localhost.google.com",

    "https://localhost.tiangolo.com",

    "https://gmial.com",

    "http://localhost",

    "http://localhost:8080",

]

#add to middleware

app.add\_middleware(

    CORSMiddleware,

    allow\_origins = origins,  # allow any origin

    allow\_credentials = True,

    allow\_methods = ["\*"], # allow get ,post etc..

    allow\_headers = ["\*"],  # requests all types of https headers

)

@app.get("/")

async def main():

    return {"message": "Hello World"}

note:

allow\_origins cannot be set to ['\*'] for credentials to be allowed, origins must be specified.

Allow\_method:that allows any request standard methods [“post”,”GET”,”put”]

Allow\_header: allows any cross- origin request

Max\_age: set max time in seconds for browser to cache response

UPLOAD FILE AND FORM IN FASTAPI:

from fastapi import  FastAPI ,Form,Body,Request,Body,File,File,UploadFile,Depends

from fastapi.templating import Jinja2Templates

from fastapi.responses import HTMLResponse

from pydantic import BaseModel

from fastapi.security import OAuth2PasswordBearer,OAuth2PasswordRequestForm

from typing import List

app = FastAPI()

#Basemodel

class read(BaseModel):

    name:str

    country:str

    age:int

    salary:float = None #not much required

templates=Jinja2Templates(directory="html.html")

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.post("/token")

async def token\_generate(form\_data:OAuth2PasswordRequestForm=Depends()):

    print(form\_data)

    return {"access\_token":"form\_data.username","token\_type":'beares'}

#if user name presnallow inthis method

@app.post("/user/profilepic") #user with profilephots

async def profile\_pic(token:str = Depends(oauth2\_scheme)):

    print(token)

    return {

        "user":"keerthi",

        "profilepic" :"my face"

    }

Connect this with html file:

<body>

        <h1>this is Home page</h1>

        <h2>USERNAME:{{username}}</h2>

        <form action = "/submitform" method="post" enctype="multipart/form-data">

            <input type="text" name="assignment">

            <input type="file" name="assignment\_file">

            <input type="submit">

        </form>

    </body>

Outh2passwordBears – will give route for us to understand where the token is from /where it is going to be stored

Oauth\_schema = oAuth2PasswordBearer(tokenUrl = “token”)

@app.get(“token”) which is going to be taken from this one

Hashpassword /password in db equal then generate a token(username)

**3.CREATE DATABASE AND SEND DATA INTO DATABASE: SQL DATA BASE**

**Stpes:**

**We can use any relational data base in FASTAPI:**

**Example dbms softwares:**

* PostgreSQL
* MySQL
* SQLite
* Oracle
* Microsoft SQL Server, etc.

ORM": an "object-relational mapping" library.

An ORM has tools to convert ("*map*") between *objects* in code and database tables ("*relations*").

With an ORM, you normally create a class that represents a table in a SQL database, each attribute of the class represents a column, with a name and a type.

**Ex:** a class Pet could represent a SQL table pets.

And each *instance* object of that class represents a row in the database.

**->install SQLAlchemy**

**->create the SQLaAlchemy parts**

**->create a database URL for SQLAlchemy**

**->create SQLAlchemy engine**

**\* sessionlocal class**

**\* Base class**

**-> create database models**

**-> create Alchemy models from the Base class**

**->create colums,attributes**

**->create relationships**

**-> use pydantic ORM\_mode**

**Ex:**

SQLALCHEMY\_DATABASE\_URL = f"postgresql://{DB\_USER}:{DB\_PASSWORD}@{DB\_HOST}:5432/{DB\_NAME}"

# print(SQLALCHEMY\_DATABASE\_URL)

engine = create\_engine(SQLALCHEMY\_DATABASE\_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

# logging.basicConfig()

# logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)

def get\_db():

    db = SessionLocal()

    try:

        yield db

except :

“SOME DATABASE ERROR”

    finally:

        db.close()

**File Structure:**

.

└── sql\_app

├── \_\_init\_\_.py

├── crud.py

├── database.py

├── main.py

├── models.py

└── schemas.py

\_\_init\_\_.py is just an empty file, but it tells Python that sql\_app with all its modules (Python files) is a package.

And also install: pip install SQLAlchemy.

Date:04-10-2023

Step by step Database Connection with fast api:

### **Import the SQLAlchemy parts**[**¶**](https://fastapi.tiangolo.com/tutorial/sql-databases/#import-the-sqlalchemy-parts)

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

create a main.py file

ex:

from fastapi import FastAPI ,Depends,HTTPExcption

from sqlalchemy import create\_engine

from sqlalchemy.orm import Session,sessionmaker  # generates new session objects

from models import Base,Item

->create engine, sessionmaker, are imported for database operations.

->engine responsible for manage database connections

->session maker responsible for providing database session to routers.

### 2. **Create a database URL for SQLAlchemy**

### 3. **Create the SQLAlchemy engine**

### 4. **Create a SessionLocal class**[**¶**](https://fastapi.tiangolo.com/tutorial/sql-databases/#create-a-sessionlocal-class)

Ex: Base = declarative\_base #class for declarative definitions,schemas ,relationships

# Database URL

DATABASE\_URL = "mysql:username:password@hostname:port/database\_name"

# create DB engine

engine = create\_engine(DATABASE\_URL)

#Crate Tables on startup

Base.metadata.create\_all(bind = engine)

#session class to handle db connection

sessionlocal = sessionmaker(autocommit=False,autoflush=False,bind =engine)

here autocommit = False,autoflush = false -> control over when commit changes to database

session local class means instance of db session -> use sessionmaker() function

## **Create the database models**

Create model.py(to define Tables ,colums for database model)

* + from database import Base

ex:

from sqlalchemy import Column,Integer,String,Boolean,ForeignKey

from sqlalchemy.ext.declarative import declarative\_base

from pydantic import BaseModel

from sqlalchemy.orm import relationship

from fastapi import FastAPI,Field

Base = declarative\_base()

class Item(Base):

    \_\_tablename\_\_ = "items"  # items as table

    id = Column(Integer, primary\_key=True,index=True) #colums

    name = Column(String, index = True)

    description = Column(String)

class employee(Base):

    \_\_tablename\_\_ = "employee"

    em\_id = Column(Integer,primary\_key=True,index = True)

    name = Column(String,index = True)

    description = Column(String)

#define user class=table

class user(Base):

    \_\_tablename\_\_ = "users"

    id = Column(Integer,primary\_key=True, index=True)

    email = Column(String,unique=True, index=True)

    hashed\_password = Column(String)

    #add relation ship

    items = relationship("Item",back\_populates="owner")

**here we create “item” , “user” ,”employee “ models(tables)**

SQLAlchemy uses the term "**model**" to refer to these classes and instances that interact with the database.

### 6. **Create model attributes/column**

We use Column from SQLAlchemy as the default value.

We can use type:string,int,etc

To define model: \_\_tablename\_\_ = “table\_name”

Col\_name = Column(type)

Id = Column(int,primary\_key = True,index=True)

7.create relation ships:

we use relationship provided by SQLAlchemy ORM.

attribute that will contain the values from other tables related to this one.

**Ex:** #add relation ship

    items = relationship("user",back\_populates="Item")

When accessing the attribute items in a User, as in my\_user.items, it will have a list of Item SQLAlchemy models (from the items table) that have a foreign key pointing to this record in the users table.

8.

## **Create the Pydantic models:**

models.Base.metadata.create\_all(bind=engine)

app = FastAPI()

books = []

class Book(BaseModel):

    title:str = Field(min\_length=1)

    author:str = Field(min\_length=1,max\_len=100)

    description:str = Field(min\_len=1,max\_length=100)

    rating:int = Field(gt= 0)

def get\_db():

    db = sessionlocal()  #databse session

    try:

        yield db      # returns the databse session to route

    except:

        return "database error"

    finally :

        db.close()

means defineing our schema(logic ) in BaseModel.

create a file name schema.py/book.py etc…

### **9.Use Pydantic's orm\_mode:**

Orm(object Relational mapping) ->used to map python objects to databse tables & vice versa

Each python class- Table

Instance individuals – rows

This [Config](https://pydantic-docs.helpmanual.io/usage/model_config/) class is used to provide configurations to Pydantic.

Ex:

class User(UserBase):

id: int

is\_active: bool

items: list[Item] = []

class Config:

orm\_mode = True

Pydantic's orm\_mode will tell the Pydantic model to read the data even if it is not a dict, but an ORM model.

**9.CRUD UTILITIES:**

**CRUD** comes from: **C**reate, **R**ead, **U**pdate, and **D**elete.

Means we can #create item, get item ,delete using Decoraters and routers

#create utem using post

@app.post("/items/")

def create\_item(item:Item,db: Session = Depends(get\_db)):

    db.add(item)

    db.commit()

    db.refresh(item)

    return item

#read item using get

@app.get("/items/{item\_id}")

def read\_item(item\_id: int, db: Session = Depends(get\_db)):

    item = db.query(Item).filter(Item.id == item\_id).first()

    if item is None:

        raise HTTPExcption(status\_code=404,detail="item not found")

    return item

Create Data:

* Create a SQLAlchemy model *instance* with your data.
* Add - that instance object to your database session.
* Commit- the changes to the database (so that they are saved).
* Refresh- your instance (so that it contains any new data from the database, like the generated ID).

### **Create a dependency:**

#dependency

app = FastAPI()

def get\_db():

    db = sessionlocal()  #databse session

    try:

        yield db      # returns the databse session to route

    except:

        return "database error"

    finally :

        db.close()

**we can creatae dependency with sessionlocal()**

**yield class retuns the values(database session) to router**

**DATE: 05-10-2023**

**BACK GROUND TASKS:**

Background tasks allow you to execute functions in the background while continuing to process the main request.

This is useful for performing time-consuming tasks without delaying the response to the client.

Some of small b.g tasks ->writing to a file

->sending an email,msg

-> send OTP msg

Big b.g tasks ->playing video

1. **Import BackgroundTasks : from fastapi import BackgroundTasks**
2. **Define a Background Task Function**
3. **Use BackgroundTasks in our API Route**
4. # we can run background tasks
5. # useful for performing time-consuming tasks without delaying response to clinet
6. from fastapi import FastAPI,BackgroundTasks
7. import time
8. app = FastAPI()
9. def handle\_email\_backgroun(email:str,data:str):       # back ground task function
10. print(email)
11. print(data)
12. for i in range(100):
13. print(i)
14. time.sleep(0.1)
15. @app.get("/users/email")
16. async def handle\_email(email:str,backgroun\_task:BackgroundTasks):
17. print(email)
18. backgroun\_task.add\_task(handle\_email\_backgroun,email,"email sent") #we sebd fun to be run in backg
19. return {'user':'keerthi','msg':'email msg sent'} # meanwhile write the response
20. # bigger applications and multiple files
21. print("path operations in router")
22. # import module
23. from fastapi import APIRouter,Header,HTTPException
24. router = APIRouter()
25. @router.get("/users/",tags=["users"])
26. async def read\_users():
27. return [{"username": "keerthi"}, {"username": "siddu"}]
28. @router.get("/users/me",tags=["users"])
29. async def read\_user\_me():
30. return {"username":"prasanna"}
31. @router.get("/users/{username}", tags=["user"])
32. async def read\_user(username: str):
33. return {"username":username}
34. # we are going to need some dependencies used in several places of the application
35. # so we can maintain our dependenci modules: app/dependencies.py
36. # to read custom read header X-Tocken

**In the above program :**

**Handle\_email\_background(email:str, data:str) # it is background function to be run,** This functions can perform any task you need, such as sending emails, processing data, or performing calculations.

we have to include the **BackgroundTasks** parameter. When you call this route,function.

Inside path operation function pass our task function to the background task object with .add\_task()

Add\_task() takes parameters: ->fun to be run in background

->arguments to be write

->any key arguments

BIGGER APPLICATIONS – MULTIPLE FILES:

When we are doing large applications , we can include files from one place to another

Using path operations/router.

Why we have to organize the files:

* Maintain to clear understandable code
* Easier to collaborate with other developers
* To manage larger projects

EX Main files:

### **Main File (**main.py**): contains fastapi instance & configurations**

### 2. **Routers (e.g.,** routers/item\_router.py**,** routers/user\_router.py**): responsible for specific set of routers**

### 3. **Models (e.g.,** models.py**): table creation**

### 4. **Dependency Modules (e.g.,** dependencies/auth.py**): if we have dependencies like authentication func.. to organize those**

### 5. **Settings (e.g.,** settings.py**): centralize your settings & configurations in separate file**

6. **Main Directory Structure:**

**Ex1:** from fastapi import FastAPI

from routers import item\_router, user\_router

app = FastAPI()

app.include\_router(item\_router.router)

app.include\_router(user\_router.router)

The main file typically contains the FastAPI app instance and the main configuration settings

In this example, **item\_router** and **user\_router** are sub-modules that contain route definitions for items and users, respectively.

Ex2: Each router is responsible for a specific set of routes.

# import module

from fastapi import APIRouter,Header,HTTPException

router = APIRouter()

# users module

@router.get("/users/",tags=["users"])

async def read\_users():

    return [{"username": "keerthi"}, {"username": "siddu"}]

@router.get("/users/me",tags=["users"])

async def read\_user\_me():

    return {"username":"prasanna"}

@router.get("/users/{username}", tags=["user"])

async def read\_user(username: str):

    return {"username":username}

**to handle that we have to import APIRouter module**

**model:**

**in this we define our table Structure:**

**Dependencies:**

**(depnedencies .py)**

**We have dependencies like authentication func etc.. for this we can organize them**

from fastapi import FastAPI , Header,HTTPException,Depends

# we are going to need some dependencies used in several places of the application

# so we can maintain our dependenci modules: app/dependencies.py

# to read custom r header X-Tocken

async def get\_token\_header(x\_token:str = Header()):

    if x\_token != "fake-secret-token":

        raise HTTPException(status\_code=400,detail="X-Token header invalid")

async def get\_query\_token(token:str):

    if token != "keerthi":

        raise HTTPException(status\_code=404,detail = "no keerthi token provided")

Settings: we can centralize our settings & configurations in a separate file

project-directory/

├── main.py

├── routers/

│ ├── \_\_init\_\_.py

│ ├── item\_router.py

│ └── user\_router.py

├── models.py

├── dependencies/

│ ├── \_\_init\_\_.py

│ └── auth.py

└── settings.py

**DATE:06-10-2023**

**META DATA and DOCS URL:**

we can use metadata and doc strings to provide additional information about your API endpoints.

Fastapi provides some parameters(attributes) etc.. to provide detailed information about the API or route.

Ex: -> Metadata or Doc strings

Meta Data for API

| **parameter** | **Type** | **Description** |
| --- | --- | --- |
| title | str | The title of the API. |
| summary | str | A short summary of the API. Available since OpenAPI 3.1.0, FastAPI 0.99.0. |
| description | str | A short description of the API. It can use Markdown. |
| version | string | The version of the API. This is the version of your own application, not of OpenAPI. For example 2.5.0. |
| terms\_of\_service | str | A URL to the Terms of Service for the API. If provided, this has to be a URL. |
| contact | dict | The contact information for the exposed API. It can contain several fields.contact fields   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |
| license\_info | dict | The license information for the exposed API. It can contain several fields.license\_info fields   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |

You can set them as follows:

from fastapi import FastAPI

**Tip**

We can write Markdown in the description field and it will be rendered in the output.

With this configuration, the automatic API docs would look like:

# to provide additional information aabout api

# we use meta data & doc urls using some attributes

print("to provide additional information aabout api \n  we use meta data & doc urls using some attributes")

print("\n attributes: tags = [] ,summary='' ,\n description etc..")

from fastapi import FastAPI

# we can also add additional metadata for different tags used to group our path operations

tags\_metadata=[{

    "name":"users",

    "description":"user operations"

},

{

    "name":"items",

    "description":"we can manage items",

    "external Docs":{

        "description":'Items external data',

        "url":"https://gmail.com"

    }

}]

app = FastAPI(

    openapi\_tags = tags\_metadata,

    title="My API",

    description = "small desc of API it can use markdown",

    version = "0.0.1",

    terms\_of\_service="https://github.com/git" , # it can be url

    contact = dict(

        name = "keerhi\_chavala", # contact is type of dict

        url = "http://keerthivandana58/github.com",

        gmail = "vkeerthichavla@gmail.com"

    ),

    license\_info=dict(name = "Startup",url="https://www.startupindia.gov.in/")

)

# define routes

@app.get("/user/",tags = ['user'])

async def get\_user():

    return [dict(name = "keerthi")]

@app.get("/items",tags=['items'])

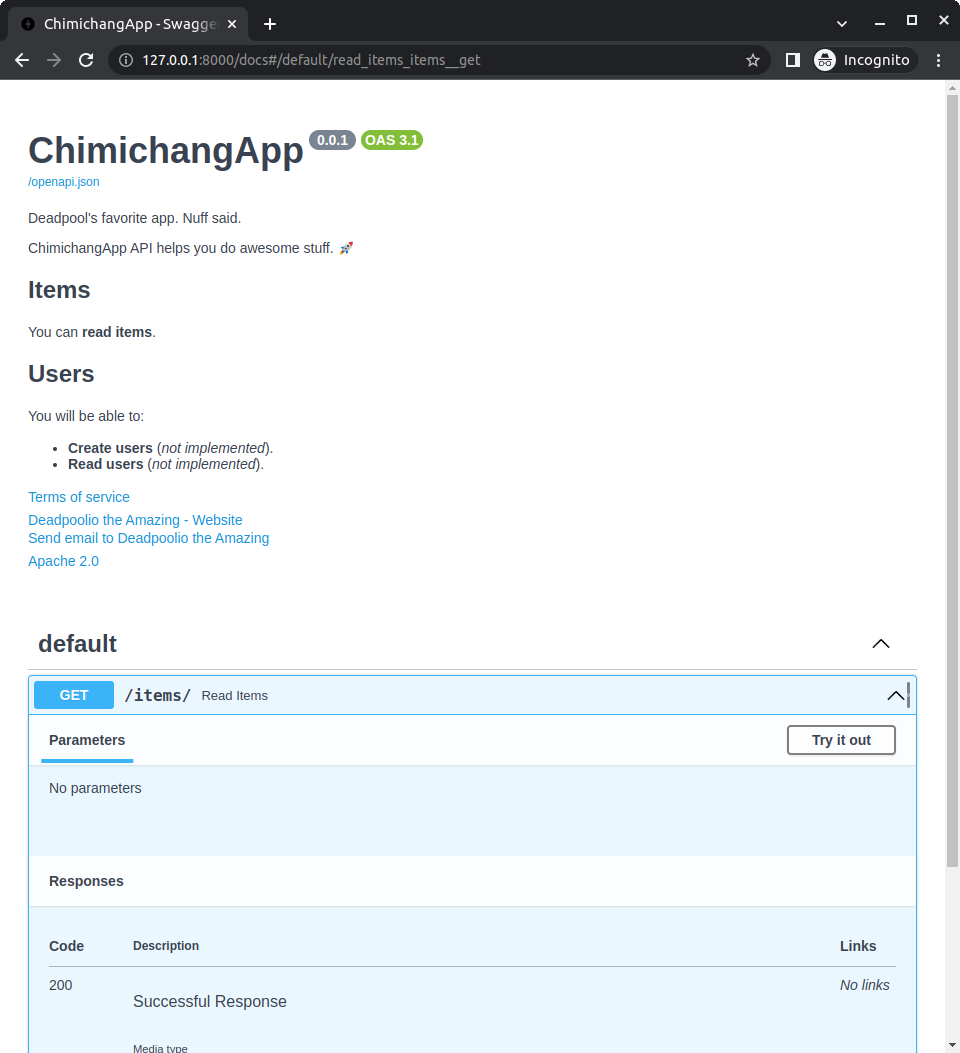
async def read\_item():

    return [dict(name = "ice cream",price = "10-100")]

# static files

# we can serve static files automatically from a directory using static files

from fastapi.staticfiles import staticFiles



## **License identifier**

you can also set the license\_info with an identifier instead of a url

## **Metadata for tags**

You can also add additional metadata for the different tags used to group your path operations with the parameter openapi\_tags.

It takes a list containing one dictionary for each tag.

Each dictionary can contain:

* name (**required**): a str with the same tag name you use in the tags parameter in your path operations and APIRouters.
* description: a str with a short description for the tag. It can have Markdown and will be shown in the docs UI.
* externalDocs: a dict describing external documentation with:
  + description: a str with a short description for the external docs.
  + url (**required**): a str with the URL for the external documentation.

### **Use your tags**

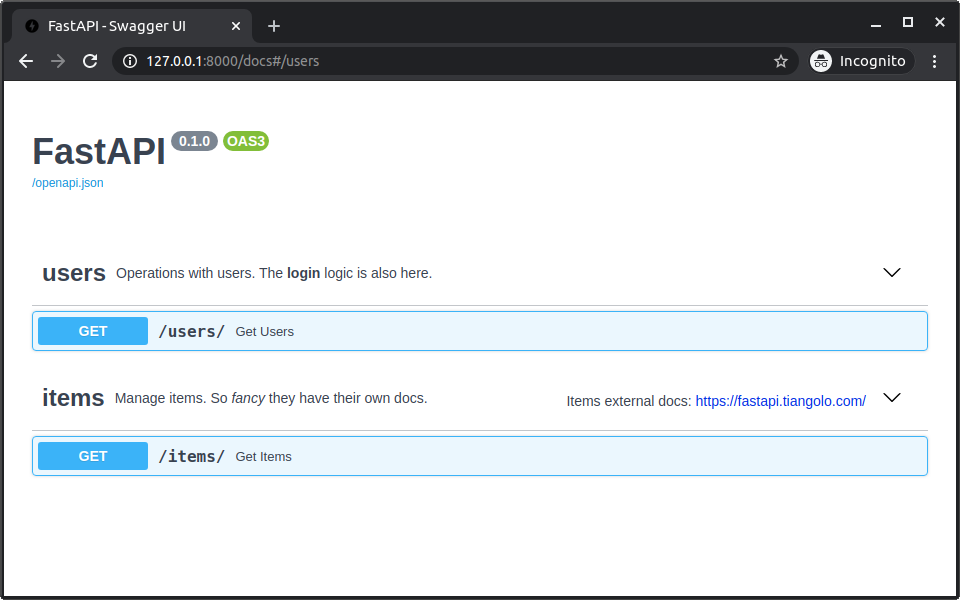
Use the tags parameter with your path operations (and APIRouters) to assign them to different

app = FastAPI(openapi\_tags=tags\_metadata)

### **Check the docs**

Now, if we check the docs, they will show all the additional metadata:

Like title, url ,description about app, links ,emails etc…



### **Order of tags**

The order of each tag metadata dictionary also defines the order shown in the docs UI.

For example, even though users would go after items in alphabetical order, it is shown before them, because we added their metadata as the first dictionary in the list.

## **OpenAPI URL:**

By default, the OpenAPI schema is served at /openapi.json.

But you can configure it with the parameter openapi\_url.

For example, to set it to be served at /api/v1/openapi.json:

from fastapi import FastAPI

app = FastAPI(openapi\_url="/api/v1/openapi.json")

If you want to disable the OpenAPI schema completely you can set openapi\_url=None, that will also disable the documentation user interfaces that use it.

### Using Doc Strings:

You can configure the two documentation user interfaces included:

* **Swagger UI**: served at /docs.
  + You can set its URL with the parameter docs\_url.
  + You can disable it by setting docs\_url=None.
* **ReDoc**: served at /redoc.
  + You can set its URL with the parameter redoc\_url.
  + You can disable it by setting redoc\_url=None.

For example, to set Swagger UI to be served at /documentation and disable ReDoc:

from fastapi import FastAPI

app = FastAPI(docs\_url="/documentation", redoc\_url=None)

ex:

“ “ “ doc string

Provides more information about code like comments

“ “ “

@app.get("/items/{item\_id}")

async def read\_item(item\_id: int, q: str = None):

    """

    Read an item.

    Args:

        - \*\*item\_id\*\*: The ID of the item to read.

        - \*\*q\*\* (optional): A query parameter for filtering items.

    Returns:

        - \*\*item\*\*: Details of the item.

    """

    # Implementation code here

    pass

we have to represent with triple quotes

To view the generated API documentation, run your FastAPI application and navigate to **http://localhost:8000/docs** (for Swagger UI).

to customize the metadata and doc strings according to your API's requirements to provide clear and comprehensive documentation for your users.

Static Files in Fast API:

In FastAPI, you can serve static files such as CSS, JavaScript, images, or any other files directly from your application using the **FastAPI.static()** function.

**1.Create a static folder:**

First, create a folder in your project directory to store your static files. By convention, this folder is often named **static or etc..**

**2.Configure FastAPI to Serve Static Files:**

use the **FastAPI.static()** function to specify the URL path under which the static files will be accessible and the directory where the static files are located.

Ex:

 static files

# we can serve static files automatically from a directory using static files

from fastapi.staticfiles import staticFiles

app.mount("/static",staticFiles(directory="static"),name="static")

#mounting means adding complete independent application in specific path

<link rel="stylesheet" href="/static/styles.css">

For static files we have to import staticFiles

And put the path in mount function

Mount: means adding a complete “independent “ application in a specific path

Or we can link our style sheet to path

<link rel = “stylesheet” href = “href= “/static/styles.css”>

The directory="static" refers to the name of the directory that contains your static files.

Testing :

Testing in FastAPI is essential to ensure your API endpoints behave as expected under different conditions. FastAPI provides built-in utilities for testing, including the **TestClient** class, which allows you to simulate HTTP requests and test your API endpoints.

Install: pip install httpx. For teating and to use Testclient

For testing we have to import TestClient Module