In [1]:

```
def print_name():
    '''This is
        demo program to print the name'''
    print("Kavita")
```

In [2]:

```
print_name() #calling of the function
```

```
Kavita
```

In [3]:

```
#Functions can also take the parameter
def printing_name(name):
    for i in range(5):
        print(name)

printing_name("Yogita")
```

```
Yogita
Yogita
Yogita
Yogita
Yogita
```

In [6]:

```
#A function can return the value
def return_name(name):
    string1="Hello "+name
    return string1


print(return_name("Ritika"))
print('End of the function')
```

```
Hello Ritika
End of the function
```

In [8]:

```
#adding the two numbers
def add(num1,num2):
    return num1+num2
add('4','5')
```

Out[8]:

```
'45'
```

In [9]:

```
#program to return all the even numbers from the given list
def even_numbers(given_list):
    even_list=[]
    for num in given_list:
        if num%2==0:
            even_list.append(num)
        else:
            pass
    return even_list

list2=[1,2,4,5,9,23,12,56,34,7,22]
even_numbers(list2)
```

Out[9]:

```
[2, 4, 12, 56, 34, 22]
```

In [28]:

```
#Tuple unpacking with functions
employee_details=[('ABC',100), ('XYZ',60), ('PQR',75), ('MNO',55)]
#program to return the names of the employee who works for more than 65 hours

def emp_names(elist):
    hours=0
    names=[]
    for a,b in elist: #tuple unpacking
        if b>65:
            names.append(a)
            hours=b

    return names,hours
emp_names(employee_details)
```

Out[28]:

```
(['ABC', 'PQR'], 75)
```

In [ ]:

```

```

In [31]:

```
myList = [2, 109, False, 10, "Demo", 482, "List"]
print(myList)
print(id(myList))
# Function definition is here
def changeme( mylist ):
    print(id(mylist))
    "This changes the passed list into this function"
    print("Values inside the function before change: ", mylist)
    mylist[2]=50
    print("Values inside the function after change: ", mylist)

changeme(myList)
print(myList)
#myList #This will return the modified list from the function
```

```
[2, 109, False, 10, 'Demo', 482, 'List']
77763656
77763656
Values inside the function before change:  [2, 109, False, 10, 'Demo', 482, 'List']
Values inside the function after change:  [2, 109, 50, 10, 'Demo', 482, 'List']
[2, 109, 50, 10, 'Demo', 482, 'List']
```

In [14]:

```
#pass by reference in Python


def modify_list(demolist):
    demolist=[1,2,3] # This would assign new reference
    print('Values inside the function:', demolist)
    return


demolist=[1,20,30,40]
modify_list(demolist)
print('Values outside the function', demolist)
```

```
Values inside the function: [1, 2, 3]
Values outside the function [1, 20, 30, 40]
```

In [18]:

```
#From the given tuple pairs, find the costliest item and its name
```

```
#from the given tuple pairs, find the costliest item and its name
list2=[('Book', 500),('Pen',100),('Writing pad',5000),('Notebook',80),('Eraser',20)]

def find_max(dlist):
    name=''
    price=0
    for a,b in dlist:
        if b>price:
            price=b
            name=a
        else:
            pass
    return name,price

find_max(list2)
```

Out[18]:

```
('Writing pad', 5000)
```

In [19]:

```
print(find_max(list2))
```

```
('Writing pad', 5000)
```

In [17]:

```
#tuple unpacking can be applied at the time of calling the function
x,y=find_max(list2)
print('The ',x , ' is costliest with ',y, 'rupees')
```

```
The  Writing pad  is costliest with  5000 rupees
```

In [19]:

```
#Using keyword arguments

def print_details(name, age):
    print('Name is ', name)
    print('Age is ', age)
    print('After 10 years, age will be ', age+10)

name='Jack'
age=20
print_details(name, age)
print_details(age=25, name='Jill')
```

```
Name is  Jack
Age is  20
After 10 years, age will be  30
Name is  Jill
Age is  25
After 10 years, age will be  35
```

In [21]:

```
#global and local variables
sum1=0
def adding(num1, num2):
    print('Adding two numbers')
    global sum1
    sum1=num1+num2
    print(f'Addition of {num1} and {num2} is {sum1}')
    return
print(f'Before addition {sum1}')
adding(10,20)
print(f'After addition {sum1}')
```

```
Before addition 0
Adding two numbers
Addition of 10 and 20 is 30
After addition 30
```

In [ ]:

```
#using global variable inside the function
sum1=0
def addingagain(num1, num2):
    print('Adding two numbers')
    global sum1
    sum1=num1+num2
    print(f'Addition of {num1} and {num2} is {sum1}')
    return
print(f'Before addition {sum1}')
addingagain(10,20)
print(f'After addition {sum1}')
```

In [38]:

```
def demo_func(*args):
    print('Inside the function')
    print(type(args))
    #print(args+args1)
    print(sum(args))
    return
demo_func(10,20,30)
```

```
Inside the function
<class 'tuple'>
60
```

In [35]:

```
demo_func(10,12)
```

```
Inside the function
<class 'int'>
22
```

In [39]:

```
demo_func(12,34,23,67) #generate an error, the solution is to introduce the default argum
ent
```

```
Inside the function
<class 'tuple'>
136
```

In [61]:

```
#Assigning function
def printing(name):
    print('Hello')
    print('Hello '+name)
    return
```

In [62]:

```
printing('Jack')
```

```
Hello
Hello Jack
```

In [64]:

```
new_func=printing #Assigning one function to another
print(id(new_func))
print(id(printing))
print(type(new_func))
new_func('Kavita')
```

```
85028376
85028376
<class 'function'>
```

```
Hello
Hello Kavita
```

In [7]:

```
var=10
def func11( x):
    var=x+1
    print(var)
    print(globals() ['var'])


func11(20)
print(var)
```

```
21
10
10
```

In [ ]:

```
def myfunc(a,b):
    return sum((a,b))*0.05
```

In [ ]:

```
myfunc(10,20)
```

In [ ]:

```
#It is possible to use multiple arguments inside the function and set the
#non compulsory variables to default values, in this case 0
```

In [ ]:

```
#alternatively we can use variable arguments

def myfunc1(*args):
    return sum(args)*0.5
```

In [ ]:

```
myfunc1(12,23,1)
```

In [13]:

```
#passing dictionary as a parameter, try to run the cell and check the output
def dfunc(D):
    for items in D:
        print(items," ",D[items])


dict_1={1:'One', 2:'Two', 3:'Three'}
dfunc(dict_1)
```

```
1   One
2   Two
3   Three
```

In [70]:

```
def new_fun(**kwargs):
    print('The variables are')
    print(f"{kwargs['var1']}  {kwargs['var2']}")
    #print(f" {kwargs['var1']}  {kwargs['var2']}")
    return
new_fun(var2=100, var1=200, var3=300, var4=400)
```

```
The variables are
200  100
 200 100
```

In [76]:

```
dict11={1:'One', 2:'Two', 3:'Three'}
for i in dict11:
    pass
    #print(dict11[i])

def nfunc(**kwargs):
    for i in kwargs:
        print("{}".format(kwargs[i]))
    return
nfunc(v1='One', v2='Two', v3='Three')
#do not use nfunc(1='One', 2='Two', 3='Three'), try to run the cell and check the error
```

```
One
Two
Three
```

In [45]:

```
# *args for variable arguments
#*kwargs for passing keyworded, variable length dictionary argument list to the calling f
unction.

def myfunc2(**kwargs):
    if 'fruit' in kwargs:
        print(type(kwargs))
        print(f"My favorite fruit is {kwargs['fruit']}")   # review String Formatting and
f-strings if this syntax is unfamiliar
    else:
        print("I don't like fruit")

myfunc2(fruits='pineapple', softdrink='coke', veggies='Potato')
```

```
I don't like fruit
```

In [77]:

```
#Using args and **kwargs in the same function
def myfunc3(*args, **kwargs): #changing the sequence will give the syntax error
    print(args)
    print(type(args))
    print(kwargs)
    print('I would like {} {}'.format(args[0], kwargs['food']))
    return
myfunc3(10,20,30, fruit='Grapes', food='breads', veggies='lettuce' )
```

```
(10, 20, 30)
<class 'tuple'>
{'fruit': 'Grapes', 'food': 'breads', 'veggies': 'lettuce'}
I would like 10 breads
```

In [67]:

```
#Define one function in another

def calc(a,b):
    print('a = {} b= {}' .format(a,b))

    def addition():
        print('{} + {} = {}'.format(a,b,(a+b)))

    addition()
    return

calc(10,20)
```

```
a = 10 b= 20
10 + 20 = 30
```

In [66]:

```
#Pass one function as parameter in another function
def Function1(text):
    return text.upper()

def Function2(text):
    return text.lower()

def Demo(Another_function):
    # storing the function in a variable
    greeting = Another_function("Demonstration of function passed as an argument")
    print(greeting)

Demo(Function1)
Demo(Function2)
```

```
DEMONSTRATION OF FUNCTION PASSED AS AN ARGUMENT
demonstration of function passed as an argument
```

In [65]:

```
#Possible that a function can return another function
def display():
    def message():
        return 'This is a function that returns another function'

    return message

#Call display() which inturn calls message()
#Returing function which is assigned to a variable
#print(display())
Function=display()


#Demonstrating the printing of messages
print(Function())
print(Function)
```

```
This is a function that returns another function
<function display.<locals>.message at 0x0000000004E9BD90>
```

In [ ]:

```
def another_func():
    def afunc():
        print("Inside the nested function")

    return afunc
print(another_func)
function1=another_func()
function1()
print('last statement')
```

In [16]:

```
#returning function
def f1():
    print( 'Inside the function1')
def f2():
    print('Inside the function2')
    return f1()
f2()
```

```
Inside the function2
Inside the function1
```

In [ ]:

```
# defining a decorator
def hello_decorator(func):
```

```python
    # inner1 is a Wrapper function in
    # which the argument is called

    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")

        # calling the actual function now
        # inside the wrapper function.
        func()

        print("This is after function execution")

    return inner1


# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function !!")


# passing 'function_to_be_used' inside the
# decorator to control its behavior
function_to_be_used = hello_decorator(function_to_be_used)


# calling the function
function_to_be_used()
```