6. Function

1. Introduction

If a group of statements is repeatedly required then it is not recommended to write these statements everytime seperately.We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

1. Smallest independent part of a program
2. Collection of Statement to Solve a problem
3. Code Resuable at demand(on calling)
4. Scoping - remove ambiguity, Recursion
5. Easy Debugging
6. Modular Programming
7. Mantinace of code becomes easy

i) Python supports 2 types of functions

1. Built in Functions
2. User Defined Functions

1. Built in Functions:

The functions which are coming along with Python software automatically,are called built in functions or pre defined functions
Eg:
id()
type()
input()
eval()
etc..

2. User Defined Functions:

The functions which are developed by programmer explicitly according to requirements ,are called user defined functions.

Syntax to create user defined functions:

```
def function_name(parameters) :
    """ doc string"""
    ----
    body
    -----
    return value

Note:- 1. def (mandatory)
       2. return (optional)
```

In [2]:

```
#write a function to print Hello world

def func():
    print("Hello World")

func()
func()
```

```
Hello World
Hello World
```

ii) Parameters

Parameters are inputs to the function. If a function contains parameters,then at the time of calling,compulsory we should provide values otherwise,otherwise we will get error.

In [4]:
```
#write a wisher a function

def wish(name): #name is Parameter
    print("Hello",name,"Good Morning")

wish("Pankaj")
```

```
Hello Pankaj Good Morning
```

iii) return

Function can take input values as parameters and executes business logic, and returns output to the caller with return statement.

Q. Write a function to accept 2 numbers as input and return sum.

In [5]:
```
def add(x,y):
    return x+y

result=add(10,20)
print("The sum is",result)
print("The sum is",add(100,200))
```

```
The sum is 30
The sum is 300
```

note: If we are not writing return statement then default return value is None

In [6]:
```
def f1():
    print("Without return")

print(f1())
```

```
Without return
None
```

Note: Returning multiple values from a function

In [7]:
```
def sum_sub(a,b):
    sum=a+b
    sub=a-b
    return sum,sub
```

```
x,y=sum_sub(100,50)
print("The Sum is :",x)
print("The Subtraction is :",y)
```

```
The Sum is : 150
The Subtraction is : 50
```

In [8]:

```
def calc(a,b):
    sum=a+b
    sub=a-b
    mul=a*b
    div=a/b
    return sum,sub,mul,div

t=calc(100,50)
print("The Results are")
for i in t:
    print(i)
```

```
The Results are
150
50
5000
2.0
```

2. Different Types of function

1. Without Argument Without Return Type

```
void func():
    body
```

2. Without Arguemnt With Return Type

```
int func():
    body
    return
```

3. With Argument Without Return Type

```
void func(int a, int b):
    body
```

4. With Argument With Return Type

```
int main(int a, int b):
    body
    return
```

In [11]:

```
def hello():
    print("without Argument Without Return Type")

v = hello()
print(v)
```

```
without Argument Without Return Type
None
```

In [13]:

```
def greet():
```

```
    name = input() # not a good practice
    print(f"hello user, {name} Welcome to function")
    return 0

if greet():
    print("do you get it")
else:
    print("without argument with return type")
```

```
pankaj
hello user, pankaj Welcome to function
without argument with return type
```

In [14]:

```
def is_even(num):
    return num % 2

for i in range(1, 11):
    if is_even(i):
        print(i, end=', ')
else:
    print("\nWith Argument With Return Type")
```

```
1, 3, 5, 7, 9,
With Argument With Return Type
```

In [15]:

```
def square(a: int,b: int) -> int:
    print('type-4 with argument with return type')
    return a**2 + b**2

r = square(4.30,6.04)
print(r)
```

```
type-4 with argument with return type
54.971599999999995
```

3. Function description

In [20]:

```
def hello(a,b):
    """
        doc string
    """

help(hello)
```

```
Help on function hello in module __main__:

hello(a, b)
    doc string
```

In [21]:

```
print(hello.__doc__)
```

```
        doc string
```

In [ ]:

In [23]:

```
def sq_add(a: int, b:int) -> int:
    """
```

```
        sq_add returns addition of a and b squared
    """
    return a** + b**2
```

In [24]:
```
help(sq_add)
```

```
Help on function sq_add in module __main__:

sq_add(a: int, b: int) -> int
    sq_add returns addition of a and b squared
```

In [25]:
```
print(sq_add.__doc__)
```

```
        sq_add returns addition of a and b squared
```

Note

In [22]:
```
#help(int) #detaild Description about int
print(int.__doc__)  #short description only provided in string
```

```
int([x]) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

4. Types of Arguments

```
def f1(a,b):
 ------
 ------
 ------
f1(10,20)
```

```
    a,b are formal arguments where as 10,20 are actual arguments
```

There are 4 types are actual arguments are allowed in Python

1. Positional Argument
2. keyword arguments
3. Default Arguments
4. Variable Length Arguments
    tuple arguments
    dict arguments

i). positional arguments

These are the arguments passed to function in correct positional order

In [26]:
```
def add(x, y):
    print(f"x + y = {x+y}")

add(10, 20) # calling using position
#    0   1
```

```
x + y = 30
```

The number of arguments and position of arguments must be matched. If we change the order then result may be changed

ii) Keyword arguments

We can pass argument values by keyword i.e by parameter name

In [31]:
```
def add(x, y):
    print(f"x + y = {x+y}")

add(x=20, y=10) # calling using keywords
```

```
x + y = 30
```

In [32]:
```
add(y=20, x=10) # calling using keywords
```

```
x + y = 30
```

Here the order of arguments is not important but number of arguments must be matched

We can use both positional and keyword arguments simultaneously. But first we have to take positional arguments and then keyword arguments,otherwise we will get syntaxerror.

In [35]:
```
add(10, y=20)
```

```
x + y = 30
```

In [36]:
```
add(x=20, 10)
```

```
  File "C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/2856028354.py", line 1
    add(x=20, 10)
              ^
SyntaxError: positional argument follows keyword argument
```

In [37]:
```
add(y=20, 10)
```

```
  File "C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/2278029257.py", line 1
    add(y=20, 10)
              ^
SyntaxError: positional argument follows keyword argument
```

iii) default Argument

Sometimes we can provide default values for our positional arguments.

In [40]:
```
def sq_add(a,b,output=False):
    r = a**2 + b**2
    if output:
```

```
        print(f"a = {a}")
        print(f"b = {b}")
        print(f"a^2 + b^2 = {r}")
    return r

r = sq_add(3,4,output=True)
print(r)
```

```
a = 3
b = 4
a^2 + b^2 = 25
25
```

If we are not passing any output then only default value will be considered.

In [41]:
```
r = sq_add(4,5)
print(r)
```

```
41
```

After default arguments we should not take non default argument

In [42]:
```
def wish(name="Guest",msg="Good Morning"): #Valid
    pass
```

In [43]:
```
def wish(name,msg="Good Morning"): #Valid
    pass
```

In [44]:
```
def wish(name="Guest",msg): #Invalid
    pass
```

```
  File "C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/2928495406.py", line 1
    def wish(name="Guest",msg): #Invalid
             ^
SyntaxError: non-default argument follows default argument
```

iv) Variable Length Arugment

Sometimes we can pass variable number of arguments to our function,such type of arguments are called variable length arguments

We can declare a variable length argument with * symbol as follows

```
def f1(*n):
    pass
```

We can call this function by passing any number of arguments including zero number.Internally all these values represented in the form of tuple.

In [47]:
```
print(1,4,5,6,6)
print('h','e','l','l','o')
```

```
1 4 5 6 6
h e l l o
```

In [48]:
```
print(*"hello")
```

```
h e l l o
```

In [ ]:

```
def func(*agrs, **kwargs):
    pass

*args -->argument --> tuple
**kwargs -->keyword argument --> dict
```

argument as tuple

In [50]:

```
def fun(*args):
    """
        *args will create a tuple of all
        given positional arguments
    """
    print(args)
    print(type(args))

fun(10,'hi',1,2,3,4,5)
```

```
(10, 'hi', 1, 2, 3, 4, 5)
<class 'tuple'>
```

In [54]:

```
fun()
```

```
()
<class 'tuple'>
```

In [51]:

```
def sum_of_nums(*nums):
    s = 0
    for item in nums:
        s += item**2
    return s

r = sum_of_nums(1, 2, 3, 4, 5)
print(r)
```

```
55
```

keyword argument as dict

In [52]:

```
def func(**kwargs):
    print(kwargs)
    print(type(kwargs))

func(a=10, b=20, c=30)
```

```
{'a': 10, 'b': 20, 'c': 30}
<class 'dict'>
```

In [53]:

```
func()
```

```
{}
<class 'dict'>
```

In [ ]:

In [55]:
```
def func(pos, default=10, *args, **kwargs):
    print("positional: ", pos)
    print("default: ", default)
    print("args: ", args)
    print("kwargs: ", kwargs)

func(1, 2, 3, 4, a='b', b='c')
```

```
positional:  1
default:  2
args:  (3, 4)
kwargs:  {'a': 'b', 'b': 'c'}
```

In [56]:
```
func(5, 20, 2, 3,4 ,5 , 76)
```

```
positional:  5
default:  20
args:  (2, 3, 4, 5, 76)
kwargs:  {}
```

In [57]:
```
func(5, 20)
```

```
positional:  5
default:  20
args:  ()
kwargs:  {}
```

In [58]:
```
func(5)
```

```
positional:  5
default:  10
args:  ()
kwargs:  {}
```

Note: After variable length argument,if we are taking any other arguments then we should provide values as keyword arguments

In [60]:
```
def f1(*s,n1):
    for s1 in s:
        print(s1)
    print(n1)

f1("A","B",n1=10)
```

```
A
B
10
```

In [61]:
```
f1("A","B",10) #Invalid
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/1839537257.py in <module>
----> 1 f1("A","B",10)

TypeError: f1() missing 1 required keyword-only argument: 'n1'
```

5. Scopes

built-in scope

global (main) scope
module scope
local scope
nonlocal scope

i) Global variable

The variables which are declared outside of function are called global variables. These variables can be accessed in all functions of that module.

In [63]:
```
a = 10
def f1():
    print(a)

def f2():
    print(a)

f1()
f2()
```

```
10
10
```

Note: without explicitly defined we can not modify a global variable inside a local scope

In [75]:
```
b = 10
def f1():
    c = c*2
    print(c)

f1()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/4049267372.py in <module>
      4     print(c)
      5
----> 6 f1()

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/4049267372.py in f1()
      1 b = 10
      2 def f1():
----> 3     c = c*2
      4     print(c)
      5

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

Note : this is not modyfying global variable it's creating another local variable d

In [78]:
```
d = 10
print(id(d))
def f1():
    d = 10 * 2
    print(d)
    print(id(d))

f1()
```

```
140732175210432
20
140732175210752
```

ii) Local Variables

The variables which are declared inside a function are called local variables. Local variables are available only for the function in which we declared it.i.e from outside of function we cannot access.

In [77]:
```
def f1():
    e = 10
    print(e)

def f2():
    print(e)

f1()
f2()
```

```
10
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/1272219139.py in <module>
      7
      8 f1()
----> 9 f2()

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/1272219139.py in f2()
      4
      5 def f2():
----> 6     print(e)
      7
      8 f1()

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

iii) global keyword

We can use global keyword for the following 2 purposes:

1. To declare global variable inside function
2. To make global variable available to the function so that we can perform required modification

In [82]:
```
f = 10
def f1():
    f = 777    # it will not modify the gobal variable
    print(f)   # it will create another f local variable

def f2():
    print(f)

f1()
f2()
```

```
777
10
```

modify global variable inside function

In [83]:
```
f = 10
def f1():
    global f
```

```
    f = 777   # it will modify the gobal variable
    print(f)

def f2():
    print(f)

f1()
f2()
```

```
777
777
```

declare global variable inside function

In [88]:

```
def f1():
    global g
    g = 30   # it will create gobal variable g
    print(g)

def f2():
    print(g)

f1()
f2()
```

```
30
30
```

Note: If global variable and local variable having the same name then we can access global variable inside a function as follows

here we are using as dictionary key to using global variable bcz every object which will create is storing in dictionary(key-value) form in global scope

In [90]:

```
a=10 #global variable

def f1():
    a=777 #local variable
    print(a)
    print(globals()['a'])

f1()
```

```
777
10
```

In [104…

```
%%writefile one.py
from pprint import pprint
x = 10
y = 20
def func():
    z = 30
    print("\nIn Local Space")
    pprint(locals())
    print("\nIn global Space")
    pprint(globals())

print("\nIn global Space")
pprint(globals())
func()
```

```
Overwriting one.py
```

In [105…

```
!python one.py
```

```
In global Space
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': 'one.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000001A8818826A0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'func': <function func at 0x000001A8813B5160>,
 'pprint': <function pprint at 0x000001A881A8E280>,
 'x': 10,
 'y': 20}

In Local Space
{'z': 30}

In global Space
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': 'one.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000001A8818826A0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'func': <function func at 0x000001A8813B5160>,
 'pprint': <function pprint at 0x000001A881A8E280>,
 'x': 10,
 'y': 20}
```

In [112…

```
%%writefile one.py
x = 10
def func():
    print(locals()==globals())

print(locals()==globals())
func()
```

```
Overwriting one.py
```

In [113…

```
!python one.py
```

```
True
False
```

iv) non local variable

In [118…

```
x = 10
def outer():
    x = 20 # Local for outer, nonlocal for inner
    def inner():
        print("inner", x) # nonlocal
    inner()
    print("outer", x) # local

print('before', x)
outer()
print('after',x)
```

```
before 10
inner 20
outer 20
after 10
```

Note: In local we can not modify non local variable and global variable

In [121…

```
x = 10 # global
def outer():
    x = 20 # local for outer, nonlocal for inner
    def inner():
        x = x + 10 # local
        print("inner", x)
    inner()
    print("outer", x)

print('before',x)
outer()
print('after',x)
```

```
before 10
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/4052610792.py in <module>
      9
     10 print('before',x)
---> 11 outer()
     12 print('after',x)

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/4052610792.py in outer()
      5         x = x + 10 # local
      6         print("inner", x)
----> 7     inner()
      8     print("outer", x)
      9

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/4052610792.py in inner()
      3     x = 20 # local for outer, nonlocal for inner
      4     def inner():
----> 5         x = x + 10 # local
      6         print("inner", x)
      7     inner()

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

we can modify nonlocal by using nonlocal keyword

we can modify global by using global keyword

In [122…

```
x = 5 # global
def outer():
    x = 20
    def inner():
        nonlocal x
        x = x + 10
        print("inner", x)
    inner()
    print("outer", x)
print('before', x)
outer()
print('after', x)
```

```
before 5
inner 30
```

```
outer 30
after 5
```

In [123…

```
x = 5 # global
def outer():
    x = 20
    def inner():
        global x
        x = x + 10
        print("inner", x) # 15
    inner()
    print("outer", x) # 20

print('before', x) # 5
outer()
print('after', x) # 15
```

```
before 5
inner 15
outer 20
after 15
```

6. Function Aliasing

For the existing function we can give another name, which is nothing but function aliasing.

In [2]:

```
def wish(name):
    print("Good Morning:",name)

greeting=wish
print(id(wish))
print(id(greeting))

greeting('Pankaj')
wish('Pankaj')
```

```
2207648648928
2207648648928
Good Morning: Pankaj
Good Morning: Pankaj
```

Note: If we delete one name still we can access that function by using alias name

In [3]:

```
def wish(name):
    print("Good Morning:",name)

greeting=wish
del wish

greeting('Pankaj')
#wish('Pankaj') # NameError:name 'wish' is not defined
```

```
Good Morning: Pankaj
```

7. Nested Function

We can declare a function inside another function, such type of functions are called Nested functions

In [4]:
```
def outer():
    print("outer function started")
    def inner():
        print("inner function execution")
    print("outer function calling inner function")
    inner()

outer()
#inner() ==>NameError: name 'inner' is not defined
```

```
outer function started
outer function calling inner function
inner function execution
```

Note: A function can return another function

In [5]:
```
def outer():
    print("outer function started")
    def inner():
        print("inner function execution")
    print("outer function returning inner function")
    return inner

f1=outer()
f1()
f1()
f1()
```

```
outer function started
outer function returning inner function
inner function execution
inner function execution
inner function execution
```

Q. What is the differenece between the following lines?

```
f1 = outer
f1 = outer()
```

In the first case for the outer() function we are providing another name f1(function aliasing).

But in the second case we calling outer() function,which returns inner function.For that inner function() we are providing another name f1

8. Problems

What is the output ?

In [69]:
```
def extendList(val, list=[]):
    print(id(list))
    list.append(val)
    return list

list1 = extendList(10)
list2 = extendList(123,[])
list3 = extendList('a')

print(f"list1 = {list1}")
```

```
print(f"list2 = {list2}")
print(f"list3 = {list3}")
```

```
2406438721472
2406438721600
2406438721472
list1 = [10, 'a']
list2 = [123]
list3 = [10, 'a']
```

Problems on Scope

In [92]:

```
x = 5
def func():
    x = 10
    print(f'inside x is {x}')

print(f"before x is {x}.")
func()
print(f"after x is {x}.")
```

```
before x is 5.
inside x is 10
after x is 5.
```

In [93]:

```
x = 10 # global
def func(x):
    x = x * 2 # local
    print('inside x is ', x)

print('before x is ', x)
func(x)
print('after x is ', x)
```

```
before x is  10
inside x is  20
after x is  10
```

In [94]:

```
x = 100
def func():
    print(f"Inside Function x is ", x**2)

print('before x is ', x)
func()
print('after x is ', x)
```

```
before x is  100
Inside Function x is  10000
after x is  100
```

note without explicitly defined we can not modify a global variable inside a local scope

In [95]:

```
x = 10
def func():
    x = x * 2
    print('inside', x)

print('before', x)
func()
print('after', x)
```

```
before 10
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/3308667349.py in <module>
      5
      6 print('before', x)
----> 7 func()
      8 print('after', x)

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_10332/3308667349.py in func()
      1 x = 10
      2 def func():
----> 3     x = x * 2
      4     print('inside', x)
      5

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

In [96]:

```
x = 10
def func():
    global x
    x = x * 2
    print('inside', x)

print('before', x)
func()
print('after', x)
```

```
before 10
inside 20
after 20
```

In [97]:

```
x = []
def func():
    global x
    x = x.append(100)
    print('inside x is ', x)

print('before x is ', x)
func()
print('after x is ', x)
```

```
before x is  []
inside x is  None
after x is  None
```

In [114…

```
%%writefile one.py
from pprint import pprint
x = 10
def func():
    print(locals())
    print(globals() == locals())
#pprint(globals())
print(globals() == locals()) # global
func()
```

```
Overwriting one.py
```

In [115…

```
!python one.py
```

```
True
{}
False
```

In [116…

```
x = 10
def func():
    x = 20
    globals()['x'] = 100
    print("inner", x)
print("before", x)
func()
print("after", x)
```

```
before 10
inner 20
after 100
```

In [ ]: