Python Booleans & Operators

Booleans represent one of two values: True or False.

Boolean Values In programming you often need to know if an expression is True or False.

You can evaluate any expression in Python, and get one of two answers, True or False.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

The logical operators and, or and not are also referred to as boolean operators.

```
#About and operator
a=50
b=25

a>40 and b>40

a>100 and b<50

a==0 and b==0

a>0 and b>0

#about or operator
a>40 or b>40

a>100 or b<50

a==0 or b==0

a>0 or b>0

#About not operator
a=10
a>10

not(a>10)

# Example
print(10 > 9)
print(10 == 9)
print(10 < 9)

#When you run a condition in an if statement, Python returns True or
False:

#Example
#Print a message based on whether the condition is True or False:

a = 200
b = 33
```

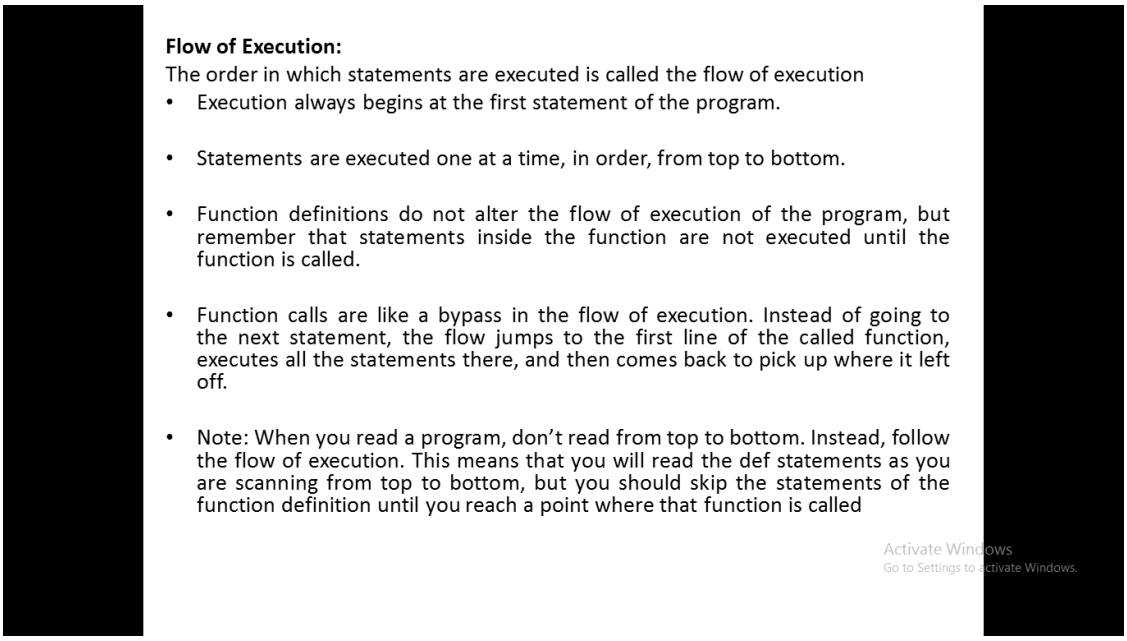
```python
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")

#Evaluate Values and Variables
#The bool() function allows you to evaluate any value, and give you
True or False in return,

#Example
#Evaluate a string and a number:

print(bool("Hello"))
print(bool(15))


#Example: Evaluate two variables:

x = "Hello"
y = 15

print(bool(x))
print(bool(y))

#Most Values are True
#Almost any value is evaluated to True if it has some sort of content.
#Any string is True, except empty strings.
#Any number is True, except 0.
#Any list, tuple, set, and dictionary are True, except empty ones.

# Example  The following will return True:

bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])


# Some Values are False
# In fact, there are not many values that evaluates to False, except
empty values, such as (), [], {}, "", the number 0, and the value
None. And of course the value False evaluates to False.

# Example The following will return False:

bool(False)
bool(None)
bool(0)
bool("")
```

```
bool(())
bool([])
bool({})
```

Conditionals in Python (if, if..else,if..elif..else,nested if)

'If' Statement

If statements are control flow statements which helps us to run a particular code only when a certain condition is satisfied.



Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called

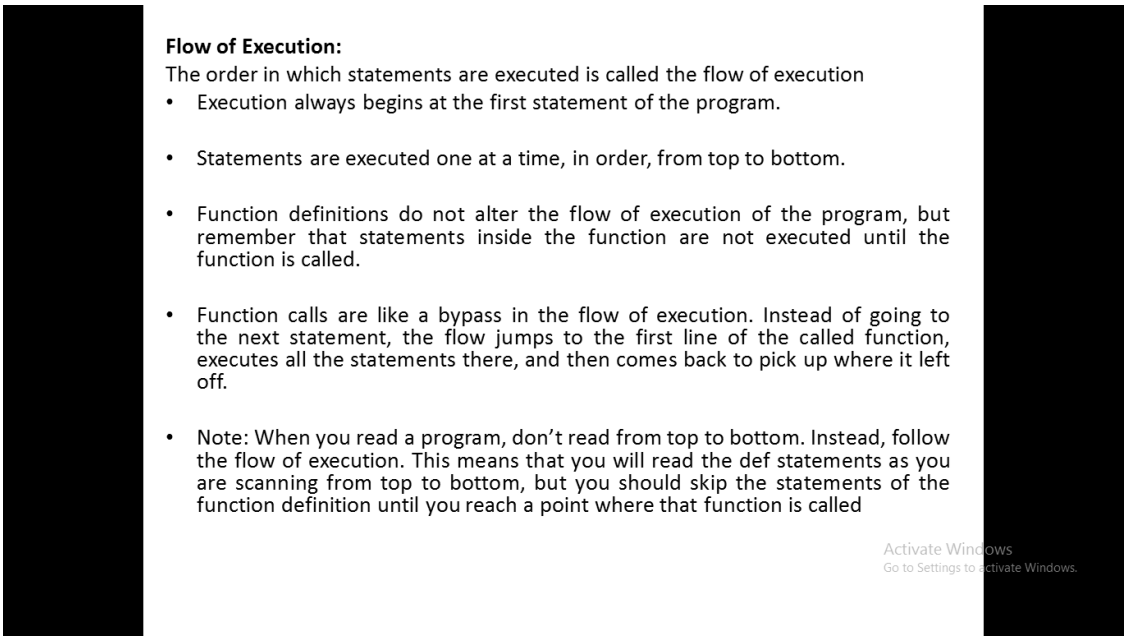

```
#Python – If statement Example
flag = True
if flag==True:
    print("Welcome")
    print("To")
    print("MRCET College")
print()

if flag:
    print("Welcome")
    print("To")
    print("MRCET College")
print()


num = 100
if num < 200:
    print("num is less than 200")
print()
```

```
i = 10
if (i > 15):
   print ("10 is less than 15")
print ("I am Not in if")
print()


# short hand if
a=2
b=6
if a > b: print("a is greater than b")
```

```
Welcome
To
MRCET College

Welcome
To
MRCET College

num is less than 200

I am Not in if
```

if- else

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

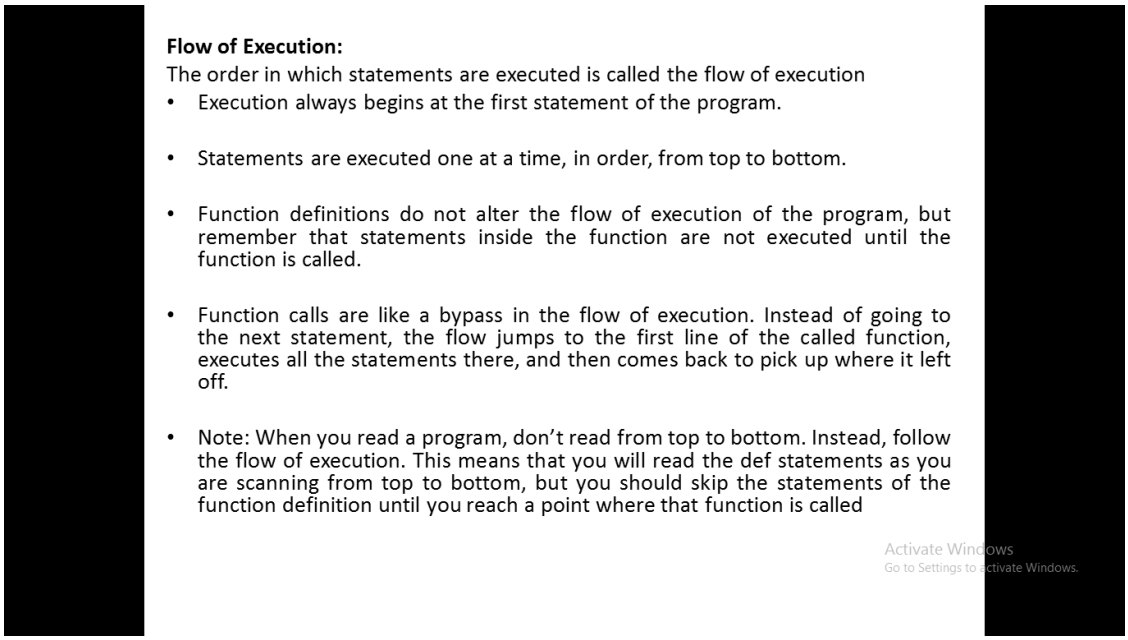



```
# If-else example in Python
num = 21
if num % 2 == 0:
    print("Even Number")
else:
    print("Odd Number")
print()


i = 20
if (i < 15):
    print ("i is smaller than 15")
    print ("i'm in if Block")
else:
    print ("i is greater than 15")
    print ("i'm in else Block")
print ("i'm not in if and not in else Block")
print()


a = 2
b = 330
print("A") if a > b else print("B")
```

```
Odd Number

i is greater than 15
i'm in else Block
i'm not in if and not in else Block

B
```

if-elif-else ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.




```
# if-elif-else  example
num = 1122
if 9 < num < 99:
    print("Two digit number")
elif 99 < num < 999:
    print("Three digit number")
elif 999 < num < 9999:
    print("Four digit number")
else:
    print("number is <= 9 or >= 9999")
print()


i = 30
if (i == 10):
```

```
    print ("i is 10")
elif (i == 15):
    print ("i is 15")
elif (i == 20):
    print ("i is 20")
else:
    print ("i is not present")
```

```
Four digit number

i is not present
```

Python Nested If else statement

In the previous tutorials, we have covered the if statement, if..else statement and if..elif..else statement. In this tutorial, we will learn the nesting of these control statements.

When there is an if statement (or if..else or if..elif..else) is present inside another if statement (or if..else or if..elif..else) then this is calling the nesting of control statements.

```
num = -99
if num > 0:
    print("Positive Number")
else:
    print("Negative Number")
    #nested if
    if -99<=num:
        print("Two digit Negative Number")
print()


x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called


```
# Python program to illustrate
# Iterating over a list
print("List Iteration")
l = [1,2,3,4,5,6]
for i in l:
    print(i)
print()

# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("A","B","C","D")
for i in t:
    print(i)
print()

# Iterating over a String
print("\nString Iteration")
s = "MRCET"
for i in s :
    print(i)
print()

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
    print("% s % d" %(i, d[i]))
```

```
List Iteration
1
2
3
4
5
6


Tuple Iteration
A
B
C
D


String Iteration
M
R
C
E
T


Dictionary Iteration
xyz  123
abc  345
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Continue Statement: It returns the control to the beginning of the loop

Break Statement: It brings control out of the loop

Pass Statement: We use pass statement to write empty loops. Pass is also used for empty control statements, function and classes.

```
# Prints all letters except 'M' and 'R'
for letter in 'MALLAREDDY':
    if letter == 'M' or letter == 'R':
        continue
    print('Current Letter :', letter)
print()

# Breaks the loop after seeing 'A' or 'R'
for letter in 'MALLAREDDY':
    if letter == 'L' or letter == 'R':
```

```
        break
    print('Current Letter :', letter)
print()


# An empty loop
for letter in 'MALLAREDDY':
    pass
print('Current Letter :', letter)
```

```
Current Letter : A
Current Letter : L
Current Letter : L
Current Letter : A
Current Letter : E
Current Letter : D
Current Letter : D
Current Letter : Y

Current Letter : M
Current Letter : A

Current Letter : Y
```

Function range()

In the above example, we have iterated over a list,tuple,string using for loop. However we can also use a range() function in for loop to iterate over numbers defined by range().

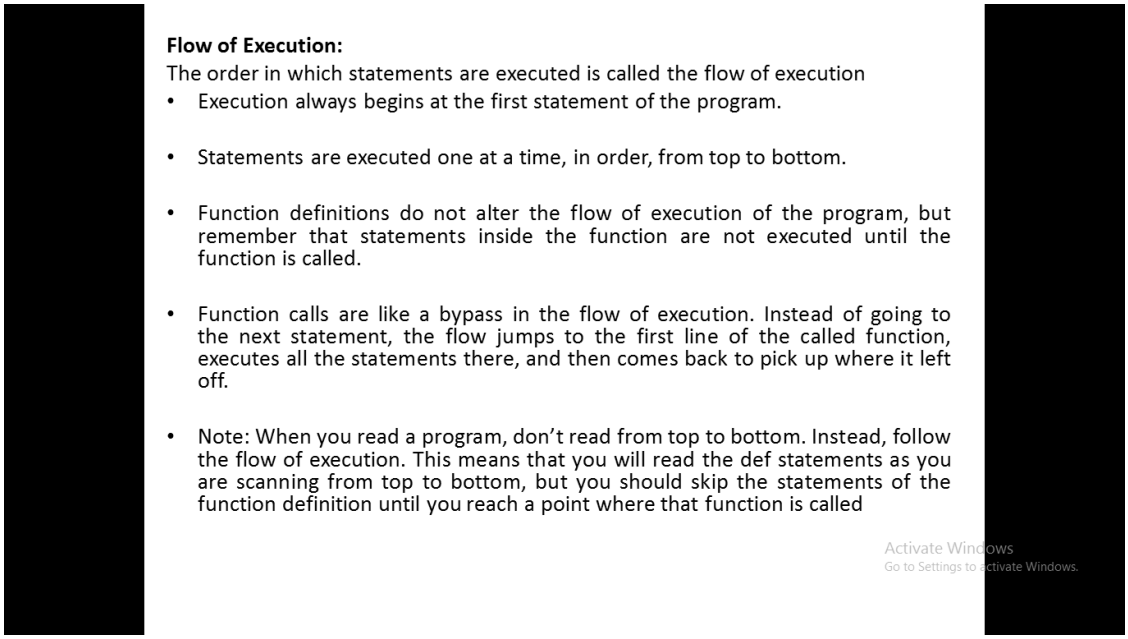
range(n): generates a set of whole numbers starting from 0 to (n-1). For example: range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

range(start, stop): generates a set of whole numbers starting from start to stop-1. For example: range(5, 9) is equivalent to [5, 6, 7, 8]

range(start, stop, step_size): The default step_size is 1 which is why when we didn't specify the step_size, the numbers generated are having difference of 1. However by specifying step_size we can generate numbers having the difference of step_size. For example: range(1, 10, 2) is equivalent to [1, 3, 5, 7, 9]

```
#Python for loop example using range() function

# Program to print the sum of first 5 natural numbers

sum = 0

# iterating over natural numbers using range()
for val in range(1, 6):
      sum = sum + val
```

```
print(sum)
print()


# For loop with else block
for val in range(5):
    print(val)
else:
    print("The loop has completed execution")
print()

#Nested For loop in Python
#When a for loop is present inside another for loop then it is called
a nested for loop.

for num1 in range(3):
    for num2 in range(10, 14):
        print(num1, ",", num2)
print()

# Python Program to show range() basics

# printing a number
for i in range(10):
    print(i, end =" ")
print()

# using range for iteration
l = [10, 20, 30, 40]
for i in range(len(l)):
    print(l[i], end =" ")
print()

# performing sum of natural number
sum = 0
for i in range(1, 10):
    sum = sum + i
print("Sum of first 10 natural number :", sum)
```



Flow of while loop

1. First the given condition is checked, if the condition returns false, the loop is terminated and the control jumps to the next statement in the program after the loop.
2. If the condition returns true, the set of statements inside loop are executed and then the control jumps to the beginning of the loop for next iteration.

These two steps happen repeatedly as long as the condition specified in while loop remains true.

```
num = 1
# loop will repeat itself as long as
# num < 10 remains true
while num < 10:
    print(num)
    #incrementing the value of num
    num = num + 3
print()
#Infinite while loop
"""while True:
   print("hello")

num = 1
while num<5:
   print(num)"""

# Nested while loop in Python
# When a while loop is present inside another while loop then it is
called nested while loop.
```

```python
i = 1
j = 5
while i < 4:
    while j < 8:
        print(i, ",", j)
        j = j + 1
        i = i + 1
print()

# Python – while loop with else block
num = 10
while num > 6:
   print(num)
   num = num-1
else:
   print("loop is finished")
print()

# single statement while block
count = 0
while (count < 5): count += 1; print("Hello MRCET")

#WHILE WITH break statement
#With the break statement we can stop the loop even if the while
condition is true:
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
print()

#With the continue statement we can stop the current iteration, and
continue with the next:
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
print()
```

Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called


```
def my_function():
  print("Hello from a function")

my_function()
```

Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called

Activate Windows
Go to Settings to activate Windows.

```
def printme( str ):
    print(str)
    return
printme("MRCET")
```

Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called


```
def printinfo( name, age ):
  print ("Name: ", name)
  print ("Age ", age)
  return
printinfo(age=50, name="miki")
```

Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called

Activate Windows
Go to Settings to activate Windows.

```
def printinfo( name, age=40):
    print( "Name: ", name)
    print ("Age ", age)
    return
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called


```
def printinfo(*vartuple ):
    print( "Output is: " )
    #print(arg1)
    for var in vartuple:
      print(var)
    return


printinfo( 10 )
printinfo(20,30)
printinfo( 70, 60, 50 )
```

```
def addition(a,b):
    sum=a+b
    return sum
sum1=addition(10,20)
print(sum1)
```

Flow of Execution:

The order in which statements are executed is called the flow of execution

- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.
- Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called


```
#Program to write sum different product and   using arguments with
return value function.
def calculete(a,b):
    total=a+b
    diff=a-b
    prod=a*b
    div=a/b
    mod=a%b
    return total,diff,prod,div,mod
a=int(input("Enter a value"))
b=int(input("Enter b value"))
 #function call
s,d,p,q,m = calculete(a,b)
print("Sum= ",s,"diff= ",d,"mul= ",p,"div= ",q,"mod= ",m)
#print("diff= ",d)
#print("mul= ",p)
#print("div= ",q)
#print("mod= ",m)

#program to find biggest of two numbers using functions.
def biggest(a,b):
    if a>b :
        return a
    else :
        return b


a=int(input("Enter a value"))
b=int(input("Enter b value"))
 #function call
big= biggest(a,b)
```

```
print("big number= ",big)


#program to find biggest of two numbers using functions. (nested if)

def biggest(a,b,c):
    if a>b :
        if a>c :
            return a
        else :
            return c
    else :
        if b>c :
            return b
        else :
            return c


a=int(input("Enter a value"))
b=int(input("Enter b value"))
c=int(input("Enter c value"))
 #function call
big= biggest(a,b,c)
print("big number= ",big)

#Writer a program to read one subject mark and print pass or fail use
single return values function with argument.
def result(a):
    if a>40:
       return "pass"
    else:
        return "fail"
a=int(input("Enter one subject marks"))

print(result(a))

#Write a program to display mrecet cse dept 10 times on the screen.
(while loop)
def usingFunctions():
    count =0
    while count<10:
        print("mrcet cse dept",count)
        count=count+1

usingFunctions()
```