Python Programming

Module –2
Functions

Overview of this Lecture

Functions
Introduction
built-in functions
User defined functions
Passing parameters
Return values

Recursive function

Lambda function

Functions

- Function is sequence of programming statements does some specific task orcomputation.
- Functions allows us to break the complex logic into several small independent modules (each module is a function). Advantages of using functions are:
  - Decompose the complex problem into smaller sub modules
  - Code Readability
  - Reusability.
- A good programming practice is to divide the large-scale program into several smaller units (functions) and repeatedly use these functions in the program.
- A function written once can be used for any number of times in a program.
- This is referred as reusability of code.

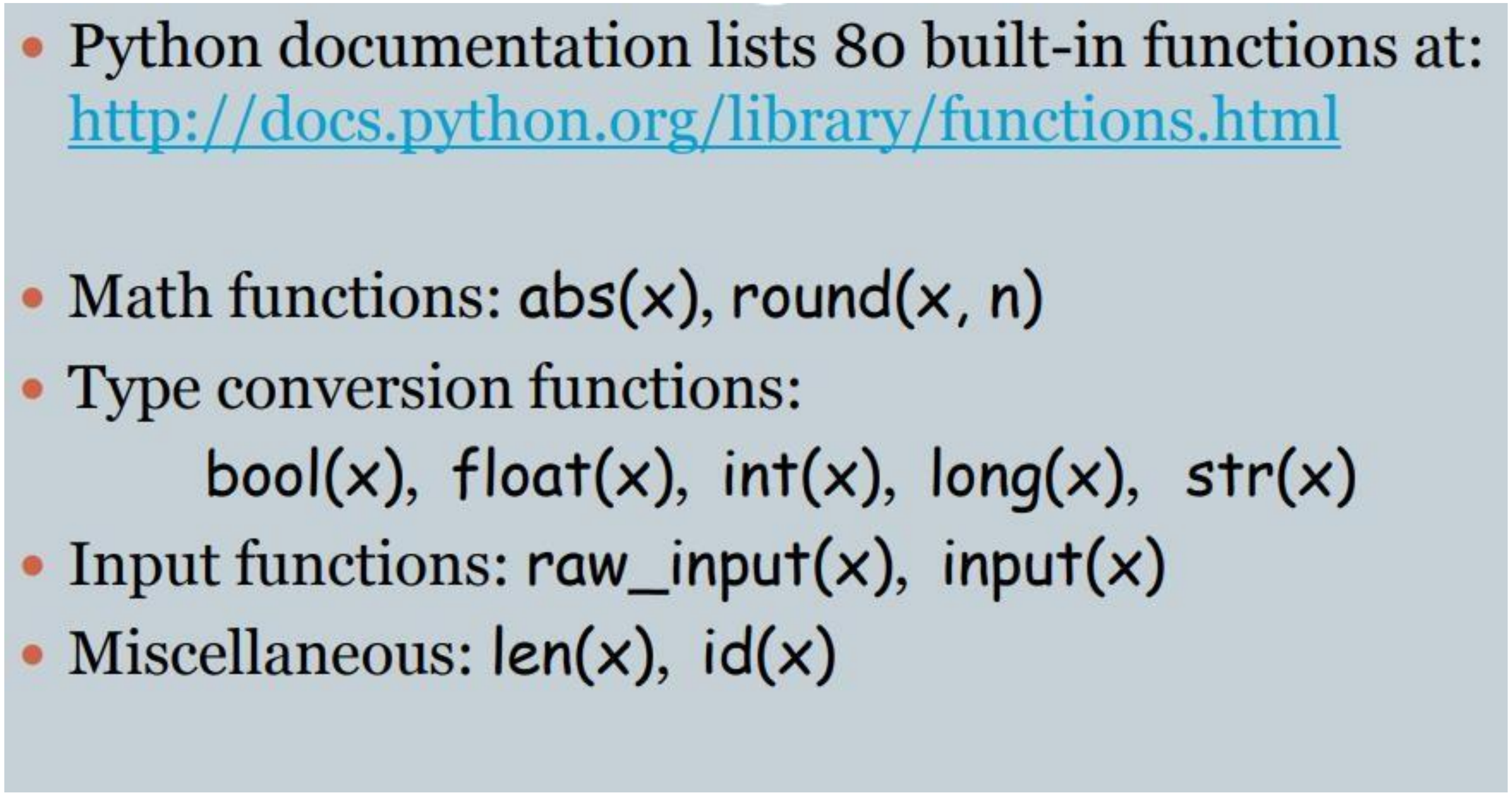
Types of Functions

- In Python, functions can be broadly classified into two types:
  - Predefined functions
  - User defined functions.

Predefined functions

- Python supports built-in functions whose functionality is predefined in Python.
- These predefined functions can be readily used in programs.
- For example, print() is a predefined function used for displaying contents specified in the print() on the screen.
- There are several built -in functions available in Python

Built in functions



- Python documentation lists 80 built-in functions at: http://docs.python.org/library/functions.html
- Math functions: abs(x), round(x, n)
- Type conversion functions:
  bool(x), float(x), int(x), long(x), str(x)
- Input functions: raw_input(x), input(x)
- Miscellaneous: len(x), id(x)

Predefined function	Description	Example	Output
abs ()	Returns the absolute value of a number	**abs(-9)**	9
ascii ()	Displays the printable version of the object	**ascii(‘~’)**	“‘~’”
bin ()	Converts an integer to binary string	**bin(10)**	'0b1010'
bool ()	Converts given value to true or false	**bool(0.99)**	True
chr()	Returns the character for the ascii value	**chr(97)**	‘a’
complex ()	Create a complex number	**complex(3.6)**	(3.6+0j)
divmod()	Takes two numbers and gives the quotient and remainder	**divmod(7,3)**	(2,1)
float()	Converts an integer into float point number	**float(2)**	2.0
help()	To get details about keyword,module,symbol or topic	**help(id)**	Details about id() will be displayed
hex ()	Returns the hexadecimal value for given number	**hex(16)**	'0x10'
id ()	Returns the object identity	**A=10** **id(a)**	499180448
input ()	Take the input from keyboard	**input()**	
int ()	Converts a value to integer	**int(‘8’)**	8
Len()	Return the length (the number of items) of an object	**len(“Python”)**	6
max ()	Find the maximum number	**max(12,1,3)**	12
min ()	Find the minimum number	**min(12,1,3)**	1
ord ()	Returns the integer value for Unicode character	**ord(‘A’)**	65
pow (x, y)	Returns x to the power of y	**pow(3,4)**	81
reversed ()	Reverses the content of iterable	**reversed([1,2,3])**	[3,2,1]
sorted ()	Prints the sorted version of iterable	**sorted('Python')**	['P', 'h', 'n', 'o', 't', 'y']
type ()	Find the type of the object	**type(7)**	<class 'int'>

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()`	

User defined Functions

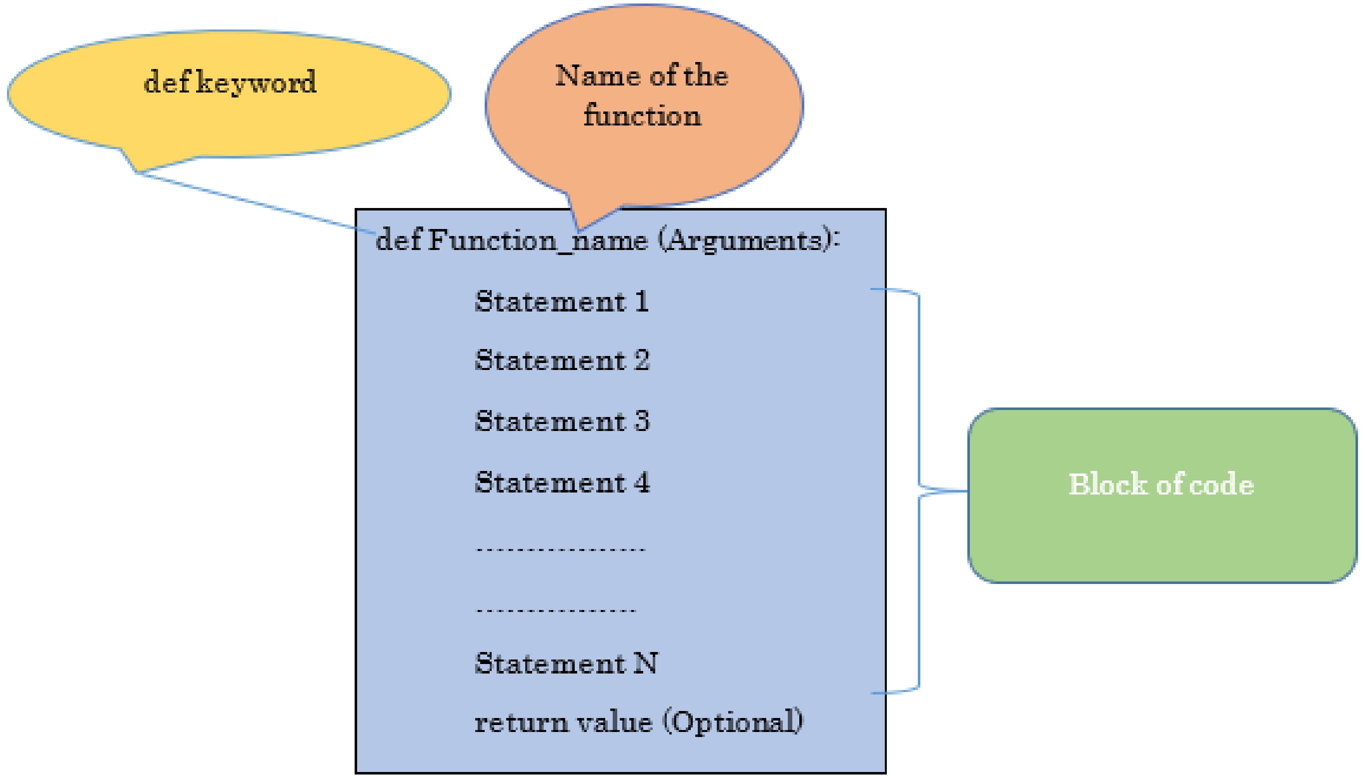
- If we want to write our own functions to perform a specific task, it is referred as user defined functions.

User defined function and syntax:

- A function in Python is an organized block of code to perform a specific task.
- Usually writing functions in a program has two parts:
  - Defining the function
  - Calling the function.

Defining the function

- In function definition, programmer will be stating the name of the function with /without arguments and block of code for performing the task.
- The syntax for function definition is given below:



```
def Function_name (Arguments):
        Statement 1
        Statement 2
        Statement 3
        Statement 4
        ----------------
        ----------------
        Statement N
        return value (Optional)
```

Syntax for Functiondefinition

- The function definition consists of **function header and function body**.
- The function header begins with def keyword followed by function name.
- Function may have zero or more arguments, where the arguments are the variables.
- Arguments can also be termed as parameters and these parameters are termed as **Formal parameters**.
- If there are more than one parameter, then the parameters are separated by commas.
- Function header ends with the colon **(:)**
- Function body includes statement (s) for performing specific task.
- Python interpreter recognizes the block of code(Function body) for the function through indentation.
- This block of code defines the action that the function must perform upon function call.

Calling the function

- Without the function call, function definition will not be executed and perform the task to be carried out.
- Hence, function call is mandatory to execute the function body.
- When the function is invoked with its name along with arguments, then it is said to be as function call.
- Arguments in function call is actual parameters, if the function definition has parameters, Python interpreter expects the parameters to be passed in function call.
- A function may or may not return value(s) upon execution.
- If the function returns value, it has to be returned by the function definition.

Write a function to display the type of the values passed to afunction

#Function Definition

```
def checkType(val):
	print(type(val))
```

#Function call

```
checkType(5)
```

Output:

```
<class 'int'>
```

Write a Simple function to square the number passed as argument.

Program

```
# Function with return value
# Function Definition
def square(x):
    sq=x*x
    return sq        #End of function body

num=int (input ("Enter the number: “))
#Function call
sqNum=square(num)
print ("Square of the number is:”, sqNum)
```

Output:

```
Enter the number: 5
Square of the number is: 25
```

Functions

Use of a function

Ex: Sum of numbers starting from 1to 25, 50 to 75 and 90 to 100

```
 def sum(x,y):
        s=0;
        for I in range(x, y+1):
                s=s+I
        print(‘ Sum of integers ‘, x,’ to ‘,y,’ is , s)
sum(1,25)
sum(50,75)
sum(90.100)
```

Functions

Write a program to find maximum of 2 numbers

```
def printMax(num1, num2):
    print("num1 =",num1)
    print("num2 =",num2)

    if num1 >num2:
        print( "The number ", num1, " is greater than ", num2)
    elif num2 >num1:
        print( "The number ", num2, " is greater than ", num1)
    else:
        print( "Both the number ", num1, " ,and" , num2, "are equal")

printMax(20,13) # function calling
```

```
num1 =20
num2 =13
The number 20 is greater than 13
```

Example Program

```
#To find sum_of digits of a number using function
 #function definition
def sum_digits(n):
    total=0
    while(n>0):     # loop continues until n>0
        digit=n%10   # to separate a last digit of a number
        total = total +digit    # sum of digits
        n=n//10
    return total
 num=int(input("Enter a number: "))   # input is stored in "num"
result= sum_digits(num) # function calling
print("The sum of digits of a number" , num, "is", result)
```

Passing arguments to functions:

- Arguments are passed by reference in Python
- Any change made to parameter passed by reference in the called function will reflect in the calling function based on whether data type of argument passed is mutable or immutable in Python
    - Mutable Data types include Lists, Sets, Dictionary
    - Immutable Data types include Number, Strings, Tuples

Pass arguments to functions: Immutable Data Type -Number

```
#Function Definition
def change(cust_id):
        cust_id += 1
        print("Customer Id in function definition: ", cust_id)return
# Function Invocation with arguments of immutable data type
cust_id = 100
print("Customer Id before function invocation: ", cust_id)
change(cust_id)
print("Customer Id after function invocation: ", cust_id)
```

```
Output:
Customer Id before function invocation: 100
Customer Id in function definition: 101
Customer Id after function invocation: 100
```

Observe that customerid remains unchanged even after function invocation

Pass arguments to functions: Mutable Data Type -List

```
#Function Definition
def change(list_cust_id):
#Assign new values inside the function
list_cust_id.append([10, 20, 30])
print("Customer Id in function definition: ", list_cust_id)return
# Function Invocation with arguments of immutable data type
list_cust_id = [100, 101, 102]
print("List of Customer Id before function invocation: ", list_cust_id)
change(list_cust_id)
print("Customer Id after function invocation: ", list_cust_id)
```

Output:

```
List of Customer Id before function invocation: [100, 101, 102]
List of Customer Id in function definition:[100, 101, 102, [10, 20, 30]]
List of Customer Id after function invocation: [100, 101, 102, [10, 20, 30]]
```

Function Arguments

- Function can be called by using the following types of formal arguments –
  - Required arguments/Positional arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Required arguments

- Arguments follow positional order
- No. of arguments and the order of arguments in the function call should be exactly same as that in function definition
- Parameters are assigned by default according to their position.

Example:

```
# Function Definition
def print_str(str1):
    print("This function prints the string passed as an argument")
    print(str1)
    return

# Function Invocation without required arguments
print_str()
```

Output:

```
TypeError: print_str() missing 1 required positional argument: 'str1'
```

Example

```
def display(name, age):
        print("Name =",name, "Age =",age)
display("Arun", 20)
display(40, "Sachin")
```

```
Name = Arun Age = 20
Name = 40 Age =  Sachin
```

Keyword arguments

- Programmer knows the parameter name used within the function then he/she can explicitly use the parameter name while calling the function.
- A programmer can pass a keyword argument to a function by using its corresponding parameter name rather than its position.
- when used in function call, the calling function identifies the argument by parameter name
- Allows you to place the arguments out of order
- Python Interpreter uses the keyword provided to match the values with parameters

Example

```
def display(name, age):
        print("Name =",name, "age =",age)
display(age=25, name="Arun")
```

```
Output:
Name =Arun age =25
```

Keyword arguments

Example:

```
# Function Definition
def customer_details (cust_id, cust_name):
    print("This function prints Customer details")
    print("Customer Id: ",cust_id)
    print("Customer Name: ",cust_name)
    return

# Function Invocation with Keyword arguments
customer_details(cust_name = "John", cust_id = 101)
```

Output:

```
This function prints Customer details
Customer Id:  101
Customer Name:  John
```

Observe the change in positional order of arguments

Default Arguments:

- Assumes a default value if the value is not specified for that argument in the function call

Example:

```
# Function Definition
def customer_details (cust_name, cust_age = 30):
    print("This function prints Customer details")
    print("Customer Name: ",cust_name)
    print("Customer Age: ",cust_age)
    return

# Function Invocation with Default arguments
customer_details(cust_age = 25, cust_name = "John")
customer_details(cust_name = "John")
```

Output:

```
This function prints Customer details
Customer Name:  John
Customer Age:  25
This function prints Customer details
Customer Name:  John
Customer Age:  30
```

```
def message(name, msg="Welcome to Python!!"):
        print(" Hello", name, msg)
message("Sachin")
```

```
Hello Sachin Welcome to Python!!
```

Variable-length arguments

- Used to execute functions with more arguments than specified during function definition
- unlike required and default arguments, variable arguments are not named while defining a function

Syntax:

```
def functionname([formal_args,] *var_args_tuple ):
    function_body

    return [expression]
```

- An asterisk '*' is placed before variable name to hold all non-keyword variable arguments
- *var_args_tuple is empty if no additional arguments are specified during function call

- **Variable-length arguments**

Example:

```
# Function Definition
def customer_details (cust_name, *var_tuple):
    print("This function prints Customer Names")
    print("Customer Name: ",cust_name)
    for var in var_tuple:
        print(var)
    return

# Function Invocation with Variable length arguments
customer_details("John", "Joy", "Jim", "Harry")
customer_details("Mary")
```

Output:

```
This function prints Customer Names
Customer Name:  John
Joy
Jim
Harry
This function prints Customer Names
Customer Name:  Mary
```

Invocate this function without arguments and observe the output

Scope of variable

- Determines accessibility of a variable at various portions of the program
- There are two basic scopes of variables in Python
  - Global variables
  - Local variables

Global vs. Local variables

Local

- Variables defined inside the function have local scope
- Can be accessed only inside the function in which it is defined

Global

- Variables defined outside the function have global scope
- Variables can be accessed throughout the program by all other functions aswell

Example

```
# Function definition is here
def sum( arg1, arg2 ):
    total = arg1 + arg2; # Here total is local variable.
    print("Inside the function local total : ", total)
    return ;

# Now you can call sum function
sum( 10, 20 );
print("Outside the function global total : ", total)
```

Output:

```
Inside the function local total :  30
Traceback (most recent call last):
  File "C:\Users\ADMIN\AppData\Local\Programs\Python\Python37-32\fun.py",
0, in <module>
    print("Outside the function global total : ", total)
NameError: name 'total' is not defined
```

Usage of Global Keyword

```
# Function definition is here
def sum( arg1, arg2 ):
    global total
    total = arg1 + arg2; # Here total is local variable.
    print("Inside the function local total : ", total)
    return;

# Now you can call sum function
sum( 10, 20 );
print("Outside the function global total : ", total)
```

Output:

```
 Inside the function local total :   30
 Outside the function global total :   30
```

Functions

The return statement

The return statement is used to return a value from the function.

Used to return from a function, ie. break out of the function

```
def minimum(a,b):
    if (a<b):
        return a
    elif b<a:
        return b
    else:
        return "Both the numbers are equal"
print(minimum(100,100))
```

```
Output:
Both the numbers are equal
```

Functions

Returning multiple values

It is possible to return multiple values in Python.

```
def arith_op(num1, num2):
    return num1+num2, num1-num2    # Return multiple values

print(arith_op(10,20))
```

```
Output:
(30, -10)
```

KUMARAGURU college of technology character is life

KCED Learning Augmented

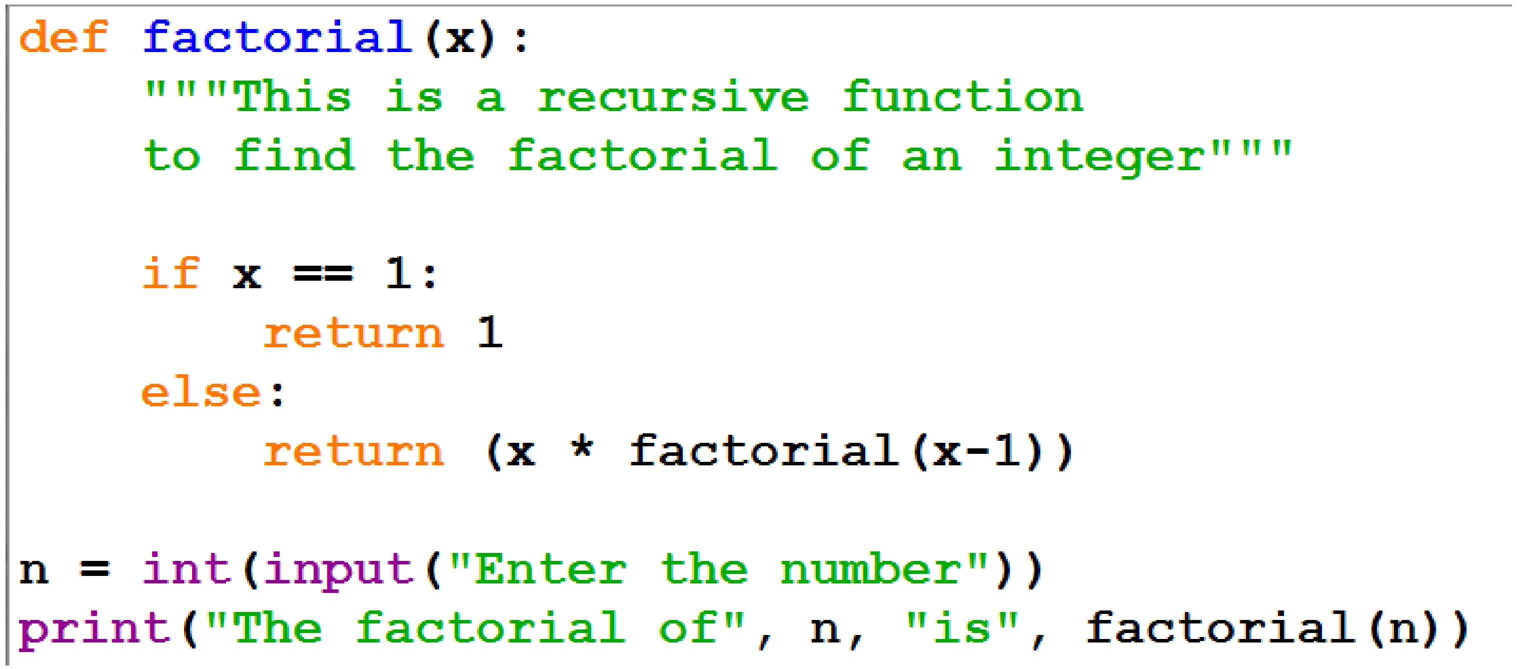
Recursion

- Recursive function is a function that calls itself
- Example:
  Factorial n! can be defined as n!=n*(n-1)!

```
4!  =4 *3!
    =4 *3 *2!
    =4 *3 *2 *1!
    =4 *3 *2 *1*0!
    =4 *3 *2 *1*1
    =24
```

LEARNING AUGMENTED

Example Program - Factorial



```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

n = int(input("Enter the number"))
print("The factorial of", n, "is", factorial(n))
```

Example Program - Fibonacci

```
def fibo(n):
    if (n==0):
        return 1
    elif (n==1):
        return 1
    else:
        return fibo(n-1)+fibo(n-2)

print('value of 4th fibonacci number is ',fibo(4))
```

Lambda Function

- Also known as anonymous functions
- Not bound to a name
- These functions are called anonymous because they are not declared in the standard manner by using the def keyword.
- It contains a single expression as a body and not a block of statements as a body
- Syntax:

```
name =lambda arguments :expression
```

Example Program

Functions

```
def fun(x):
    return x *x

print(fun(2))
```

Lambda Function

```
square =lambda x:x*x
print(square(2))
```

Output:

```
4
```

Lambda function- Key points

- Lambda functions have no name
- Lambda functions can take any no.of arguments
- Can return just one values in the form of an expression
- Does not have explicit return statement
- Can have only single expression