USN | | | | | | | | | | |

Internal Assessment Test I – March 2019

Sub:	Python Application Programming				Sub Code:	15CS664	Branch :	ISE/ECE/TCE/EEE
Date:	07/03/2019	Duration:	90 mins	Max Marks:	50	Sem/Sec :	6th Sem Open Elective	OBE

Answer any **FIVE** FULL Questions | MARKS | CO | RBT

1 (a) List and briefly explain the building blocks of programs [04] CO1 L1

Scheme: Listing all 6 building blocks of a program - 1M
Explaining the 6 blocks – 3M

Solution:

- **A program is a sequence of Python statements that is intended to do something.**
- We use some general constructs while writing a program. These constructs are not just for Python programs, they are part of every programming language from machine language to high-level languages.
  - **Input**: Get data from "outside world". It can be reading data from a file or some kind of sensor like a microphone or GPS. Generally input will be given from the user through keyboard.
  - **Output**: Display the result of the program on the screen or store them in a file or write them to a device like a speaker to play music or speak text.
  - **Sequential Execution**: Statements are executed one after the other in the order they appear in the script.
  - **Conditional Execution**: Check for certain conditions and based on that either execute or skip a sequence of statements.
  - **Repeated Execution**: Repeat some set of statements, usually with some variation.
  - **Reuse**: Write a set of instructions/statements once and give them a name, then reuse those instructions in the program whenever needed.

(b) Bring out the differences between the following [06] CO1 L2

i) syntax errors, logic errors and semantic errors.
ii) interpreter and compiler

Scheme: Syntax Errors – 1M
Logic Error – 1M
Semantic Error – 1M
Interpreter – 1.5M
Compiler 1.5M

Solution:

- **Syntax Errors**: A syntax error means that you have violated the "grammar" rules of Python.

  OR

  Syntax refers to the structure of a program or the rules about that structure.

  Python points right at the line and character where the error occurred. Sometimes the mistake that needs to be fixed is actually earlier in the program than where python noticed. So the line and character that Python indicates in a syntax error may just be a starting point of our investigation. The syntax errors are easy to fix.

  Examples: missing colon, commas or brackets, misspelling a keyword, incorrect indentation and so on.

- **Logic Errors**: A logic error is when the program is syntactically correct but there is a mistake in the order of statements or mistake in how the statements relate to one another. It causes the program to operate incorrectly. It produces unintended or undesired output. These are difficult to fix.

  Examples: indenting a block to a wrong level, using integer division instead of floating-point division, wrong operator precedence and so on.

- **Semantic Errors**: An error in a program that makes it to do something other than what the programmer intended.

  OR

  A semantic (meaning) error is when the program is syntactically correct and in the right order, but there is simply a mistake in the program.

  Example: Colourless green ideas sleep furiously – This is grammatically correct but cannot be combined into one meaning.

  OR

  A semantic error is a violation of the rules of meaning of a natural language or a programming language.

Summary

- ✓ **Syntax relate to spelling and grammar.**

  If the code fails to execute due to typos, invalid names, a missing parenthesis or some other grammatical flaw, you have a syntax error.

- ✓ **Logic relate to program flow.**

  If the syntax is correct but a piece of code is (inadvertently) never executed, operations are not done in the correct order, the operation itself is wrong or code is operating on the wrong data, you have a logical error. Using a wrong conditional operator is a common example, so is inadvertently creating an infinite loop or mixing up (valid) names of variables or functions.

- ✓ **Semantics relate to meaning and context.**

  If both your program logic and syntax is correct so the code runs as intended, but the result is still wrong: you likely have a semantic error.

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

2 (a) Write a program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature. [04] CO1 L3

Scheme: Input – 1M
Formula – 1M
Output – 1M
Sample run – 1M

Solution:

```
#Program to convert temperature from celsius to farenheit.
cel = float(input("Enter temperature in celsius: "))
far = ((9/5)*cel)+32
print("The temperature in farenheit is: ",far)
```

Output:

```
Enter temperature in celsius: 33
The temperature in farenheit is:  91.4
```

(b) Briefly explain how to catch exceptions and short-circuit evaluation of logical expressions is done in python with code snippets.	[06]	CO1	L1

Scheme: try and except syntax – 2M
try and except example – 1M
Short-circuit evaluation of logical expression explain – 1M
Short-circuit evaluation of logical expression explain – 2M

Solution:

- **An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program.**
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- **For example:**

```
# Python program to demonstrate an exception.

a = int(input('Enter a: '))
b = int(input('Enter b: '))

res = a/b

print(res)
```

Output:

I. **Enter a: 10**
Enter b: 5
2.0

II. **Enter a: 4**
Enter b: 0
Traceback (most recent call last):

File"C:/Users/akhil/AppData/Local/Programs/Python/Python36 32/tryexcept.py", line 6, in <module>
res = a/b
ZeroDivisionError: division by zero → **This is an exception**

- **The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.**
- You can think of the try and except feature in Python as an “insurance policy” on a sequence of statements.

- **Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.**
- We can rewrite the above program to handle exceptions as follows:

```
# Python program to demonstrate catching an exception using try and catch.

a = int(input('Enter a: '))
b = int(input('Enter b: '))

try:
   res = a/b
   print(res)
except:
   print('Divide by Zero Exception!!!')
```

Output:

I. **Enter a: 8**
   Enter b: 0
   Divide by Zero Exception!!!

II. **Enter a: 53**
   Enter b: 35
   1.5142857142857142

- **Handling an exception with a try statement is called *catching* an exception.**
- **When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right**. Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.
- **When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression**. **When the evaluation of a logical expression stops because the overall value is already known, it is called short-circuiting the evaluation.**
- While this may seem like a fine point, **the short-circuit behaviour leads to a clever technique called the guardian pattern.**
- Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y)>2
True
>>> x = 1
>>> y = 0
```

```
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

- The third calculation failed because Python was evaluating (x/y) and y was zero, which causes a runtime error.
- But the second example did *not* fail because the first part of the expression x >= 2 evaluated to False so the (x/y) was not ever executed due to the *short-circuit* rule and there was no error.

➢ **We can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:**

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

- In the first logical expression, x >= 2 is False so the evaluation stops at the and.
- In the second logical expression, x >= 2 is True but y != 0 is False so we never reach (x/y).
- In the third logical expression, the y != 0 is *after* the (x/y) calculation so the expression fails with an error.
- In the second expression, we say that y != 0 acts as a *guard* to insure that we only execute (x/y) if y is non-zero.

3 (a)	Briefly explain why we need functions in program? And write the syntax to create functions in python?	[6]	CO1	L1

Scheme: 3 Advantages – 3M
Function syntax – 2M
Example - 1M

Solution:

➢ Creating a new function gives you an opportunity to name a group of statements, which makes your program **easier to read, understand, and debug**.

- **Functions can make a program smaller by eliminating repetitive code**. Later**, if you make a change, you only have to make it in one place.**
- **Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole**.
- Well-designed functions are often useful for many programs. **Once you write and debug one, you can reuse it.**
- **A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.**
- **def** keyword is used to define the function (marks the start of function header).
- A function name to uniquely identify it. **Function naming follows the same <u>rules of writing variable names (identifiers) in Python</u>.**
    - Letters, numbers and some punctuation marks are legal, but the first character can't be a number.
    - You can't use a keyword as the name of a function.
    - Avoid having a variable and a function with the same name.
- **Parameters (arguments) through which we pass values to a function. They are optional**. The **empty parentheses** after the name indicate that this **function doesn't take any arguments**.
- A **colon (:) to mark the end of function header**.
- The first statement of a function can be an optional statement - the **documentation string (docstring) to describe what the function does.**
- **One or more valid python statements that make up the function body.** The body can contain any number of statements.
- **Statements must have same indentation level (usually 4 spaces).**
- An optional return statement to return a value from the function.
- The general syntax of defining a function is:

```
def function_name (parameters/arguments):
        """docstring"""  → Optional
        Statement 1 ⎤
        Statement 2 ⎬          Body of the function
        …           ⎪
        Statement n ⎦
```

For Example:

```
def print_lyrics():
        print('Haan hum badalne lage,  Girne sambhalne lage')
        print('Jab se hai jaana tumhein, Teri ore chalne lage')
```

- If you type a function definition in interactive mode, **the interpreter prints ellipses (. . .) to let you know that the definition isn't complete**. To end the function, you have to enter an empty line (this is not necessary in a script).

```
>>> def print_lyrics():
...     print('Haan hum badalne lage, Girne sambhalne lage')
...     print('Jab se hain jaana tumhein, Teri ore chalne lage')
```

...

Note: The below is an example that shows an error if you are not indenting the statements after the Function. header

```
>>> def print_lyrics():
... print('Jab se hai jaana tumhein, Teri ore chalne lage')
File "<stdin>", line 2
print('Jab se hai jaana tumhein, Teri ore chalne lage')
^
IndentationError: expected an indented block
```

- **Defining a function creates a variable with the same name.**

```
>>> print(print_lyrics)
<function print_lyrics at 0x05916228>
```

- **The value of print_lyrics is a function object, which has type "function".**

```
>>> print(type(print_lyrics))
<class 'function'>
```

- **A function call is a statement that executes a function. It consists of the function name followed by an argument list.**
- The general syntax of calling a function is:

```
function_name(parameters/arguments)
```

- **For Example:**

```
>>> print_lyrics()        → Function call
Haan hum badalne lage, Girne sambhalne lage
Jab se hain jaana tumhein, Teri ore chalne lage
```

- Once you have defined a function, you can use it inside another function.

```
>>> def repeat_lyrics():
...     print_lyrics()    → Function call
...     print_lyrics()    → Function call
...
>>> repeat_lyrics()       → Function call
Haan hum badalne lage, Girne sambhalne lage
Jab se hain jaana tumhein, Teri ore chalne lage
Haan hum badalne lage, Girne sambhalne lage
Jab se hain jaana tumhein, Teri ore chalne lage
```

(b) Explain any 5 built in functions in python with examples. [4] CO1 L1

Scheme: Any 5 built in functions with syntax and example : 4M

Solution:

- Python provides a number of important built-in functions that we can use without needing to provide the function definition.

- The **max** and **min** gives the **largest and smallest element** in the list respectively.

  For Example:

```
>>> max('Hello World')
'r'
>>> min('Hello World')
' '
>>> max(10,45,67,6,-6,348.6,58)
348.6
>>> min(10,45,67,6,-6,348.6,58)
-6
```

  The max function tells us the "largest character" in the string (which turns out to be the letter "r") and the min function shows us the smallest character (which turns out to be a space).

- **len function** which tells us **how many items are in its argument.**

  For Example:

  If the **argument to len is a string, it returns the number of characters in the string**.

```
>>> len('Hello World')
11
>>> list = [10, 325, 52, 25, 55.2, 436]
>>> print(len(list))
6
>>> len(10,45,67,6,-6,348.6,58)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (7 given)
```

- The **int function takes any value and converts it to an integer, if it can, or complains otherwise**:

  For Example:

  - >>> int('32') → **When 32 is passed as string it converts it into integer.**
    32
  - >>> int('hello') → **Gives an error.**

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
```

- **int can convert floating-point values to integers**, but **it doesn't round off; it chops off the fraction part**.

  For Example:

  - >>> int(4.1926345)
    4

- >>> int(-2.6035)
  -2
- >>> int('25')
  25
- >>> int('6.62536') → When 6.62536 (floating-point) is passed as string it Traceback (most recent call last): gives an error.
  File "<stdin>", line 1, in <module>
  ValueError: invalid literal for int() with base 10: '6.62536'

➢ **float** converts integers and strings to floating-point numbers.
For Example:

- >>> float(32)
  32.0
- >>> float('3.1412')
  3.1412
- float('hello')
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: could not convert string to float: 'hello'

➢ **str** converts its argument to a string.
For Example:

- >>> str(32)
  '32'
- >>> str('3.1412')
  '3.1412'

4 (a) Write a program to prompt for a score between 1 to 100. If the score is out of range, print an error message. If the score is between 1 to 100, print the grade using the following table: [10] CO1 L3

Score	Grade
>=90	A
>=80	B
>=70	C
>=60	D
<60	E

Note: Run the program repeatedly till an invalid value is entered.

Scheme: Input - 1M
Score validation – 1M
While loop – 2M
Conditions – 4M
Output – 1M
Sample run – 1M

Solution:

```
#Program to prompt score and find out grade.

while(True):
    score = float(input('Enter the score: '))
    if score < 0 or score > 1.0 :
        print('Error, Score out of range')
        break
    else:
        if score >= 0.9 and score <= 1.0 :
            print('Grade A')
        elif score >= 0.8 and score < 0.9 :
            print('Grade B')
        elif score >= 0.7 and score < 0.8 :
            print('Grade C')
        elif score >= 0.6 and score < 0.7 :
            print('Grade D')
        elif score >= 0 and score <0.6 :
            print('Fail')
```

Output:

```
Enter the score: 0.5
Fail
Enter the score: 0.8
Grade B
Enter the score: 0.9
Grade A
Enter the score: 0.65
Grade D
Enter the score: 1.5
Error, Score out of range
```

5 (a)	Briefly explain the iteration statements in python with code snippets. With an example explain the use of break, continue and pass keywords in iteration and conditional statements.	[10]	CO1	L2

Scheme: while statement syntax and example – 2.5 Marks
for statement syntax and example – 2.5 Marks
break statement syntax and example – 2 Marks
continue statement syntax and example – 2 Marks
pass statement syntax and example – 1 Marks

Solution:

- **A set of statements that are executed repeatedly until a condition is satisfied is called a loop. When a condition is not satisfied then the loop is terminated. It useful in performing repetitive tasks.**
- **Whenever we want to do some task repeatedly we use while statement.**

➢ The syntax of while statement is:

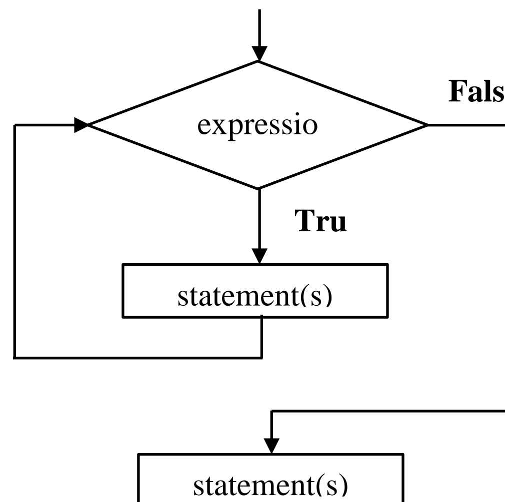
while expression **:**
    Statement 1
    Statement 2
    …..
    Statement n
Statement(s)

➢ The **flow of execution of while statement** is:

i. **Evaluate the condition, which results in either True or False.**
ii. **If the condition is False, exit the while statement and continue execution at the next statement.**
iii. **If the condition is True, then execute the body and then goto step 1.**

- ✓ This type of flow is called **loop**, since the third step loops back to the top.
- ✓ Each time we execute the body of the loop we call it an **iteration**.
- ✓ The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the ***iteration variable***.
- ✓ If there is no iteration variable, the loop will repeat forever, resulting in an ***infinite loop***.

➢ The flowchart of while statement is:



➢ Consider a simple program that counts from 5 to 1.

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print("Good Bye!!!")
```

Output:

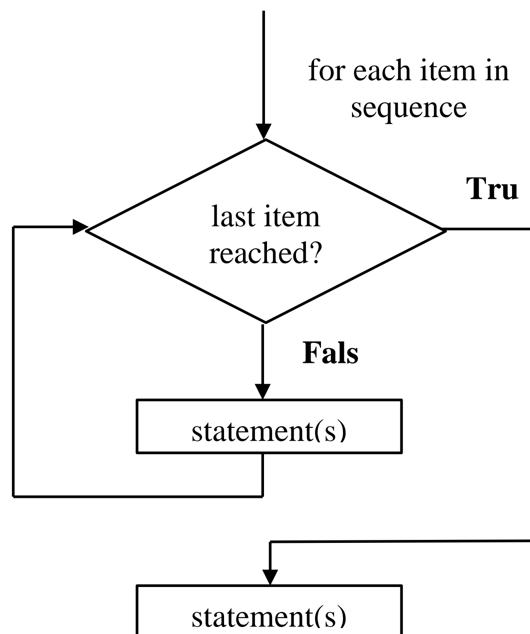
5
4
3
2
1
Good Bye!!!

While n is greater than 0, display the value of n and then decrement the value of n by 1. When you get to 0, exit the while statement and display “Good Bye!!!”.

- **The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.**
- The syntax of for loop is:

```
for variable_name in sequence:
        Statement 1
        Statement 2
        …..
        Statement n
Statement(s)
```

  - Here, variable_name is the variable that takes the value of the item inside the sequence on each iteration.
  - Loop continues until we reach the last item in the sequence.
  - The body of for loop is separated from the rest of the code using indentation.
- The flowchart of while statement is:



- Consider a program to print each character in a word.

```
for letter in 'python':
   print(letter)
```

Output:

p
y
t
h
o
n

- Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.
- Break statement is a **jump statement which is used to transfer execution control**.
- It **terminates the current loop** and **resumes execution at the next statement**.
- The general syntax of break statement is:

  break
- Consider the following examples:
  - ```
    for var in 'python':
       if var == 't':
          break
       print(var)
    print('End of program!!!')
    ```
    Output:
    ```
    p
    y
    End of program!!!
    ```
  - ```
    for i in range(10):
       if i == 5:
          break
       print(i)
    print('End of program!!!')
    ```
    Output:
    ```
    0
    1
    2
    3
    4
    End of program!!!
    ```
  - ```
    #To check if 11 is prime or not.
    n=11
    flag = 0
    for i in range(2,n):
       if n%i == 0:
          flag = 1
          break
    if flag == 1:
       print('Not Prime number')
    else:
       print('Prime number')
    ```
    ```

Output:

Prime number

- The **continue statement is used to skip the rest of the code inside a loop for the current iteration only**. **Loop does not terminate but continues on with the next iteration**.
- The general syntax of continue statement is:

  continue

- Consider the following examples:
  - ```
    for var in 'python':
       if var == 't':
          continue
       print(var)
    print('End of program!!!')
    ```

    Output:

    ```
    p
    y
    h
    o
    n
    End of program!!!
    ```

  - ```
    for i in range(10):
       if i == 5:
          continue
       print(i)
    print('End of program!!!')
    ```

    Output:

    ```
    0
    1
    2
    3
    4
    6
    7
    8
    9
    End of program!!!
    ```

Note: Loop control/Jump statements can be used only within the loops.

- **Sometimes, it is useful to have a body with no statements (usually as a place holder for code you haven't written yet). In that case, you can use the pass statement, which does nothing.**

  For example:

  ```
  if n < 0 :
  ```

pass # need to handle negative values

6 (a)	Write a program to prompt the user to enter a list of values (eg: [3,41,12,9,74,15]), where these values are passed as a parameter for minimum and maximum functions. Process the list and display the minimum value and maximum value in the list of values. **Note:** Do not use built in functions for get min and max values.	[10]	CO1	L3

Scheme: Input – 2M
Min user defined function – 2M
Max user defined function – 2M
Function call – 2M
Output – 1M
Sample runs – 1M

Solution:

```
#Function definition to find minimum number in a list.

def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest

#Function definition to find maximum number in a list.

def max(values):
    largest = None
    for value in values:
        if largest is None or value > largest:
            largest = value
    return largest
num = [25, 58, 78, 19, -84, 67, 412, -15]

min_no = min(num)
print("Minimum number is the list is: ", min_no)
max_no = max(num)
print("Maximum number is the list is: ", max_no)
```

Output:

Minimum number is the list is: -84
Maximum number is the list is: 412

7 (a)	Explain the conditional statements, chained conditionals statements, nested conditional statements with code snippets.	[06]	CO1	L3

Scheme: Conditional statement syntax – 1M
Conditional statement explain and example – 1M
Chained conditional statement syntax – 1M

Chained conditional statement explain and example – 1M
Nested conditional statement syntax – 1M
Nested conditional statement explain and example – 1M

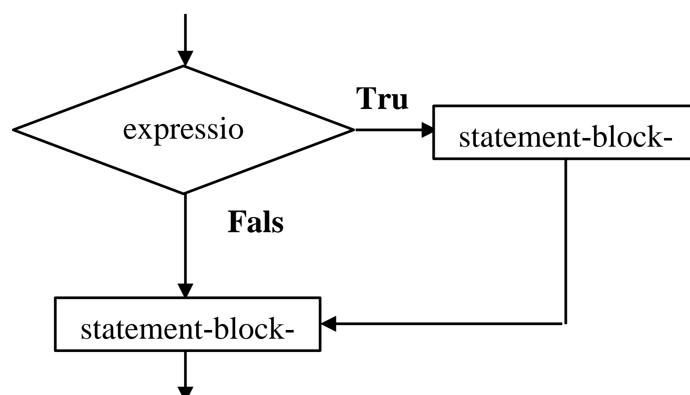
Solution:

- **Conditional statements gives the ability to check conditions change the behaviour of the program accordingly**.
- Decision making is required when we want to execute a code only if a certain condition is satisfied.
- The syntax of if-statement is:

```
if expression :
        statement-block-1
statement-block-2
```

The boolean expression after the if statement is called the *condition*. We end the if statement with a colon character (:) and the line(s) after the if statement are indented.

- **The expression is evaluated to either True or False**.
    - **If True then the intended statements (statement-block-1) get executed** and the control comes outside the if statement and the execution of further statements (statement-block-2) continues if any.
    - **If the expression is evaluated to be false, then intended block (statement-block-1) is skipped.**
- **There is no limit on the number of statements that can appear in the body, but there must be at least one.**
- **Note: Parenthesis can be used for the expression in the if statement but it is optional.**
- The flowchart is shown below:



- **Python interprets non-zero values as `True`. `None` and 0 are interpreted as `False`.**
- **# Python program to check if a given number is positive.**

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

```
if n > 0 :
    print('Positive')

print('Bye')
```

Output:

I. **Enter a number: 10**
Positive
Bye

II. **Enter a number: -74**
Bye

III. **Enter a number: 0**
Bye

- It is multi-way selection statement. **It is used when we have to make a choice among many alternatives**. One way to express a computation like that is a *chained conditional.*
- The syntax of else-if statement is:

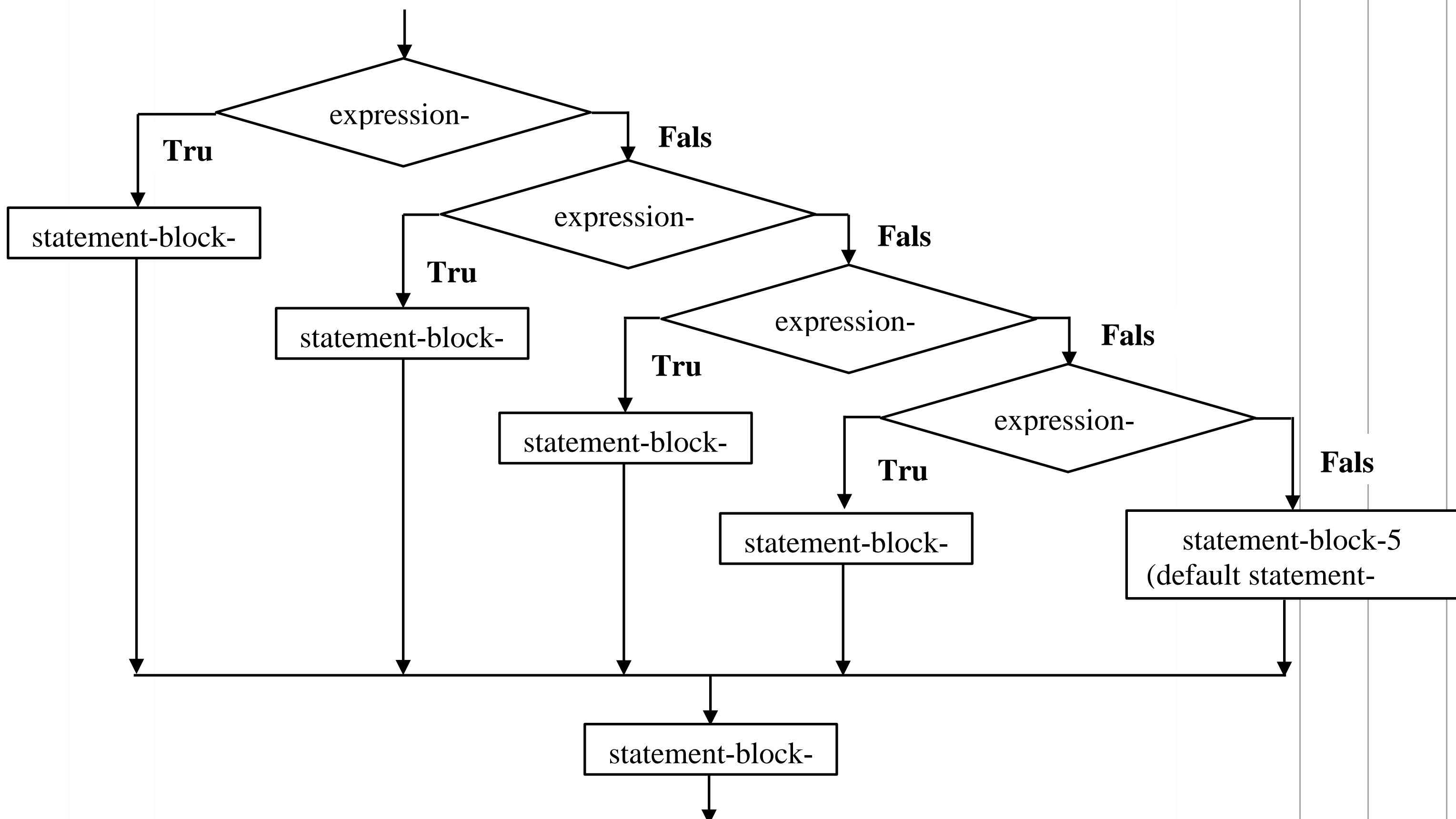
```
if expression-1 :
        statement-block-1
elif expression-2 :
        statement-block-2
elif expression-3 :
        statement-block-3
elif expression-4 :
        statement-block-4
elif expression-5 :
        statement-block-5
statement-block-6
```

Note: elif is an abbreviation of "else if"

- **The expression is evaluated in top to bottom order. If an expression is evaluated to true, then the statement-block associated with that expression is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.**
- **There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.**
- For example:
    - If expression-1 is evaluated to true, then statement-block-1 is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.

- If expression-1 is evaluated to false, then expression-2 is evaluated. If expression-2 is evaluated to true then statement-block-2 is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.
- If all the expressions are evaluated to false, then the last statement-block-5 (default) is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.

➢ The flowchart is shown below:



➢ **# Python program to check if a number is positive, negative or zero.**

```
n = int(input('Enter the number: '))

if n > 0 :
   print('The number',n,'is Positive')
elif n < 0 :
   print('The number',n,'is Negative')
else :
   print('The number',n,'is Zero')

print('Good Bye')
```

Output:

I. **Enter the number: 0**
   The number 0 is Zero
   Good Bye

II. **Enter the number: 73**
   The number 73 is Positive
   Good Bye

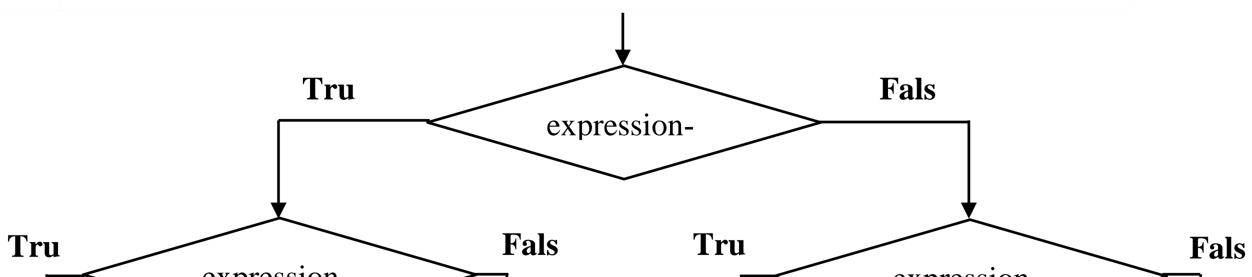
III. **Enter the number: 0**
   The number 0 is Zero
   Good Bye

- It is used when an action has to be performed based on many decisions.
- **An if-else statement within another if-else statement is called nested if-else statement.**
- The syntax of nested if-else statement is:

```
if expression-1 :
      if expression-2 :
            statement-block-1
      else :
            statement-block-2
else
      if expression-3 :
            statement-block-3
      else :
            statement-block-4
statement-block-5
```

- The expression-1 is evaluated to true or false.
  - If expression-1 is evaluated to true, then expression-2 is evaluated to true or false. If expression-2 is evaluated to true, then statement-block-1 is executed. If expression-2 is evaluated to false, then statement-block-2 is executed.
  - If expression-1 is evaluated to false, then expression-3 is evaluated to true or false. If expression-3 is evaluated to true, then statement-block-3 is executed. If expression-3 is evaluated to false, then statement-block-4 is executed.

  In either of the cases, after execution of a particular statement-block the control comes outside the nested if-else statement and continues execution from statement-block-5.
- Although the indentation of the statements makes the structure apparent, *nested conditionals* become difficult to read very quickly. In general, it is a good idea to avoid them when you can.
- The flowchart is shown below:



- **# Python program to check if a number is lesser, greater or equal to another number.**

```
x = int(input('Enter X: '))
y = int(input('Enter Y: '))

if x == y :
  print('X and Y are Equal')
else :
  if x < y :
    print('X is lesser than Y')
  else :
    print('X is greater than Y')
```

Output:

I. **Enter X: 6**
   Enter Y: 9
   X is lesser than Y
II. **Enter X: 84**
   Enter Y: 62
   X is greater than Y
III. **Enter X: 5**
   Enter Y: 5
   X and Y are Equal

(b)	Briefly explain few built in functions in math module and random module with examples. **Scheme: math module – 2M** **random module – 2M** **Solution:**	[04]	CO1	L1

```
import math

#math.ceil(x) function.
print('Ceiling function:', math.ceil(23.56))
print('Ceiling function:', math.ceil(-23.56))
print('Ceiling function:', math.ceil(12.32))
print('Ceiling function:', math.ceil(-12.32))

print('\n')

#math.floor(x) function.
print('Floor function:', math.floor(23.56))
print('Floor function:', math.floor(-23.56))
print('Floor function:', math.floor(12.32))
print('Floor function:', math.floor(-12.32))

print('\n')

#math.factorial(x) function.
print('Factorial Function:', math.factorial(15))
#print('Factorial Function:', math.factorial(5.2))    --> Gives a valueError exception
#print('Factorial Function:', math.factorial(-75.34)) --> Gives a valueError exception

print('\n')

#math.fabs(x) function.
print('Fabs Function:', math.fabs(-876.54))
print('Fabs Function:', math.fabs(-7235))

print('\n')

#math.gcd(x,y) function.
print('Greatest Common Divisor:', math.gcd(10,24))
#print('Greatest Common Divisor:', math.gcd(12.30,24))   --> Gives an error
print('Greatest Common Divisor:', math.gcd(-10,24))
```

```
print('\n')

#math.exp(x) function i.e., e**x.
print('Exponent function:', math.exp(34))
print('Exponent function:', math.exp(-67.45))

print('\n')

#math.pow(x,y) function i.e., x**y.
print('Power function:', math.pow(2,31))
print('Power function:', math.pow(2,-8))
print('Power function:', math.pow(12.5,53.5))

print('\n')

#math.sqrt(x) fubction.
print('Square root function:', math.sqrt(625))
#print('Square root function:', math.sqrt(-3984))    --> Gives ValueError exception
print('Square root function:', math.sqrt(2467.38))

print('\n')

print('Sin function:', math.sin(64.47))
print('Cos function:', math.cos(64.47))
print('Tan function:', math.tan(64.47))

print('\n')

#math.log2(x) function.
print('Log2 function:', math.log2(25))
#print('Log2 function:', math.log2(-25.8))  --> Gives ValueError exception
print('Log2 function:', math.log2(43.769))

print('\n')

#math.log10(x) function.
print('Log10 function:', math.log10(25))
#print('Log10 function:', math.log10(-25.8))  --> Gives ValueError exception
print('Log10 function:', math.log10(43.769))

print('\n')
```

```
print('Degrees to radians:', math.radians(90))
print('Radians to degrees:', math.degrees(1.5707))

print('\n')

#Constants
print('The value of pi:', math.pi)
print('The value of e:', math.e)
print('The value of tau:', math.tau)
```

Output:

```
Ceiling function: 24
Ceiling function: -23
Ceiling function: 13
Ceiling function: -12

Floor function: 23
Floor function: -24
Floor function: 12
Floor function: -13

Factorial Function: 1307674368000

Fabs Function: 876.54
Fabs Function: 7235.0

Greatest Common Divisor: 2
Greatest Common Divisor: 2

Exponent function: 583461742527454.9
Exponent function: 5.0913997350542235e-30

Power function: 2147483648.0
Power function: 0.00390625
Power function: 4.8382209306170583e+58

Square root function: 25.0
Square root function: 49.67272893651002

Sin function: 0.9977328054584116
```

Cos function: -0.06729969473992774
Tan function: -14.825220371563946

Log2 function: 4.643856189774724
Log2 function: 5.451837517668948

Log10 function: 1.3979400086720377
Log10 function: 1.6411666243046132

Degrees to radians: 1.5707963267948966
Radians to degrees: 89.9944808811984

The value of pi: 3.141592653589793
The value of e: 2.718281828459045
The value of tau: 6.283185307179586

- Python offers random module that can generate random numbers.

1. For Integers:

Syntax :

random.randrange (start(opt), stop, step(opt))

Parameters :

start(opt) : Number consideration for generation starts from this, default value is 0. This parameter is optional.

stop : Numbers less than this are generated. This parameter is mandatory.

step(opt) : Step point of range, this won't be included. This is optional. Default value is 1.

Return Value :

This function generated the numbers in the sequence start-stop skipping step.

Exceptions :

Raises ValueError if **stop <= start** and number is **non- integral**.

- ✓ **random.randrange(stop)**

```
import random

print(random.randrange(30))
```

Output:

I. **5**
II. **2**
III. **21**

IV. **27**

- ✓ **random.randrange(*start*, *stop*[, *step*])**

  Return a randomly selected element from range(start, stop, step).

  Note: [] means optional.

  Program

```
import random

# Using randrange() to generate numbers from 50-100
print ("\nRandom number from 50-100 is : ")
print (random.randrange(50,100))

# Using randrange() to generate numbers from 50-100 skipping 5
print ("\nRandom number from 50-100 skip 5 is : ")
print (random.randrange(50,100,5))

#Throws exception
#print (random.randrange(50,10))
#print (random.randrange(50.3,100))
```

  Output:

  I. **Random number from 50-100 is :**
  95

  Random number from 50-100 skip 5 is :
  75

  II. **Random number from 50-100 is :**
  60

  Random number from 50-100 skip 5 is :
  55

- ✓ **random.randint(a,b)**

  Return a random integer N such that $a <= N <= b$

```
import random

print(random.randint(1,100))
```

  → Inclusive of 1 and 100.

Output:

I. 96
II. 76
III. 15

2. For Real/Floating:

✓ **random.random()**

Return the next random floating point number in the range (0.0, 1.0) i.e., excluding 1.0.

```
import random

print(random.random())
```

Output:

I. 0.48644020748452865
II. 0.9207854756869541
III. 0.8515254897126728

✓ **random.uniform(a,b)**

Return a random floating point number *N* such that a <= N <= b for a <= b and b <= N <= a for b < a.

The end-point value b may or may not be included in the range depending on floating-point rounding in the equation a + (b-a) * random().

```
import random

print(random.uniform(1,100))
```

Output:

I. 51.433547569607704
II. 90.0503177294139
III. 14.294244323244204

3. For Sequences

✓ **random.choice(seq)**

Return a random element from the non-empty sequence *seq (*list, tuple or string). If *seq* is empty, raises Index Error.

Program:

```
import random

color_list = ['Red', 'Blue', 'Green', 'White', 'Black']
print(random.choice(color_list))

print(random.choice([1, 2, 3, 4, 5,6]))

print(random.choice('Python Application Programming'))

my_list = [2, 109, False, 10, "Lorem", 482, "Ipsum"]
print(random.choice(my_list))
```

Output:

I. Black
   1
   y
   Lorem
II. Green
   5
   m
   10
III. White
   1
   y
   482