Function

- Functions in Python are essential for code organization, reuse, and modularity, as they allow you to break down complex tasks into smaller, manageable parts and encapsulate functionality for easier maintenance and readability.
- In Python, a function is a reusable block of code that performs a specific task or a set of tasks. Functions are defined using the def keyword, followed by the function name, a pair of parentheses for optional parameters, and a colon to indicate the beginning of the function body.

We will cover the following topics in this chapter:

Built-in functions

User-defined functions

Defining a function

Function calling

Docstring

Return statement

Pass by reference vs value

Function arguments

Required arguments

Default arguments

Keyword arguments

Arbitrary arguments

Scope of variable

Recursion

Built-in functions

Just like Python has built-in operators (such as plus and minus for addition, asterisk for multiplication, and so on), it has built-in functions. You can use them in your programs as pieces of code. A built-in function can be used by specifying its name and, typically, by providing some information. With that information, the function performs some work and returns the results. For example, and so on.

In [2]:
```python
#built in function
print("hello")
print(4+5)
list=[1,2,3]
list.append([5])
print(list)
a=6
print(type(a))
n=int(input("enter number"))
```

```
hello
9
[1, 2, 3, [5]]
<class 'int'>
enter number5
```

User-defined functions

#These are functions that we define ourselves to accomplish specific tasks. #User-defined functions have the following advantages: Using user-defined functions, a large program can be broken up into smaller parts that are easier to understand, maintain, and debug. #When repeated code appears in a program, you can include it and execute it when needed by calling a function.

Defining a function

> The def keyword marks the beginning of the function header, followed by the function name and parentheses

> The function name identifies it. In Python, function names also follow the same rules that are followed by identifiers. Parameters and arguments should be enclosed in parentheses; parameters can also be defined inside these parentheses.

> It describes what the function does with an optional documentation string (docstring).

> A colon marks the end of the function header. At least one valid Python statement must appear in the body of the function; statements must be indented equally (usually 4 spaces).

> The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return

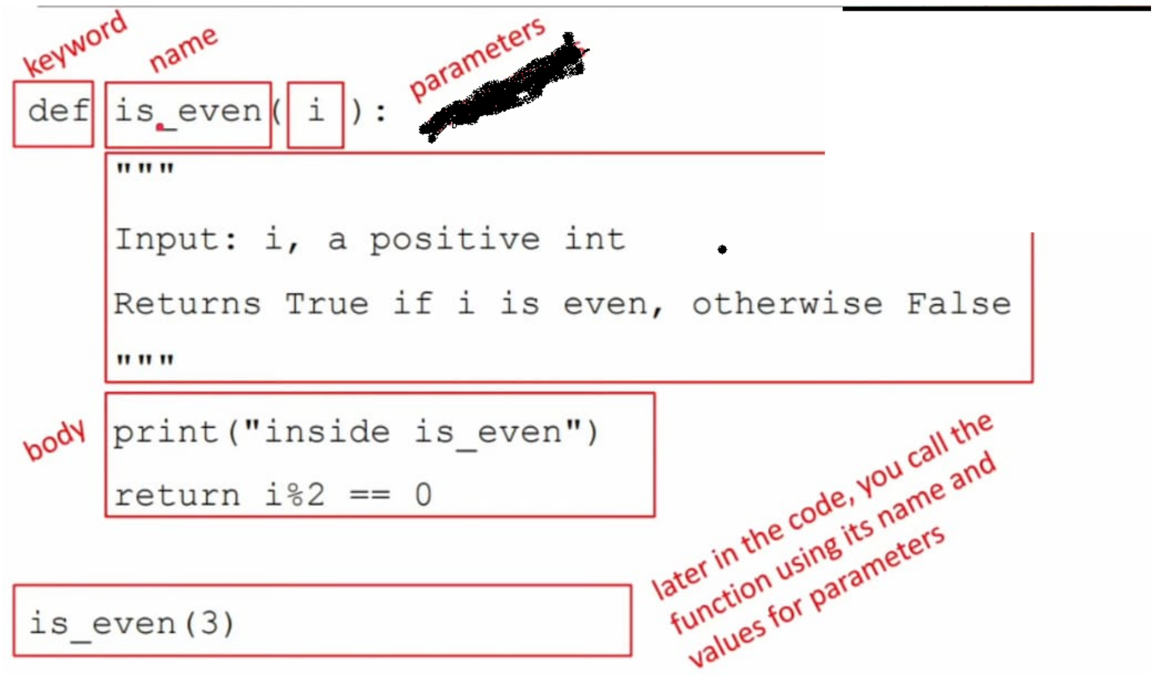
Abstraction and decomposition

Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user.

Functional decomposition is a method of analysis that dissects a complex process in order to examine its individual elements. A function, in this context, is a task in a larger process whereby

```
# Syntax:
deffunction_name(parameters):
    """docstring"""
    statement(s)
```

component of function


keyword
name
parameters
[redacted]
body
later in the code, you call the function using its name and values for parameters

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0

is_even(3)
```

- def: This keyword is used to declare a function.
- function_name: This is the name you give to the function. Function names should follow Python's naming conventions and be descriptive of the task the function performs.
- parameters: These are optional inputs that a function can accept. You can specify zero or more parameters within the parentheses. Parameters are used to pass values into the function so that it can work with the data provided.
- Function body: This is an indented block of code following the colon. It contains the instructions that define what the function does. You can use the parameters and perform various operations inside the function.
- return statement (optional): Functions can optionally return a value using the return statement. The return value can be used when calling the function to retrieve the result of its computation.

create a function(with docstring)

In [3]:
```
def is_even(num):#num is parameter
  """
  This function returns if a given number is odd or even
  input - any valid integer
  output - odd/even
  created on - 06th Oct 2023
  """
  if type(num) == int:
    if num % 2 == 0:
      return 'even'
    else:
      return 'odd'
  else:
    return 'pagal hai kya?'
```

Function calling Defining a function gives it a name, which specifies the parameter that will be included within the function and the structure of the blocks of code in it.

When we define a function, we can call it from another function, program, or even the Python prompt. The name of the function and its parameters are typed to call it. Here’s an example to call the is_even() function:

In [4]:
```
is_even(8)# 8 is argument function calling
```

Out[4]:
```
'even'
```

In [5]:
```
for i in range(1,11):
    x= is_even(i)
    print(x)
```

```
odd
even
odd
even
odd
even
odd
even
odd
even
```

return [expression_list]

The statement can contain expressions that are evaluated, after which the value is returned. The function returns None if there is no expression in the statement or if the return statement itself is absent inside the function.

In [19]:
```
def display(num):
    return
display(7)
```

return none

In [20]:
```python
def display(num):
    return num
display(7)
```

Out[20]:
```
7
```

In [5]:
```python
is_even(5)# function calling
```

Out[5]:
```
'odd'
```

In [9]:
```python
is_even("a")
```

Out[9]:
```
'pagal hai kya?'
```

Docstring

The docstring is the first string after the function header, and it stands for documentation string. It is used for briefly describing a function; documentation is a good programming practice, even if it is optional.

There is a docstring immediately below the function header in the preceding example. Usually, triple quotes are used as docstrings often span multiple lines. The string is available in the **doc** attribute of the function.

In [11]:
```python
print(is_even.__doc__)# print docstring
```

```

 This function returns if a given number is odd or even
 input - any valid integer
 output - odd/even
 created on - 29th Nov 2022
```

In [12]:
```python
# function
# function_name(input)
for i in range(1,11):
  x = is_even(i)
  print(x)
```

```
odd
even
odd
even
odd
even
odd
even
odd
even
```

2 point of view

1. programmer view: logic is correct and optimize program
2. customer view: if any error done by user,programmer make sure display the correct error in common words not a display any type of error (syntax error,value error,name error)

In [13]:
```
def is_even(num):#num is parameter
    if num % 2 == 0:
        return 'even'
    else:
        return 'odd'

```

In [14]:
```
is_even("vishal")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4288\3276565115.py in <module>
----> 1 is_even("vishal")

~\AppData\Local\Temp\ipykernel_4288\1238403626.py in is_even(num)
      1 def is_even(num):#num is parameter
----> 2     if num % 2 == 0:
      3         return 'even'
      4     else:
      5         return 'odd'

TypeError: not all arguments converted during string formatting
```

Different categories of user defined function

In [26]:
```
#1 function with no parameters and no return type
def println():
    s=input("enter your name:  " )
    print(s)
println()
```

```
enter your name:  ljiet
ljiet
```

In [29]:
```
#2 function with parameters and no return type
def println(s):

    print(s)
s="vishal Acharya"
println(s)
```

```
vishal Acharya
```

In [30]:
```
#3 function with parameters and with return type
def println(s):

    return (s)
s="vishal Acharya"
print(println(s))
```

```
vishal Acharya
```

In [31]:
```
#3 function with parameters and with return type
def printline(s):

    return (s)
s="vishal Acharya"
t=printline(s)
print(t)
```

```
vishal Acharya
```

In [32]:
```
#4 function with no parameters and with return type
def printline():
    s=input("enter name:")
    return (s)

t=printline()
print(t)
```

```
enter name:vishal
vishal
```

returning multiple values

In [34]:
```
def calc(a,b):
    sum=a+b
    mul=a*b
    div=a/b
    return sum,mul,div
calc(8,4)
```

Out[34]:
```
(12, 32, 2.0)
```

In [35]:
```
def calc(a,b):
    sum=a+b
    mul=a*b
    div=a/b
    return sum,mul,div
x,y,z=calc(8,4)
print(x)
print(y)
print(z)
```

```
12
32
2.0
```

In [36]:
```
def calc(a,b):
    sum=a+b
    mul=a*b
    div=a/b
    return sum,mul,div
t=calc(8,4)
for i in t:
    print(i)
```

```
12
32
2.0
```

pass by refrance (in python)

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Pass by Value: The method parameter values are copied to another variable and then the copied object is passed, that’s why it’s called pass by value.

Pass by Reference: An reference to the actual parameter is passed to the method, that’s why it’s called pass by reference.

In [21]:
```
# Function definition is here pass by refrance
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist.append([1,2,3,4]);
    print ("Values inside the function: ", mylist)
    return
# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

Python pass-by-value?

In the pass-by-value model, when you call a function with a set of arguments, the data is copied into the function. This means that you can modify the arguments however you please and that you won't be able to alter the state of the program outside the function. This is not what Python does, Python does not use the pass-by-value model.

In [22]:
```
def foo(x):
    x = 4

a = 3
foo(a)
print(a)
```

```
3
```

In [23]:
```python
def clearly_not_pass_by_value(my_list):
    my_list[0] = 42

l = [1, 2, 3]
clearly_not_pass_by_value(l)
print(l)
```

```
[42, 2, 3]
```

Function arguments

These types of formal arguments can be used to call a function:

Required arguments

Keyword arguments

Default arguments

Variable-length arguments

Required arguments (Positional argument)

- Positional arguments are known as required arguments and are passed to a function in the correct order. The arguments in the function call should match the number in the function definition.

In [37]:
```python
def sub(a,b):
    return a-b
print(sub(5,10))
print(sub(10,5))
```

```
-5
5
```

In [38]:
```python
def pow(a,b):
    return a**b

print(pow(5,2))
print(pow(5))
```

```
25

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_11004\2810920390.py in <module>
      3
      4 print(pow(5,2))
----> 5 print(pow(5))

TypeError: pow() missing 1 required positional argument: 'b'
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

The user may not want to provide values for some parameters for some functions, so you may want to make them optional and use default values. Default argument values are used to accomplish this. Add the assignment operator followed by the default value to the parameter name in the function definition if you want to specify default argument values for parameters.

A constant value should be used for the default argument; it should be an immutable value by default.

In [39]:
```
def pow(a,b=2):
    return a**b

print(pow(5,2))
print(pow(6))
```

```
25
36
```

We cannot have a parameter with a default argument value

preceding a parameter without a default argument value in a function's parameter list.

The parameters are assigned values based on their positions. For example, defcal(a, b=5) is valid, but defcal(a=5, b) is not valid. Take a look at this:

In [15]:
```
def power(a=2,b):
    return a**2
print(power(6))

```

```
  File "C:\Users\VISHAL\AppData\Local\Temp\ipykernel_4288\1007664959.py", lin
e 1
    def power(a=2,b):
                  ^
SyntaxError: non-default argument follows default argument
```

In [41]:
```
def power(a,b=2):
    return a**2
print(power(6))
```

```
36
```

Keyword arguments

Arguments related to function calls are known as keyword In function calls, parameter names are used to identify keyword arguments.

If you have some functions with many parameters and only want to specify some of them, you can call them keyword arguments because we now use the name (keyword) instead of the position (which we have been using all along) as the parameter name for the arguments to the function.

You can skip arguments or place them out of order because the Python interpreter can match the values with the parameters using the keywords provided. There are two advantages of Keyword arguments:

The function is easier to use since we don't have to worry about the order of the arguments.

Only the parameters we wish to set values for can be given values, provided that the others have default arguments.

In [18]:
```python
def total(python=50,fsd=100,ps=70,De=40):
    sum=python+fsd+ps+De
    return sum
print(total(python=70,De=40,ps=60))
```

```
270
```

In [20]:
```python
def total(fsd=50,python=100,De=70,ps=40):
    sum=python+fsd+ps+De
    return sum
print(total(python=70,De=40,ps=60))
```

```
220
```

Arbitrary arguments

They are also known as variable length arguments. In some cases, we do not know in advance how many arguments will be passed to a function. This kind of a situation can be handled in Python through function calls with arbitrary For this kind of an argument, we use an asterisk before the parameter name in the function definition.

*args and *kwargs*

args and *kwargs are special Python keywords that are used to pass the variable length of arguments to a function

args

allows us to pass a variable number of non-keyword arguments to a function.

In [21]:
```python
def total(a,*b):
    sum=a
    for i in b:
        sum+=i
    print(b)
    return sum
print(total(5,2,3,1))
print(total(7,2,3,1))

```

```
(2, 3, 1)
11
(2, 3, 1)
13
```

In [23]:
```
def multiply(*v):
  product = 1

  for i in v:
    product = product * i

  print(v)
  return product
multiply(1,2,3,4,5,6,7,8,9,10)
```

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Out[23]:
```
3628800
```

In [4]:
```
def my_function(food):
    for x in food:
        print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

```
apple
banana
cherry
```

**kwargs

***kwargs allows us to pass any number of keyword arguments.*

Keyword arguments mean that they contain a key-value pair, like a Python dictionary.

In [27]:
```
def display(**vishal):
    print(vishal)
    print("items")
    for (key,value) in vishal.items():
        print(key,'->',value)
    print("key")
    for i in vishal:
        print(i)
    print("values")
    for i in vishal.values():
        print(i)
display(india='delhi',srilanka='colombo',nepal='kathmandu',pakistan='isla
```

```
{'india': 'delhi', 'srilanka': 'colombo', 'nepal': 'kathmandu', 'pakistan':
'islamabad'}
items
india -> delhi
srilanka -> colombo
nepal -> kathmandu
pakistan -> islamabad
key
india
srilanka
nepal
pakistan
values
delhi
colombo
kathmandu
islamabad
```

Points to remember while using *args and* *kwargs

order of the arguments matter(normal -> *args* -> *kwargs)

The words “args” and “kwargs” are only a convention, you can use any name of your choice

In [28]:
```
def add(a,*b,**c):
    sum=0
    print(a)
    sum+=a
    print(b)
    for i in b:
        sum+=i
    print(sum)
    print(c)
    for i in c.values():
        sum+=i
    return sum
print(add(5,3,5,7,8,x=5,y=7))

```

```
5
(3, 5, 7, 8)
28
{'x': 5, 'y': 7}
40
```

Without return statement

In [29]:
```
def is_even(x):
    if x%2==0:
        print("even")
    else:
        print("odd")
print(is_even(5))
```

```
odd
None
```

In [52]:
```
L = [1,2,3]
print(L.append(4))
print(L)
```

```
None
[1, 2, 3, 4]
```

Scope of variable

- A variable's scope is the area of a program where it is recognized. A function's parameters and variables are not visible from outside, so they have local scope.
- The life time of a variable is the period during which it remains in memory. Variables inside a function exist for the duration of the function; they are destroyed once we return from the function. As a result, a function does not remember the value of a variable from its previous calls.
- Variable scope defines which part of the program can access a particular identifier. In Python, variables fall into two basic scopes:

  Global variables (program variables)

  Local variables (function variables)

  The variables inside a function body have a local scope, while the variables defined outside have a global scope.
- Local variables can only be accessed within the function in which they are declared, while global variables can be accessed by all functions throughout the program body. The variables declared inside a function are brought into scope as soon as you call them.

In [56]:
```
def g(y):
    print("local",x)
    print("local",x+1)
x = 5
g(x)
print("global",x)
```

```
local 5
local 6
global 5
```

In [57]:
```
def f(y):
    x = 1
    x += 1
    print("local",x)
x = 5
f(x)
print("global",x)
```

```
local 2
global 5
```

In [5]:
```
def f(x):
    x+=7
    message="hello"
    print(x,message)
f(5)
print(message)
```

```
12 hello

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_19980\1558889760.py in <module>
      4     print(x,message)
      5 f(5)
----> 6 print(message)

NameError: name 'message' is not defined
```

In [24]:
```
def h(y):
    x += 1
x = 5
h(x)
print(x)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4288\1829714546.py in <module>
      2     x += 1
      3 x = 5
----> 4 h(x)
      5 print(x)

~\AppData\Local\Temp\ipykernel_4288\1829714546.py in h(y)
      1 def h(y):
----> 2     x += 1
      3 x = 5
      4 h(x)
      5 print(x)

UnboundLocalError: local variable 'x' referenced before assignment
```

In [60]:
```
def h(y):
    global x
    x += 1
x = 5
h(x)
print(x)
```

```
6
```

In [30]:
```
def f(x):
   x = x + 1
   print('in f(x): x =', x)
   return x

x = 3
z = f(x)
print('in function scope: z =', z)
print('in main program scope: x =', x)
```

```
in f(x): x = 4
in function scope: z = 4
in main program scope: x = 3
```

In [7]:
```
def f(y):
    print(x)
x=5
f(7)
```

```
5
```

In [8]:
```
# declare global variable
message = 'Hello'
def greet():
    # declare local variable
    print('Local', message)
greet()
print('Global', message)
```

```
Local Hello
Global Hello
```

Python Nonlocal Variables

- In Python, nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.
- We use the nonlocal keyword to create nonlocal variables.For example,

In [11]:
```
# outside function

def outer():
    message = 'local'
    # nested function
    def inner():
    # declare nonlocal variable
        nonlocal message
        message = 'nonlocal'
        print("inner:", message)
    inner()
    print("outer:", message)
outer()
```

```
inner: nonlocal
outer: nonlocal
```

In [1]:
```
# outside function
message="vishal"
def outer():
    message = 'local'
    # nested function
    def inner():
    # declare nonlocal variable
        nonlocal message
        message = 'nonlocal'
        print("inner:", message)
    inner()
    print("outer:", message)
outer()
print(message)
```

```
inner: nonlocal
outer: nonlocal
vishal
```

- We shouldn't use same variable name for gloabal and local variable if they are supposed act as different variables. for ex.

In [12]:
```
x=3
def fun():
    print(x) #can't access global x as it contains local variable with sam
    x=2
    x+=5
    print(x)
fun()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_19980\3103221620.py in <module>
      5     x+=5
      6     print(x)
----> 7 fun()

~\AppData\Local\Temp\ipykernel_19980\3103221620.py in fun()
      1 x=3
      2 def fun():
----> 3     print(x) #can't access global x as it contains local variable wit
h same name.
      4     x=2
      5     x+=5

UnboundLocalError: local variable 'x' referenced before assignment
```

In [ ]:

In [13]:
```
x=3
def fun(a):
    x=5
    global x
    return x*a
x=6
y=fun(5)
print(y)
print(x)
print(x)
```

```
  File "C:\Users\VISHAL\AppData\Local\Temp\ipykernel_19980\1522758319.py", li
ne 4
    global x
    ^
SyntaxError: name 'x' is assigned to before global declaration
```

Nested Functions

In [62]:
```
def f():
  def g():
    print('inside function g')

  print('inside function f')
f()
```

```
inside function f
```

In [63]:
```python
def f():
  def g():
    print('inside function g')

  g()
  print('inside function f')
f()
```

```
inside function g
inside function f
```

In [1]:
```python
def f():
  def g():
    print('inside function g')

  g()
  print('inside function f')
g()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16124\946121982.py in <module>
      5   g()
      6   print('inside function f')
----> 7 g()

NameError: name 'g' is not defined
```

In [ ]:
```python
def f():
  def g():
    print('inside function g')
    f()
  g()
  print('inside function f')
f()
```

```python
# infinate loop
```

In [3]:
```python
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('in g(x): x =', x)
    h()
    return x

x = 3
z = g(x)
print(z)
```

```
in g(x): x = 4
4
```

In [32]:
```
def g(x):
    def h():
        x = 'abc'
        return x
    x = x + 1
    print('in g(x): x =', x)
    print(h())
    return x

x = 3
z = g(x)
print(z)
```

```
in g(x): x = 4
abc
4
```

In [4]:
```
def g(x):
    def h(x):
        x = x+1
        print("in h(x): x = ", x)

    x = x + 1
    print('in g(x): x = ', x)
    h(x)
    return x

x = 3
z = g(x)
print('in main program scope: x = ', x)
print('in main program scope: z = ', z)
```

```
in g(x): x =  4
in h(x): x =  5
in main program scope: x =  3
in main program scope: z =  4
```

In [17]:
```
fire='wait'
def missile(): # a main function
        fire='detect'
        def launch (): # an inner function
            global fire
            fire='launch'
            print("Before calling inner function launch(): ",fire)
        launch() # calling inner function within outer functioin
        print("After calling inner function launch(): ",fire)
print('Before calling outer function missile():',fire)
missile() # calling outer fucntion
print("Outside both function: ",fire)
```

```
Before calling outer function missile(): wait
Before calling inner function launch():  launch
After calling inner function launch():  detect
Outside both function:  launch
```

Functions are 1st class citizens

If any programming language has the ability to treat functions as values, to pass them as arguments and to return a function from another function then it is said that programming

In [7]:
```
# type and id
def square(num):
  return num**2

type(square)
```

Out[7]:
```
function
```

In [8]:
```
id(square)
```

Out[8]:
```
1804414459040
```

In [9]:
```
# reassign
x = square
print(id(x))
x(3)
```

```
1804414459040
```

Out[9]:
```
9
```

In [10]:
```
a = 2
b = a
b
```

Out[10]:
```
2
```

In [12]:
```
# deleting a function
del square
```

In [13]:
```
square(3)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16124\1056684087.py in <module>
----> 1 square(3)

NameError: name 'square' is not defined
```

In [15]:
```
# storing
def square(num):
  return num**2
L = [1,2,3,4,square]
L[-1](3)
```

Out[15]:
```
9
```

In [16]:
```
s = {square}
s
```

Out[16]:
```
{<function __main__.square(num)>}
```

returning a function

In [17]:
```python
def f():
    def x(a, b):
        return a+b
    return x

val = f()(3,4)
print(val)
```

```
7
```

function as argument

In [18]:
```python
def func_a():
    print('inside func_a')

def func_b(z):
    print('inside func_c')
    return z()

print(func_b(func_a))
```

```
inside func_c
inside func_a
None
```

Benefits of using a function

- code modulaarity
- code readibillity
- code reusabillity