Topics:

- **Introduction to Functions**
  - Declaration and Definition
  - Variable Scope and Lifetime
  - Return Statements
  - Types of Arguments
  - Lambda function
  - Recursion

Functions

- **A Function is a self block of code.**
- A function is a block of organized and reusable program code that performs a single, specific and well defined task
- A Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.
- A Function is a subprogram that works on data and produce some output.
- . Functions provide better modularity for your application and a high degree of code reusing.

There are two types of Functions.

- **Built-in Functions**: Functions that are predefined by python interpreter. We have used many predefined functions in Python.
- **User- Defined Functions**: Functions that are created according to the requirements.

Benefits of Modularizing a Program with Functions

- A program benefits in the following ways when it is broken down into functions:
- **Simpler Code**
- A program's code tends to be simpler and easier to understand when it is broken down into functions.
- **Code Reuse**
- Functions also reduce the duplication of code within a program. If a specific operation is performed in several places in a program, a function can be written once to perform that operation and then be executed any time it is needed
- **Better Testing**
- When each task within a program is contained in its own function, testing and debugging becomes simpler

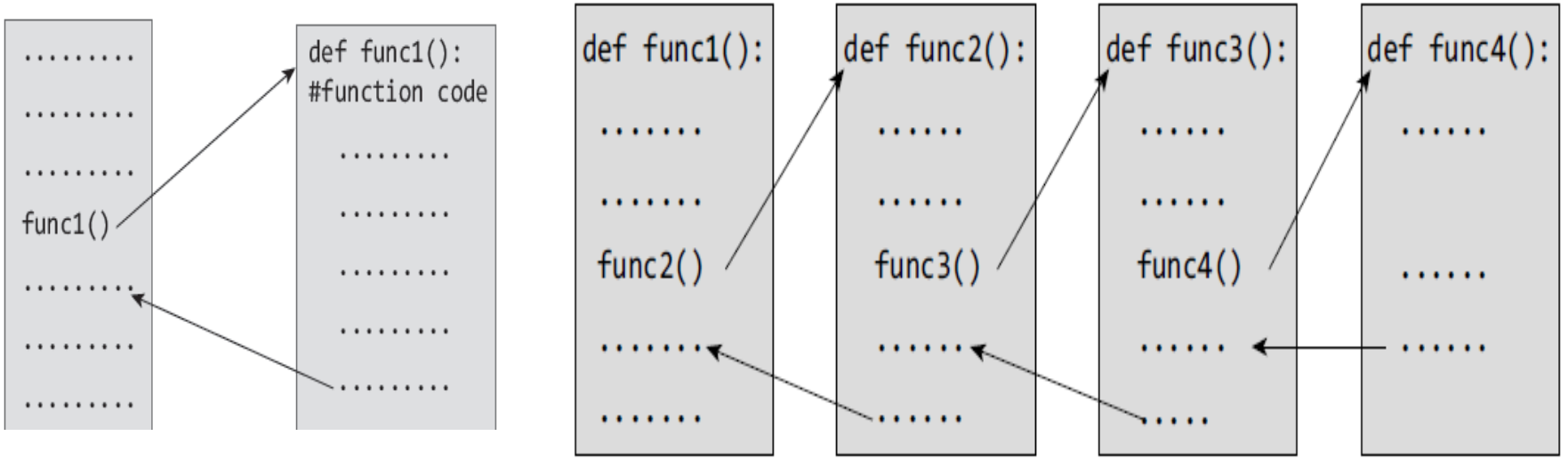
- **Faster Development**
- It doesn't make sense to write the code for these tasks multiple times. Instead, functions can be written for commonly needed tasks & those functions can be incorporated into each program that needs them.
- **Easier Facilitation of Teamwork**
- Functions also make it easier for programmers to work in teams. When a program is developed as a set of functions that each performs an individual task, then different programmers can be assigned the job of writing different functions.

Need for Functions

- Each function to be written and tested separately.
- Understanding, coding and testing multiple separate functions is far easier.
- Without the use of any function, then there will be countless lines in the code and maintaining it will be a big mess.
- Programmers use functions without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that he has to write.
- Different programmers working on that project can divide the workload by writing different functions.
- Like Python libraries, programmers can also make their functions and use them from different point in the main program or any other program that needs its functionalities

Functions

Python enables its programmers to break up a program into segments commonly known as functions, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task



func1() calls func2(), therefore func1() is known as calling function and func2() is known as called function

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Function Declaration and Definition

- A function, **f** that uses another function **g,** known as the **calling function** and g is known as the **called function.**
- The inputs that the function takes are known as **arguments/parameters.**
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- **Function declaration** is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.
- **Function definition** consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

User Defined Function Declaration and Definition

- When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts
  - **Function Header** **Function Body**
- A Function defined in Python should follow the following format:

1) Keyword **def** is used to start the Function Definition.def specifies the starting of Function block.

2) def is followed by function-name followed by parenthesis ( ).

3) Arguments/Parameters are passed inside the parenthesis. [Optional], At the end a colon **:** is marked.

4)First statement of a function is Optional documentation string (docstring) to describe what the function does.

5) Statements must have same indentation level (usually 4 spaces

6) An optional return statement to return a value from the function

- A function may have a return[expression] statement. That is, the return statement is optional.
- You can assign the function name to a variable. Doing this will allow you to call same function using the name of that variable

Syntax of function definition:

```
def  function_name ([parameters]):   # Function Header
        Documentation String         # Function Body
        Statement Block
        return[ Expression]
```

- The parameter list in the function definition as well as function declaration must match with each other
- Before calling a function you must define it as you assign variables before using them

```
# Example of a function

def greet(name):
    '''This function greets
    to the person passed in
    as parameter'''
    print("Hello,"+name+".Good Morning!")

# Function Call
greet(" CSE")
```

- Once we have defined a function, we can call it from another function, Program or even the Python Prompt. To call a function we simply type the function name with appropriate parameters.

Doc string

- The first string after the function header is called the doc string and is short for documentation string. It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice.
- We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as __doc__ attribute of the function

```
def func():
    """ The Program prints a msg
    Hello World"""
    print("Hello world!!")
print(func.__doc__)
```

```python
#WAPP to add two numbers using a function
def add(a,b): # Function to add two values
    return a+b
a=20
b=30
task=add      # function name assigned to a variable
print(task(a,b))# function called using variable name



#WAPP to display a string using function repetatedly
def func(): # Function Definition
    for i in range(10):
        print("Hello Python")
func()        # Function Declaration
```

Invoking a Function:

- To execute a function it needs to be called. This is called function calling.
- Function Definition provides the information about function name, parameters and the definition what operation is to be performed. In order to execute the Function Definition it is to be called.
- The function call statement invokes the function. When a function is invoked the program control jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

Function Parameters

- A function can take parameters which are nothing but some values that are passed to it so that the function can manipulate them to produce the desired result.
- These parameters are normal variables with a small difference that the values of these variables are defined (initialized) when we call the function and are then passed to the function.
- Function name and the number and type of arguments in the function call must be same as that given in the function definition.
- If the data type of the argument passed does not matches with that expected in function then an error is generated.

```
# This program demonstrates a function.
# First, we define a function named message.
def msg():
    print("Iam working on it,")
    print("shall let you know once I finish it off")
# Call the message function.
msg()
```

NOTE: A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must call it. This is how we would call the message function

```
# This program has two functions. First we
# define the main function.
def main( ):
    print('I have a message for you.')
    message( )
    print('Goodbye!')
# Next we define the message function.
def message( ):
    print("Iam working on it,")
    print("shall let you know once I finish it off")
# Call the main function.
main( )
```

```
 #Providing Function Definition
def sum(x,y):
    "Going to add x and y"
    s=x+y
    print("Sum of two numbers is")
    print(s)
  #Calling the sum Function
sum(10,20)
sum(20,30)
```

Output:

- NOTE: Function call will be executed in the order in which it is called

```
def func(i):
    print("Hello Please join us!",i)
func(5+3*2)
```

```
                # func. definition header accepts a var. with name i
def func(i):
    print("Hello Please join us!",i)
j=10
func(j)      # Function is called using variable j
```

```
def total(a,b):# function accepting parameters
    result=a+b
    print("sum of ",a,"and",b,"=",result)
a=int(input("Enter the first number:"))
b=int(input("Enter the Second number:"))
total(a,b) # Function call with two arguments
```

Advantages of user-defined functions

•User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.

•If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.

•Programmers working on large project can divide the workload by making different functions.

Scope of Variable

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.
- Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.
- *Scope* defines the visibility of a name within a block.
  - If a local variable is defined in a block, its scope is that particular block.
  - If it is defined in a function, then its scope is all blocks within that function. .
- Scope of a variable can be determined by the part in which variable is defined.
- Each variable cannot be accessed in each part of a program.

Local and Global Variables

- **There are two types of variables based on Scope:**

  1) Local Variable.

  2) Global Variable.

- A variable which is defined within a function is local to that function. A local variable can be accessed from the point of its definition until the end of the function in which it is defined. It exists as long as the function is executing. Function parameters behave like local variables in the function. Moreover, whenever we use the assignment operator (=) inside a function, a new local variable is created.

- Global variables are those variables which are defined in the main body of the program file. They are visible throughout the program file. As a good programming habit, you must try to avoid the use of global variables because they may get altered by mistake and then result in erroneous output.

1) Local Variables:

- Variables declared inside a function body is known as Local Variable. These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

Eg:

```
def msg():
    a=10
    print("Value of a is",a)
    return
msg()
print(a)
#it will show error since variable is local
```

```
  print(a)
NameError: name 'a' is not defined
```

- When a variable name is used in a code block, it is resolved using the nearest enclosing scope. If no variable of that name is found, then a NameError is raised.
- In the code given below, str is a global string because it has been defined before calling the function

```
def func():
    print(str)
str="Welcome"
func()
```

- **b) Global Variable:**
- Variable defined outside the function is called Global Variable. Global variable is accessed all over program thus global variable have widest accessibility.
- **Ex:**

```
b=20
def msg():
    a=10
    print("value of a is",a)
    print("value of b is",b)
    return
msg()
print(b)
print(a)
```

```
def my_func():
    x=10
    print("Prints value inside function",x)
x=20
my_func()
print("Prints value outside function",x)
```

- Here, we can see that the value of *x* is 20 initially. Even though the function my_func( )changed the value of *x* to 10, it did not effect the value outside the function. This is because the variable *x* inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.
- On the other hand, variables outside of the function are visible from inside. They have a global scope. We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the **keyword global.**

Local and Global Variables

```
num1=10 # Global Variable
print("global variable num1=",num1)
def fun(num2):  #num2 is function parameter
    print("In function-local variable num2=",num2)
    num3=30     # num3 is a local variable
    print("In function-local variable num3=",num3)
fun(20)     # 20 is passed as an argument to the function
print("num1 again =", num1)
print("num3 outside function=",num3)
```

Using the Global Statement

•To define a variable defined inside a function as global, you must use the global statement. This declares the local or the inner variable of the function to have module scope.

```
def show():
    global var1
    var1="Good Morning"
    print("In function var is-",var)
    print("In function var is-",var1)
var="Good"
show()
print("Outside function,var1 is ",var1)
print("var is", var)
```

•All the variables have the Scope of the Block

•Variables can only be used after the point of their declaration

The Return Statement

•The return statement is used to exit a function and go back to the place from where it was called

Syntax of return statement :

return [expression]

•The expression is written in brackets because it is optional.

•If the expression is present, it is evaluated and the **resultant value** is returned to the calling function, if no expression is specified then the function will return **None**.

The return statement is used for two things.

• Return a value to the caller

• To end and exit a function and go back to its caller

```
def greet(name):
    """This function greets to
    the person passed in as
    parameter"""
    print("Hello, " + name + ". Good morning!")
print(greet("May"))
```

```
def cube(x):
    return(x*x*x)
    #return
num=20
result=cube(num)
print("cube of",num,'=',result)
```


K.RaviKanth, Asst.Prof., Dept. of CSE, IIIT- RGUKT-Basara, T.S, IND.

```
def absolute_value(num):
    '''Ths fucntion returns the absolute
value of the entered number'''
    if num>=0:
        return num
    else:
        return -num
print(absolute_value(2))
print(absolute_value(-4))
```

```
def sum(a,b):
    "Adding the two values"
    print("Printing within Function")
    print(a+b)
    return (a+b)        # Try with out return
def msg():
    print("Hello")
    return
total=sum(10,20)
print("Printing Outside:",total)
msg( )
print("Rest of code")
```

Python Function Argument and Parameter:

- There can be two types of data passed in the function.

1) The First type of data is the data passed in the function call. This data is called arguments.

2) The second type of data is the data received in the function definition. This data is called parameters.

- Arguments can be variables and expressions.
- Parameters must be variable to hold incoming values.
- Alternatively, arguments can be called as actual parameters or actual arguments and parameters can be called as formal parameters or formal arguments.

```
def add(x,y):
    print(x+y)
x=15
add(x,10)
add(x,x)
y=20
add(x,y)
```

```
def greet(name,msg):
    """This function greets to
    the person with the provided message"""
    print("Hello",name + ', ' + msg)
greet("Monica","Good morning!")
```

- Here, the function greet() has two parameters. Since, we have called this function with two arguments, it runs smoothly and we do not get any error. But if we call it with different number of arguments, the interpreter will complain. Below is a call to this function with one and no arguments along with their respective error messages.

```
greet("Monica")     # only one argument
```

```
TypeError: greet() missing 1 required positional argument: 'msg'
```

```
greet()         # no arguments
```

```
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

Passing Parameters

- Apart from matching the parameters, there are other ways of matching the parameters.
- Python supports following types of formal argument:

  1) Required argument or Positional argument

  2) Keyword argument or Named argument

  3)Default argument

  4) Variable length argument

Positional/Required Arguments

- In the required arguments, the arguments are passed to a function in correct positional order. Also, the number of arguments in the function call should exactly match with the number of arguments specified in the function definition
- When the function call statement must match the number and order of arguments as defined in the function definition it is Positional Argument matching
- Ex:

```
#Function definition of sum
def sum(a,b):
    "Function having two parameters"
    c=a+b
    print(c)
sum(10,20)
sum(20)
```

- **Explanation:**

  1) In the first case, when sum() function is called passing two values i.e., 10 and 20 it matches with function definition parameter and hence 10 and 20 is assigned to a and b respectively. The sum is calculated and printed.

  2) In the second case, when sum() function is called passing a single value i.e., 20 , it is passed to function definition. Function definition accepts two parameters whereas only one value is being passed, hence it will show an error.

```
def display():
    print("hello")
display("Hi")
```

```
    display("Hi")
TypeError: display() takes 0 positional arguments but 1
was given
```

```
  def display(str):
      print(str)
  display()
```

```
    display()
TypeError: display() missing 1 required positional argu
ment: 'str'
```

```
   def display(str):
       print(str)
   str="Hello"
   display(str)
```

```
Hello
>>>
```

Keyword Arguments

- When we call a function with some values, the values are assigned to the arguments based on their position.
- Python also allow functions to be called using keyword arguments in which the order (or position) of the arguments can be changed.
- The values are not assigned to arguments according to their position but based on their name (or keyword).
- Keyword arguments are beneficial in two cases.
  - First, if you skip arguments.
  - Second, if in the function call you change the order of parameters.
- Using the Keyword Argument, the argument passed in function call is matched with function definition on the basis of the name of the parameter.

```
def msg(s_id,name):
    "Printing passed value"
    print(s_id)
    print(name)
    return
msg(s_id=100,name='Raj')
msg(name='Rahul',s_id=101)


def display(str,int_x,float_y):
    print("The string is:",str)
    print("The integer value is:",int_x)
    print("The Floating value is:",float_y)
display(float_y=34545.90,str="WElcome",int_x=2345)
```

- **Output:**

  >>>
  100
  Raj
  101
  Rahul

- **Explanation:**
- 1) In the first case, when msg() function is called passing two values i.e., s_id and name the position of parameter passed is same as that of function definition and hence values are initialized to respective parameters in function definition. This is done on the basis of the name of the parameter.
- 2) In second case, when msg() function is called passing two values i.e., name and s_id, although the position of two parameters is different it initialize the value of id in Function call to id in Function Definition. same with name parameter. Hence, values are initialized on the basis of name of the parameter.

Default Arguments

- Python allows users to specify function arguments that can have default values. This means that a function can be called with fewer arguments than it is defined to have. That is, if the function accepts three parameters, but function call provides only two arguments, then the third parameter will be assigned the default (already specified) value.
- Default Argument is the argument which provides the default values to the parameters passed in the function definition, in case value is not provided in the function call.
- The default value to an argument is provided by using the assignment operator (=).
- Users can specify a default value for one or more arguments.

```
#Function Definition
def msg(Id,Name,Age=20):
    "Printing the passed value"
    print(Id)
    print(Name)
    print(Age)
    return
#Function call
msg(Id=100,Name='Rajesh',Age=20)
msg(Id=101,Name='Richa')
```

- **Output:**

  100

  Rajesh

  20

  101

  Richa

  20

- **Explanation:**

  1) In first case, when msg() function is called passing three different values i.e., 100 , Rajesh and 20, these values will be assigned to respective parameters and thus respective values will be printed.

  2) In second case, when msg() function is called passing two values i.e., 101 and Richa, these values will be assigned to Id and Name respectively. No value is assigned for third argument via function call and hence it will retain its default value i.e, 20.

```
def display(name,course="B.Tech"):
    print("name:",name)
    print("course:",course)
display(course="B.Tech", name="Joseph")#Keyword Arg
display(name="Rajesh")#Default Argument
```

Variable-length Arguments

- In some situations, it is not known in advance how many arguments will be passed to a function.
- In such cases, Python allows programmers to make function calls with arbitrary (or any) number of arguments.
- When we use arbitrary arguments or variable length arguments, then the function definition use an asterisk (*) before the parameter name.
- The syntax for a function using variable arguments can be given as,

```
def function_name([arg1,arg2,....]*var_args_tuple):
    function statements
    return[expression]
```

```
def func(name,*fav_subjects):
    print("\n",name,"likes to read")
    for subject in fav_subjects:
        print(subject)
func("Ravi","Python","Java")
func("Vamshi","C","C++","JAVA",".NET")
func("Hari")
```

- In the function definition, we have two parameters- one is name and the other is var_length_argu fav_subjects.
- The fucntion is called 3 times with 3,5,1 parameters
- The first value is assigned to name and the other values are assigned to parameter fav_subjects
- Everyone can have any number of fav_subjects and some can have none.

Lambda or Anonymous Function

- In Python, anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called lambda functions
- Can take any number of arguments but, return only one value in the form of an expression
- Cannot be direct call to print as lambda requires an expression
- They have their own namespace and cannot access variables in either in local or global namespaces
- The lambda feature was added to Python due to the demand from LISP programmers.

- Lambda functions contain only a single line. Its syntax can be given as
- **Syntax:** **lambda arguments : expression**

lambda[arg1,[args2,[args3…..argsn]:expression

Ex:

```
#Function Definiton
total= lambda num1,num2:num1+num2
# Function Call
print("Value of total:",total(10,20))
print("Value of total:", total(100,200))
```

- **Difference between Normal Functions and Anonymous Function:**

  Have a look over two examples:

```
def square(x):
    return x*x
print("square of number is",square(10))
```

```
square=lambda a:a*a
print("Square of number is",square(10))
```

```
#Program that passes lambda function
#as an argument to a function
def func(f,n):
    print(f(n))
twice=lambda x:x*2
thrice=lambda x:x*3

func(twice,4)
func(thrice,3)
```

- You can define a lambda that receives no arguments but simply returns an expression

```
# Program that uses a lambda function to
#find the sum of first 10 natural numbers
x=lambda:sum(range(1,11))
# Invoke lambda expression that accepts
#no arguments but returns a value
print(x())
```

- You can call a lambda function from another function

```
add=lambda x,y:x+y
# lambda function calls another lambda func
mult_add=lambda x,y,z:x*add(y,z)
print(mult_add(5,10,15))
```

Recursive Function

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Every recursive solution has two major cases, which are as follows:
  - base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - recursive case, in which first the problem at hand is divided into simpler sub parts.
- Recursion utilized divide and conquer technique of problem solving.
- A method invoking itself is referred to as a Recursion
- Typically when a program employs recursion the function invokes itself with a smaller argument
- Computing factorial(5) involves computing factorial(4), computing factorial(4) involves computing factorial(3) and so on
- Try out with Factorial & GCD Problems

```
# Recursive Function
def fact(n):
    if(n==1 or n==0):
        return 1
    else:
        return n*fact(n-1)
n=int(input("Enter the value of n:"))
print("The Factorial of ", n,"is",fact(n))
```

Topics:

- **Functional Programming**
  - filter( ) function
  - map( )function
  - reduce( )function
- It decomposes a given problem into a set of functions. map( ), filter( ) and reduce( ) functions tools will work on all list items

filter( ) function

- The filter( ) function constructs a list from those elements of the list for which a function returns TRUE.
- Syntax: `filter(function,sequence)`
- The filter function returns a sequence that contains items from the sequence for which the function is TRUE
- If the sequence is a string, Unicode or atuple then the result will be the same type, otherwise it is allways list

```
#with out filter()
def check(x):
    if(x%2==0):
        return x
l=[1,2,3,4,5,6,7,8,9]
for x in l:
    print(x,end=' ')
print()

#with filter()
def check(x):
    if(x%2==0 or x%4==0):
        return 1
x=list(filter(check,range(1,20)))
print(x)
print(type(x))
```

map() function

- The map( ) function applies a particular function to every element of a list
- Syntax: **map(function, sequence)**
- After applying the specified function on the sequence, the map( ) function returns the modified list.
- The map() function calls function(item) for each item in the sequence and returns a list of the return values

```
#with map()
def check(x,y):
    return (x+y)
x=list(map(check,range(1,5),range(10,15)))
print(x)
print(type(x))
```

reduce( ) function

- The reduce( ) function returns a single value generated by calling the function on the first two items of the sequence, then on the result and the next item and so on
- Syntax: **reduce(function, sequence)**

```
#with reduce()
import functools
def check(x,y):
    return x+y
list1=[1,2,3,4,5]
print("Sum of values:")
print(functools.reduce(check,list1))
```

For Details Contact Me @ :

9247448766

ravikanth27787@gmail.com

python

Programming Language