Python  Programming

Day 7: Control Statement

Control Statement

if	for
elif	while

Color and symbol meaning

Hint

Preferred

Student's activity

Code to run

	Keyword
	In-built modules
	Strings
	Output

Conditional Execution

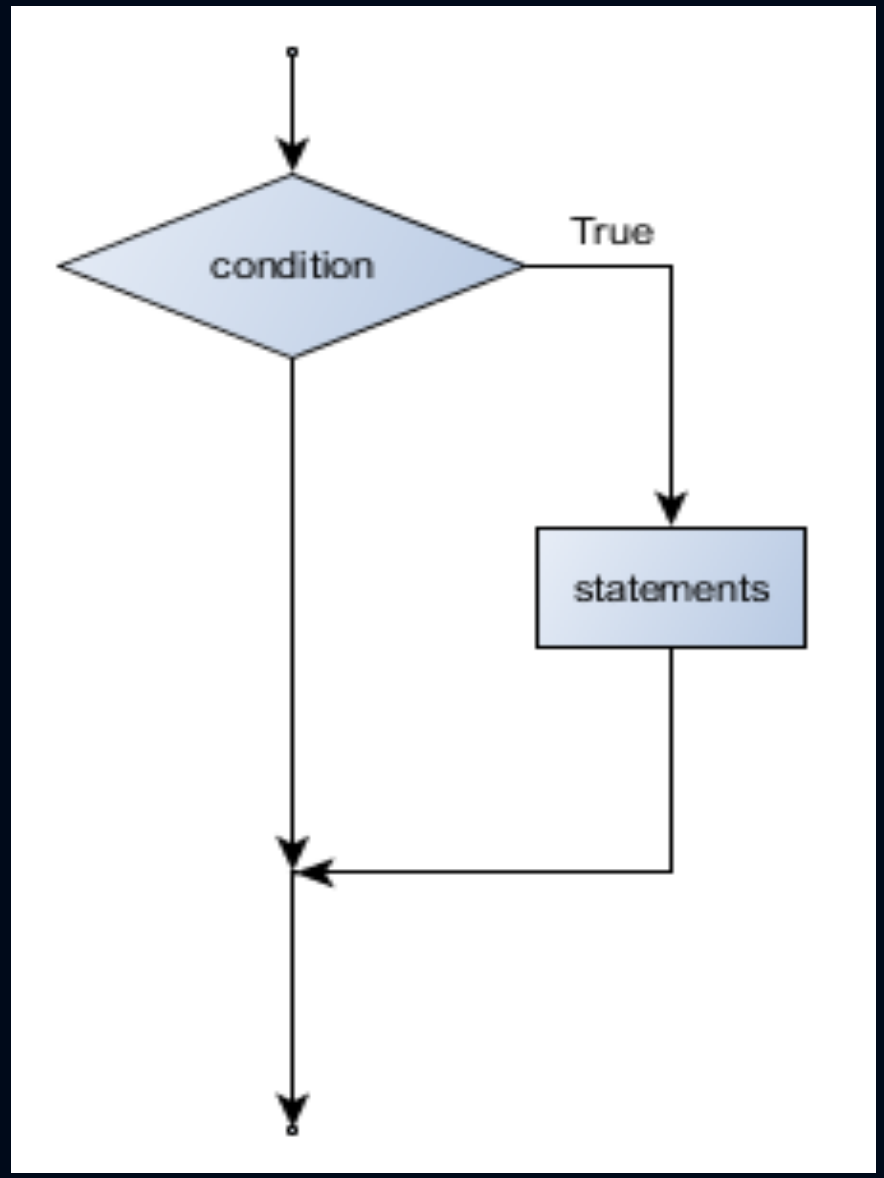
Conditional statements give us the ability to check conditions and change the behaviour of the program accordingly.

The simplest form is the ** **if statement**:

If statement

The syntax for an if statement looks like this:

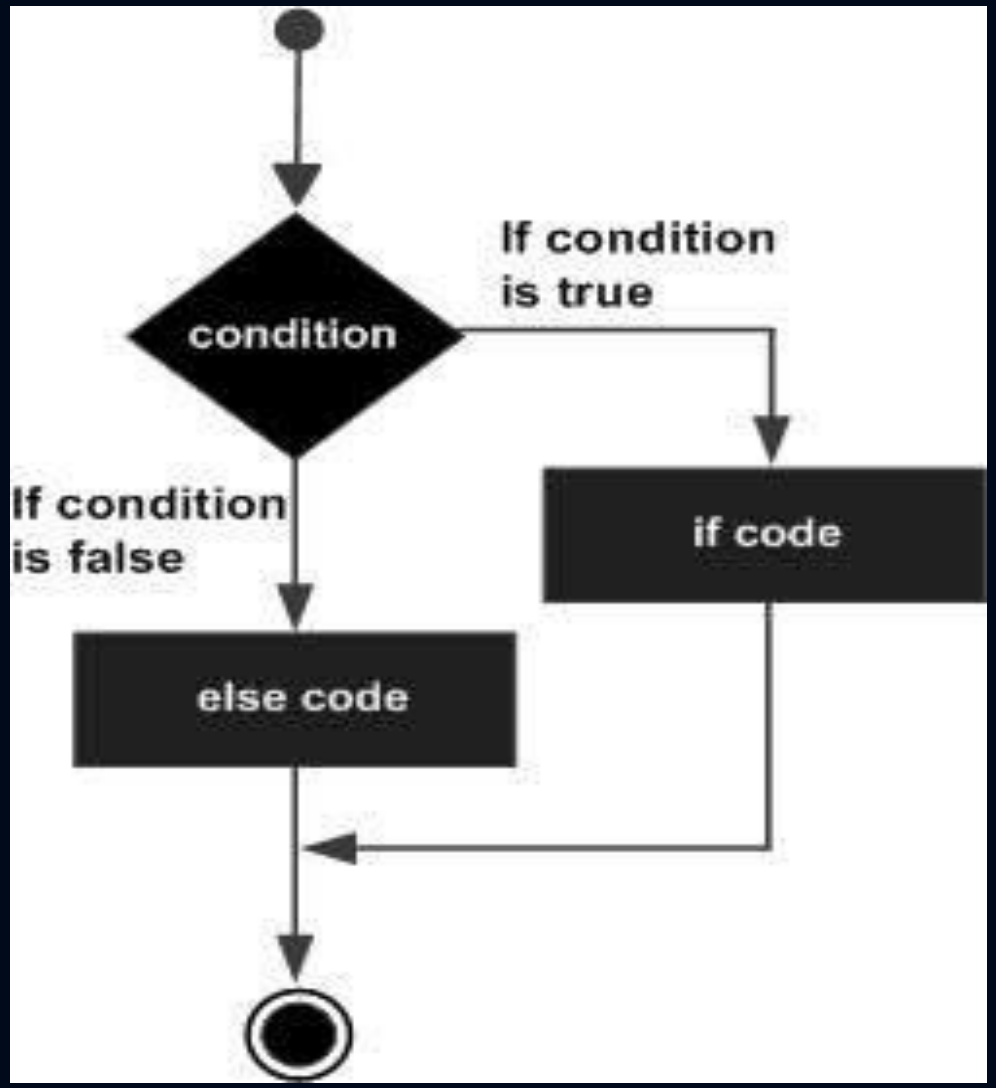
```
if BOOLEAN EXPRESSION:
    STATEMENTS
```



Each of the statements inside the body are executed in order if the Boolean expression evaluates to True. The entire block is skipped if the Boolean expression evaluates to False.

Alternative execution – else statement

An **else statement** contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.



Alternative execution – else statement

The else statement is an optional statement and there could be at most only one else statement following if

```
var1 = 100
if var1:
   print ("1 - Got a true expression value“)
   print (var1)
else:
   print ("1 - Got a false expression value“)
   print (var1)

print ("Good bye!“)
```


Single Statement Suites

if your if clause consists only of a single statement, it may be placed on the same line as the if header.

```
flag = 1
if (flag): print ('Given
flag is really true!‘)
```

This is also applicable to other conditional statements.

Class Activity 1

Write a program, graduate.py, that prompts students for how many credits they have. Print whether of not they have enough credits for graduation.

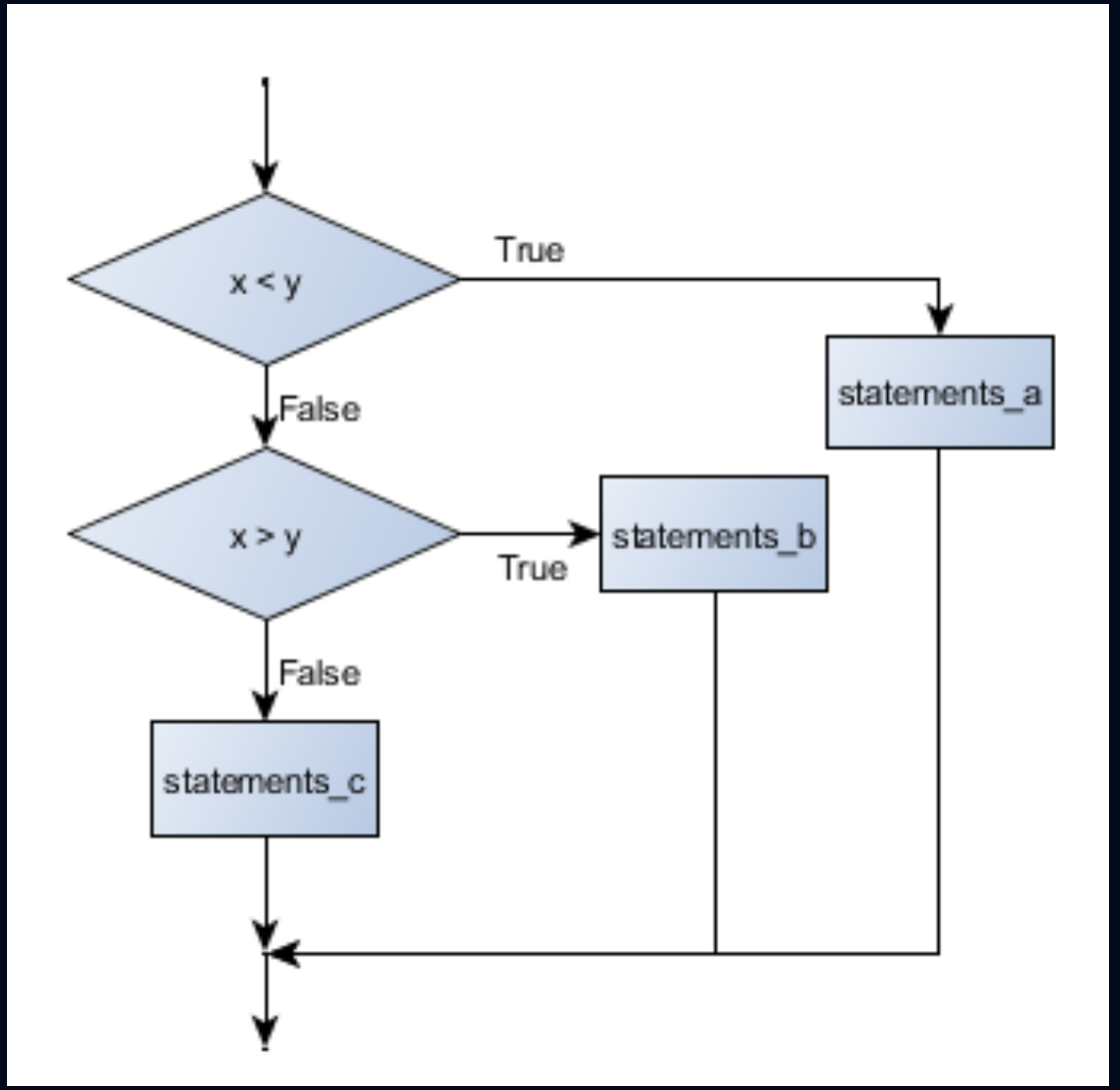
Chained conditionals– elif statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```


Chained conditionals– elif statement

Flowchart of chained conditional statements



Chained conditionals– elif statement

Python does not provide switch or case statements as in other languages

```
var = 100
if var == 200:
   print ("1 - Got a true expression value")
   print (var)
elif var == 150:
   print ("2 - Got a true expression value")
   print (var)
elif var == 100:
   print ("3 - Got a true expression value")
   print (var)
else:
   print ("4 - Got a false expression value")
   print (var)
print ("Good bye!")
```

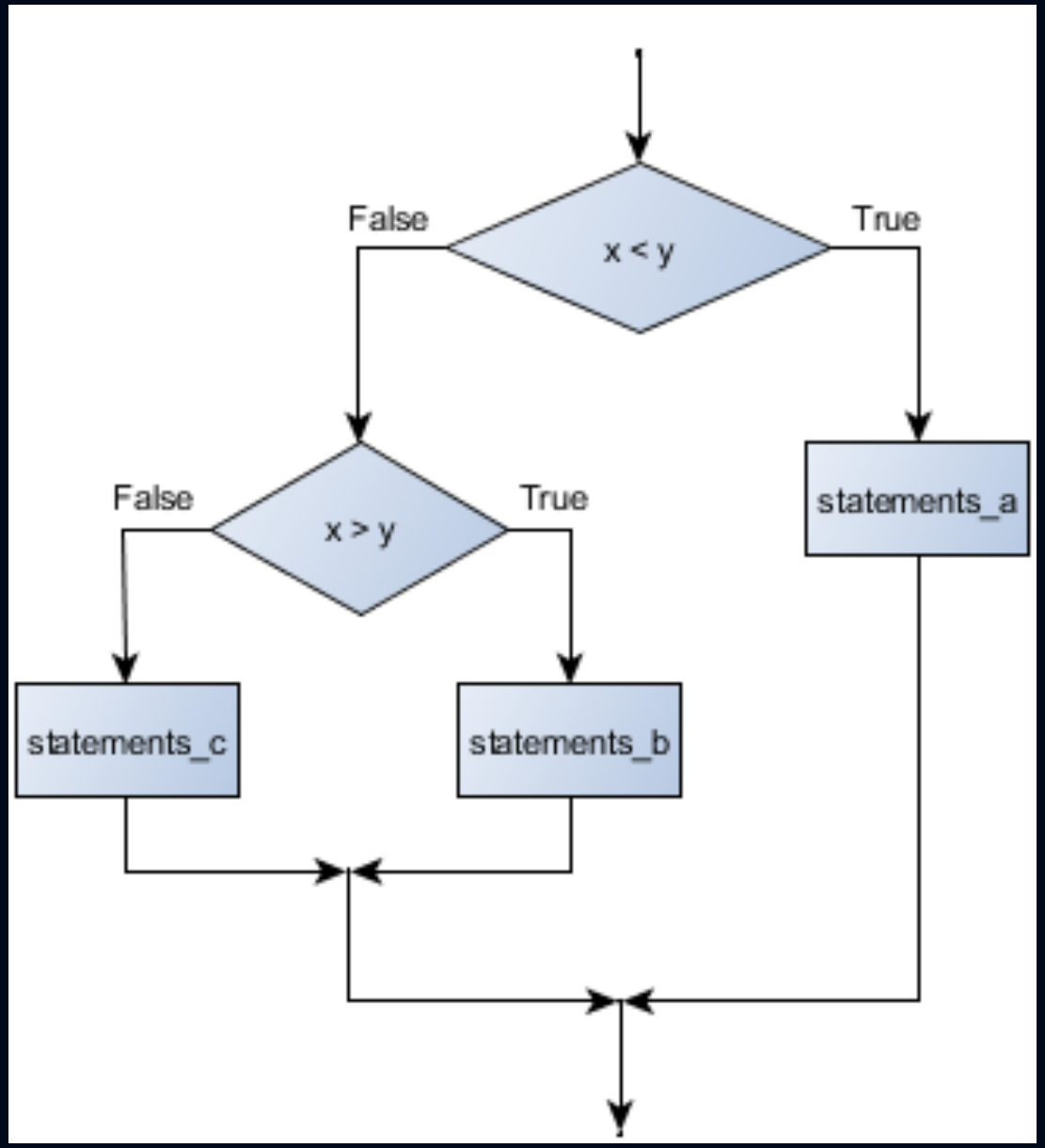

Nested conditionals

In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

```
if expression1:
  statement(s)
  if expression2:
    statement(s)
  elif expression3:
    statement(s)
  else:
    statement(s)
elif expression4:
  statement(s)
else:
  statement(s)
```


Nested conditionals

Flowchart of nested conditional statements



Nested conditionals

Sample Code

```
var = 100
if var < 200:
   print ("Expression value is less than 200")
   if var == 150:
      print ("Which is 150")
   elif var == 100:
      print ("Which is 100")
   elif var == 50:
      print ("Which is 50")
elif var < 50:
   print ("Expression value is less than 50")
else:
   print ("Could not find true expression")

print ("Good bye!")
```


Pass Statement

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

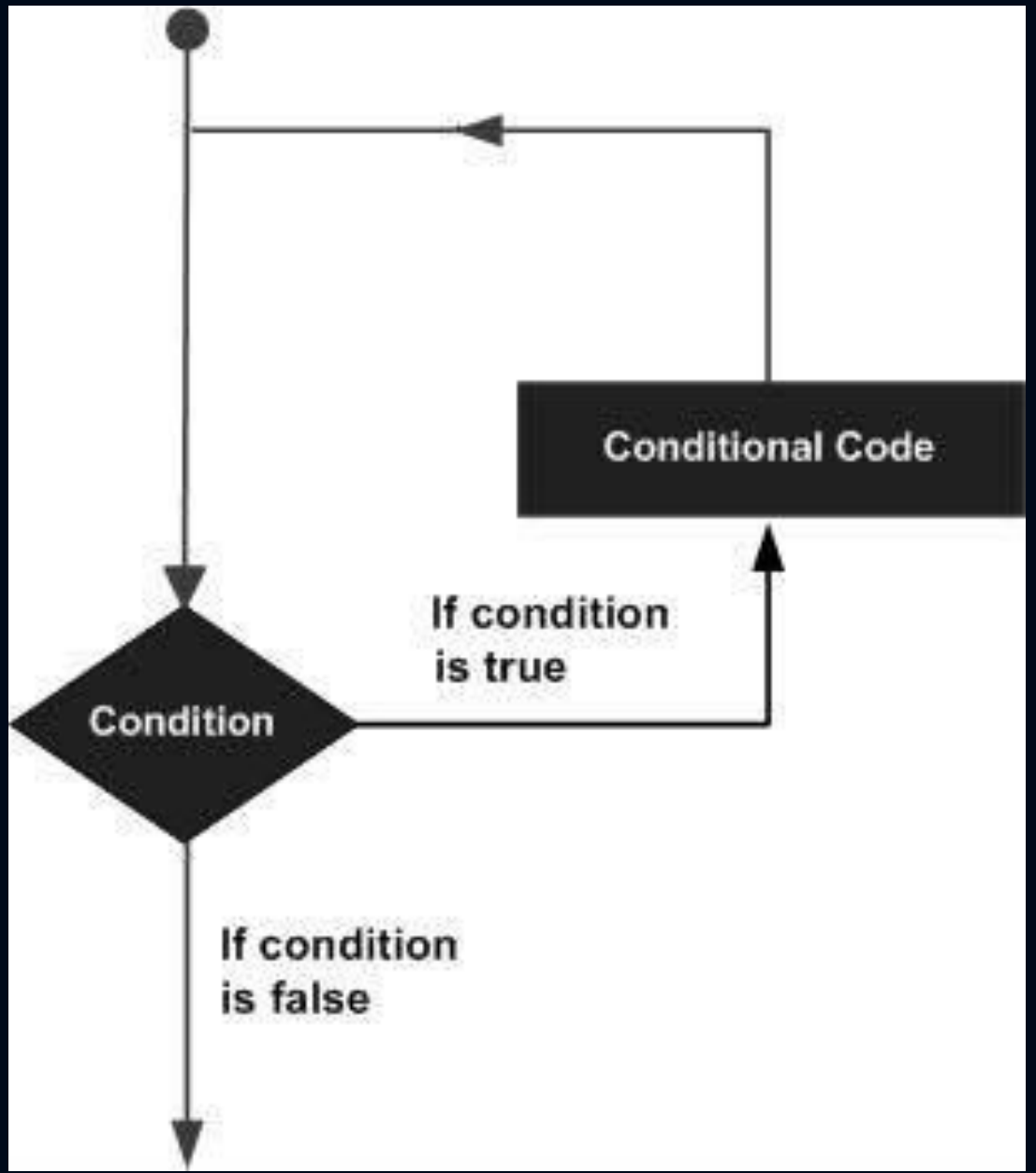
```
for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :',
letter)

print ("Good bye!")
```

The pass statement is a null operation; nothing happens when it executes.

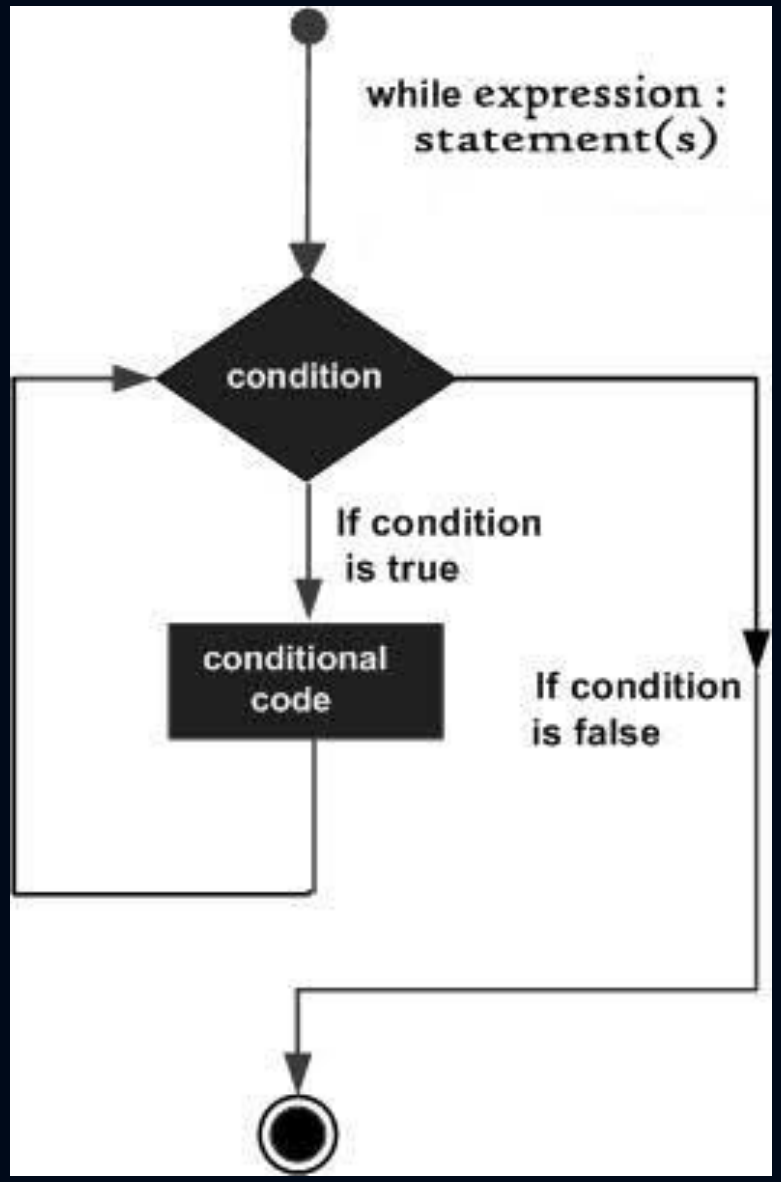
Loops and Iteration

A loop statement allows us to execute a statement or group of statements multiple times.



While loop

A while loop statement in repeatedly executes a target statement as long as a given condition is true



While loop

Python uses indentation as its method of grouping statements

Statement(s) may be a single statement or a block of statements.

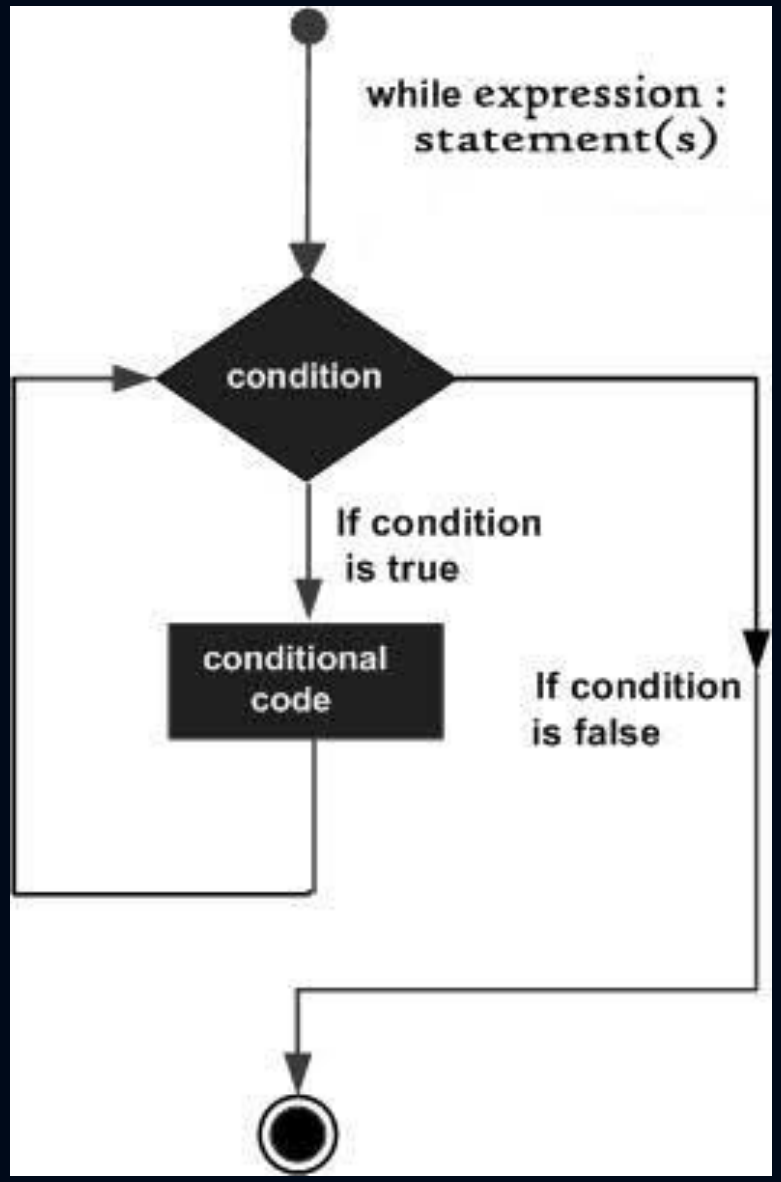
```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1

print ("Good bye!")
```


Nested conditionals

When the condition becomes false, program control passes to the line immediately following the loop.

A loop becomes **infinite loop** if a condition never becomes FALSE. You must use caution when using while loops.



else statement with while Loop

Python supports to have an else statement associated with a loop statement.

If the else statement is used with a **while** loop, the else statement is executed when the condition becomes false.

else Statement with while Loop

Sample Code

```
count = 0
while count < 5:
  print (count, " is less than 5“)
  count = count + 1
else:
  print (count, " is not less than 5“)
```

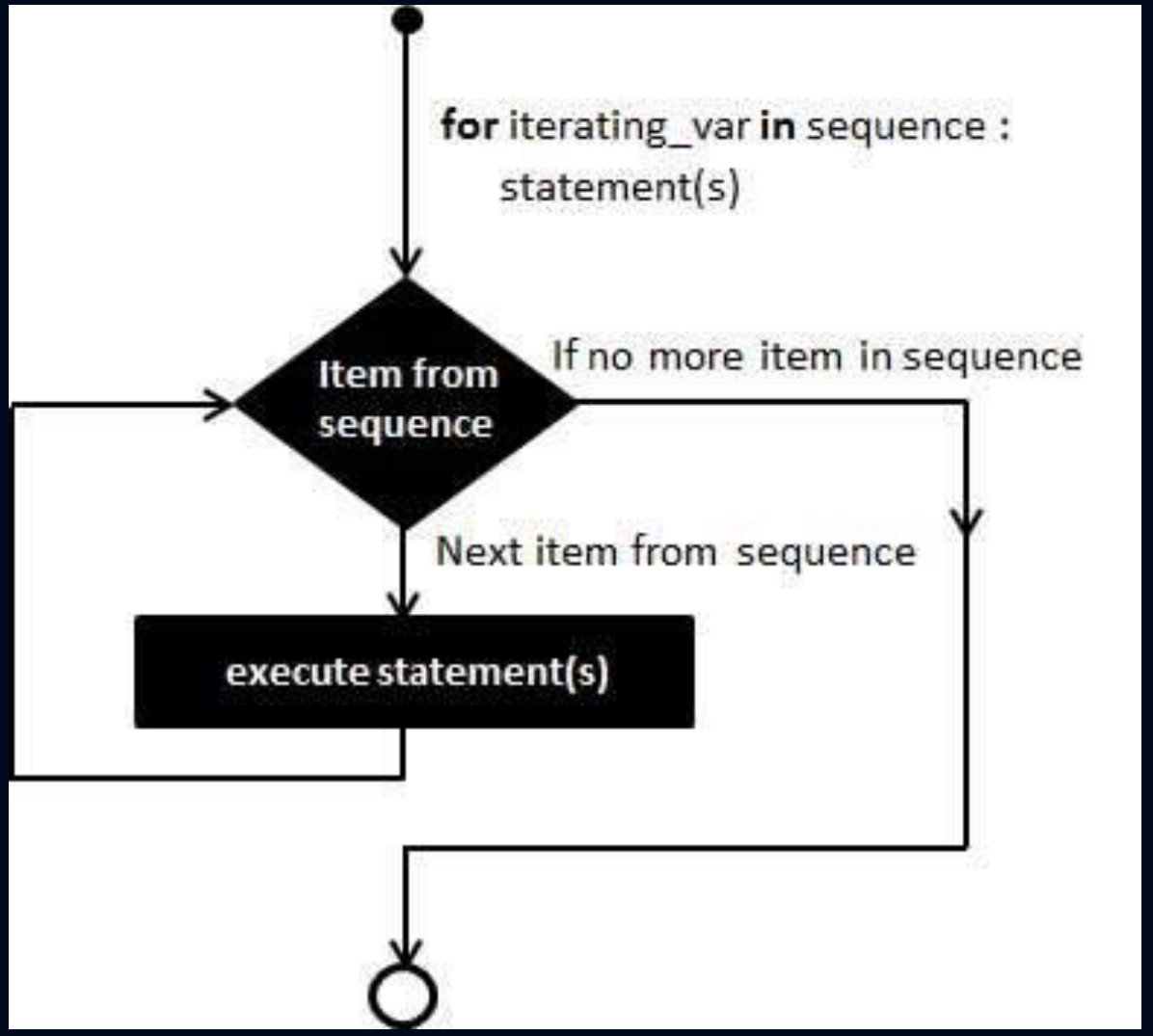
OUTPUT

0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5

for Loop Statements

for loop has the ability to iterate over the items of any sequence, such as a list or a string.

It Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.



Nested conditionals

Sample Code

```
for letter in 'Python':  # First Example
   print ('Current Letter :', letter)

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:   # Second Example
   print ('Current fruit :', fruit)

print ("Good bye!")
```

OUTPUT

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```


Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :',
fruits[index])

print ("Good bye!")
```


else Statement with for Loop

Sample Code

```
for num in range(10,20):  #to iterate between 10 to 20
  for i in range(2,num): #to iterate on the factors of the number
    if num%i == 0:      #to determine the first factor
      j=num/i          #to calculate the second factor
      print ('%d equals %d * %d' % (num,i,j))
      break #to move to the next number, the #first FOR
  else:                # else part of the loop
    print (num, 'is a prime number')
```

If the else statement is used with a **for** loop, the else statement is executed when the loop has exhausted iterating the list.

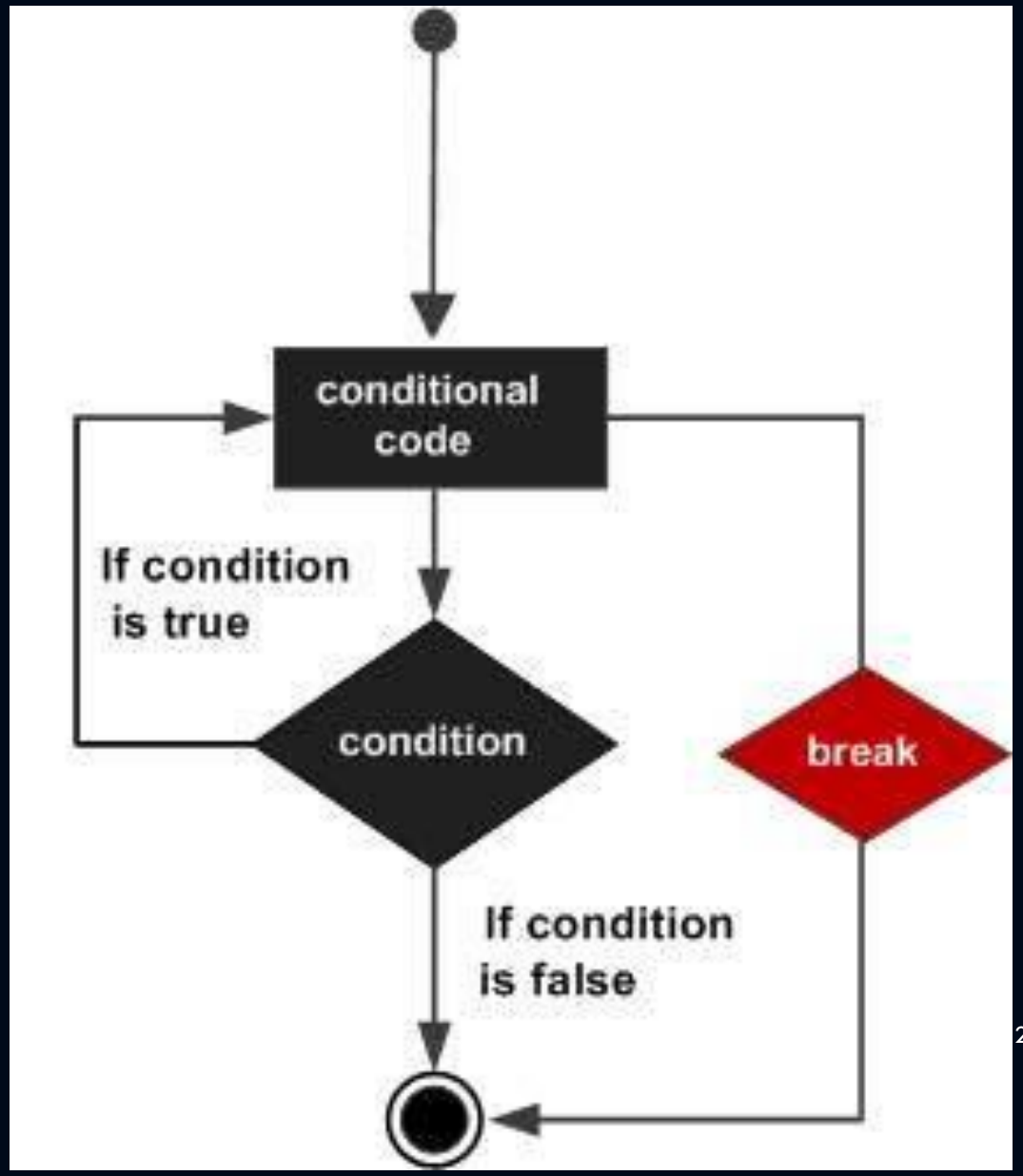
break statement

The keyword **break** stops the processing of a loop and exits the loop. Any code in the block that comes after the break statement is ignored (any else clause is also skipped).

The break statement can be used in both while and for loops.

break statement

If the loop is nested inside another code block, the program goes back to the block that the loop was nested in.



break statement

```
for letter in 'Python':  # First Example
   if letter == 'h':
      break
   print('Current Letter :', letter)

var = 10  # Second Example
while var > 0:
   print ('Current variable value :', var)
   var = var - 1
   if var == 5:
      break

print ("Good bye!")
```

Sample Code

OUTPUT

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```


continue statement

Continue statement returns the control to the beginning of the while loop….

The continue statement can be used in both while and for loops.

The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

continue statement

Sample Code

```
for letter in 'Python': # First Example
   if letter == 'h':
      continue
   print ('Current Letter :', letter)

var = 10                # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print ('Current variable value :',
var)
print ("Good bye!")
```

OUTPUT

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```


Tables

One of the things loops are good for is **generating tables**.

```
# Generate numbers 0 to 12

for x in range(13):

print(x, '\t', 2**x)
```

Using the tab character ('\t') makes the output align nicely.

Choosing between for and while

Use a **for loop** if you know, before you start looping, the maximum number of times that you will need to execute the body.

For example, if you're traversing a list of elements, you know that the maximum number of loop iterations you can possibly need is "all the elements in the list"

for loop is referred to as definite iteration because we have some definite bounds for what is needed.

Choosing between for and while

By contrast, if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when this will happen, then you will need a **while loop.**

while loop is referred to as indefinite iteration because we're not sure how many iterations we'll need

Class Activity 2

Write a program to implement a simple guessing game

Solution:

```
import random                    # Import the random module

number = random.randrange(1, 1000) # Get random number between [1 and 1000)
guesses = 0
guess = int(input("Guess my number between 1 and 1000: "))

while guess != number:
    guesses += 1
    if guess > number:
        print(guess, "is too high.")
    elif guess < number:
        print(guess, " is too low.")
    guess = int(input("Guess again: "))

print("\n\nGreat, you got it in", guesses,  "guesses!")
```


Next Lecture ...

Python Programming

Day 8: Functions