Mc Graw Hill

Because learning changes everything.®

TITLE: Python Programming: Problem Solving, Packages and Libraries

Edition

Lecture PPT: Chapter 6 Flow Control

Flow and need for flow control

The term "flow control" in Python refers to the "order" in which the Python statements in a script are executed.

Need:

- **Normally in a script, the lines of code are executed sequentially, that is, one after another. But in many situations, one needs to change the "sequential execution" of statements.**
- **Some common situations where one may have to change the "normal or sequential" flow can be as follows:**
  - **Function calls (Functions have already been discussed)**
  - **Conditional statements or branches (In conditional statements, a block of code is executed depending upon a specific condition)**
  - **Iterations or loops (In iterations, a block of code is repeatedly executed)**

Learning objectives

- Understand “flow control” in a program.
- Identify and use the various types of “conditional statements”, that is, (i) if-elif-else(ii)if-else (iii) Nested if and (iv) while loops.
- Explain why too deep nesting may cause problems and how to avoid/ reduce nesting.
- Use (i) break (ii) continue and (iii) pass
- Decide when to use “while” loop and when to use “for” loop.
- Appreciate the use of “range” function and its use in looping.
- Comprehend the significance of infinite loops.
- Discuss some common errors in flow control and ways to avoid them.

Using “if”

The “if” statement can be used in a number of ways. Some common usages are:

- if -- elif – else
- if -- else (Without elif)
- Omitting the else clause (that is, using an ‘if’ without an ‘else’)
- Nested if -- else Statements
- How to avoid nested if (Good programming technique)

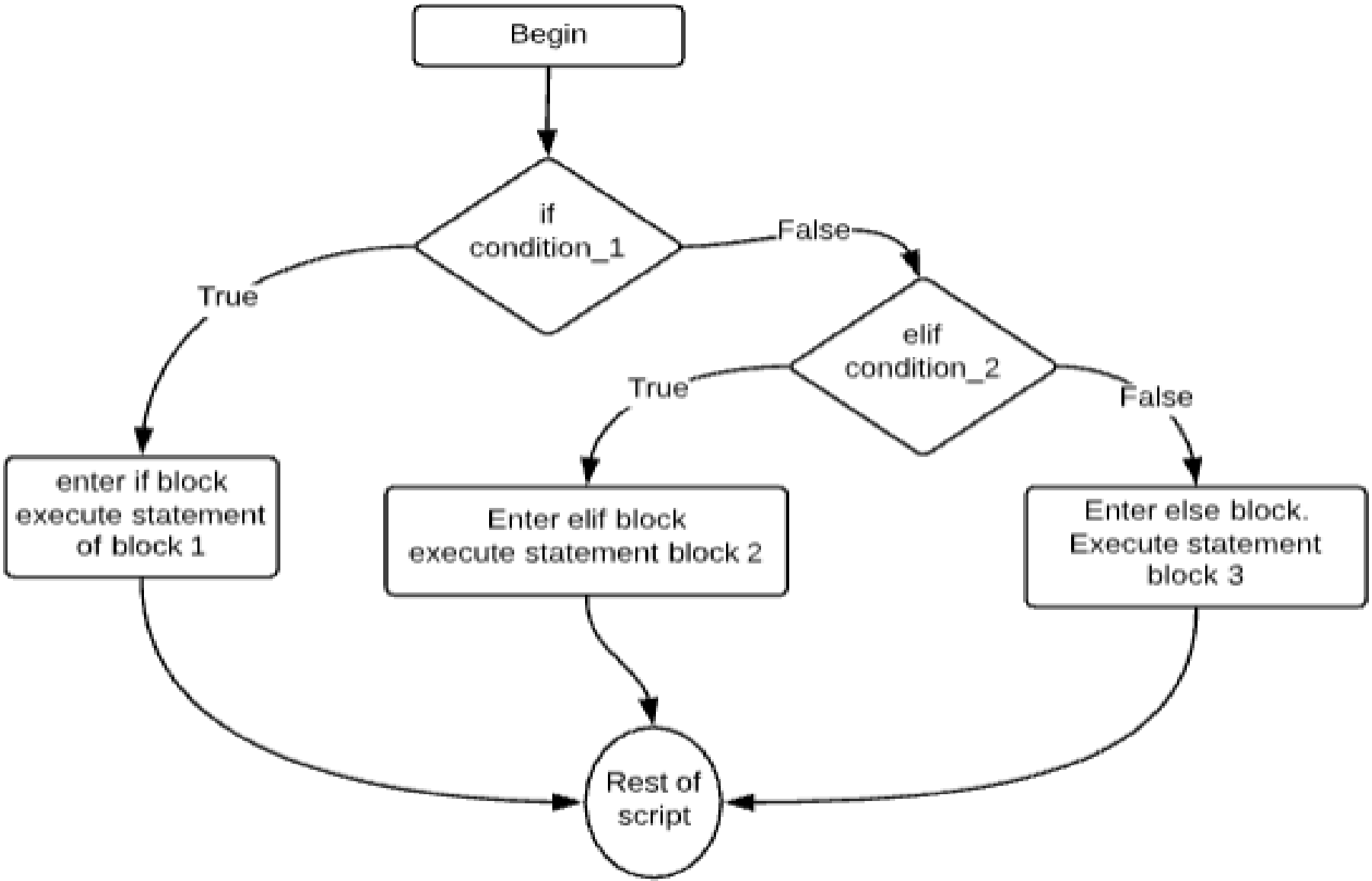
Usage of “if --elif -- else”

(See the pseudo-code below)

```
1  if condition_1:# condition_1 must be a Bool expression and eval to True or False
2      statement_block_1
3  elif condition_2: # condition_2 also must be a bool expression
4      statement_block_2
5  else:
6      statement_block_3
```

- The ‘if’ condition and the ‘elif’ conditions must all be boolean expressions. So all these expressions must evaluate to either True or False.
- This evaluation of a statement to either True or False is also called “boolean context”.
- If the condition "condition_1" is True, the statements in the block statement_block_1 will be executed.
- In Python, it is the indentation which indicates the beginning of a “block or body” of code.
- Further, the “first un-indented” statement indicates the end of a block of code.
- If condition_1 evaluates to False, then condition_2 will be evaluated. If condition_2 evaluates to True, statement_block_2 will be executed. But if condition_2 is False, then the statements in statement_block_3 will be executed.

Flow chart showing how “if --elif -- else” works



Working of "**if – else**" statement

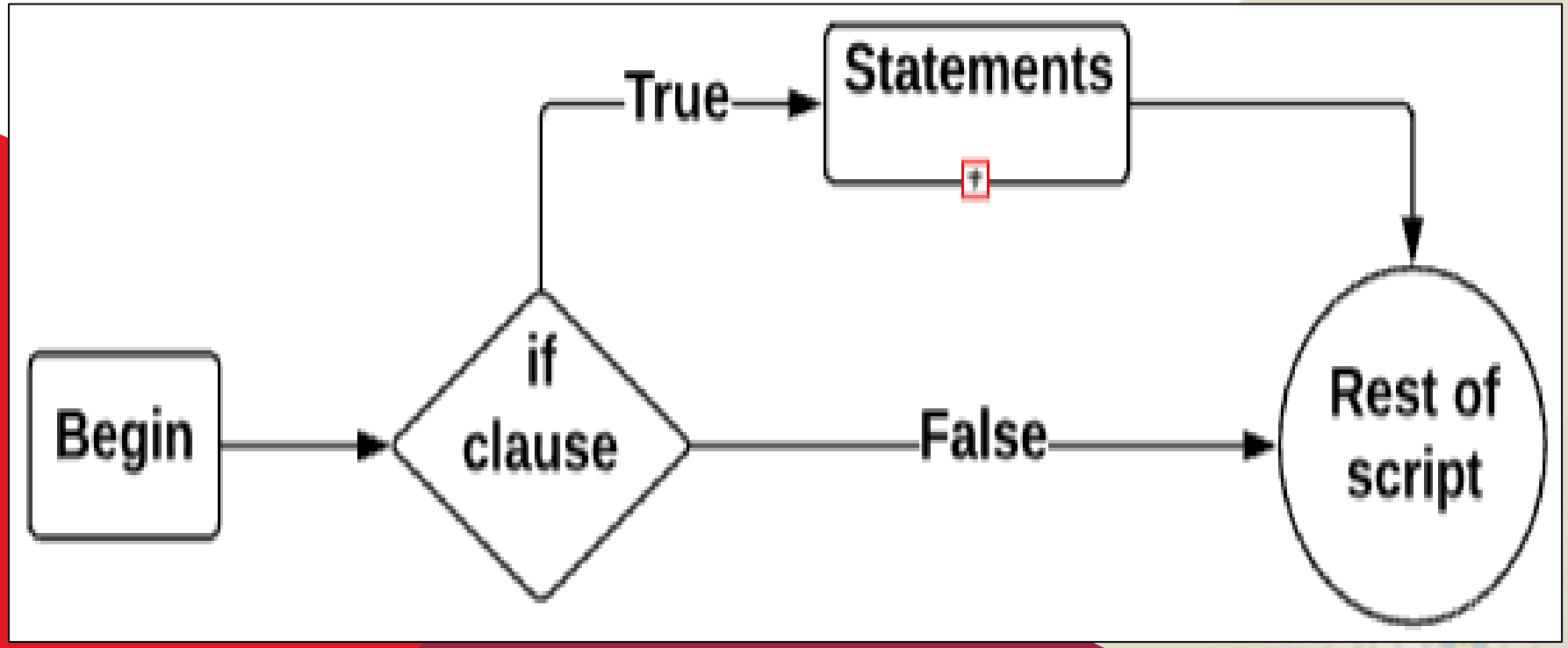
- **First, it evaluates the boolean test expression. If this expression is True, then the "if" block is executed.**
- **However if the "if" boolean expression evaluates to False, then the entire "if" block is skipped and the entire "else" block is executed.**
- **Note that the "if" block has a "boolean expression", but the "else" block does not have any expression.**

Omitting the else clause (that is, using an 'if' without an 'else')

One can use an "if" statement without an "else" block. The scheme works as follows:

- If the "if" boolean expression evaluates to True, then the "if" block is executed and the rest of the script is also executed.
- However, if the "if" boolean expression evaluates to False, then the "if" block is not executed and only the "rest of the script is executed".
- So, in conclusion, if the "if" boolean expression is True then both the "if" block as well as the "rest of the script" are both executed. But if the "if" boolean expression is False then only the "rest of the script" is executed.

Omitting the else clause shown as a figure



The “while” loop works as follows:

- **At first the “boolean test expression” is checked.**
- **If the test expression is False, the “while loop” is not entered at all and the program moves to the next line of code after the “while loop”.**
- **However, if the test expression evaluates to True, then the “while loop” is entered and one iteration of the entire “while loop” is executed.**
- **After completion of one iteration, the expression of the “while loop” is again tested. If the test expression evaluates to False, then the while loop is not entered again. However, if the “while loop” test expression again evaluates to True, then the “while loop” is executed again. This process goes on until the test expression of the ‘while’ loop evaluates to False.**
- **If the “while boolean expression” does not evaluate to False ever, then you will get an “infinite loop, that is, a loop from which the script cannot exit.**
- **If initially the boolean expression of the “while loop” evaluates to False, then the “while loop” is not run even once.**
- **Like in all other cases, the beginning of the “while loop” is marked by a level of indentation and the end of the “while loop” is marked by the “first un-indented” line.**

The following points regarding the "break" statement are noteworthy:

- **"break" statements are used inside "loops" or "nested loops".**
- **In case the "break" statement is in a loop which is not nested, then the "break" statement terminates the loop containing the "break" statement. So, the control of the program will pass to the statement immediately after the body of the loop containing the "break" statement.**
- **In case the "break" statement is used in "nested loops", then the inner loop containing the "break" statement will terminate. Further, the control of the program flows to the loop containing the inner loop.**

“continue” statements

The following points regarding the “continue” statement are noteworthy:

- **The “continue” statement is useful where you don’t want to terminate a loop but only want to “skip” the rest of the code in a loop.**
- **So with the “continue” statement, the loop does not terminate, only the “remaining statements” are skipped.**

“pass” statement

Some important aspects relating to the “pass” statement are as follows:

- **The “pass” statement is basically a “null” statement, that is, nothing happens when a “pass” statement is executed. So a “pass” statement is a no-operation, that is, “NOP”.**
- **The “pass” statement is basically used in programming as a “place holder”. A “place holder” in programming refers to a piece of code which has not been implemented.**
- **Note that a “pass” statement is different from a “comment”. The Python interpreter ignores comments entirely. However, the interpreter does not ignore the “pass” statement. The interpreter simply does not do anything when it comes across a “pass” statement.**

The need for a place holder like “pass”

A question which often arises is, ‘Why does one need a ‘place holder’ like the “pass” statement?

- **The answer is that many blocks of code in Python cannot be empty.**
- **For instance, if you don’t give an indented block after say an “if” or a “while” or “def” or a “class”, then the interpreter will throw an error.**
- **You cannot use a comment as an indent block.**

A ‘for’ loop is a definite loop whereas a ‘while’ loop is an indefinite loop. A ‘while’ loop is an indefinite loop because it simply loops till a condition becomes False. A ‘for’ loop , on the other hand, runs as many times as there are items in the set. The general format of a ‘for’ loop is as follows:

```
1 for each_item in mySet: #mySet is some collection of items
2     #do something
3 else:
4     #do something else
```

When to use ‘while’ and when to use ‘for’ loop

Why have two kinds of loops, that is, ‘while’ and ‘for’?

Suggestions for using a ‘for’ loop:

- **You are already aware of the maximum number of times that you’ll need to execute the body of the loop.**
- **or else you have a collection (Such as a string, a list, dictionary, tuple etc) and you want to go over the individual items in the collection.**
- **The use of ‘for’ loop is called a definite iteration because there are some definite boundaries.**

Suggestions for using a ‘while’ loop:

- **If you need to repeatedly do some computations until some condition is met, and you have no way of knowing in advance as to when this will happen, then a ‘while’ loop is a better idea.**
- **A ‘while’ loop is probably not a good idea for iterating or stepping over the items in a collection.**
- **A “while” loop is called indefinite iteration because you don’t know in advance what are the boundaries and how many iterations will be required.**

The function range([*start*], *stop*[, *step*])

Note:

- **The square brackets indicate that the parameter is optional. Hence, if you have say range(10), it means that the parameters, start and step, have been omitted. The default values of these are as follows:**
- **- for start it is 0 and**
- **- for step it is 1**
- **Hence, range(10) is the same as range(0,10) and the same as range(0,10,1).**
- **This means, if one parameter is given it will be presumed to be 'stop'.**
- **If two parameters are given they will be presumed to be 'start' and 'stop'.**
- **With a single argument, range gives integers from zero up to but not including the argument's value.**
- **If one gives two arguments to the range() function, then the first is taken as the lower bound. An optional third argument can give a step; if it is used.**

Form of range	Details
range(n)	Creates sequence from 0 to n-1 with a step of 1
range(m,n)	Creates sequence from m to n-1 with a step of 1
range(m,n,k)	Creates sequence from m to n-1 with step k.

The next 3 slides show 12 important points regarding the range() function -- 1

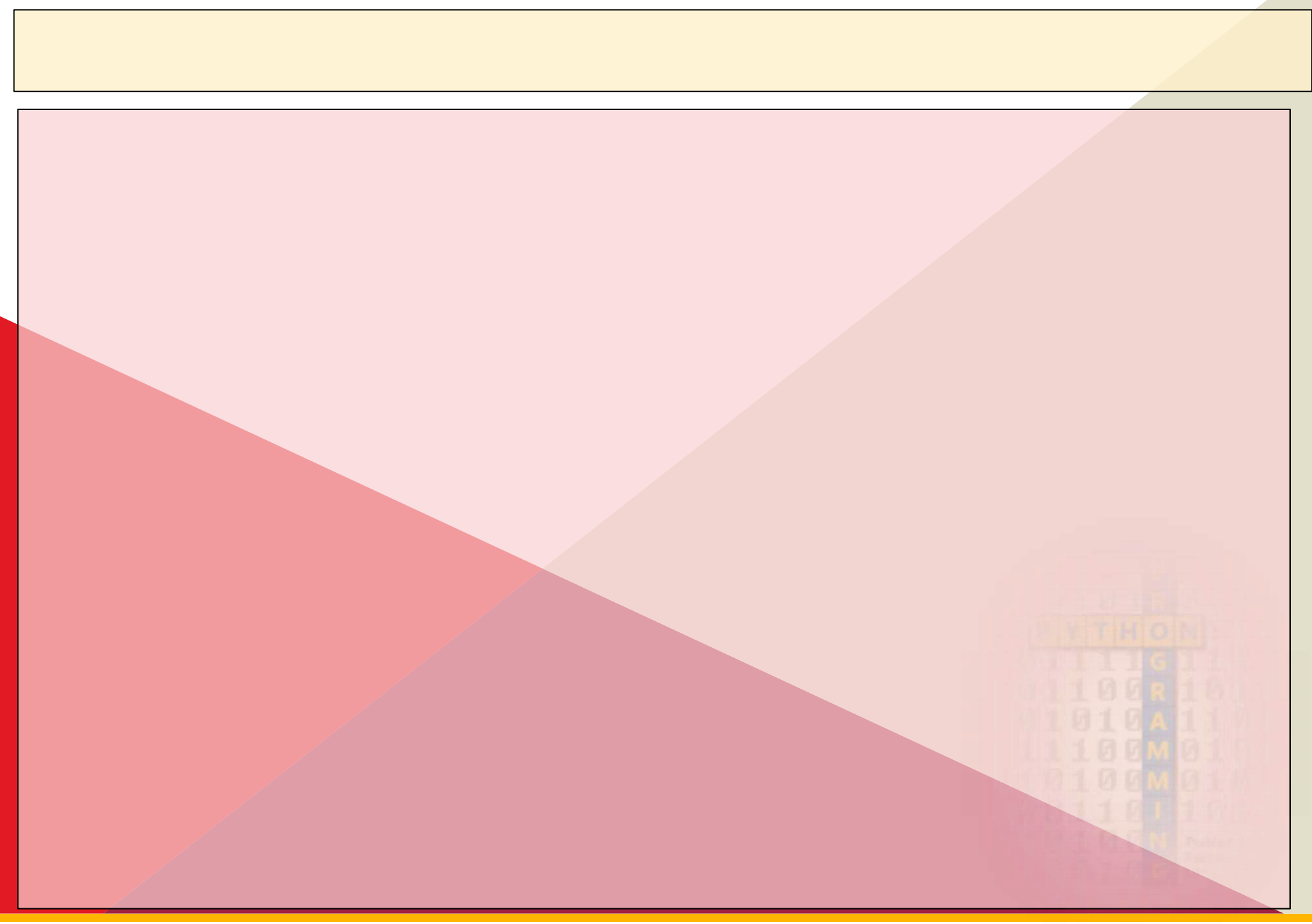
1. The range() function can at most take three arguments, which must all be integers. These three arguments are called “start”, “stop” and “step”. Out of these three, “start” and “step” are optional but the “stop” argument is mandatory.
2. If the “start” argument is omitted, it gets a default value of 0.
3. If the “step” argument is omitted, then it will get a default value of 1.
4. So the range function can be written as:- range([*start= 0]*, *stop*[, *step= 1*]). The “start = 0” and “step = 1” indicate the default values for these two arguments.
5. The full form returns a list in the following format: [start, (start + step), (start + 2 * step), ... (Upto but not including stop)].

range() function -- 2

6. If the “step” parameter has a positive *value*, and start < stop, then the contents of a range x are determined by the formula x[i] = start + step*i where i >= 0 and x[i] <stop.
7. Note that the “step” can be negative also. If it is negative, then the progression will be a decreasing progression, that is, each successive element is smaller than its predecessor. If the “step” is negative, the formula for determining the members of the range() function output is:- x[i] = start + step*i. However here x[i] > stop. So if step is negative, the last element is the smallest start + i * step greater than stop. (Will be clear from example below)
8. You can omit the parameter “step” but “step” cannot be o. If you use 0 for “stop”, it will lead to ValueError.

range() function -- 3

9. If step is positive, then one can never reach stop. So, the range will give an empty sequence. This will be clear from example that follows.
10. Similarly, if 'start' < 'stop' is and the 'step' is negative, then also one can never reach 'stop'. So, here also the range will give empty sequence. This will be clear from the example.
11. The range() function can be used to create an "arithmetic progression". The term "arithmetic progression" is used for a sequence of numbers such that the difference between any two successive elements is the same.
12. The range() function is generally used in "for" loops.



Because learning changes everything.®

Thank You!

For any queries or feedback contact us at:

support.india@mheducation.com

1800-103-5875

www.mheducation.co.in