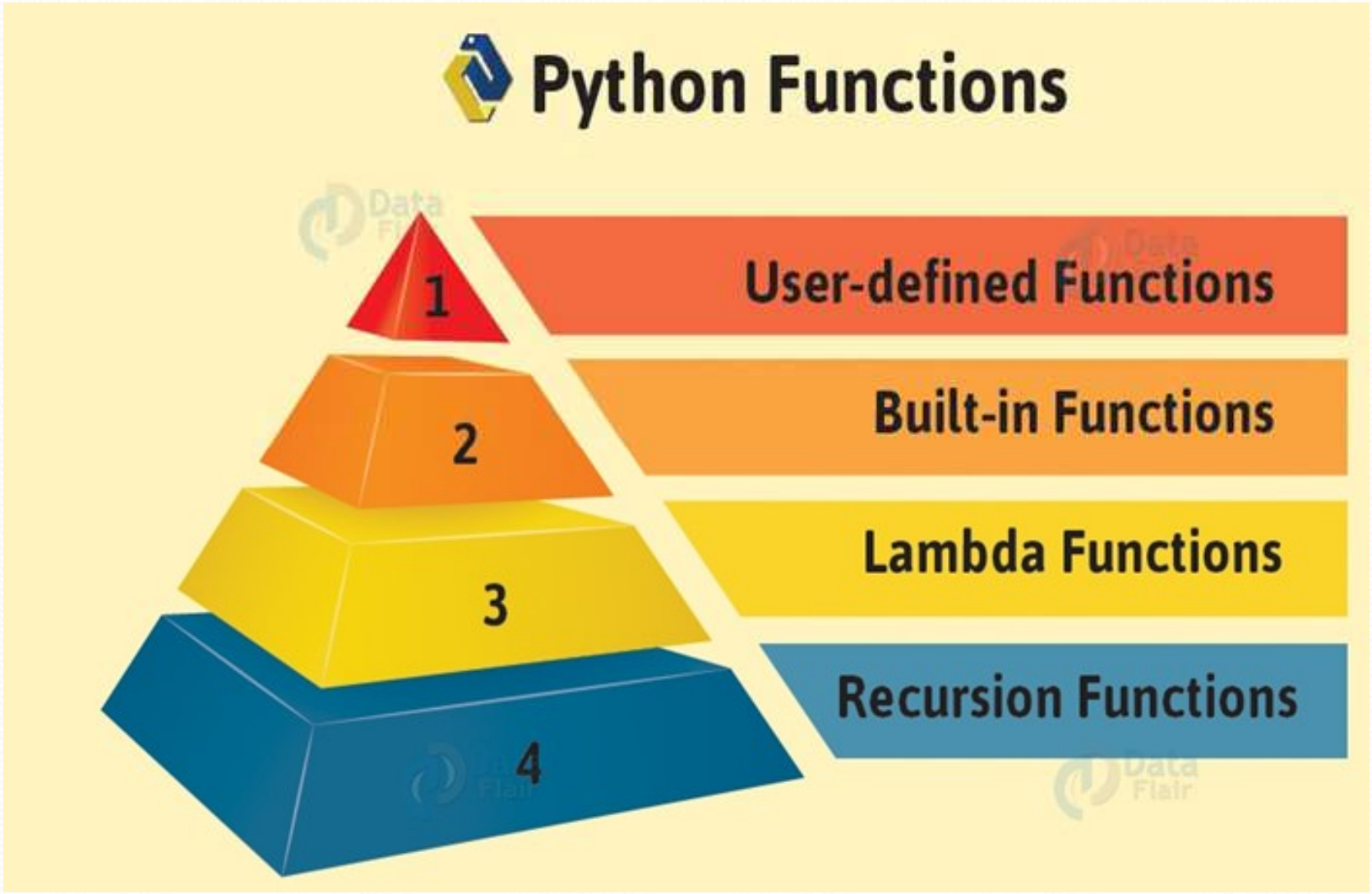

Python Functions
1
User-defined Functions
2
Built-in Functions
3
Lambda Functions
4
Recursion Functions

What is function?

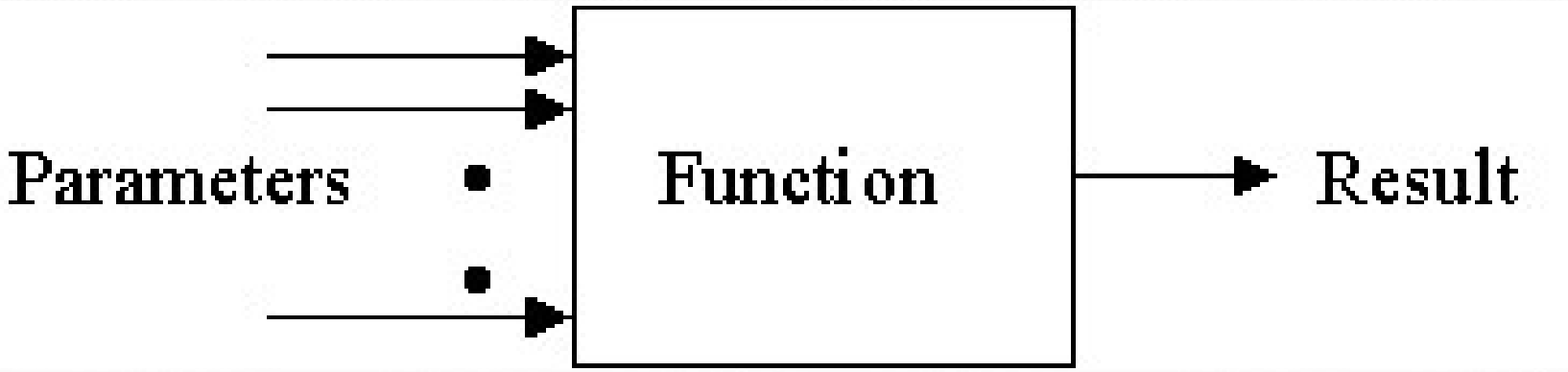
"Set of instructions to perform a specific task"

Ex: 1. To find area of circle

2. To add two numbers

3. To find simple interest

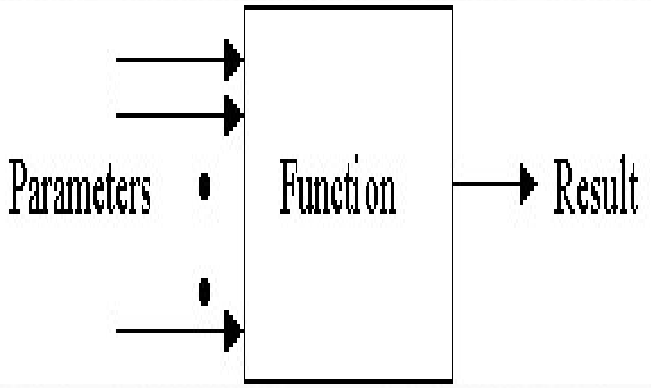
4. Sorting numbers



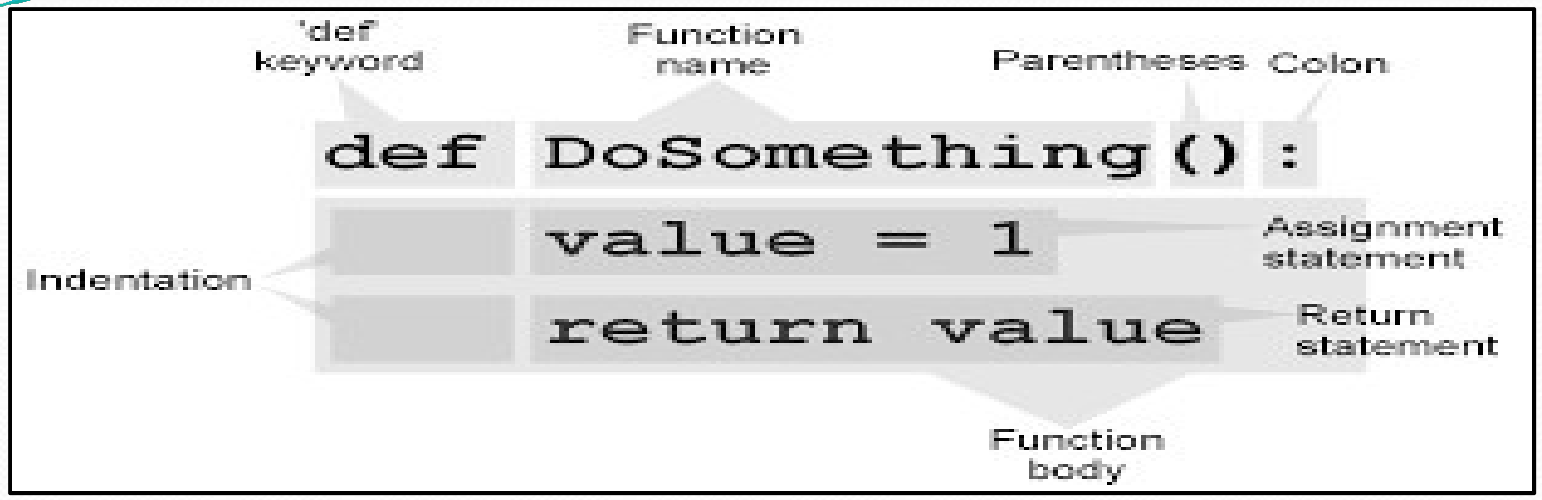
Functions

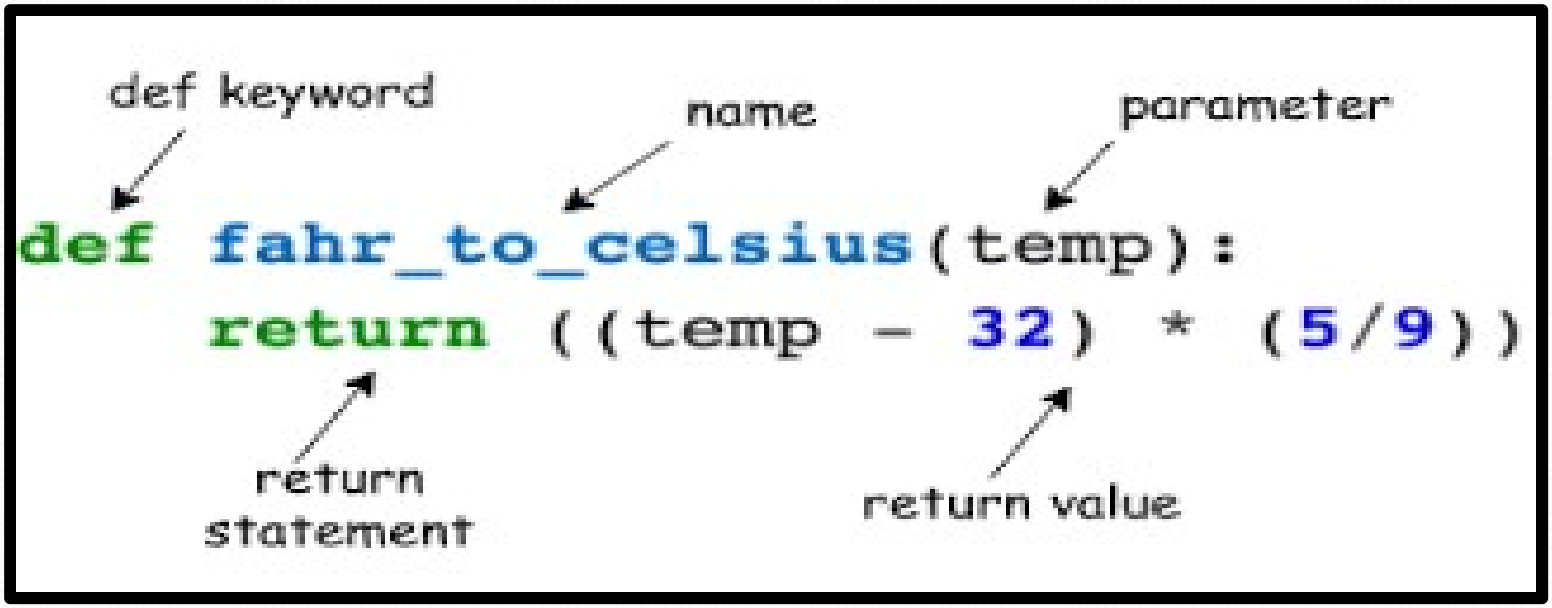
A complex problem is easier to solve by dividing it into several smaller parts, each of which can be called as function

```
def fun_name(param list):

        // statements
        //return statement
```

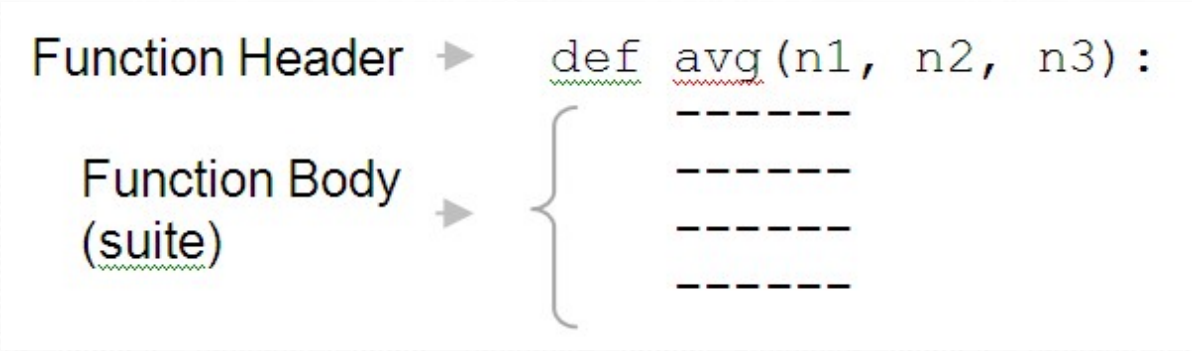


- A **routine** is a named group of instructions performing some task. A routine can be **invoked** (called) as many times as needed in a given program.


'def' keyword
Function name
Parentheses
Colon
def DoSomething():
value = 1
return value
Indentation
Assignment statement
Return statement
Function body


def keyword
name
parameter
def fahr_to_celsius(temp):
return ((temp - 32) * (5/9))
return statement
return value

Defining Functions



- A **function header** starts with the keyword **def,** followed by an identifier (`avg`), which is the function's name.
- The function name is followed by a comma-separated (possibly empty) list of identifiers (`n1, n2, n3`) called **formal parameters**, or simply "parameters." Following the parameter list is a colon (:).
- Following the function header is the **function body**, a suite (program block) containing the function's instructions. As with all suites, the statements must be indented at the same level, relative to the function header.

The number of items in a parameter list indicates the number of values that must be passed to the function, called **actual arguments** (or simply "arguments"), such as variables `num1`, `num2`, and `num3` below.

```
>>> num1 = 10
>>> num2 = 25
>>> num3 = 16

>>> avg(num1,num2,num3)
```

Functions are generally defined at the top of a program. However, every function must be defined before it is called.

Functions

Why functions

Program that we study from textbooks to learn a language are very small when compared to real world problems, if the program is very big, there are some disadvantages.

- Difficulty to understand the big program without modules
- It is very difficulty for the programmer to write large programs
- It is difficulty to identify the logical errors and to debug
- Large programs are more prone to errors

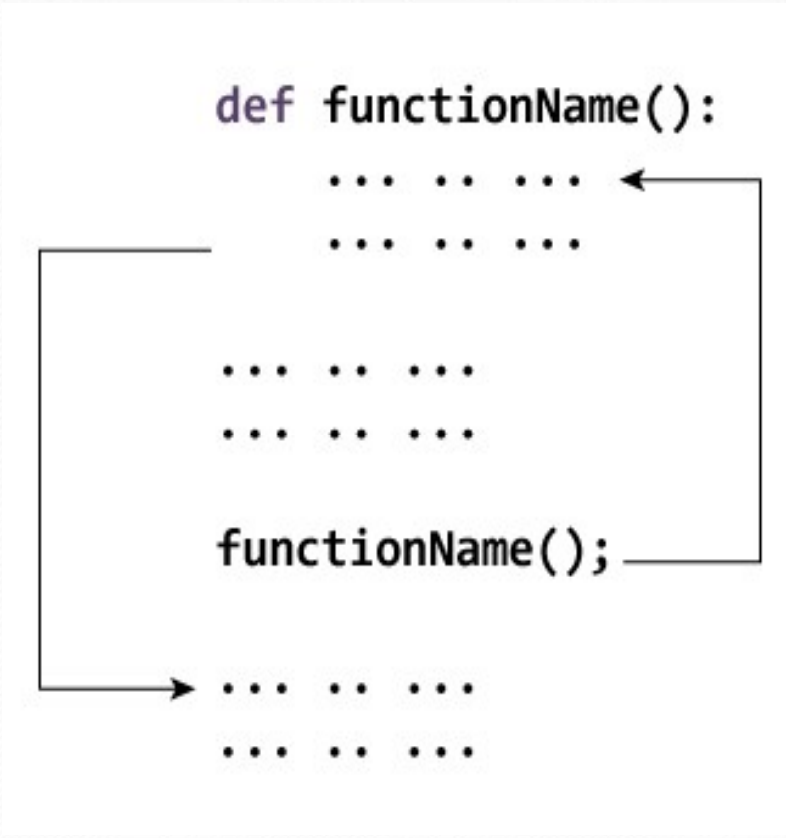
These disadvantages can be overcome using functions.

Program to add two numbers using functions

```
def add(x,y):
    z=x+y
    print(z)


a=input("enter first number ")
b=input("enter second number ")
add(int(a),int(20))
```

How functions work

```
def add(x,y):
        res=x+y
        print(res)
def prod(x,y):
        print(x*y)

def div(x,y):
        print(x/y)

a=10;b=20
add(a,b)  #function call

prod(a,b) #function call

div(a,b) #function call
```



Advantages of functions

Reusability: code can be reused number of times.

Less space: functions reduce the length of the program and thereby program takes less space.

Debugging is easy.

Program readability increases.

Program becomes more understandable.

Complex Program can be divided into modules.

Types of functions

- Built-in/Library functions
  - Ex: int(),float(),round(),sum()……
- User defined functions
  - Ex: Add(), Area(a,b)
- Lambda function
- Recursive functions

Value returning
Non value returning

Value returning and no value returning

```
#Adding 2 numbers
def add(x,y):
    z=x+y
    return z

a=10;b=20
res=add(a,b)  #function call
print(res)
```

Return z return value

Collect the return value in variable

```
#Adding 2 numbers
def add(x,y):
    print(x+y)

a=10;b=20
add(a,b)  #function call
```

No return value

function definition and function name: internals

```
# function definition and function name: internals
# foo : is a function; therefore callable
def foo() :
      print("I am foo")

print("one")
foo # no function call
print("two")
print(foo)   # <function foo at 0x________>

bar = foo # bar also becomes callable
print(bar)   # <function foo at 0x________>

# both give the same output
foo()
bar()

# remove foo
del foo
bar() # still works!
```

Return statement

There are three forms of return statement, they are

- Simple return
- return with value or return with exp
- multiple return statements

Simple return statement

Example:

```
def greet(name):
        print("Hi",name)
        return  //simple return statement, that returns the control
```

return statement with value or expression

Example:

```
Def Add(x,y) :

    z=x+y;                  // statements
    reurn z;        //return statement with value of z
```

Multiple return statements: more than one return statements in a function

Example:

```
def Max(x,y):

    if(x>y):
        return x;
    else:
        return y;
```

Mulitple values can be returned in python
return x,y
It makes a tuple and returns
You can also make a set,dict ,list,then return

Returning multiple values as a container

```
def calci(x,y):
    return x+y,x*y,x/y,x-y


a,b,c,d=calci(10,20)
print(a,b,c,d)
```

Actual arguments and Formal parameters

Formal parameters: parameters present in the function definition

Ex:

```
def add(x,y): #  x and y are formal parameters
    z=x+y
    return z
```

Actual parameters: parameters present in the function call

Ex:

```
a=3;b=4
sum=add(a,b); //  a and b are actual arguments/parameters
```

Formal parameters and actual parameters names can be same

The number of actual arguments are equal to number of formal parameters

Difference between actual and formal parameters

Actual parameters	Formal(dummy) parameters
Present in function call **EX:** **Sum=Max(a,b)// actual**	Present in function definition **Def Max(x,y) //formal** **print(max(x,y))**
Actual parameters can be constants, variables or exp Sum=Add(10,20) Sum=Add(a,b); Sum=Add(a+4,b);//expressions	Formal parameters should only be variables **def Add(x, y):** **print(x+y)**
They send values (to formal parameters)	They receive values (from actual parameters)

Scope of variables in functions

```
def add(x,y):
    z=x+y
    return z
```

x,y ,z are local to add Local variables

```
a=10;b=20
res=add(a,b)  #function call
print(res)
```

a ,b are global variables

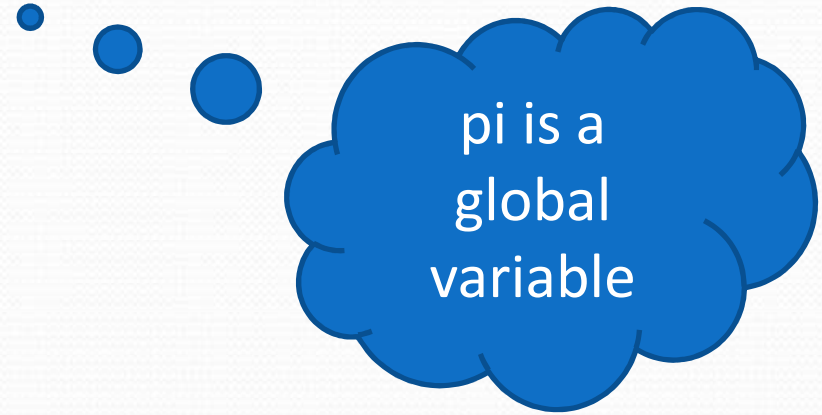
Global variables:

Global variables: We talk about two terms in programming – life and scope. Life of a variable is about existence of the variable – the variable has a location and therefore some value. The variable loses its life when the reference count becomes 0. The variable has scope if it can be seen – is visible – in the current suite.

We talk about local and global symbols. All names created outside of functions are global. All names created within a function by default are local to those functions.

Global variables

```
pi=3.14  #global
def area(r):
    res=pi*r*r   #res is local
    print(res)
def perimeter(r):
    res=2*pi*r   #res is local
    print(res)
```




```
a=1  #global
area(a)
perimeter(a)
```

The use of global variables is generally considered to be bad programming style.

Global variables

```
pi=3.14  # pi is global

def area(r):
    global pi
    pi=100    #value of global changes
    res=pi*r*r   #res is local
    print(res)
def perimeter(r):
    res=2*pi*r   #res is local
    print(res)

a=1  #a is also global
area(a)
perimeter(a)
```

Inside the function

- Cannot directly modify a global variable
- To modify a global variable, we should declare that it is global
- To access the global variable, for its value, nothing is required

```
pi=3.14  # pi is global
def area(r):
    res=pi*r*r
    print(res)
def perimeter(r):
    global pi
    pi=100
    res=2*pi*r
    print(res)
def disp():
    print(pi)


a=1
area(a)
perimeter(a)
disp()
```

Output

3.14

200

100

Global variables

```
k=30
def increment():
    global k
    k=k+100

print("before",k)
increment()
print("after",k)
k=4444
print(k)
```

```
def increment(k):
    k=k+100

k=30
print("before",k)
increment(k)
print("after",k)
30
30
```

Global var

```
k=30
def increment():
    k=k+100


print("before",k)
increment()
print("after",k)
```


```
k=30
def increment():
    global k    #to modify a global
    k=k+100     #modifying here
```

```
k=30
def increment(k):
    k=k+100
    print(“in the fun”,k)  //k is local variable, k is available within increment

print("before",k)
increment(k)
print("after",k)
```

```
before 30
In the fun 130
after 30
```

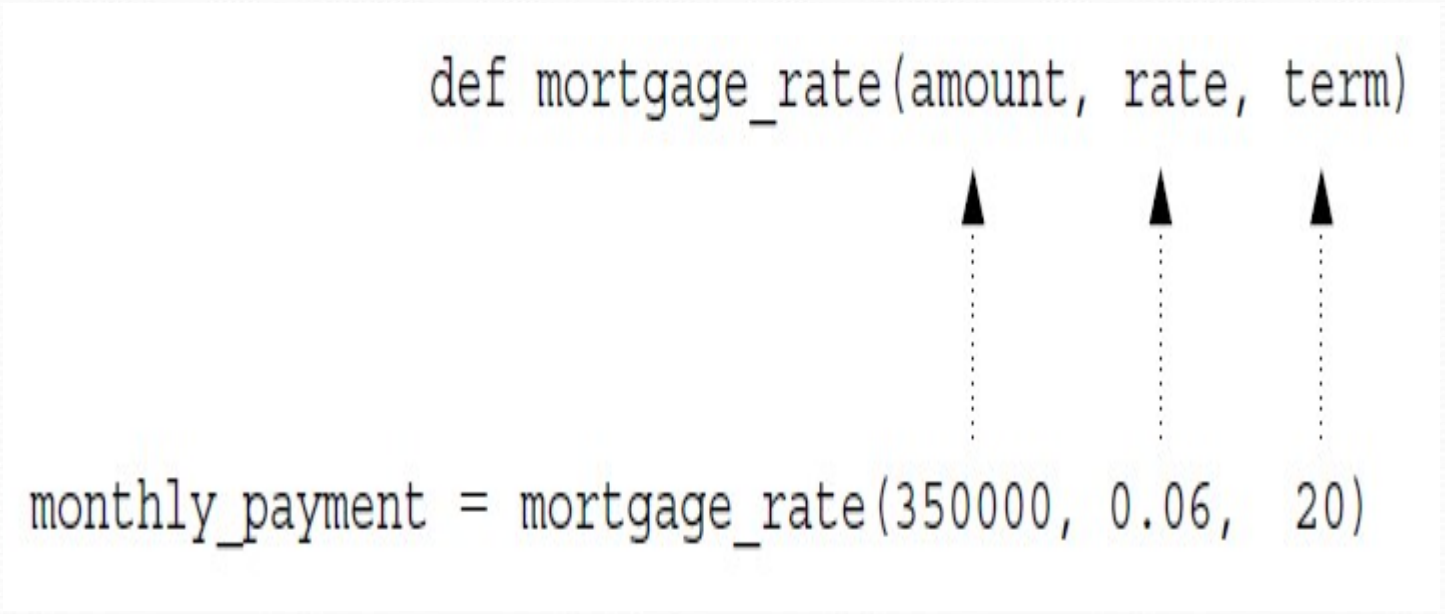
Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments or positional arguments
- Keyword arguments
- Default arguments(**Optional Parameters)**
- Variable-length arguments (Arbitrary Arguments)
- Key-value pair

Positional Arguments in Python

The functions we have looked at so far were called with a fixed number of *positional arguments*. A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list,



Positional arguments

```
def greet(name,msg):
    print("Mr.",name ,",", msg)


greet('amar','goood morning')
greet('goood morning','amar‘)
```


```
def wd(balance,amount):
      balance=balance-amount
      print("present bal is",balance)

pb=5000
a=int(input("ente the amount to withdraw"))
wd(a,pb)      //order matters, incorrect order
```

Keyword Arguments in Python

Python provides the option of calling any function by the use of **keyword arguments**. A keyword argument is an argument that is specified by parameter name, rather than as a positional argument.

```
def mortgage_rate(amount, rate, term)
```

```
monthly_payment = mortgage_rate(rate=0.06, term=20, amount=350000)
```

This can be a useful way of calling a function if it is easier to remember the parameter names than it is to remember their order.

Required arguments and Keyword arguments

```
def greet(name,msg):

        print("Hello",name , msg)

greet("Kumar","Good morning!")
greet("Kumar") #error ,  Required 2 arguments, 1 given

# 2 keyword arguments
greet(name = "Amar",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Amar")

# 1 positional, 1 keyword argument
greet(“Amar",msg = "How do you do?")
```

Default arguments(**Optional Parameters)**

```
def Greet(name, msg = "Good morning!"):
        print("Hello",name ,msg)
    #This function greets to  the person with the provided message.
    #If message is not provided, it defaults to "Good  morning!“

Greet("akbar")
Greet("anthony","How do you do?")
```

```
def Greet(name="Hi", msg = "Good morning!"):
    print(name,msg)

Greet()
```

Variable-length arguments (Arbitrary Arguments)

```
def Greet(*names):
    #This function greets all  the person in the names tuple.

    # names is a tuple with arguments
    for name in names:
        print("Hello",name)

Greet("Amar","Akbar","Anthony")
```

Key-value pair arguments

```
def pair(**d):
    print(d)


#key-value pair
pair(name="Amar",addr="Bangalore",phno="123345")
```

```
# key value pairs
def foo(**kw):
        print(kw, type(kw))
```

```
foo(king = 'dasharatha', wives = ['kousalya', 'sumitra', 'kaikeyi'],sons = ['rama', 'lakshmana'])
```

Output:

```
{'king': 'dasharatha', 'wives': ['kousalya', 'sumitra', 'kaikeyi'], 'sons': ['rama', 'lakshmana']}
<class 'dict'>
```

Passing mutable and immutable….

Int ,Float, Bool

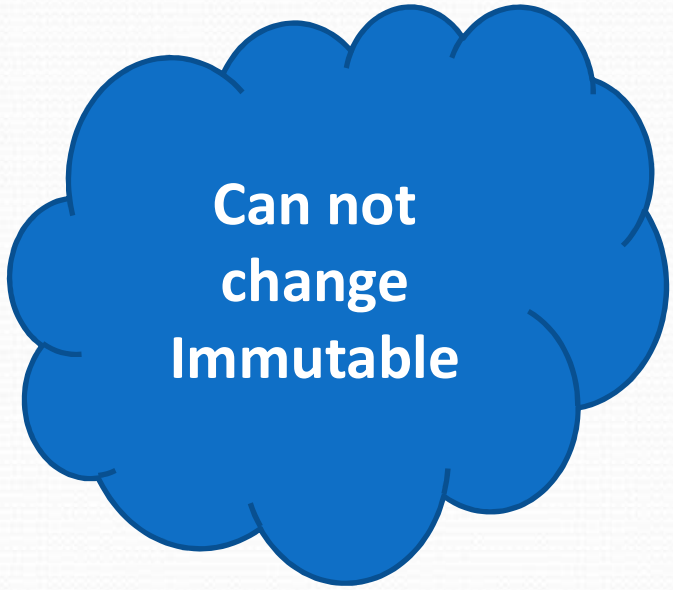
Str,Tuple

List, Set ,dict

Global

Write a program to swap two numbers using functions

```
def swap(x,y):
    temp=x
    x=y
    y=temp

#swap 2 nums using functions
a=10;b=20
print(a,b)
swap(a,b)
print(a,b)
```



Modifying the value of an integer variable using function

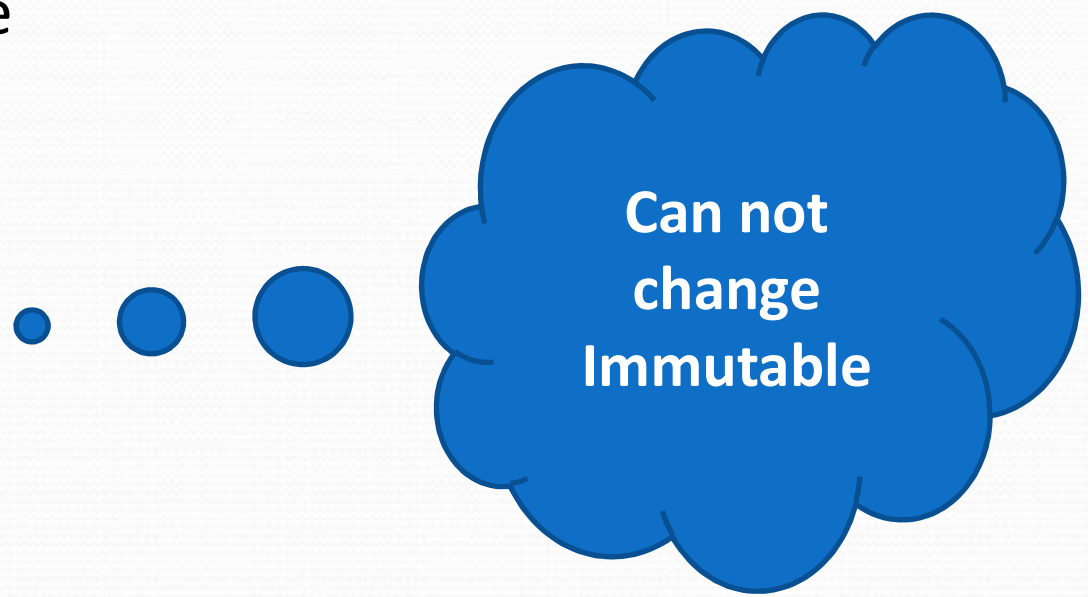
```
def increment(k):
    k=k+100
```

A dummy copy of the k is created, it changes in dummy, not the actual

```
k=30
print(k)
increment(k)
print(k)
```

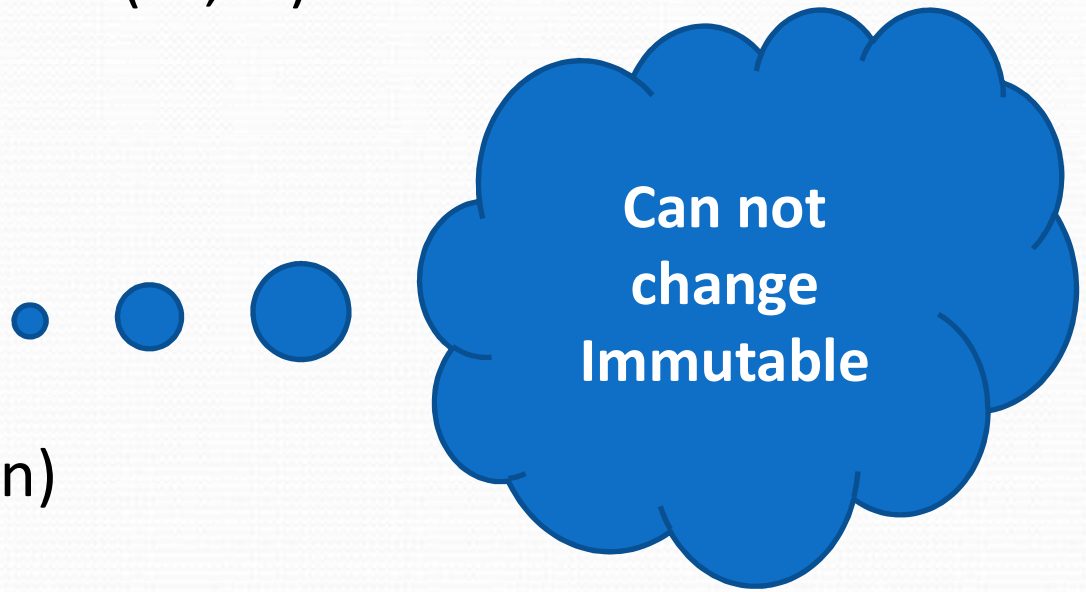
Modify the Boolean variable content using function

```
def change(devil):
    devil=False


devil=True
print(devil)
change(devil)
print(devil)
```



Modify the string content using function

```
def convert(person):
    person.replace('S','K')

person="SRK"
print(person)
convert(person)
print(person)
```



Modify(Extend) the list content using function

```
def change(list):

    list.extend([13,21,34])

fib = [0,1,1,2,3,5,8]
print("before",fib)
change(fib)
print ("after",fib)
```

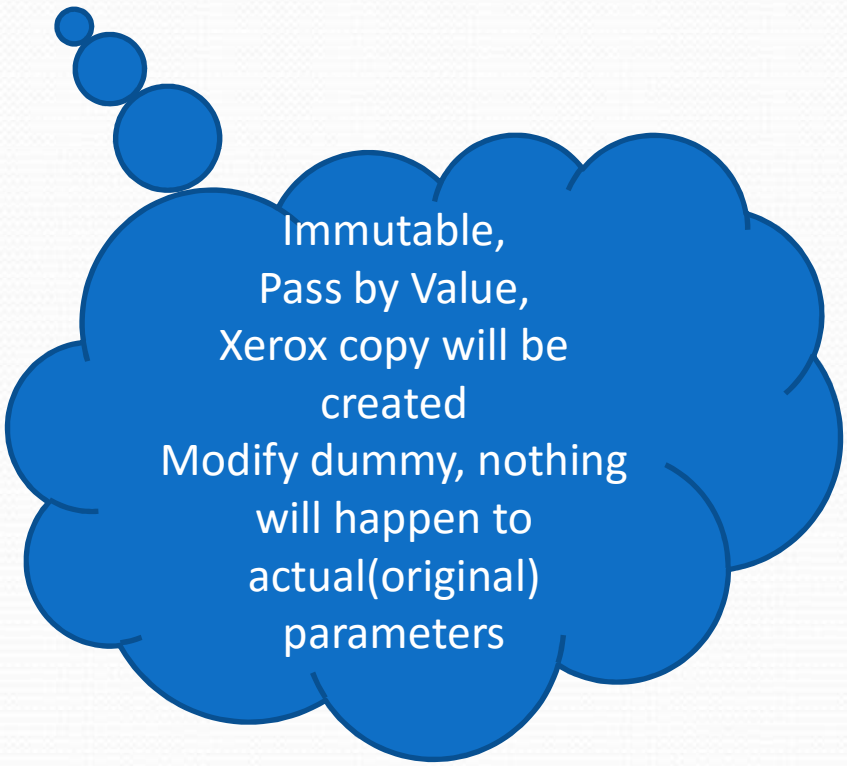
Modify the dict content using function

```
def disp(d):

    d.update({'b':'ball'})


d={'a':'apple'}
print("before",d)
disp(d)
print ("after",d)
```

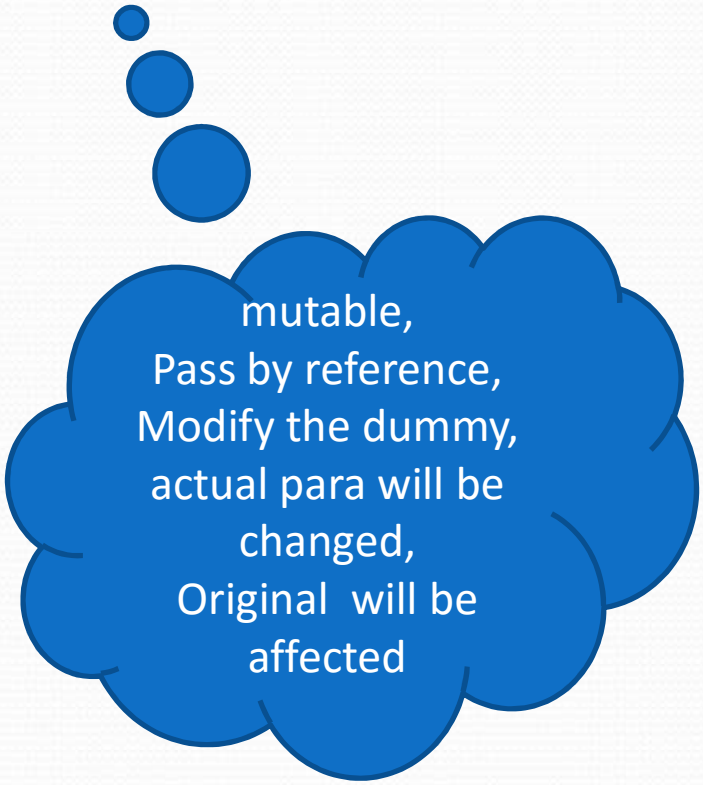
Passing immutable and mutable

Int ,Float, Bool

Str,Tuple



List, Set ,dict



Functions in Python

```
def add(value1, value2):
    return value1 + value2

result = add(3, 5)
print(result)
# Output: 8
```

```
def add(value1, value2):
    result = value1 + value2

add(2, 4)
print(result)
```

```
def add(value1,value2):
    global result
    result = value1 + value2

add(3,5)
print(result)
```

Types of functions

- Built-in/Library functions
  - Ex: int(),float(),round(),sum()……
- User defined functions
  - Ex: Add(), Area(a,b)
- Lambda function
- Recursive functions

Value returning
Non value returning

lambda function

lambda operator or lambda function is used for creating small, one-time and anonymous function objects in Python.

Basic syntax:

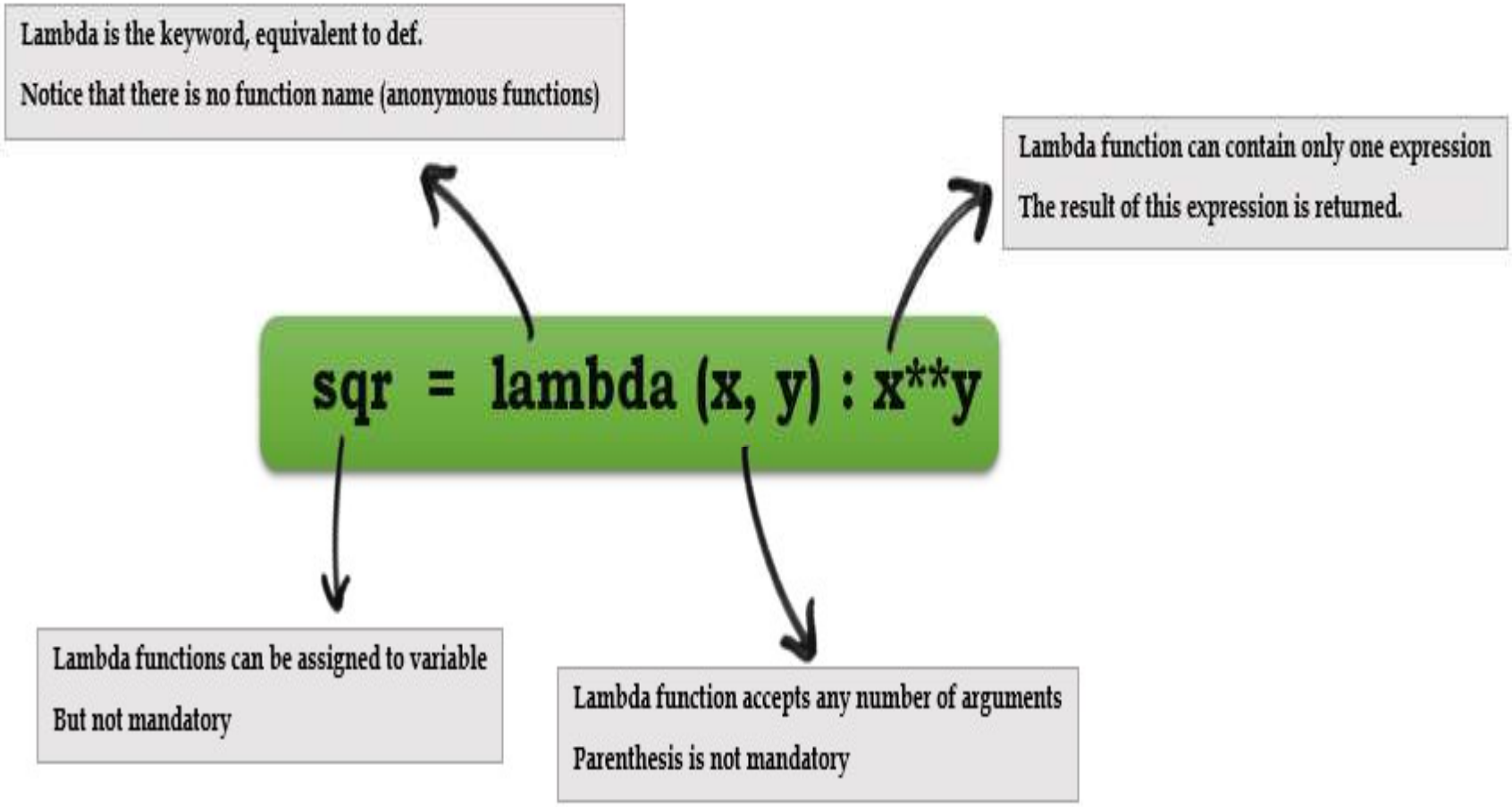
lambda arguments : expression

Ex :

lamda x: x*x

Lambda function

- It is a way to create small anonymous functions
- function without a name(nameless fellow)
- These functions are throw-away functions, i.e. they are just needed where they have been created.
  - Because of throw away, also called as One time function
- Lambda function is mainly used in combination with the functions filter(), map() and reduce()



Why to you declare a function if you don’t want to reuse the code , just go for lambda function

Lamb function

```
sqr=lambda  x:x*x
print(sqr(5))
OR
print((lambda x:x*x)(4))
```

```
f = lambda x : x * x
print(f, type(f))
print(f(10))
```

```
cube=lambda x:x*x*x
print(cube(3))
OR
print((lambda x:x*x*x)(3))
```

Lambda function

```
add=lambda x, y:   x + y   #  def add(x,y): return x + y
print(add(5,5))

cube=lambda x:x*x*x  #def cube(x):return x*x*x
print(cube(5))

sqr=lambda x:x*x #def sqr(x):return x*x
print(sqr(5))
```

Checking even or odd

```
def even(x):
    if x%2==0:
        print("even")
    else:
        print("odd")
```

```
even=lambda x:  'Even' if x%2==0 else 'Odd'
print(even(4))
```

Functional programming

- No side effects
  - no side effects in functions
- Immutability
  - no change in values of variables
- Simpler code
- It uses expressions instead of statements(avoid using loops)
- Recursion is an example for functional programming

To convert all chars of a list to uppercase

```
def convert(chars):
    res=[]
    for char in chars:
        res.append(char.upper())
    return res

chars=['a','b','c','d']
print(convert(chars))
```


```
chars=['a','b','c','d']
print(list(map(str.upper,chars)))
```

Map:

- Applies a task (function) to all the items in a list
  - Squaring, finding len, upper case
- Allows us to walk through an iterable and Collect the result
- Returns an iterable
  - Creates a new list from the results of applying the given *function* to the items of the given *sequence*
- input : n elements
- output : n elements

Basic syntax of map

map(function, iterables)

Name of the function without parenthesis

MAP :Applies a task (function) to all the items in a list

find the length of all the words in a given list

```
def findlen(words):
    res=[]
    for word in words:
        res.append(len(word))
    return res
```


```
a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
b = findlen(a)
print(b) # 5 9 3 7
```

```
a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
b = list(map(len, a))
print(b)
```

given a list of strings, convert to uppercase

```
def upper(x):
    res = []
    for w in x:
            res.append(str.upper(w))
    return res

a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
b = upper(a)
print(b) # [‘APPLE’, ‘PINEAPPLE’, ‘FIG’, ‘MANGOES’]
```

```
a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
b = list(map(str.upper, a))
print(b)
```

given a list of numbers, square each number

```
def square(x):
    res = []
    for w in x:
            res.append(w * w)
    return res

a = [11, 33, 22, 44]
b = square(a)
print(b)
```


```
a = [11, 33, 22, 44]
b = list(map(lambda x : x * x, a))
print(b)
```

Replace a with b in a given list of names

```
rep=lambda x:x.replace('a','b')

a = [ 'rama', 'krishna', 'parama', 'hamsa' ]
b = map(rep , a)
print(list(b))
```

```
a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
print(list(map(lambda s: s.replace('a','b'), a )))
```

capitalize the first letter of each word in a list

```
def foo(x):
    #return (x.replace(x[0],x[0].upper()))
    # return x.title()
    return x.capitalize()

words = ['raja', 'ram', 'mohan', 'roy']
print(list(map(foo(),words))
```

```
words = ['raja', 'ram', 'mohan', 'roy‘]
print(list(map(lambda x:x.title(),words)))
```

```
line="ram ram mohan roy"
print(line.title())
```

Print the first char of each word of a given list

```
words = ['raja', 'ram', 'mohan', 'roy']
print(list(map(lambda x:x[0],words)))
```

To find the position of some char in each word

```
return pos
words = ['raja', 'ram', 'mohan', 'roy']
print(list(map(lambda x:x.find('a'),words)))
```

To add corresponding elements of two lists

```
a=[1,2,3,4]
b=[4,3,2,1]
print(list(map(lambda x,y:x+y,a,b)))
```

5 5 5 5

```
a=[1,2,3,4]
b=[4,3,2]
print(list(map(lambda x,y:x+y,a,b)))
```

5 5 5

Map Stops when the shortest iterable is exhausted

Concluding the map:

Basic syntax of map

map(function, iterables)

- Applies a task (function) to all the items in a list
  - Squaring, finding len, upper case
- Allows us to walk through an iterable and Collect the result
- returns an iterable
  - Create a new list from the results of applying the given *function* to the items of the the given *sequence*
- *Stops when the shortest iterable is exhausted*

Filter:finding all even numbers in a list

```
numbers=[1,2,3,4,5,6,7,89,44,22,33,11,22,55]
def isEven(n):
    if n%2==0: return True
print(list(filter(isEven,numbers)))
```

Even better way

```
numbers=[1,2,3,4,5,6,7,89,44,22,33,11,22,55]
print(list(filter(lambda x:x%2==0,numbers)))
```

filter

- returns an iterable
- input : n elements
- output : anything between 0 and n elements
- output : has elements given in the input itself

  not the result of the function call
- gives the output when the function returns a true value

To pick of all words having a particular char

```
a = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]

# pickup all words which have r

print(list(filter(lambda s : 'r' in s , a)))

# pickup all words whose length exceeds 4

print(list(filter(lambda s : len(s) > 4 , a)))
```

Given a list of strings, count and print the number of strings where the string length is 2 or more & the 1st & last characters are same.

```
words= ["abc","bbc", "madam", "dad","hi","pp"]
count=0
for word in words:
    if(len(word)>1 and word[0] == word[-1]):
        count+=1
        print(word)
print(count)
```

```
words = ["abc","bbc", "madam", "dad","hi","pp"]
print(list(filter(lambda x:x[0]==x[-1] and len(x)>1,words)))
```

```
#to print all words having odd length
words = ["abc","bbc", "madam", "dad","hi","pp"]
print(list(filter(lambda x:len(x)%2==1,words)))
```

To print all names ending with a particular string

```
def ends(names):
    for name in names:
        if(name.endswith('sha')):
            print(name)

names=['amar','asha','akbar','isha','usha']
ends(names)
```

```
names=['amar','asha','akbar','isha','usha']
print(list(filter(lambda string:string.endswith('sha'),names)))
```

Map and filter

convert strings to uppercase and find all strings whose length exceeds 4

(Return all uppercase strings whose length exceeds 4)

```
# terrible code
print(list(filter(lambda s : len(s) > 4 , map(str.upper , a))))

# good code
print(list(map(str.upper, filter(lambda s : len(s) > 4, a))))
```

map

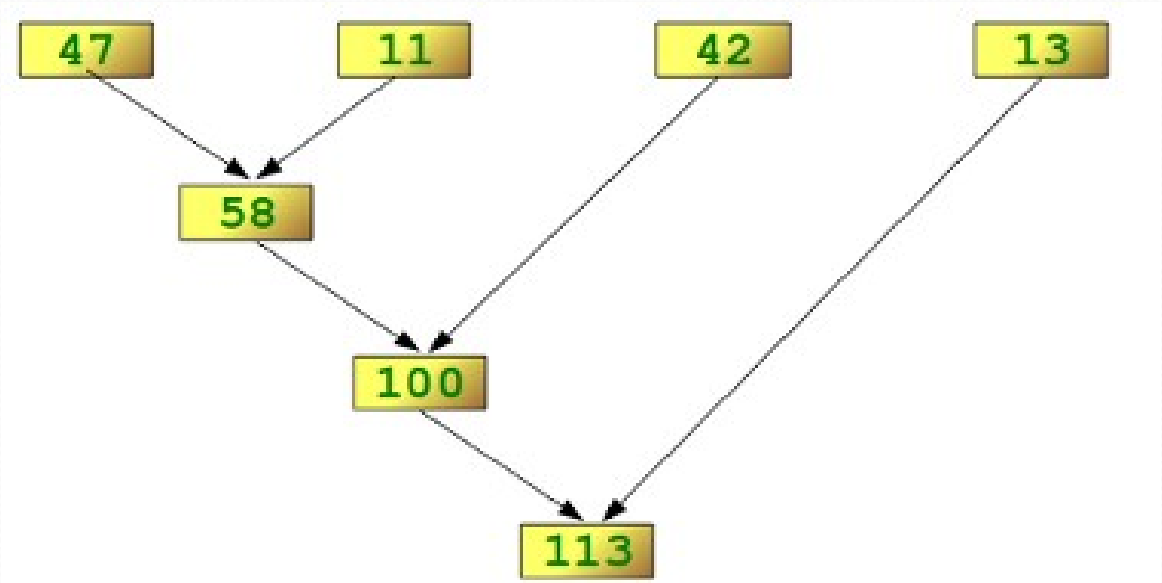
```
a=['1','2','3']
print(list(map(len,a)))

a=[1,2,3]
print(list(map(len,a)))

a=[[1],[2],[3]]
print(list(map(len,a)))

k=30
print(len(k))
k="30"
print(len(k))
```

Reduce

- Reduce the sequence to a single value.
- input : n elements
- output : 1 element
- Function(callable) : takes two arguments
- called n - 1 times

reduce

```
import functools
A= [47,11,42,13]
functools.reduce(lambda x,y: x+y,a)
113
```



reduce

```
# to find the product all numbers in a list

a=[10, 20, 30, 40]

from functools import reduce

product = reduce((lambda x,y: x*y),a)

print(product)
```

reduce

print(sum(range(1,101)))

find the sum of the numbers from 1 to 100:

```
from functools import reduce
reduce(lambda x, y: x+y, range(1,101))
 5050
```

Find the maximum of a list of numerical values by using reduce:

```
from functools import reduce
f = lambda a,b: a if (a > b) else b
a= [47,11,42,102,13])
reduce(f, a)
102
```

print(max(a))

```
import functools
a=[4,3,2,1]
print(functools.reduce(lambda x,y:x/y,a))
```

(((4/3)/2)/1)

Adding elements of a list

```
import functools
a=[1,2,3,4]
b=[4,3,2,1]
print(list(map(lambda x,y:x+y,a,b)))

print((functools.reduce(lambda x,y:x+y,a)))

print((functools.reduce(lambda x,y:x+y,b)))
```

```
print(sum(a))
print(sum(b))
```

To print multiples of 5 upto 100

```
print(list(filter(lambda x: x%5==0,range(1,100))))
```

Given a list of strings

a)find the longest string

b)find all string sending witha given suffix

c)find all strings starting with a given prefix

d)find all strings having a given substring

e)find the average length of the string-use reduce to find the total length

```
import functools
a=['aaaaa','bbb','c']
print(max(a))
print(max(a ,key=len))
a=list(map(len,a))
avg= functools.reduce(lambda x,y:x+y,a)/len(a)
print(avg)
```

Adding bonus to all employees and create another salary list.

Sal=[4000,8000,90000,8900,9300,9100]

```
def DasaraBonus(sal,bonus):
        blist=[]
        for i in Sal:
                blist.append(i+bonus)
        print(blist)
Sal=[4000,8000,90000,8900,9300,9100]
bonus=2000
DasaraBonus(Sal,bonus)
```

```
Sal=[4000,8000,90000,8900,9300,9100]
print(list(map(lambda x: x+2000,Sal)))
```

Separate even and odd elements from the list.

Put all even elements in list called elist and odd elements in a list called olist

```
a=[2,3,5,6,7,8,9,11,23,44,55,11,56]
elist=[]
olist=[]
for i in a:
      if(i%2==0):
            elist.append(i)
      else:
            olist.append(i)
print(elist,olist,sep="\n")
```

```
a=[2,3,5,6,7,8,9,11,23,44,55,11,56]
print(list(filter(lambda x:x%2==0,a)))
print(list(filter(lambda x:x%2==1,a)))
```

#info of students is given as list of tuples
s=[("890","x",(95,78,99)),("123","a",(90,98,89)),("567","p",(59,68,100))]

```
s=[("890","x",(95,78,99)),("123","a",(90,98,89)),("567","p",(59,68,100))]
srn=sorted(s)
print("Student list is sorted based on 1st field,SRN:\n",srn)

name=sorted(s,key=lambda t:t[1])
print("Student list is sorted based on 2nd field,Name:\n",name)

PCM=sorted(s,reverse=True,key=lambda t:sum(t[2]))
print(“list is sorted based on total of PCM marks in descending order:\n",PCM)

foo=lambda x:x[2][0]< 35 or x[2][1]<35 or x[2][2]<35
print(list(filter(foo ,s)))   #failure list
```

Zip in Python

- zip() in Python
- The purpose of zip() is to **map the similar index of multiple containers** so that they can be used just using as single entity.

Syntax :

*zip(*iterators)*

Parameters :

Python iterables or containers (list, string etc)

Return Value :

Returns a single iterator object, having mapped values from all the containers.

Zip in python

```
a=[1,2,3,4]
b=[6,7,8,9]
c=zip(a,b)
print(list(c))
[(1, 6), (2, 7), (3, 8), (4, 9)]
```

```
a=['a','b','c','d']
b=['apple','ball']
c=zip(a,b)
print(dict(c))
{'a': 'apple', 'b': 'ball'}
```

Zip() function

```
name = [ "amar", "akbar", "anthony"]

srn = [ 1, 2,3 ]

marks = [ 50, 60, 70 ]

c=zip(srn,name,marks)

print(list(c))
```

Given a list : SRN, P_marks, C_marks, M_marks and B_marks.

a) Create a dict with SRN as the key and marks in P, C, M, B as a list.
b) Find the class average in each subject
c) Find the maximum and minimum score in each subject
d) Find the number of failures in each subject
e) Find the percentage of each student

Given data

srns = ["17CS001","17CS002","17CS003"]

p_marks = [1,10,80]

c_marks = [2,20,84]

m_marks = [3,30,60]

b_marks = [4,40,78]

#Hint , create a dict as shown below, then proceed

#students={'17CS001':(1,2,3,4),'17CS002':(10,20,30,40)....}

Expected output

Name	Physics	Chem	Maths	Bio	percentage
17CS001	1	2	3	4	2.5
17CS002	10	20	30	40	25.0
17CS003	80	84	80	78	80.5

Avg is Phys= 30.333333333333332

Maxi in phy 17CS003 80

```
students={}
i=0;
for srn in srns:
        students[srn]=[]
        students[srn].extend([p[i], c[i],m[i],b[i]])
        i=i+1
print(students)
```

```
students={}
for i in range(0,3):
        students.update({srns[i]:[p[i],c[i],m[i],b[i]]})
print(students)
```

```
students=dict(zip(srns,(zip(p,c,m,b))))
print(students)
```

```
srns = ["17CS001","17CS002","17CS003"]
p = [1,10,80]
c = [2,20,84]
m= [3,30,60]
b= [4,40,78]

students=dict(zip(srns,(zip(p,c,m,b))))

print('-------------------------------------------------------------------')
print('Name \t  Physics \t Chem \t Maths \tBio \t percentage')
print('-------------------------------------------------------------------')
for student in students:
    print(student,end=" ")
    for marks in students[student]:
        print('\t',marks,end=" ")
    print('\t',sum(students[student])*100/400)
print('-------------------------------------------------------------------')
```

List of failures

```
students=(zip(srns,(zip(p,c,m,b))))


print(list(filter(lambda x:x[1][0]<35,students)))

print("no of failures in phy",len(list(filter(lambda x:x<35,p))))
print("no of failures in chem",len(list(filter(lambda x:x<35,c))))
print("no of failures in math",len(list(filter(lambda x:x<35,m))))
```

Finding highest score

```
srns = ["17CS001","17CS002","17CS003"]
p = [1,10,80]
c = [2,20,84]
m = [3,30,60]
b= [45,40,78]

#students=tuple((zip(srns,zip(p,c,m,b))))

#print(max(students,key=lambda x:sum(x[1])))

students=dict((zip(srns,zip(p,c,m,b))))

key_max = max(students.keys(), key=(lambda k: students[k]))
print(key_max,students[key_max])

key_min = min(students.keys(), key=(lambda k: students[k]))
print(key_min,students[key_min])
```

Call backs

A callback is a function that's called from within another

the name of a function can be used as a variable

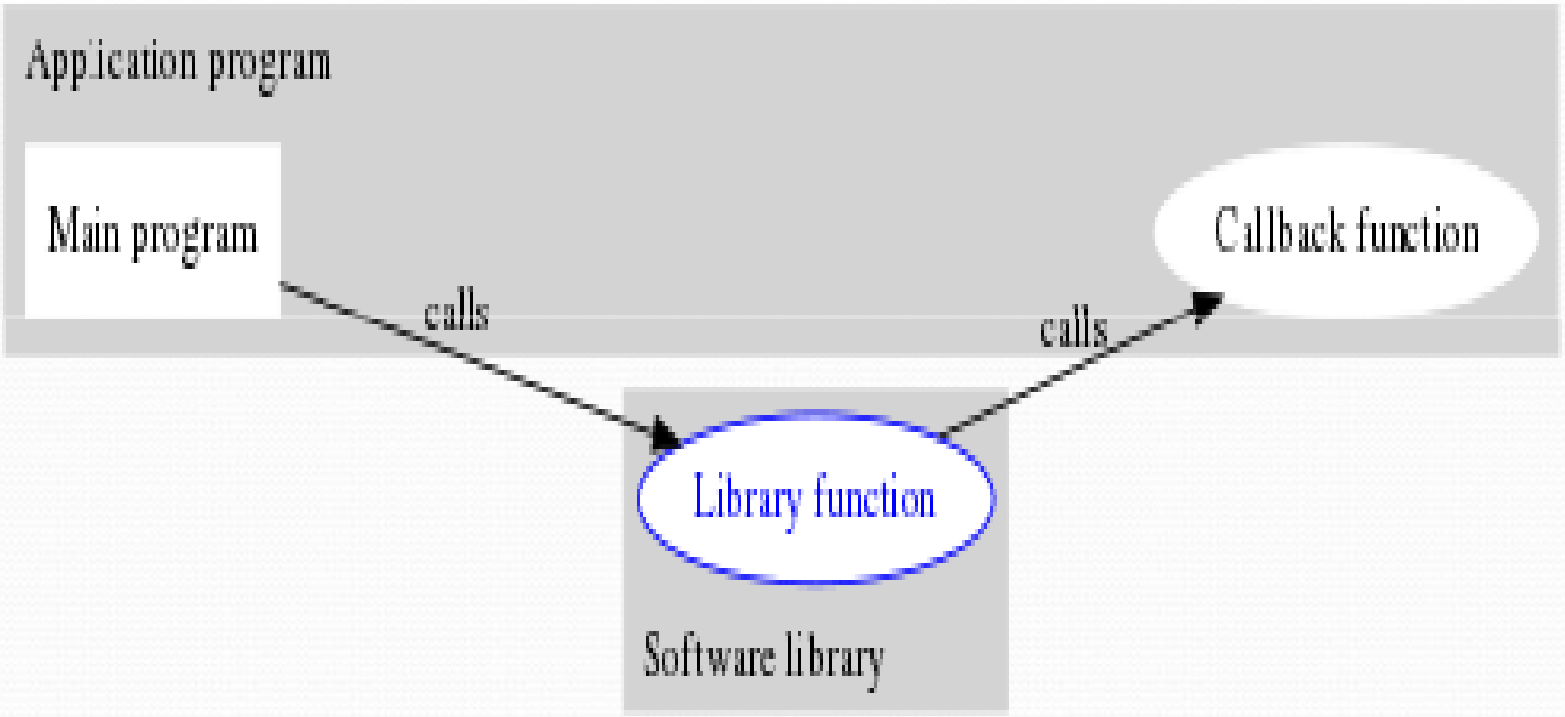
```
def func():
    ...

something(func)
```

Call backs

```
#name: 5_fn_callback.py
def fun1():
    print("this is function1")

def fun2():
    print("this is function2")

def callthis(fn_name):
    fn_name()

callthis(fun1)
callthis(fun2)
```

Observe that the function callthis is flexible. It does not know which function will be invoked when fn_name is called.
It depends on what the user specifies as the argument.
“This mechanism of passing a function name as argument and calling it indirectly is called callback”.

Call backs


Application program
Main program
calls
Callback function
calls
Library function
Software library

Call backs

```
# sort
a = ['apple', 'orange', 'ant', 'ball', 'zibra']
print(sorted(a))

# sort in a case insensitive way
print(sorted(a, key = str.upper))

a = ['apple', 'orange', 'ant', 'ball', 'zibra']
# combine 0th and 2nd char
#ap,oa,at,bl,zb
print(sorted(a, key = lambda s : s[0] + s[2]))
```

Call backs

```
a=['aaaaak','bbz','zc']

print(max(a))

print(max(a,key=len))

print(max(a,key=lambda x:x[-1]))
```

Call backs

```
def nice(msg):
    print("----------------------------------------")
    print('\t',msg)
    print("----------------------------------------")
    print()

def display(msg,fun=""):
    if(fun):
        fun(msg)
    else:
        print(msg)

msg="good morning"
display(msg,nice)

display(msg)
```

```
def prime(n):
    count=0
    for i in range(1,n//2):
            if(n%i==0): count+=1
    if(count>2): return "not prime"
    else:   return "prime“
def fact(n):
    if n==0:  return 1
    else:  return(n*fact(n-1))

def DoThis(job,num):
        return job(num)

n=int(input("enter the number"))
print(DoThis(job=prime,num=n))

n=int(input("enter the number"))
print(DoThis(job=fact,num=n))
```

Given name list and marks list, Write a function to find the student(s) with maximum marks

```
x = [90, 99, 95, 60, 99, 97]
y = ['Kumar', 'Rama', 'Amar', 'Akbar','Ravi','Jhon']
print(find_all_pairs(y, x))
```

Expected output

```
[('Rama', 99), ('Ravi', 99)]
```

Given name list and marks list, find the student who has scored maximum marks

```
def find_all_pairs(namelist, markslist) :
# find max
    m = max(markslist)
    pos = markslist.index(m)

    res = []
    for i in range(pos, len(markslist)):
        if markslist[i] == m :
            res.append((namelist[i], m))
    return res

x = [90, 99, 95, 60, 99, 97]
y = ['Kumar', 'Rama', 'Amar', 'Akbar','Ravi','Jhon']
print(find_all_pairs(y, x))
```

Given a dict where values are not unique, write a function to create a new dict where the key is the value and the value is concatenated keys of the original dict and return it

Given

d ={ 'apple' : 'fruit', 'cat' : 'mammal', 'beans' : 'veg', 'dog' : 'mammal','mango' : 'fruit' }

Expected output:

y= { 'fruit' : ['apple', 'mango'], 'mammal' : ['cat', 'dog'], 'veg' : ['beans']}

Hint:

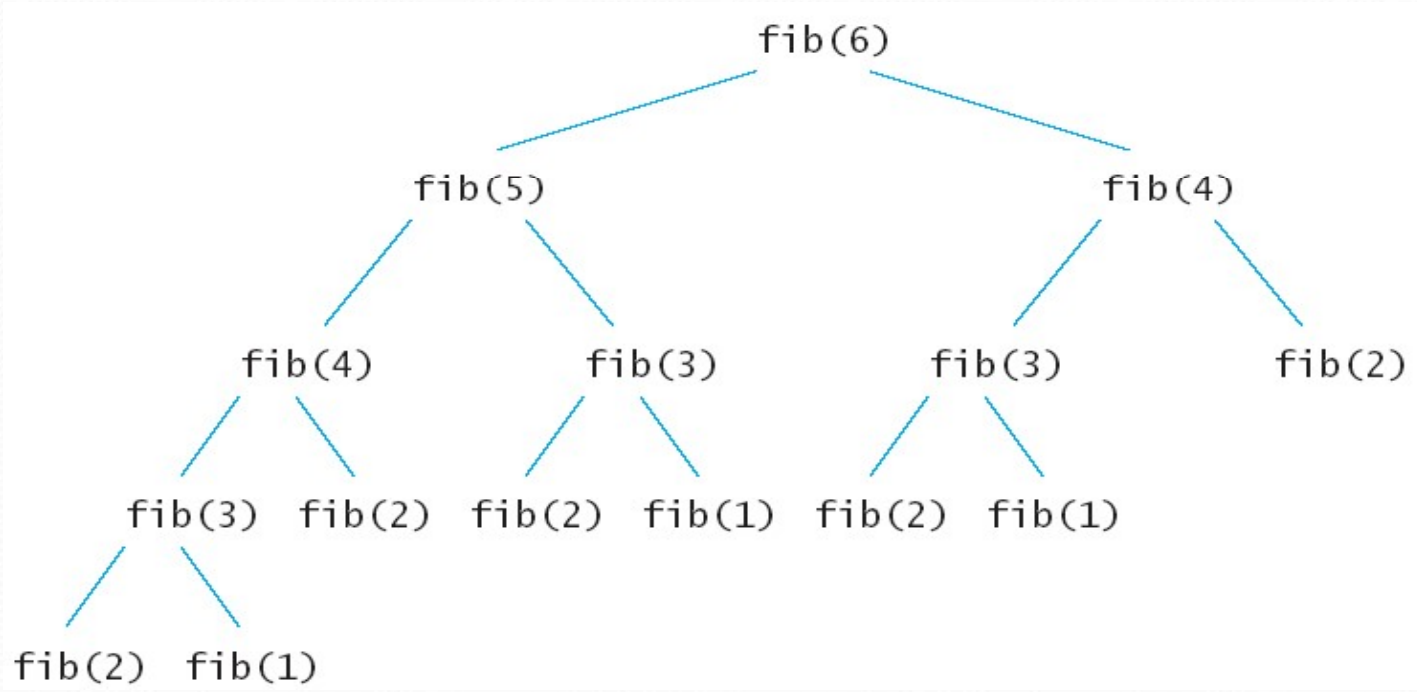
{'fruit': [], 'mammal': [], 'veg': []}

```
d ={ 'apple' : 'fruit', 'cat' : 'mammal', 'beans' : 'veg', 'dog' :
   'mammal','mango' : 'fruit' }
res={}
for key in d.keys():
     val=d[key]
     if val not in res:
          res[val]=[]
     res[val].append(key)
print(res)
```

```
def invert(d):
    res={}
    for key in d.keys():
        val=d[key]
        if val not in  res:
            res[val]=[]
        res[val].append(key)
    return res

d = { 'apple' : 'fruit', 'cat' : 'mammal', 'beans' : 'veg', 'dog' :
   'mammal','mango' : 'fruit' }
print(invert(d))
```

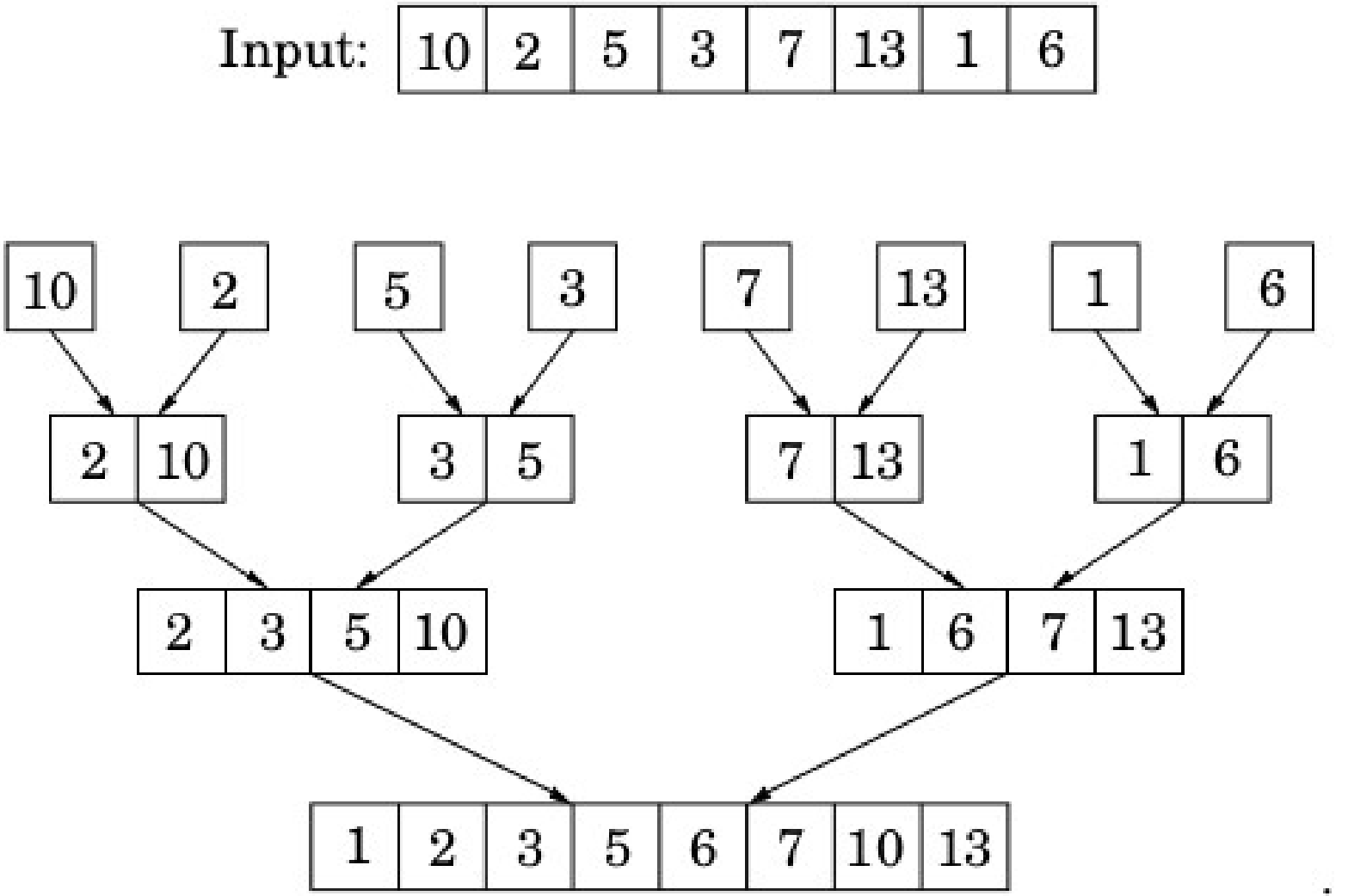
Recursion A function that calls by itself

Recursive problem :Fibonacci series



Recursive problems

Merge sort



Factorial using recursion

Fact(n) = 1 (if n ==1)

= n*fact(n-1) (if n>0)

Fact(7)

7xfact(6)

6xfact(5)

5xfact(4)

4xfact(3)

3xfact(2)

2xfact(1)

1

Recursive function to find factorial of a number

Fact(n) = 1 if n ==1

= n*fact(n-1) if n>1

```
def fact(n):
    if n == 1 :
        return 1    #stopping condition
    else:
        return n * fact(n - 1)

print(fact(5)) # 120
```

Factorial using recursion

Fact(n) = 1 if n ==1

= n*fact(n-1) if n>1

Fact(6)

return 6xfact(5)

return 5xfact(4)

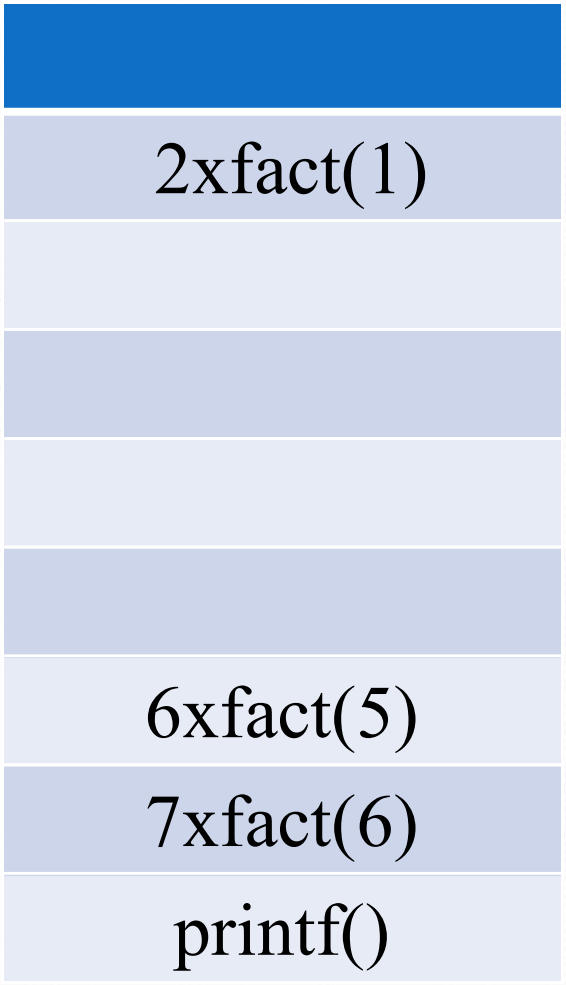
return 4xfact(3)

return 3xfact(2)

return 2xfact(1)

1 ----------

```
def fact(n):
    if n == 1 :
            res = 1
    else:
            res = n * fact(n - 1)
    return res

print(fact(7)) # 720
print(fact(0)) # 1
```



1x1
2x1=2
3x2=6
4x6
5x24
6x120
7x720
printf()

GCD of two numbers

```
a=10
b=20
if a < b:
      small = a
else:
     small = b
for i in range(1, small+1):
      if((a % i == 0) and (b % i == 0)):
          gcd = i

print(gcd)
```

```
If 10%1 ==0  and  20%1 ==0   true
  Gcd=1
If 10%2 ==0  and  20%2 ==0  true
  Gcd=2
If 10%3 ==0  and  20%3 ==0  false
If 10%4 ==0  and  20%4 ==0  False
  -----------
If 10%10 ==0  and  20%10 ==0
Gcd=10
```

GCD of two numbers-**Euclid's Algorithm**

```
a=20;b=10
while(b):
    r=a%b
    if(r==0):
        print(b)
        break
    else:
        a=b
        b=r
```

```
a=20;b=10
while(b):
    r=a%b
    a=b
    b=r
print(a)
```

```
a=20;b=10
while(b):
    a, b = b, a % b
  print a
```

Recursive function to find GCD

```
def gcd(a,b):
    if(b==0):
        return a
    else:
        return gcd(b,a%b)

a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
GCD=gcd(a,b)
print("GCD is: ")
print(GCD)
```

write a recursive function to find the factorial of a number, n!, defined by

fact(n)=1, if n=0. Otherwise fact(n)=n*fact(n-1)

Using this function, write a program to compute the binomial coefficient nCr.

Using recursive functions, write a program to solve

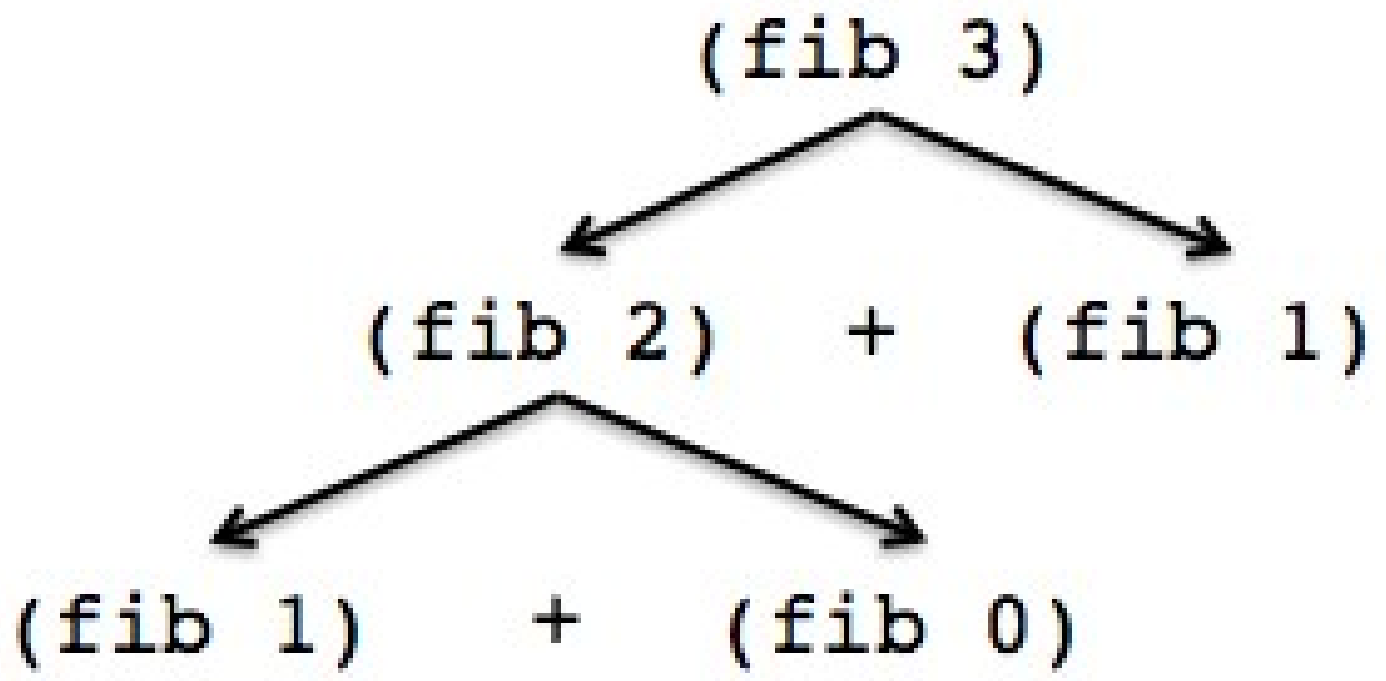
nCr =n!/(n-r)!*r!

```
def factorial(n):
    if(n==0): return 1
    else: return n*factorial(n-1)
```

```
n=int(input('enter the vaue of n'))
r=int(input('enter the vaue of r'))
ncr=factorial(n)/(factorial(n-r)*factorial(r))
print(ncr)
```

To find ncr

Fibonacci using recursion



Fibonacci using recursion

Write a recursive pgm to print Fibonacci series up to n

0 1 1 2 3 5 8 13

n indicates no of terms

If n=1 then fib term is 0

If n=2 then fib term is 1

If n>2 then term is fib(n-1)+ fib(n-2)

Pgm to print Fibonacci series up to n

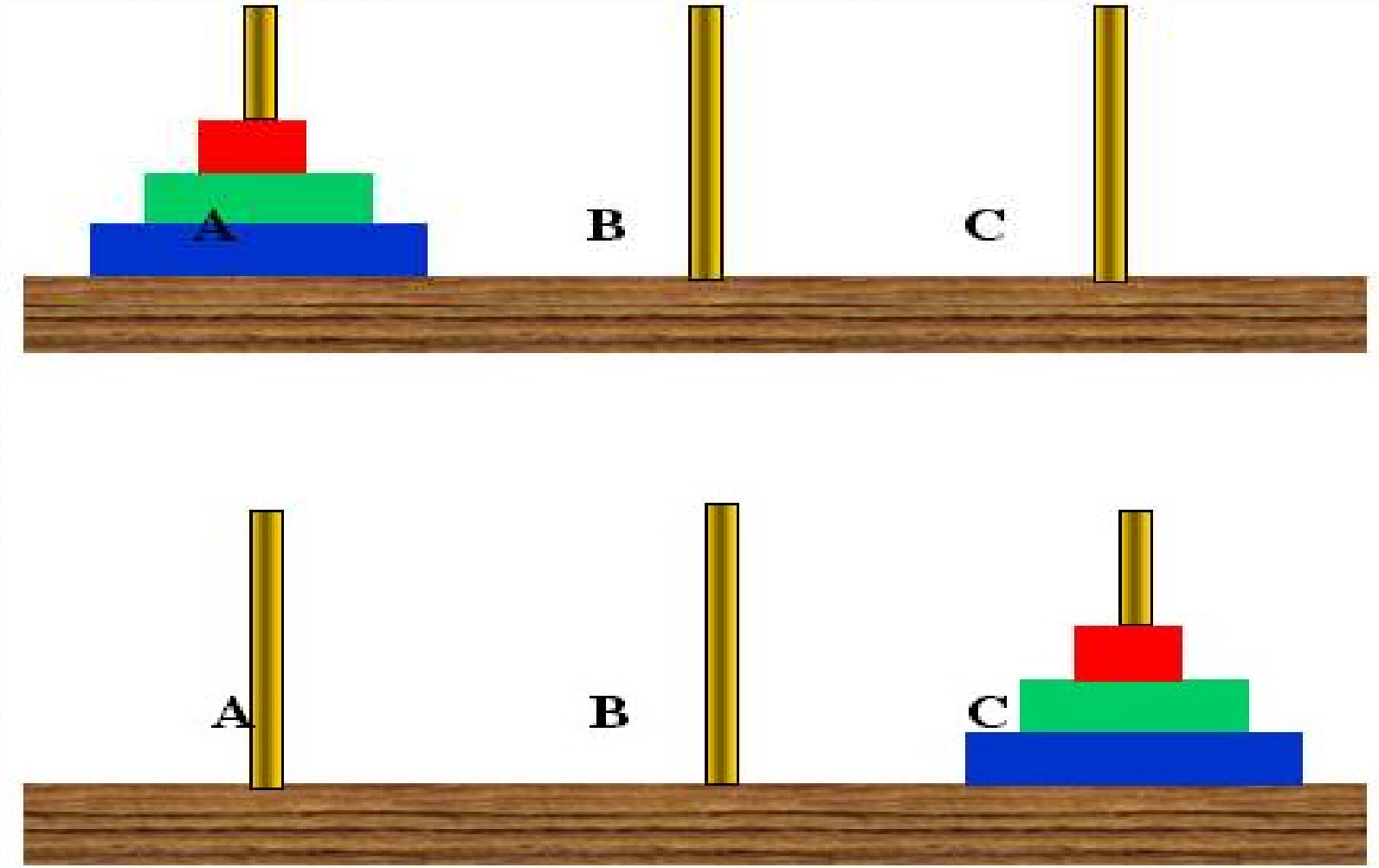
```
n=100
first=0
second=1
third=1
print(first,second,end=" ")
while(third<=n):
        print(third,end=" ")
        third=first+second
        first=second
        second=third
```

Fibonacci series upto n

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

for i in range(1,20):
    print(fibo(i))
```

Recursive problems

Tower of hanoi



```
# name: 3_tower_of_hanoi.py

# tower of hanoi
# if there are disks(say n) then
#             move n-1 disks from source to temp using dest
#    move one disk from source to dest
#    move n-1 disks from temp to dest using source


def move(n, source, dest, temp) :
    if n > 0 :
                move(n - 1, source, temp, dest)
                print(source, "=>", dest)
                move(n - 1, temp, dest, source)

move(3, 'A', 'C', 'B')
```

Recursion to find length of the string

```
Length('Python')
    return(1+ Length('Pytho'))
        return(1+ Length('Pyth'))
            return(1+ Length('Pyt'))
                return(1+ Length('Py'))
                    return(1+ Length('P'))
                        return(1+ Length(''))
```

Python
Pytho
Pyth
Pyt
Py
P

Recursion to find length of the string

```
def length(s):
    #print(s)
    if s == '':
        return 0
    else:
        return 1 + length(s[:-1])


s = 'bangalore'
print("Length of a given string ", s, " is ",length(s))
```

```
Python
Pytho
Pyth
Pyt
Py
P
```

String reverse using recursion

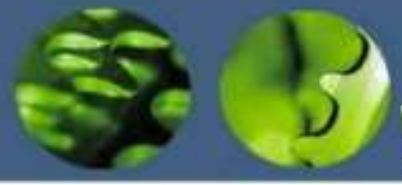
```
Reverse('ABCD')
    return Reverse('BCD')+A
        return Reverse('CD')+B
            return Reverse('D')+C
                return Reverse('')+D
```

ABCD
BCD
CD
D

String reverse using recursion

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:])+ s[0]

s = input("Enter a string: ")
print("Reverse of a string ", s, " is ",reverse(s))
```

```
ABCD
BCD
CD
D
```



Why Recursion?

- Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems.
- Recursive solutions can be easier to understand and to describe than iterative solutions.

Advantages of functions

Reusability: code can be reused number of times.

Less space: functions reduce the length of the program and thereby program takes less space.

Debugging is easy.

Program readability increases.

Program becomes more understandable.

Complex Program can be divided into modules.

Function redefine

```
#name : 4_fn_redefine.py
def foo():
    print("one")
foo() # one
# replaces the old function with the new one
def foo():
    print("two")
foo() # two
```

```
Variable redefine
k=11
k=22
print(k)
print(k)
```

```
import statistics
a=[1,2,3,4,3,2,2,1,1,1,1]
print(statistics.mean(a))
print(statistics.mode(a))
```