Functions: Function is a name given to a piece of code.
i.e Function is a block of code which only runs when it is called.

```
Print('hi')
x=1
y=1
z=x+y
```
Name → function

These statements will only be saved when the function is defined. statements will only be executed when we call the name of the function.

Function
three ways
- Inbuilt function
- user defined functions
} we will learn these now
- (we will learn later)

Inbuilt function:

These are functions which are pre-built in Python and will have a special meaning inside python.

ex:- Type(), len(), print(), input()....

python developers initially created these to meet basic programming requirements.

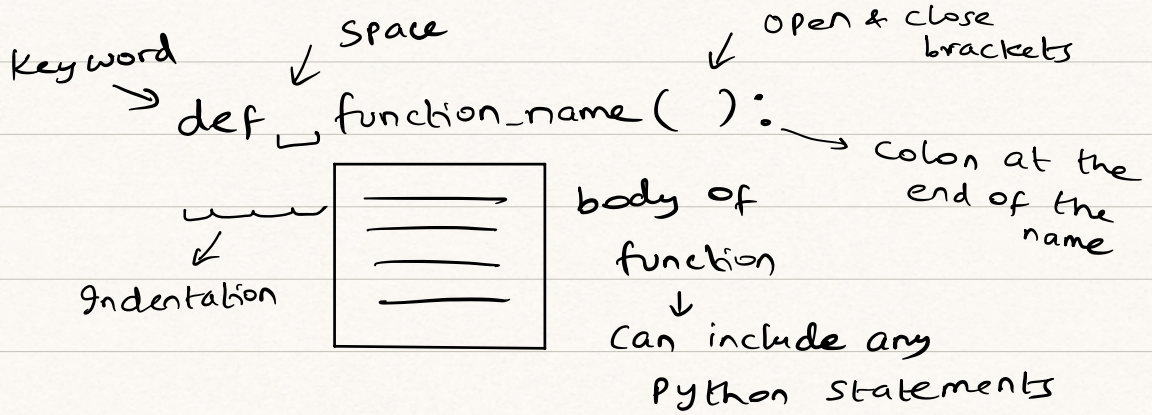
User defined functions:

These are the functions defined by the user. So the programmer builds his own functions.

we will learn how to create our own function.

To create a function

Syntax



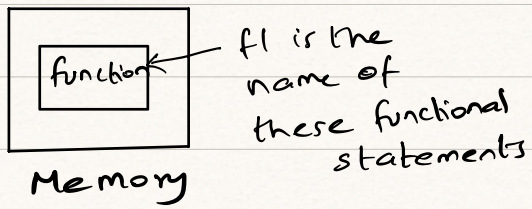
1) 'def' (definition) keyword is used to define the function.
2) Space is left between the def and the function name.
3) function name should follow the naming convention and should be programmer friendly (name should give clarity on what the function is doing).
   - can start with alphabets or underscore
   - can contain numbers, alphabets
   - special characters not allowed
   - case-sensitive.
4) After function name we use an opening and closing brackets followed by colon to specify the end of name.
5) Indentation rules need to be followed.

6) Body of the function can contain any python statements like Arithmetic, Assignment, Logical, conditional, looping etc.

Simple function

First step - define a function

```
def f1():
    Print('hi')
```
} defining the function



During function defining phase python statements will not be executed. They will only be stored.

* so any coding errors in the function will not be known until we execute it.

second step - execute the function (calling a function)

```
f1( )
```
← we call using function name followed by opening and closing brackets.

↓

python will check if the function exists in the memory. If it does the statements inside will be executed.

* Function has to be defined first and called only then.
* Once a function is defined we can call it n no. of times.

  So we can create a function once and call it like 10, 100, 1000 times.

Advantages of functions:

1) Reusability of the code.
2) Chunks: When we are writing huge python code, with the help of functions we can divide the code in to small parts called chunks. Each chunk is given a function name so that when an error occurs it will be easy to debug it.

So far we learned simple functions. If we want to write advanced functions we use two important concepts called

- parameters
- arguments

Parameters: we define these when we are defining the functions (these will be mentioned inside the brackets).

```
def sum1(x,y):
    print(x,y)
```

Parameters

As parameters are given we do not need to write the input statements in the body of function.

* we can create n no.of parameters.
* when we call this function we need to mention the parameter values.
  ↳ these values are called arguments

```
sum1(10,20)
```

arguments

↳ output 30

* As a programmer we create the parameters that are being used in the function. Additional parameters waste memory space.
* parameters
  - Simple parameters
  - default parameters

Simple parameters - If while defining the function we define n parameters then at the time of calling the function also we need to mention 10 arguments.

default Parameters -

If we are using simple parameters by while Calling we do not mention n arguments we will get an error. To avoid this error we can use default parameters.

default parameter is given at the time of definition.

default parameter

```
def sum1(x1, x2, x3, x4, x5=10):
```

so during the function calling if we do not pass the argument value then the default value is taken.

* when we pass arguments to default parameters then the default value will be replaced by arguments passed.
* default Parameter should always follow non-default Parameters

Arguments are of two types.

- Positional arguments
- Keyword arguments.

Positional arguments : These arguments will be assigned to the parameters based on the position.

ex:- bio(name, age, height)

so if bio("xyz", 23, 170) ✓ this is correct

but if we give

bio("xyz", 170, 23) ✗ here though we do not get error logic is wrong.

So positional arguments might create issues when we do not know the position of the argument to be passed. Instead better to use keyword arguments.

Keyword arguments: Here we use keywords of the original value when calling the function.

ex:- bio(name='xyz', age=23, height=170)

Keyword arguments are the best.

* when we use default parameters. If we call a function with no arguments also it will execute with out any errors.

ex:- def f1(x=10, y=10):
⋮

f1()
↳ no error.

* If we use default parameters and we call a function with more arguments than parameters we will get an error.

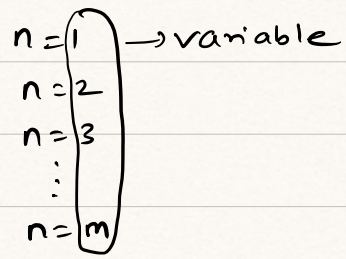
ex:- f1(10, 20, 30) X expecting only 2 arguments.

* Till now we were using limited parameters in examples.

* If we have a function with huge no.of parameters and we have to pass all these arguments we will have an issue so we use:

Variable length argument : This is an argument of variable length

two types
- args
- kwargs



args : Non-keyworded variable length arguments.
(If arguments do not contain any keys)

Symbol for arg *

Kwargs: Keyword variable length arguments.
(If arguments contain keys and values).

Symbol for kwarg **

when we use these symbols within the function paranthesis they will have this special meaning.

i.e f1(*x) or f1(**x)

Till now each parameter was holding one value but if we want the parameter to hold 'n' no.of data we use:

list, tuple, sets → args

(default is tuple)

Dictionary → kwargs

so,

for args

```
def f1(*x):          # * → represents arg; x → parameter name
    for y in x:      # here since it is a tuple we dont mention (*)
        print(y)
```

f1(1, 2, 3, 4, 5) → Internally x = (1, 2, 3, 4, 5)

So here x is tuple which has 5 elements.

↳ 1
2
3
4
5

for kwargs (Dichionary)

```
def f1(**x):         # ** → represents kwarg
    for y in x:      # here it is a Dichionary.
        print(y)
```

when we call the function

f1(Name = "P1", Age = 25, sex = 'Male')

here

* key we use is mandated to be string.
* when we are calling the function we use '=' Assign operator between key and value instead of ':' as we do in Dictionary.

```
f1('Name'='P1', 'Age'= 25, 'Sex'='Male')
```

Internally

```
x = {'Name':'P1', 'Age': 25, 'sex'='Male'}
```

dictionary

```
x.keys() → ['Name', 'Age', 'Sex']
x.values() → ['P1', 25, 'Male']
x.items() → [('Name','P1'), ('Age',25),
             ('Sex','Male']
```

Q) Create a function which will output all the even numbers.

input → n numbers

output → even numbers from this n numbers

```
def even(*n):
    for y in n:
        if y%2==0
            print(y)
```

even (1, 2, 3, 4, 5, 6, 7, 8)

↳ 2
4
6
8

but if we want to reuse this printed output we will not have anything.

return keyword

user-defined
built-in

Even though whenever we call a function it will definetly return something but the default is 'None'.

So we use keyword 'return' inside the function.

default for return is None.

If we want to use the output we need to explicitly mention in the body of the function.

* During execution when the program sees 'return' it will return that value and comes out of the function.
* return can return one thing only** But this can be a collection data type.

ex:-

```
def even(*n):
    l=[]
    for y in n:
```

```
if y%2==0:
    l.append(y)
```

return l ← this will return the value.

this needs to be outside the for or if because otherwise it will exit the function before code completion.

ex:-
```
def even(*n):
    l=[]
    for y in n:
        if y%2==0:
            return y
            l.append(y)
```

X

program stops here and will not execute the next commands

return syntax

return _ any_value

ex:- return 'hi'

default is return None

If we want to use the value that was returned to perform further operations on it then when we are calling the function we need to assign it to a variable.

ex:-

x = even (1, 2, 3, 4, 5, 6, 7, 8)

↳ [2, 4, 6, 8] (return list)

print(x)

[2, 4, 6, 8]

So we can now use this x at a different place.

* This is calling the function and storing the return value in a variable.

Assigning a function to a variable

*This is different do not get confused.

we can assign a function to a variable in python.

ex:- W1 = even → no paranthesis is needed.

↳ variable → function

here even ✓ we are not calling just assigning.

even ( ) ✗

Assigning the function to a variable	Calling function and storing return value in variable
Var = function	Var = function ( )

* function only executes the statement.

↓ if we want to use the output

return needs to be used in the function.

↓

var = function (parameters)

↓

this variable can be used in other places.

or var1 = function

var2 = var1 (....)

↳ temporary variable assigned a function.

↳ has the return value

ex:- x = even

y = x(1, 2, 3, 4, 5, 6, 7, 8)

print(y)

↳ [2, 4, 6, 8]

* Till now we mentioned return value can return a single data. This is right.

So if return 'hi', 1, 2, 3, 4

it will still return and consider this as a tuple i.e ('hi', 1, 2, 3, 4).

Practice Programs

Q) Create a function where we will pass a string of length 10 and output will return odd indexed characters.

i.e input = "abcdefghijk"

output = (b, d, f, h, j)

↳ only one data so no need of arg or kwarg

```
def odd_in(n):
    l = []
    for y in range(len(n)):
        if y%2!=0:
            l.append(n[y])
    return l

x = odd_in("abcdefghijk")
```

↳ x value stores the return value.

```
print(tuple(x))
```

↳ (b, d, f, h, j)

Q) Create a function

Input: Dynamic string

Output: string with odd indexed characters upper case and even indexed lower case.

```
def od_up_e_lo(n):
    l=[]
    for y in range(len(n)):
        if y%2!=0:
            l.append(n[y].upper())
        else:
            l.append(n[y].lower())
    S=" ".join(l)
    return S
```

if n="abcdefgh"
S → "aBcDeFgH"

Q) Create a multiplication table with 2 input parameters. 1st parameter which table
2nd parameter no. of iterations.

```
def mult_tab(x,n):
    for y in range(1,(n+1)):
        print("{}*{}={}".format(x,y,x*y))
```

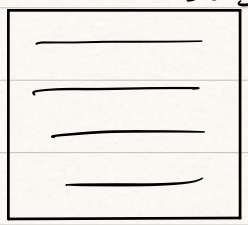
so c = mult_tab(5,4)

c
↳ 5x1 = 5
5x2 = 10
5x3 = 15
5x4 = 20

doc-string (document-string)

Inside a function if we want to document what the function is doing then we can write a string describing what it is doing.

```
def bio(name, age, height):
    """ message will come here """
    other python
    statements
```



* The message can be single line or multiline within three double quotes (""" """)
* document string has to come inside the function before all the other python statements.
* If we want to view the doc string we can use a magic function (function_name.__doc__)

2 underscores.

* Doc string comment should begin with a capital letter and end with a period.

Generators

Function can only return one value. we cannot return multiple values in a function.

return 1
return 2 X
return 3 X

Not possible as python exits out of function when it sees return. so we can only use 1 return.

So instead we use Generate function and yield.

Generate function : A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with yield keyword, rather than return.
If the body of the function contains yield it automatically becomes Generator function.

* yield is used when we want to return multiple values from a function.
* yield is similar to return but we use it multiple number of times and python does not come out of the function upon seeing a yield.

Yield & Generator function syntax

```
def f1( ):
    yield ('hi')
    yield (123)
    yield ('bye')
```

} generator

* When python sees yield it will not stop until it sees the last yield.

Generator object: Generator function returns a generator object. These are iterable objects. Generator objects are used either by calling the next( ) function on the generator object or by using generator object in a 'for in' loop.

f1( ) → returns a generator object. → 'hi', 123, 'bye'

* Generator object is a iterable object or sequential object.

* We use next( ) function to see all these yields on the generator object.

first time → 'hi'

second time → 123

third time → 'bye'

ex:-

```
def f2():
    yield 'hi'
    yield 123
    yield 'bye'
```

q = f2() ← calling a function and assigning output to variable

generator object

```
next(q)
  ↳ 'hi'
next(q)
  ↳ 123
next(q)
  ↳ 'bye'
next(q)
  ↳ X  Iteration is done so we get a error.
```

* Instead we can use for loop as it understands how many elements are in the generator object.

```
for y in f2():
    print(y)
      ↳ 'hi'
        123
        'bye'
```

Enumerators

Often when dealing with iterable objects, we also need to keep a count of iterations. Python eases the programmers task by providing a built-in function enumerate() for this task.

enumerate() function adds a counter to the iterable and returns it in a form of enumerate object. The enumerated object can then be used directly for loops or converted into a list of tuples using the list() constructor.

Syntax

enumerate(Iterable object, starting index value)

↑ all iterable objects except dictionary

↓ by default is zero but we can give our own index

ex:-

```
x = "abcdef"
for y in enumerate(x,100):
    print(y)
```

↳ (100, a)
(101, b)
(102, c)
(103, d)
(104, e)
(105, f)

* we can use enumerate() function to change the index value temporarily and it will only function with in a loop.

* The index value change is not permanent.

ex:-

x[100] → X because it was only a temporary change. x[100] does not exist.

ex:-

```
x = "abcdefgh"
index = []
value = []
for y in enumerate(x,100):
    index.append(y[0])
    value.append(y[1])
```

↳ y = (100,'a')
(101,'b')
⋮
(107,'h')

index
↳ [100, 101, ... 107]

value
↳ ['a', 'b', 'c', ..., 'h']

Anonymous function (lambda function)

Python lambda functions: are anonymous functions i.e the function is without a name.
As we already know that the 'def' keyword is used to define a normal function in python.
Similarly, the 'lambda' keyword is used to define a anonymous function in python.

Syntax

n no. of arguments

```
lambda _ arguments : expression
```

only one expression

* whenever we want to execute a single expression we use lambda function.

* This function can have any number of arguments but only one expression, which is evaluated and returned.

* Inside the lambda function there is no return used. The value will be automatically returned by the lambda function.

* we have to assign the lambda function to a variable to call it.

```
y = lambda arguments: expression
↓
name
```

```
y()
↳ output
```

only one expression

ex:- `z = lambda x,y : x+y`

```
z(10,20)
↳ 30
```

* only give the arguments that are used in the expression.
* lambda is used to write the code in short form.

ex:- `q = lambda x,y,z : x+y*z`

instead of

```
def f1(x,y,z):
    print(x+y*z)
```

`q()` x we need to give 3 parameters

```
q(10,20,30)
↳ 610
```

or we can store the output of the lambda function in a variable

```
P = q(10,20,30)
```

P
↳ 610

ex:- q = lambda z, x=10, y=20 : x+y*z

q(1) (one argument is enough as x, y are default parameters).
↳ 30

* default parameters always follow non-default parameters.

(or)

* Keyword arguments always follow positional arguments.

Map function : (one of the beautiful functions in python)

The map() function executes a specified function for each item in a iterable. The item is sent to the function as a parameter.

map (function , iterable object)

- function: user-defined / built-in
- iterable object: sequential object

ex:-

```
def f1(x):
    return x**2
```

} returns the square of the value

```
f1(2)
  ↳ 4
```

This is calculating the square for one value, what if we want to calculate square for 10 values we need to call this function 10 times.

Instead we use map() function and give the values as a iterable object.

```
map (f1 , [1, 2, ....10])
```

↳ here paranthesis are not used as we are not calling the function but map function is calling it.

* map function first checks if the given function is in memory.

* map function then checks if iterable object is provided or not.

The output of a map function is stored in a map object.

f1(1) → 1

f1(2) → 4

⋮

f1(10) → 100

map object = map(f1, [1, 2, ... 10])

If we want to see the map object we can convert it in to list, tuple, set (not recommended) using the constructor.

list(map object) = [1, 4, 9, ... 100]

* we can use range() function inside the map function to create iterable object.

* Inbuilt functions or user defined functions can be used inside map function.

** we cannot use methods inside map.

ex:- y = map(lambda x: x**2, range(1,11))

list(y)

↳ [1, 4, 9, ... 100].

* when we are using lambda function inside the map function we give no. of iterable objects same as no. of arguments passed.

q = map(lambda x,y: x*y, [1,2,3], [4,5,6])

they need to be iterable object for both x & y

list(q)

↳ [4, 10, 18]

Q) with the help of map & lambda function convert x = "abcdef" in to upper case.

z = map(lambda x: x.upper(), "abcdef")

"".join(list(z))

(or)

```
" ".join(list(map(lambda x: x.upper(), "abcdef")))
```

↳ "ABCDEF"

Q) (1, 'a', 1.1, 2+3j) get type of each value using map & lambda

```
list(map(lambda x: type(x), (1, 'a', 1.1, 2+3j)))
```

↳ [int, str, float, complex]

Filter function

The filter() function filters the given iterable object or sequence with the help of a function that tests each element in the sequence to be true or not.

Syntax

filter(function, iterable object)

here function: function that tests if each element of the sequence is true or not.

iterable object: iterable which needs to be filtered, it can be sets, tuples, lists etc.

Returns a filter object which is an iterable that is already filtered. we can use list, tuple, set constructors to view it.

* filter function is similar to map function but the difference is filter function does the filteration.

ex:-

If we need odd values from a list of numbers. Here based on the condition we will get True or False.

x = [1, 2, 3, 4, 5, 6, 7]
T F T F T F T

filter is used to filter out based on True or False.

Q) Using map & lambda values filter out odd values.

```
list(map(lambda x: x%2!=0, [1,2,3,4,5,6,7]))
```

↳ [True, False, True, False, True, False, True]

Now if we don't want boolean values instead we want to return for when value is True we will use filter function.

```
list(filter(lambda x: x%2!=0, [1,2,3,4,5,6,7]))
```

↳ [1,3,5,7]

* filter is only used when we want to filter out based on a condition.

Q) Given a string filter out vowels (a,e,i,o,u)

ex:- "abcedt"

```
list(map(lambda x: x in 'aeiou', 'abcedt'))
```

↳ gives boolean values.

```
list(filter(lambda x: x in 'aeiou', 'abcedt'))
```

↳ ['a', 'e']

Q) Given a string of numbers filter out numbers divisible by 7.

need to convert to int

```
list(filter(lambda x: int(x)%7==0, "1234567890"))
```

↳ ['7', '0']

Q) Given a list of strings by to filter out the strings whose length is greater than 4.

```
list (filter (lambda x: len(x)>4, ["abcd", "acd", "abcde"]))
    ↳ ["abcd", "abcde"]
```

** filter() function evaluates the expression and prints values where expression is true.

ex:- list(filter(lambda x:x**2, [1, 2, 3, 4])

↳ [1, 2, 3, 4]

filter will not print the value of expression x**2 like map, instead it will print x for all bool(x) = True so [1, 2, 3, 4].

Zip function

The zip function returns a zip object which is an iterable object of tuples where the first item in each passed iterable object is paired together, and then the second item in each passed iterable is paired together and so on.

* If the passed iterable objects have different lengths. The iterable object with the least items decides the length of the zip object.

zip(iterable object 1, iterable object 2, ...)

ex:-

```
x = [1, 2, 3, 4]
y = ['a', 'b', 'c', 'd']
z = "abcde"
```

↑ e exceeds the length so it is ignored.

```
list(zip(x, y, z))
↳ [(1,'a','a'), (2,'b','b'), (3,'c','c'), (4,'d','d')]
```

* zip function will check if the passed objects are iterable or not.
* once check is done zip object is created.

Unzipping

If we want the original iterable objects from the zip object we need to unzip.

Syntax

```
zip (*zip (iterable object 1, iterable object 2,...))
```

we need to Assign this function to no.of variables equal to number of iterable objects we are unzipping.

Each variable will be assigned an iterable object.

ex:-

```
x = [1, 2, 3]
y = [5, 6, 7, 8]
```

↳ excluded

```
list(zip(x,y))
```

↳ [(1,5), (2,6), (3,7)]

```
z,w = zip(*zip(x,y))
```

z

↳ [1, 2, 3]

w

↳ [5, 6, 7] ← 8 will not be there here

* we use this in pandas for column combinations.

Unzip operator :

Given any variable which is iterable we can then unzip and append directly.

ex:- `l = [1, 2, 3, 4, 5]`

`z = [4, 5, 6, 7]`

if we want new-list p to have all the elements of l+z.

Original

```
p = [ ]
for y in l+z:
    p.append(y)
```

→ `[1, 2, 3, 4, 5, 4, 5, 6, 7]`

(or)

```
p[*l, *z]
```

→ `[1, 2, 3, 4, 5, 4, 5, 6, 7]`

ex:- `l = [1, 2, 3, 4]`

`z = "abcd"`

`p = [*l, *z]`

→ `[1, 2, 3, 4, 'a', 'b', 'c', 'd']`

* we can use this for set also but keep in mind that set removes all the duplicates.

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

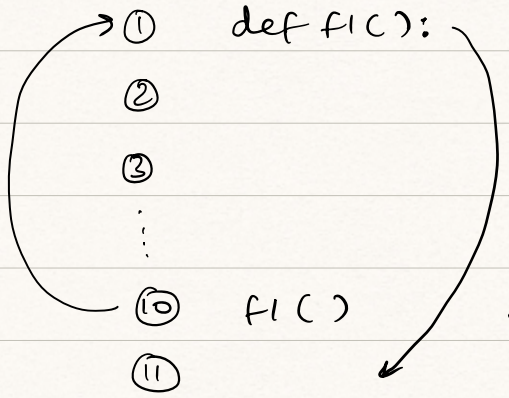
`z = [4, 5, 6, 7]`

$P = \{*1, *2\}$

$\hookrightarrow \{1, 2, 3, 4, 5, 6, 7\}$

Concept of storing return address in stack memory:

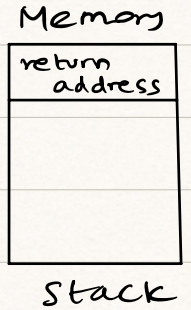
If we have a function f1() and we define it in code line 1. Then we call this function in Python code line 10.



← defined at 1st line

← Called at 10th line so it goes back to 1st line and after execution it has to come back to line 11.

Python engine knows that it has to come back to line 11 because python stores the return address in memory called stack when function is called.



So after the execution of function it goes to stack to check if return address is present. If return address exists then python goes to that location.

of the calling of the function is done and we stop using the function. Once the stack is full the return addresses will be automatically cleared.

Recursion (Tricky, extreme caution)

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that we can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip in to writing a function which never terminates, or one that uses excess amounts of memory or processor power.

However when written correctly recursion can be very efficient and mathematically elegant approach to programming.

Issue with Recursion: If a function is calling itself then return addresses keep getting saved in stack. so if there is no logic to exit out of the function then after a time the stack is full and function is still being used so stack is not cleared and the Python will stop working. As this is a ∞ loop that is happening.

ex:-

```
def f1():
    f1()
```

→ system will crash

* It is developers responsibility to break the code and come out of the function when using Recursion..

** When writing Recursion things we need:

logic 1: Terminate the recursion (i.e Base)

logic 2: How to use this recursion (correct flow)

* Recursion takes high memory, more space and time is high if not written effectively so it needs to be used with extreme caution.

Q) Find the factorial of a number using function Recursion.

$1! = 1$

$2! = 2 \times 1$

$3! = 3 \times 2 \times 1$

$4! = 4 \times 3 \times 2 \times 1$

$5! = 5 \times 4 \times 3 \times 2 \times 1$

logic: For factorial the last value multiplied is always '1'

So,

```
def f1(x):
    if x==1:        → since 1 is the last digit multiplied.
        return 1
    else:
        return x* f1 (x-1)
                      ↳ calling itself
```

f1 (4)
↓
4 * f1(3)
↓
3 * f1(2)
↓
2 * f1(1)
↓
1

4 * 3 * 2 * 1
↳ 24

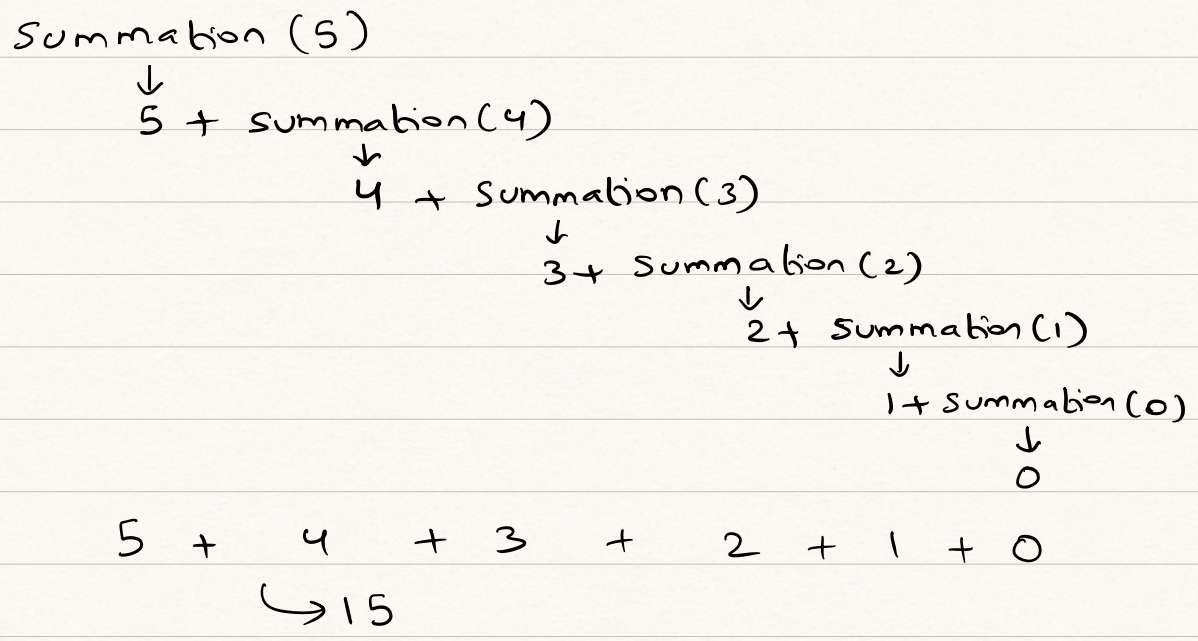
* for every problem based on the problem statement we need to come up with a logic to terminate the recursion.

Q) Given a number 'n'. Find sum of 1 to n using recursion.

summation = n+(n-1)+(n-2)+ ....+1+0

logic: we can use 1 or 0 to break the recursion.

```
def summation(n):
    if n==0:
        return 0
    else:
        return n+summation(n-1)
```



Advantages of Recursion

1) Recursive functions make the code look clean and Elegant.
2) A complex problem can be broken down into to simpler subproblems using recursion.
3) Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

1) Sometimes the logic behind recursion is hard to follow through.
2) Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3) Recursive functions are hard to debug.

Decorators: (Tricky Advanced)

One of the essential things that programmers should keep in mind is `do not repeat yourself'. This means that programmers should not repeat the same code infact, they must re-use the code. Programmers must look for an elegant solution when they face any problem of creating highly repitibive code. In python, this problem can be solved using the concept of meta-programming.

Meta-programming is the concept of building functions and classes where primary target is to manipulate code by modifying, wrapping or generating existing code.

One of the significant feature of meta-programming is Decoders.

Putting wrapper for a function:

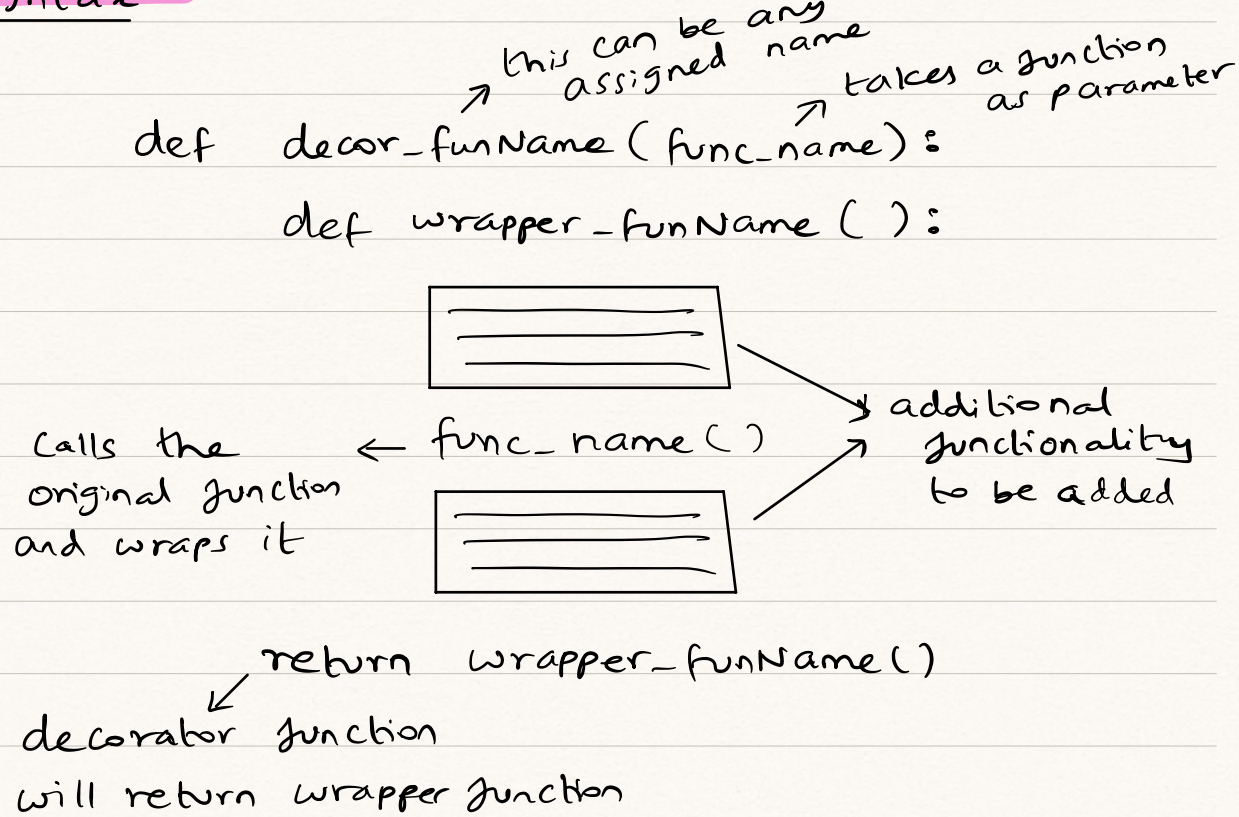
Programmers can add wrapper as a layer around an already existing function to add extra processing capabilities such as timing, logging etc.

The decorator in python is a particular form of function that takes functions as input and returns a new function as output.

Simple definition : Mainly used to decorate the function which was already defined without changing the original function.

These decorators add additional functionalities by using wrapper functions.

Syntax



So when decorator function is called it internally calls the wrapper function.

Wrapper function calls the original function and adds additional functionality to it.

ex:-

```
def f1():
    Print ('welcome')
```

This is the original function but now for this instance we want to print: 'Hi'
'welcome'
'Bye'

we can use decorator.

this is decorator

```
def decor(func):
    def wrap():
        print ('Hi')
        func()
        Print ('Bye')
    return wrap()
```

This process is wrapping

```
decor (f1)
    ↳ 'Hi'
      'welcome'
      'Bye'
```

Q) we have a function and we want to check how much time the function is taking.

```
def f2():
    c = []
    for y in range(1,10000000):
        c.append(y)
    return c
def deco1(func):
    import time
    def wrap():
        st = time.time()
        func()
        et = time.time()
        return print(et-st)
    return wrap()

deco1(f2)
```

↳ 2.961811

* decorator function which has function with parameters as parameter is advanced concept.