Computer Science (083)
CLASS-XII 2021-22

DISTRIBUTION OF MARKS:

UNIT	UNIT NAME	MARKS
I	**Computational Thinking and Programming - 2**	**40**
II	Computer Networks	10
III	Database Management	20
TOTAL		70

Unit I: Computational Thinking and Programming – 2

• Revision of Python topics covered in Class XI.
• Functions: types of function (built-in functions, functions defined in module, user defined functions), creating user defined function, arguments and parameters, default parameters, positional parameters, function returning value(s), flow of execution, scope of a variable (global scope, local scope)
• Introduction to files, types of files (Text file, Binary file, CSV file), relative and absolute paths
• Text file: opening a text file, text file open modes (r, r+, w, w+, a, a+), closing a text file, opening a file using with clause, writing/appending data to a text file using write() and writelines(), reading from a text file using read(), readline() and readlines(), seek and tell methods, manipulation of data in a text file
• Binary file: basic operations on a binary file: open using file open modes (rb, rb+, wb, wb+, ab, ab+), close a binary file, import pickle module, dump() and load() method, read, write/create, search, append and update operations in a binary file
• CSV file: import csv module, open / close csv file, write into a csv file using csv.writerow() and read from a csv file using csv.reader( )
• Python libraries: creating python libraries
• Recursion: simple programs with recursion: sum of first n natural numbers, factorial, fibonacci series
• Idea of efficiency: number of comparisons in Best, Worst and Average case for linear search
• Data Structure: Stack, operations on stack (push & pop), implementation of stack using list. Introduction to queue, operations on queue (enqueue, dequeue, is empty, peek, is full), implementation of queue using list.

Functions:

Large programs are often difficult to manage, thus large programs are divided into smaller units known as functions.

A function is a programming block of codes which is used to perform a single, related task. It is simply a group of statements under any name i.e. Function name and can be invoked (call) from other part of program. It only runs when it is called. We can pass data, known as parameters, into a function. A function can return data as a result.

We have already used some python built in functions like print (), etc. But we can also create our own functions. These functions are called user-defined functions.

Type of Functions:

- Built-in Functions
- User-Defined Functions

Advantages of functions:

- Program development made easy and fast
- Program testing becomes easy
- Code sharing becomes possible
- Code re-usability increases
- Increases program readability

User Defined Functions:

Syntax to create USER DEFINED FUNCTION

def function_name([**comma separated list of parameters**]):
    statements….
    statements….

```
def my_own_function():
    print('Hello from a function')
#program start
print('hello before calling a function')
my_own_function() #function calling.
```

Save the above source code in python file and execute it

Important points to remember:

- Keyword **def** marks the start of function header
- Function name must be unique and follows naming rules same as for identifiers
- Function can take arguments. It is optional
- A colon(:) to mark the end of function header
- Function can contains one or more statement to perform specific task

Function must be called/invoked to execute its code

Scope of variable in functions

SCOPE means in which part(s) of the program, a particular piece of code or data is accessible or known.
In Python there are broadly 2 kinds of Scopes:

Global Scope
A name declared in top level segment (main) of a program is said to have global scope and can be used in entire program.
Variable defined outside all functions are global variables.

Local Scope
A name declare in a function body is said to have local scope i.e. it can be used only within this function and the other block inside the function.
The formal parameters are also having local scope.

Let us understand with example….

Example:

```
def  area(length, breadth):
        a= length*breadth          # a is local to this function
        return a
l=int(input('Enter Length'))
b=int(input('Enter Breadth'))
ar=area(l,b)                       # l,b and ar having global scope
print('Area of Rectangle =', ar)
```

Example:

```
def  area(length, breadth):
        a= length*breadth          # a is local to this function
        return a
l=int(input('Enter Length'))
b=int(input('Enter Breadth'))
ar=area(l,b)                       # l,b and ar having global scope
print('Area of Rectangle =', a)
```

'a' is not accessible here because it is declared in function area(), so scope is local to area()

Example:

```
def  area(length, breadth):
        a= length*breadth          # a is local to this function
        return a

def showarea():
        print('Area of Rectangle =', ar)

l=int(input('Enter Length'))
b=int(input('Enter Breadth'))
ar=area(l,b)                       # l,b and ar having global scope
showarea()
```

Variable 'ar' is accessible in function showarea() because it is having Global Scope

Example:

```
count =0
def add(x,y):
   global count
   count+=1
   return x+y

def sub(x,y):
   global count
   count+=1
   return x-y

def mul(x,y):
```

```
    global count
    count+=1
    return x*y

def div(x,y):
    global count
    count+=1
    return x//y
ans=1
while ans==1:
    n1=int(input('Enter first number'))
    n2=int(input('Enter second number'))
    op=int(input('Enter 1-Add 2-Sub 3-Mul 4-Div')
    if op==1:
        print('Sum=', add(n1,n2))
    elif op==2:
        print('Sub=', sub(n1,n2))
elif op==3:
        print('Mul=', mul(n1,n2))
elif op==4:
        print('Div=', div(n1,n2))
    ans=int(input('press 1 to continue'))
print('Total operation donE', count)
```

This declaration "global count" is necessary for using global variables in function, otherwise an error "local variable 'count' referenced before assignment" will appear because local scope will create variable "count" and it will be found unassigned

Lifetime of a variable is the time for which a variable lives in memory.
For Global variables the lifetime is entire program run
i.e. as long as program is executing.

For Local variables lifetime is their function's run i.e. as long as function is executing.

Types of user defined functions

1. Function with no arguments and no return
2. Function with arguments but no return value
3. Function with arguments and return value
4. Function with no argument but return value

Function with no arguments and no return

Example:

```
def welcome():
   print('Welcome to India')
   print('Good Morning')

welcome()
```

Function with arguments but no return value

Example:

```
def table(num):
        for i in range(1,11):
                print(num,'x','=',num*i)
n=int(input('Enter any number'))
table(n)
```

Example:

```
def avg(n1,n2,n3,n4):
        av=(n1+n2+n3+n4)/4.0
        print("Average=",av)
a=int(input('Enter first number='))
b=int(input('Enter second number='))
c=int(input('Enter third number='))
d=int(input('Enter fourth number='))
avg(a,b,c,d)
```

Function with arguments and return value

We can return values from function using return keyword.

The return value must be used at the calling place by –

- Either store it any variable
- Use with print()
- Use in any expression

Example:

```
def cube(num):
        return num*num*num

x=int(input('Enter any number'))
y=cube(x)
print('C=cube of number is =', y)
```

Example:

```
def cube(num):
        return num*num*num

x=int(input('Enter any number'))
y=x+cube(x)
print('Number + Cube=', y)
```

Example:

```
def cube(num):
        return num*num*num

x=int(input('Enter any number'))
print('Cube of number is =', cube(x))
```

Example:

```
def cube(num):
        c=num*num*num
        return c

x=int(input('Enter any number'))
y= cube(x)
print('Cube of number is=',y)
```

Example:

```
import math
def area(r):
        a=math.pi*r*r
        return a

n= int(input('Enter radius='))
ar=area(n)
print('Area of circle =', ar)
```

Function with no argument but return value

Example:

```
def welcome():
        return('Welcome to India')
welcome()
print(welcome())
```

Parameters & Arguments (Actual/Formal)

Parameters are the value(s) provided in the parenthesis when we write function header. These are the values required by function to work
If there is more than one parameter, it must be separated by comma (,)
These are called FORMAL ARGUMENTS/PARAMETERS

An Argument is a value that is passed to the function when it is called. In other words arguments are the value(s) provided in function call/invoke statement.
These are called ACTUAL ARGUMENTS / PARAMETER

Example:

```
import math
def area(r):
        a=math.pi*r*r
        return a

n= int(input('Enter radius='))
ar=area(n)
print('Area of circle =', ar)
```

In this example r is Formal parameter and n is actual parameter.

Parameters / Arguments Passing and return value

These are specified after the function name, inside the parentheses. Multiple parameters are separated by comma. The following example has a function with two parameters x and y. When the function is called, we pass two values, which is used inside the function to sum up the values and store in z and then return the result(z):

```
def sum(x,y): #x, y are formal arguments
z=x+y
return z #return the value/result
```

```
x,y=4,5
r=sum(x,y) #x, y are actual arguments
print(r)
```

Note :-

1. Function Prototype is declaration of function with name, argument and return type.
2. A formal parameter, i.e. a parameter, is in the function definition. An actual parameter, i.e. an argument, is in a function call.

Functions can be called using following types of formal arguments –

1. **Positional parameter** - arguments passed in correct positional order
2. **Keyword arguments** -the caller identifies the arguments by the parameter name
3. **Default arguments** - that assumes a default value if a value is not provided to argu.
4. **Variable-length** arguments – pass multiple values with single argument name.

#Positional arguments

Example:

```
def square(x):
z=x*x
return z

r=square()
print(r)
```

#In above function square() we have to definitely need to pass some value to argument x.

Example:

```
def divide(a,b):
        c=a/b
        print(c)
```

```
x=int(input('Enter first number='))
y=int(input('Enter second number='))
divide(x,y)
```

Here x is passed to a and y is passed to b i.e. in the order of their position, if order is change result may differ
If the number of formal argument and actual differs then Python will raise an error

#Keyword arguments

```
def fun( name, age ):  #"This prints a passed info into this function"
        print ("Name: ", name)
        print ("Age ", age)
        return;

fun( age=15, name="mohak" )
```

value 15 and mohak is being passed to relevant argument based on keyword used for them.

#Default arguments

Sometimes we can provide default values for our positional arguments. In this case if we are not passing any value then default values will be considered.
Default argument must not followed by non-default arguments.

```
def interest(principal,rate,time=15):                valid
def interest(principal,rate=8.5,time=15):            valid
def interest(principal=10000,rate=8.5,time=15):      valid
def interest(principal,rate=8.5,time):               invalid
```

```
#Default arguments
def sum(x=3,y=4):
        z=x+y
        return z

r=sum()
```

```
print(r)
r=sum(x=4)
print(r)
r=sum(y=45)
print(r)
```

#default value of x and y is being used when it is not passed

#Variable length arguments

Example:

```
def sum( *vartuple ):
        s=0
        for var in vartuple:
                s=s+int(var)
        return s

r=sum( 70, 60, 50 )
print(r)
r=sum(4,5)
print(r)
#now the above function sum() can  sum n number of values
```

Mutable/immutable properties of data objects

Everything in Python is an object, and every object in Python can be either mutable or immutable.

Since everything in Python is an Object, every variable holds an object instance. When an object is initiated, it is assigned a unique object id. Its type is defined at runtime and once set can never change, however its state can be changed if it is mutable. Means a mutable object can be changed after it is created, and an immutable object can't.

Mutable objects: list, dict, set, byte array
Immutable objects: int, float, complex, string, tuple, bytes

Example:

```
def check(num):
   print('value in function check() is =', num)
   print('ID of num in function is =', id(num))
   num+=10
   print('value in function check() is =', num)
   print('ID of num in function is =', id(num))

myvalue=100
print('value in main function is =', myvalue)
print('ID of myvalue in function is =', id(myvalue))
check(myvalue)
print('value in main function is =', myvalue)
print('ID of myvalue in function is =', id(myvalue))
```

```
value in main function is = 100
ID of myvalue in function is = 1549036224
value in function check() is = 100
ID of num in function is = 1549036224
value in function check() is = 110
ID of num in function is = 1549036384
value in main function is = 100
ID of myvalue in function is = 1549036224
```

Python variables are not storage containers, rather Python variables are like memory references, they refer to memory address where the value is stored, thus any change in immutable type data will also change the memory address.

*So any change to formal argument will not reflect back to its corresponding actual argument and in case of mutable type, any change in mutable type will not change the memory address of variable.*Example:

```
def updatedata(mylist):
        print(mylist)
        for i in range (len(mylist)):
                mylist[i]+=100
        print(mylist)

List1=[10,20,30,40,50]
print(List1)
updatedata(List1)
print(List1)
```

```
Output:
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
[110, 120, 130, 140, 150]
[110, 120, 130, 140, 150]
```

Because List if Mutable type, hence any change in formal argument myList will not change the memory address, So changes done to myList will be reflected back to List1.

Passing String to a function

String can be passed in a function as argument but it is used as pass by value. It can be depicted from below program. As it will not change the value of actual argument.

```
def welcome(title):
        title="hello"+title

r="Mohan"
welcome(r)
print(r)

output:
Mohan
```

Passing tuple to a function

in function call, we have to explicitly define/pass the tuple. It is not required to specify the data type as tuple in formal argument. E.g.

```
def Max(myTuple):
        first, second = myTuple
        if first>second:
                return first
        else:
                return second

r=(3, 1)
```

```
m=Max(r)
print(m)

output:
3
```

Pass dictionary to a function

In Python, everything is an object, so the dictionary can be passed as an argument to a function like other variables are passed.

```
def func(d):
        for key in d:
                print("key:", key, "Value:", d[key])

Mydict = {'a':1, 'b':2, 'c':3}
func(Mydict)

output:
key: a Value: 1
key: b Value: 2
key: c Value: 3
```

Passing arrays/list to function

Arrays are popular in most programming languages like: Java, C/C++, JavaScript and so on. However, in Python, they are not that common. When people talk about Python arrays, more often than not, they are talking about Python lists. Array of numeric values are supported in Python by the array module/.e.g.

```
def dosomething( thelist ):
        for element in thelist:
                print (element)

dosomething( ['1','2','3'] )
alist = ['red','green','blue']
dosomething( alist )

Output:
1
2
3
red
```

green
blue

Functions using libraries

Mathematical functions:

Mathematical functions are available under math module. To use mathematical functions under this module, we have to import the module using import math.

For e.g.

To use sqrt() function we have to write statements like given below.

```
import math
r=math.sqrt(4)
print(r)
```

OUTPUT : 2.0

Functions available in Python Math Module

Function	Description	Example
ceil(n)	It returns the smallest integer greater than or equal to n.	math.ceil(4.2) returns 5
factorial(n)	It returns the factorial of value n	math.factorial(4) returns 24
floor(n)	It returns the largest integer less than or equal to n	math.floor(4.2) returns 4
fmod(x, y)	It returns the remainder when n is divided by y	math.fmod(10.5,2) returns 0.5
exp(n)	It returns e**n	math.exp(1) return 2.718281828459045
log2(n)	It returns the base-2 logarithm of n	math.log2(4) return 2.0
log10(n)	It returns the base-10 logarithm of n	math.log10(4) returns 0.6020599913279624
pow(n, y)	It returns n raised to the power y	math.pow(2,3) returns 8.0
sqrt(n)	It returns the square root of n	math.sqrt(100) returns 10.0
cos(n)	It returns the cosine of n	math.cos(100) returns 0.8623188722876839
sin(n)	It returns the sine of n	math.sin(100) returns -0.5063656411097588
tan(n)	It returns the tangent of n	math.tan(100) returns -0.5872139151569291
pi	It is pi value (3.14159...)	It is (3.14159...)
e	It is mathematical constant e (2.71828...)	It is (2.71828...)

String functions:

String functions are available in python standard module.These are always availble to use.

For e.g. capitalize() function Converts the first character of string to upper case.

```
s="i love programming"
r=s.capitalize()
print(r)
```

OUTPUT:

I love programming

String functions

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
count()	Returns the number of times a specified value occurs in a string
find()	Searches the string for a specified value and returns the position of where it was found
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isspace()	Returns True if all characters in the string are whitespaces
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string

lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
split()	Splits the string at the specified separator, and returns a list
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case

Predict the output:

```
def check():
        value=100
        print(value)

value=600
print(value)
check()
print(value)
```

```
def check():
        global value
        value=100
        print(value)

value=600
print(value)
check()
print(value)
```

```
def check():
        print(value)

num=100
print(num)
check()
print(value)
```

```
def check():
        if num1>num2:
                print(num1)
        else:
                print(num2)
```

```
num1=int(input(‘Enter first number=’))
num2=int(input(‘Enter second number=’))
check()
```