#3 LAMBDA function

* **Lambda function**
  - lambda function is a way of of writting functions without name
  - lambda functions are anonymous functions (without name) & it is defined using "lambda" keyword
  - Syntax :

```
            lambda argument_variables : expression
for calling <— variable = lambda arguments : expression
fun you need to use this variable          └> This argument is noth
& pass parameters to this variable              but the function parameters
it will behave like function name
```

  - use case : lambda functions are commonly used as arguments (parameter) to pass to high order function such as map, filter, & reduce
  - eg1:

```
def square_fun (x):
        return x*x

    or

ans = lambda x : x*x
print (ans(2)) ——>// in this ans is noth but name of
                    function
ans(4) // calling of lambda fun, this "ans(4)" will
            get evalute as 16
    or
m = ans (4) // you called the lambda fun & pass 4 to it,
               it returns 16
print (m) // it will print 16
```

  - eg2:

```
                   ┌——————————————————> fx is a function,
def fun_1 ( fx , val1, val2, val3 )          we are using lambda
        return 10 + fx(val1, val2, val3)     function to pass
                                             function as parameter

avg = lambda x, y, z : (x+y+z)/3
final_ans = fun_1 (avg , 5, 10, 15)
print(final_ans)   └——————> we are passing function as
        or                      parameter & it parameters
                                also
print (fun_1 (avg , 5, 10, 15))
```

NOTE:
- you can pass function as a parameters to other functions
- high order function : These are function which take function as parameter

(14) MAP, FILTER & REDUCE

~~(i) MAP~~

*** map, filter & Reduce**

- map, filter, & Reduce functions are built in function & are also present in JS
- These functions allows you to apply a function to a sequence of elements (list, sets etc) & return a new sequence from that OG sequence
- These functions are known as high order functions bcoz they take functions as parameters (These parameter functions are noth but lambda functions)

*** map**

- map function applies a function to each element of a sequence (like list, set, string etc) & returns new sequence
- This function returns a new sequence (like list, set, string etc) containing the transformed elements.
- Syntax:

    map (function , iterable_sequence)

fun argument, This is a function that is applied on each element of the iterable_sequence argument

This argument can be list, Tuple, string

- you can also send lambda fun to it

eg1:

```
def cube(i):
    return i*i*i

l = [1, 2, 4, 6, 4, 3]
newl = map(cube, l)
print(newl)
//output: <map object>
print(list(newl))
//output: 1, 8, 64, 216, 64, 27
```

- This is the function cube send as parameter
- The parameters of cube will be the elements of list & it will automatically send to cube

l is list, on each element of list "l" that function will get apply

* filter

- filter functions filters a sequence of elements based on the function passed to it.
- the function which is passed as parameter to filter function is called predicate
- Predicate : function passed as parameter should return a boolean value T/F, so that filter function can decide wether to add that value/element or not
- filter function returns a new sequence (list, set etc) which includes the filtered elements
- Syntax :

filter (function , iterable)

This passed function should return a boolean

This is noth but a sequence like list, & each element of this sequence is passed as parameter to the lambda function send as parameter to filter

eg:
```
l = [1, 2, 4, 6, 4, 3]
def filter_fun (a) :
        return  a>2 //returns T or F

newl = ~~list~~ filter( filter_fun , l)
print (newl) // out : filter object
print (list (newl)) //output :[4, 6, 4, 3]
```
output is all the elements from list which are greater then the function condition

or

```
l = [1, 2, 4, 6, 4, 3]
fun_1 = lambda a : a>2 // This lambda function should return T or F only
newl = filter (fun_1, l)
print (newl) // output : lambda fun
print (list (newl)) // output : [4, 6, 4, 3]
```

*reduce

- reduce function is a high order function that takes a function & sequence as parameter & then apply that function on the sequence elements & returns a single value
- "reduce" function is a part of "functools" modules, which is a built in module & you need to import it
- Syntax :

  reduce (lambda_function , iterable)

  $\leftarrow$ it will apply this fun on each element & return a single value

  $\hookrightarrow$ Can be list, tuple, string

eg: from functools import reduce

l = [1, 2, 3, 4, 5]

sum = lambda x, y : x+y

$\rightarrow$ x: it will be the previous return value for reduce function & y will be the next element

~~ans~~ = ~~sum~~

ans = reduce (sum , l)

print (ans) // output = 15

// working

$[\underbrace{1, 2}, 3, 4, 5] \Rightarrow$ sum (1, 2)

$[\underbrace{3, 3}, 4, 5] \Rightarrow$ sum (3, 3)

$[\underbrace{6, 4}, 5] \Rightarrow$ sum (6, 4)

$[10, 5] \Rightarrow$ sum (10, 5)

15 $\rightarrow$ return 15

#5) ENUMERATE FUNCTION

* Enumerate function

- Enumerate function is a built in function that allows you to loop over a sequence (like list, tuple, string) and get the value/element and index of that element both at same time
- मतलब अगर हमे एक list के elements के साथ साथ उस element का index भी देना है directly. Just हमे "enumerate" keyword use करना है.
  Generally C++ मे हम map use करके ऐ behaviour मिलता है
- Syntax :

this will help us to get both element & index of that element

```
for i, ele in enumerate(sequence)
```

this "i" is variable for index

ele is variable for elements of that sequence

sequence can be name of list, string or tuple etc

eg: marks = [10, 20, 32, 45, 65, 95]

```
for index, m in enumerate(marks):
    print(index, " = ", m)
    if (index == 5 or m == 95)
        print("Congratulation topper")
```

in this for loop, you can access the index of the elements of list using ~~index~~ "index" variable

// you can also specify diff that ~~yo~~ start indexing from a custom no like:

```
eg: for i, m in enumerator(marks, start=1):
        print(i, "=", m)
```

#6 LOCAL V/S GLOBAL VARIABLE

* Local variable: this is the variable which is defined inside a specific function
  - you can only access this variable inside the function & its scope is only till inside the function
  - if you have local & global variable with same name & you want to access the globle variable inside the the function where local variable is also present with same name, then make use of "Global" keyword
* Global variables: The variables which can be accessed from anywhere in the pgm
* Global keyword: if you use "global" keyword then you can access global variable inside function who has same name local variable

- eg:

```
x = 10 ——> // global variable
def fun_1():
    x = 20 ——> //Local variable
    print (x)
    global x = 100       ] // not allowed, you will
    print (global x)     ]    get Error
fun_1()
print (x)
```

- eg 2:

```
x = 10 ——> // global variable
def fun_1()
    global x ——> // this tells that "x" is the
                    global x
    x = 100
    print (x)

fun_1()
print (x) //output : 100
```

~~* [illegible]~~

#7) *args & **kwargs

***args** : it is a tuple, & its used inside a function to take in dynamic no. of arguments or parameters

- *args lets your function or constructor to accept any no. of positional arguments without explicitly defining each one
- it is useful for function overloading or constructor overloading

eg:
```
def funargs (x, *args):
        print ("msg is :", x)
        for i in args:
            print (i)

har_variable = ["Hari.", "vishu", "Gaikwad"]
x = "Hello this is args demo"
funargs (x, *har_variable)
```

this "*" means its a tuple & this should always come at the end or as last parameter.

store it as tuple or list & pass it while calling

while calling also you need to pass it using "*"

- syntax :
```
def fun_name (parameter1, para2, *args):
        // your code

fun_name (parameters, *tuple)
```

- you can pass tuple or list while calling the function
- & using elif you can design constructor overloading

****kwargs** : this is noth but dictionary, it works same as *args but it takes key value pair

- **kwargs lets your function accept any no. of key-value pair arguments
- when you don't know Exact no. of arguments, then you can use this for dynamic arguments accepting inside a function or Constructor

eg:
```
def demo (a, b, **kwargs):
        print (a, b)
        for key, val in kwargs.items():
                print ( key, ":", val)
```
→ you need to use "**" it tells that, it is a dictionary variable

```
    a = 69
    b = "Hello 69"
    dic = (name = "visha", age = 23 , country = "India")
    demo ( a, b, **dic)
```
→ while calling also, you need to pass dictionary using "**"