* **What are functions?**
  + Python function is a block of code that performs a specific and well defined task.
  + Two main advantages of functions are:
    - They help us divide our program into multiple tasks. For each task we can define a function that makes our code modular.
    - Functions provide a re-use mechanism. The same function can be called any number of times.
* **Types of Python Functions**
  + **Built-in Functions:** ex. len(), sorted(), min(), print(), etc.
  + **User-defined functions:**  ex. add(), isprime(), palindrome(), sum\_of\_digits(), etc.
* **Example of User-defined functions:**

# Function definition of add() . Note that the body of the function must be indented suitably.

def add():

a = int(input("Enter a number:"))

b = int(input("Enter another number:"))

c = a + b

print(a, "+", b , "=", c)

add() # Calling add function.

* A function can be called any number of times. (Re-use mechanism)
  + add() # First call to add()
  + add() # another call to the same function.
* When a function is called, control is transferred to the function, its statements are executed and control is returned to place from where the call originated.
* A function can be redefined. While calling the function its latest definition will be called.
* Function definitions can be nested. When we do so, the inner function can access the variables of outer function. The outer function has to be called for the inner function to execute.

def fun1():

print("Inside fun1...")

def fun2():

print("Inside fun2...")

print("Again back in fun1...")

fun2()

fun1() # We can call fun1()

print(type(fun1)) # Class ‘function’

fun2() # We can't call inner function from here.

The above type of functions can be written when we wish to protect the function. Means, we don’t want to call an inner function from outside.

* **Communication with Functions:**
  + Through parameters/arguments passed to it and the value(s) returned from it.

def sum\_of\_3\_numbers(a, b, c):

return a + b + c

ans = sum\_of\_3\_numbers(10, 20, 30)

x, y, z = 1, 2, 3

ans = sum\_of\_3\_numbers(x, y, z)

* + **return** statement returns control and value(s) from a function. **return** without an expression returns **None**.
  + A function that reaches end of execution without a return statement will always return **None.**
  + We can return multiple values from a function by putting them into a list/tuple/set/dictionary and then return it or even we can return multiple values directly.

def order3(a,b,c):

l = max (a,b,c)

s = min (a, b, c)

m = a + b + c - l - s

return s,m,l

print(order3(5, 10, 7)) 🡨 Output: ( 5, 7, 10 ) 🡨 Returns tuple.

s, m, l = order3(10, 7, 5) 🡨 Returns 3 values.

print(s, m, l) 🡨 Output: 5 7 10

* + Python always call functions by value, unlike in C by reference also.
  + A function can return different types of values through different return statements.
* **Types of Arguments**
  + Arguments in a Python function can be of 4 types:
    - Positional arguments (also known as required arguments)
    - Keyword arguments
    - Variable-length (also known as optional arguments) positional arguments
    - Variable-length Keyword arguments
  + **Positional arguments** must be passed in correct positional order. e.g. if a function expects an int, float and string to be passed to it, then while calling the function the arguments must be passed in the same order. While passing positional arguments, number of arguments passed must match with number of arguments received.
  + **Keyword arguments** can be passed out of order. Python interpreter uses keywords (variable names) to match the values passed with the arguments used in the function definition.

def calc\_interest(p, r, n):

print ("interest = ", p \* r \* n / 100)

calc\_interest(10000, 8, 3)

calc\_interest(r = 8, n = 3, p = 10000)

calc\_interest(n = 3, p = 10000, r = 8)

#calc\_interest(**a = 10000**, n = 3, r = 8) <-- Will generate an error as variable **a** is not a keyword name.

* + Sometimes number of positional arguments to be passed to a function is not certain. In such cases, variable-length positional arguments can be received using \*args.

def add\_many(\*args):

s = 0

for v in args:

if type(v) == int or type(v) == float:

s = s + v

return s

print(add\_many(10, 20, 30, 40, 50)) # 150

print(add\_many(10, 20, 30, 40, 50, 60.0)) # 210.0

print(add\_many(10, 20, 30, 40, 50, 'abc')) # 150 ignoring ‘abc’.

* + Sometimes number of keyword arguments to be passed to a function is not certain. In such cases, variable-length keyword arguments can be received using **\*\*args.**

def print\_dict(\*\*kwargs):

print()

for k, v in kwargs.items():

print(k, v, end = '\t')

print\_dict(a = 10)

print\_dict(a = 10, b = 20)

dict = { 'a' : 10, 'my\_name' : 'Darshit' }

print\_dict(\*\*dict)

#print\_dict(dict) <-- Will give error as dereferencing is not used.

* + We can use any variable name in place of **args** or **kwargs**. However, we can’t use more than one **args** and **kwargs**.
* **Default Arguments**
  + We can assign a default value to the function arguments. Default value will be used if we don’t pass the value for that argument during the call.
  + While defining a function, default arguments must follow non-default arguments.

def func(a, b, op = 'add'):

if op == 'add':

return a + b

elif op == 'sub':

return a - b

elif op == 'mul':

return a \* b

else:

return a / b

print(func(20, 10))

print(func(20, 10, 'sub'))

print(func(20, 10, 'mul'))

print(func(20, 10, 'd'))

* **Unpacking Arguments**
  + Suppose a function is expecting positional arguments and the arguments to be passed are in a list, tuple or set. In such a case,we need to unpack the list/tuple/set using \* operator before passing it to the function.

def sums(a, b, c):

return a + b + c

l = [ v for v in range(3)]

s = { v \* v for v in range(3) }

t = ( v \* v \* v for v in range(3) )

print (l, s, t)

print(sums(\*l))

print(sums(\*s))

print(sums(\*t))

* + Suppose a function is expecting keyword arguments and the arguments to be passed are in a dictionary, we need to unpack dictionary using \*\* operator before passing it to the function.

def print\_items(name = 'Darshit', bg = 'O+ve'):

print(name, bg)

d = { 'name' : 'Ragi', 'bg' : 'A+ve' }

print\_items(\*d) 🡨 will pass keys only. output: name bg

print\_items(\*\*d) 🡨 will pass values. output: Ragi A+ve

d = dict()

print\_items(\*\*d) <- Dict is empty. Output: Darshit O+ve

* **Recursive Functions**
  + Function that calls itself.
  + Useful, if a problem can be solved by breaking it down into sub-problems or when a problem requires unknown number of loops.

**Examples of recursive functions**

**# This function generates nth term of fibnacci series recursively.**

def recfibo(n):

if n == 1 or n == 2:

return 1

else:

return recfibo(n-1) + recfibo(n - 2)

**# The following function finds GCD of given two numbers recursively.**

def recgcd(n, d):

if n % d == 0:

return d

else:

return recgcd(d, n % d)

**# The following function returns factorial of a given no.**

def recfact(n):

if n == 0:

return 1

else:

return n \* recfact(n - 1)

**# The following function returns Sum of Digits for a given no. recursively.**

def recsod(n):

if n == 0:

return 0

else:

return n % 10 + recsod(n//10)

# Call the recfibo ( ) for finding the nth term of fibonacci series.

for x in range(1,11):

print (x, recfibo(x))

# Calling recgcd function to print GCD of two numbers.

n = int(input("Enter a number.:"))

r = int(input("Enter another number.:"))

print ("GCD of ",n,"and",r,"is",recgcd(n,r))

# Calling recfact function to get factorial value.

for x in range(11):

print (f'{x:>2} != {recfact(x):>10}')

# print sum of digits for a given number.

n = int(input("Enter a number:"))

print("Sum of digits of" , n , "is", recsod(n))

s = 0

for digit in str(n):

s = s + int(digit)

print (s)

* **Write following programs considering functions or recursive functions in mind:**

1. Write a program that defines a function count\_lower\_upper() that accepts a string and calculates the number of uppercase and lowercase alphabets in it. It should return these values as a dictionary. Call this function for some sample string.
2. Write a program that defines a function compute() that calculates the value of n + nn + nnn + nnnn, where n is digit received by the function. test the function for digits 4 to 7.
3. Write a program that defines a function create\_array() to create and return a 3D array whose dimensions are passed to the function. Also initialize each element of this aray to a value passed to the function. e.g. create\_array(3,4,5,n) where first three arguments are 3D array dimensions and 4th value is for initialing each value of the 3D array.
4. Write a program that defines a function sum\_avg() to accept marks of five subjects and calculates total and average. It should return directly both values.
5. Pangram is a sentence that uses every letter of the alphabet. Write a program to check whether a given string is pangram or not, through a user-defined function ispangram(). Test the function with “The quick brown fox jumps over the lazy dog” or “Crazy Fredrick bought many very exquisite opal jewels”. Hint: use set() to convert the string into a set of characters present in the string and use <= to check whether alphaset is a subset of the given string.
6. Write a function to create and return a list containing tuples of the form (x,x2,x3) for all x between 1 and given ending value (both inclusive).
7. A palindrome is a word or phrase that reads the same in both directions. Write a program that defines a function ispalindrome() which checks whether a given string is a palindrome or not. Ignore spaces and case mismatch while checking for palindrome.
8. Write a program that defines a function convert() that receives a string containing a sequence of whitespace separated words and returns a string after removing all duplicate words and sorting them alphanumerically. Hint: use set(), list () , sorted(), join().
9. Write a program that defines a function count\_alpha\_digits() that accepts a string and calculates the number of alphabets and digits in it. It should return these values as a dictionary.
10. Write a program that defines a function called frequency() which computes the frequency of words present in a string passed to it. The frequencies should be returned in sorted order of words in the string.
11. Write a function create\_list() that creates and returns a list which is an intersection of two lists passed to it.

* **Write following programs considering functions or recursive functions in mind:**
  1. If a positive integer is entered through the keyword, write a recursive function to obtain the prime factors of the number.
  2. A positive integer is entered through the keyboard. Write a function to find its binary equivalent of this number.
  3. A string is entered through the keyboard. Write a recursive function that counts the number of vowels in this string.
  4. Write a recursive function that reverses the list of numbers that it receives.
  5. Calculate ab where a and b received through the keyword using recursion.
  6. A list contains some negative and some positive values. Write a recursive function that sanitizes the list by replacing all negative numbers with 0.
  7. Write a recursive function to obtain average of all numbers present in a given list.
  8. Write a recursive function to obtain length of a given string.