Control Flow Statements

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Python

Simple Programming Constructs

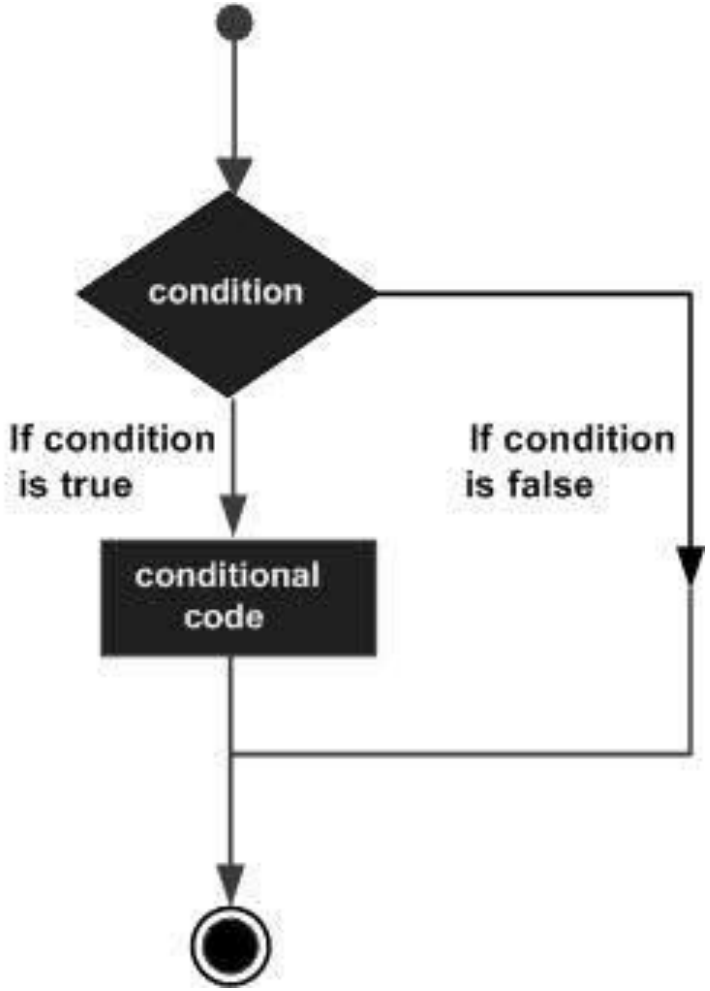
- Conditions(Decision Making) - Decide at runtime whether to perform certain statements.
- Loops - Decide at runtime how many times to perform certain statements.
- Branches

Decision Making

- Is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.
- Decision structures evaluate multiple expressions, which produce $\boldsymbol{TRUE}$ or $\boldsymbol{FALSE}$ as the outcome.
- Need to determine which action to take and which statements to execute if the outcome is $\boldsymbol{TRUE}$ or $\boldsymbol{FALSE}$ otherwise.

Decision Making (Cont.)

Statement	Description
`if` statements	An `if` statement consists of a boolean expression followed by one or more statements.
`if`...`else` statements	An `if` statement can be followed by an optional `else` statement, which executes when the boolean expression is *FALSE*.
nested `if` statements	You can use one `if` or `else if` statement inside another `if` or `else if` statement(s).



IF Statement

- The `if` statement is similar to that of other languages.
- The `if` statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

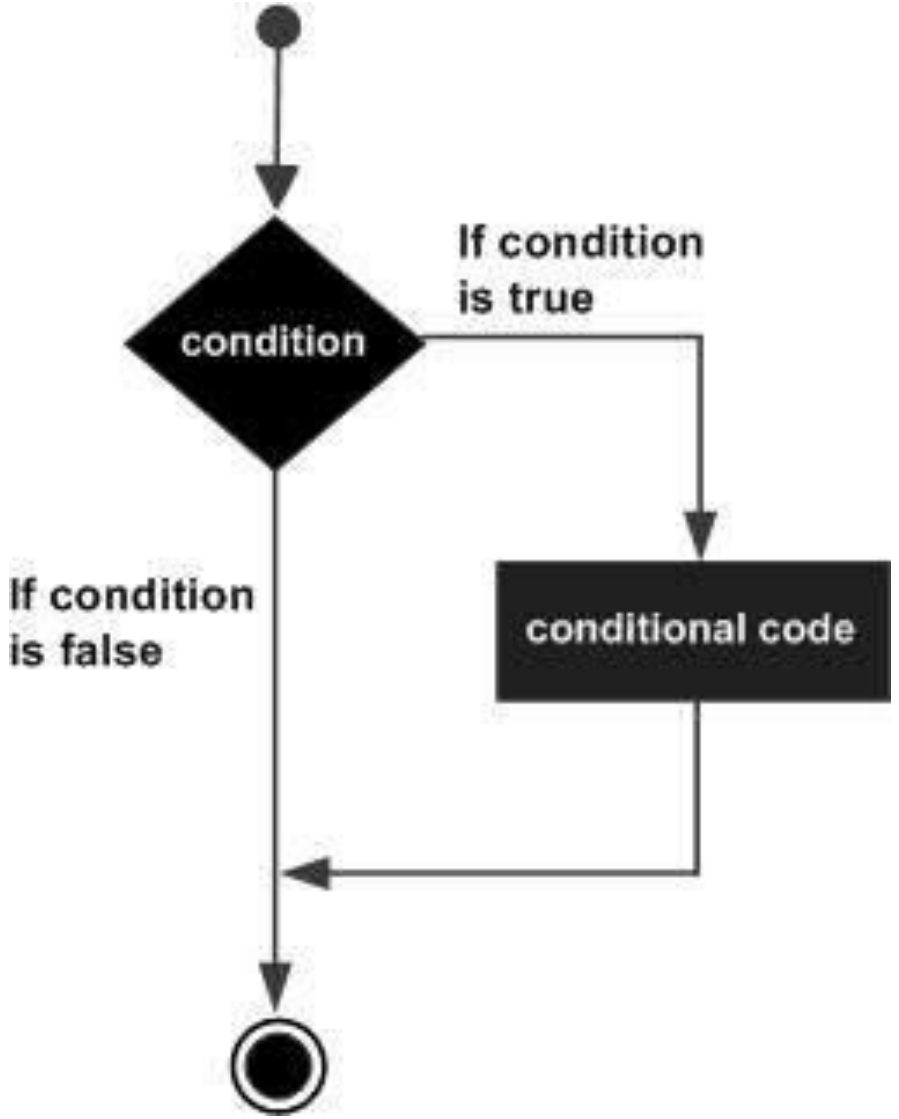
IF Statement (Cont.)

- Syntax

```
if expression :
        statement(s)
```

- If the boolean expression evaluates to $\boldsymbol{TRUE}$, then the block of statement(s) inside the if statement is executed.
- In Python, statements in a block are uniformly indented after the `:` symbol.
- If boolean expression evaluates to $\boldsymbol{FALSE}$, then the first set of code after the end of block is executed.

IF Statement (Cont.)



IF Statement (Cont.)

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


1 - Got a true expression value
100
>>>
>>> var2 = 0
>>> if var2 :
        print ("2 - Got a true expression value")
        print ( var2 )


>>> print ("Good bye!")
Good bye!
>>>
```

IF … ELSE … Statements

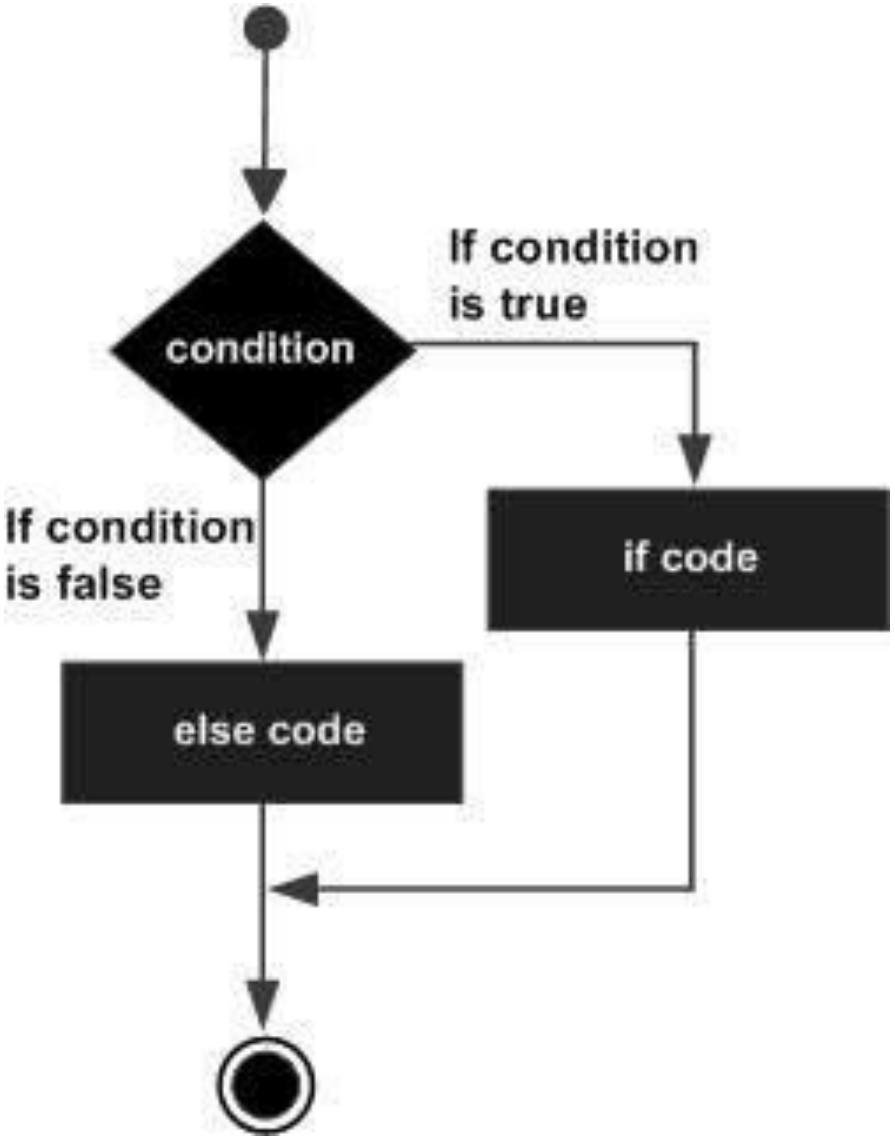
- An **`else`** statement can be combined with an **`if`** statement.
- An **`else`** statement contains a block of code that executes **`if`** the conditional expression in the **`if`** statement resolves to 0 or a ***FALSE*** value.
- The **`else`** statement is an optional statement and there could be at the most only one else statement following **`if`**.

IF … ELSE … Statements (Cont.)

- Syntax

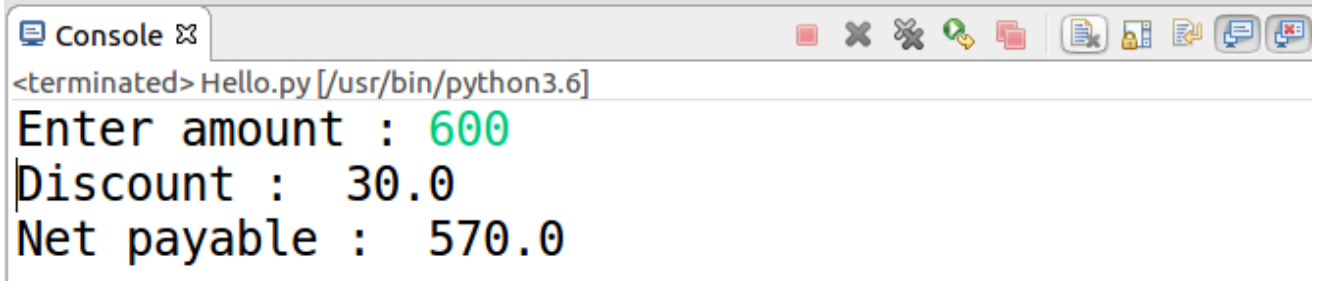
```
if expression :
        statement(s)

else :
        statement(s)
```



IF … ELSE … Statements (Cont.)

```
2  amount = int(input("Enter amount : "))
3
4  if amount < 1000 :
5      discount = amount * 0.05
6      print ("Discount : ", discount)
7  else :
8      discount = amount * 0.10
9      print ("Discount : ", discount)
10
11 print ("Net payable : ", amount - discount)
```



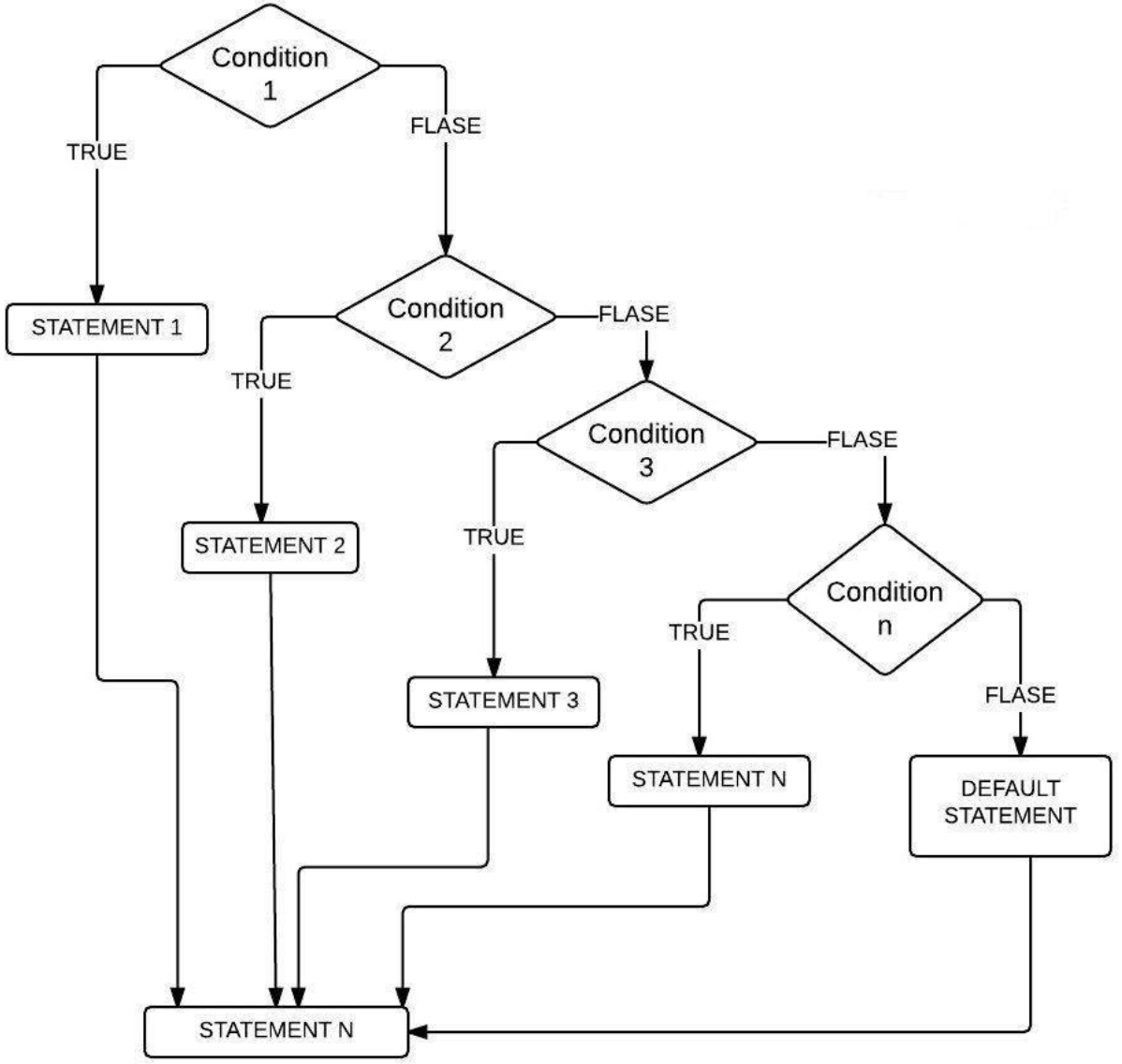
IF … ELIF … ELSE Statements

- The `elif` statement allows to check multiple expressions for $\boldsymbol{TRUE}$ and execute a block of code as soon as one of the conditions evaluates to $\boldsymbol{TRUE}$.
- Similar to the `else`, the `elif` statement is optional.
- However, unlike `else`, for which there can be at the most one statement, there can be an arbitrary number of `elif` statements following an `if`.

IF … ELIF … ELSE Statements (Cont.)

- Syntax

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

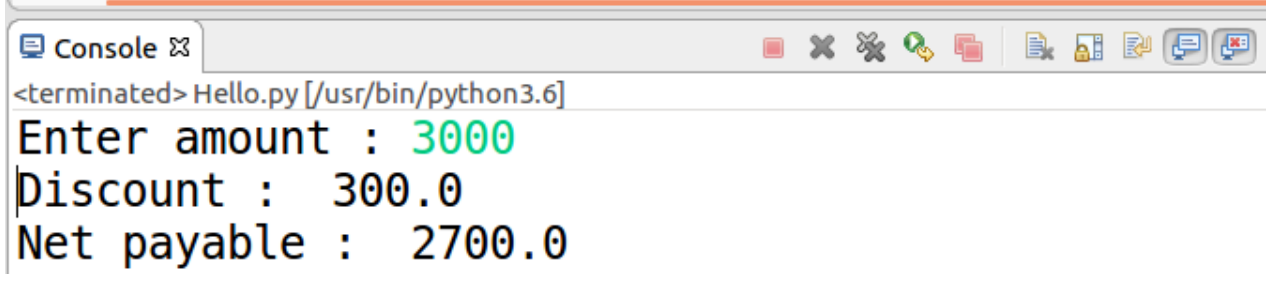


IF … ELIF … ELSE Statements (Cont.)

- Core Python does not provide **`switch`** or **`case`** statements as in other languages.
- But can use **`if..elif...`** statements to simulate **`switch case`**.

IF … ELIF … ELSE Statements (Cont.)

```
2  amount = int(input("Enter amount : "))
3
4  if amount < 1000 :
5      discount = amount * 0.05
6      print ("Discount : ", discount)
7  elif amount < 5000 :
8      discount = amount * 0.10
9      print ("Discount : ", discount)
10 else :
11     discount = amount * 0.15
12     print ("Discount : ", discount)
13
14 print ("Net payable : ", amount - discount)
```



Nested IF Statements

- There may be a situation when want to check for another condition after a condition resolves to true.
- In such a situation, can use the ***nested*** `if` construct.
- In a ***nested*** `if` construct, can have an `if...elif...else` construct inside another `if...elif...else` construct.

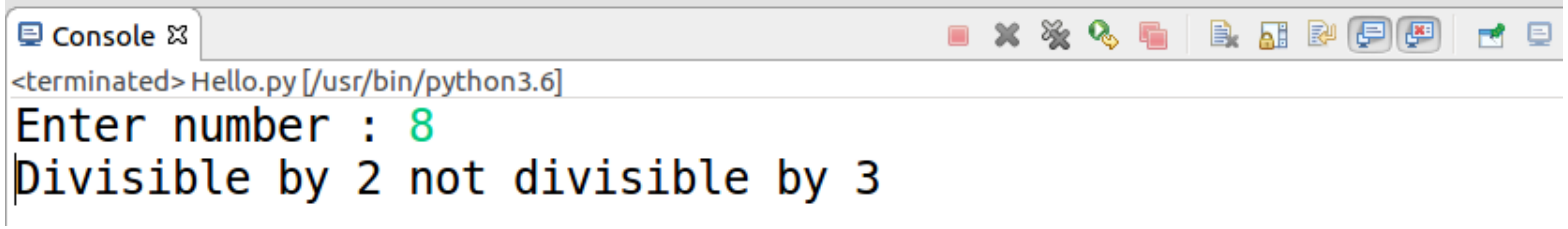
Nested IF Statements (Cont.)

- Syntax

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

Nested IF Statements (Cont.)

```
2  num = int(input("Enter number : "))
3
4  if num % 2 == 0 :
5      if num % 3 == 0 :
6          print ("Divisible by 3 and 2")
7      else :
8          print ("Divisible by 2 not divisible by 3")
9  else :
10     if num % 3 == 0 :
11         print ("Divisible by 3 not divisible by 2")
12     else :
13         print ("Not divisible by 2 not divisible by 3")
14
15
```



Single Statement Suites

- If the suite of an `if` clause consists only of a single line, it may go on the same line as the header statement.

```
>>> var = 100
>>>
>>> if ( var == 100 ) : print ("Value of expression is 100")

Value of expression is 100
>>> print ("Good bye!")
Good bye!
>>>
```

if - Lab

- What is the result of the following code ?

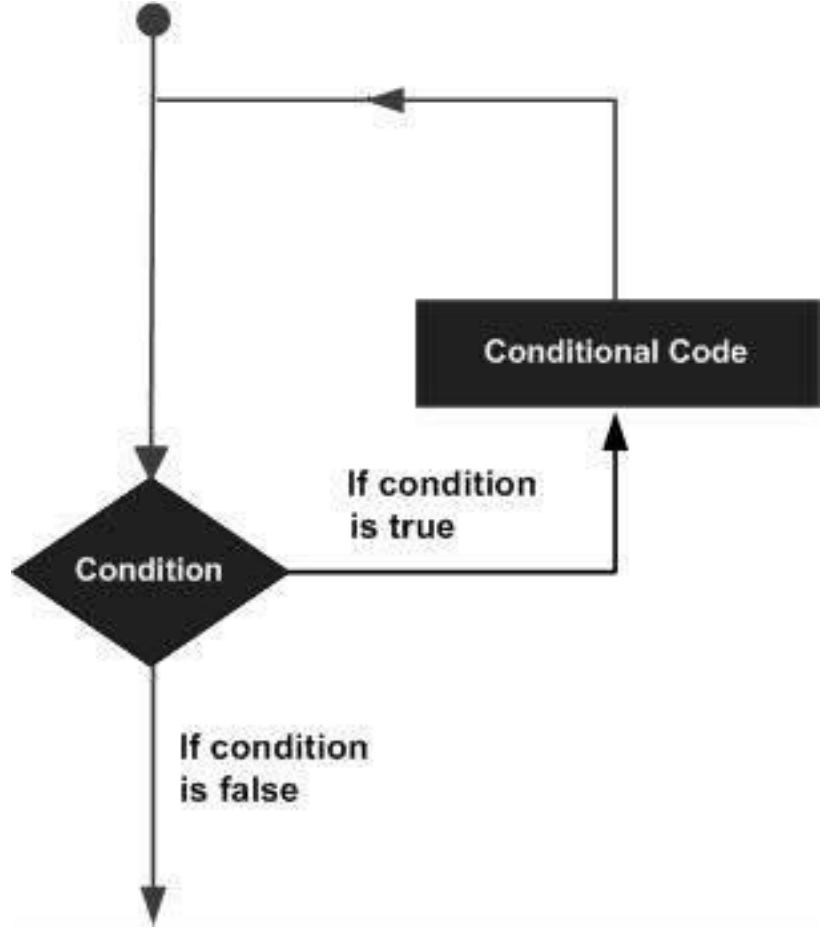
```
>>> a = "Life is too short, you need python"
>>>
>>> if "wife" in a : print("wife")
elif "python" in a and "you" not in a : print ("python")
elif "shirt" not in a : print ("shirt")
elif "need" in a : print ("need")
else : print ("none")
```

Loops

- In general, statements are executed sequentially.
- The first statement in a function is executed first, followed by the second, and so on.
- There may be a situation when need to execute a block of code several number of times.
- Programming languages provide various control structures that allow more complicated execution paths.
- A loop statement allows to execute a statement or group of statements multiple times.

Loops (Cont.)

Statement	Description
`while` loop	Repeats a statement or group of statements while a given condition is *TRUE*. It tests the condition before executing the loop body.
`for` loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loop	You can use one or more loop inside any another while, or for loop.



WHILE Loop Statements

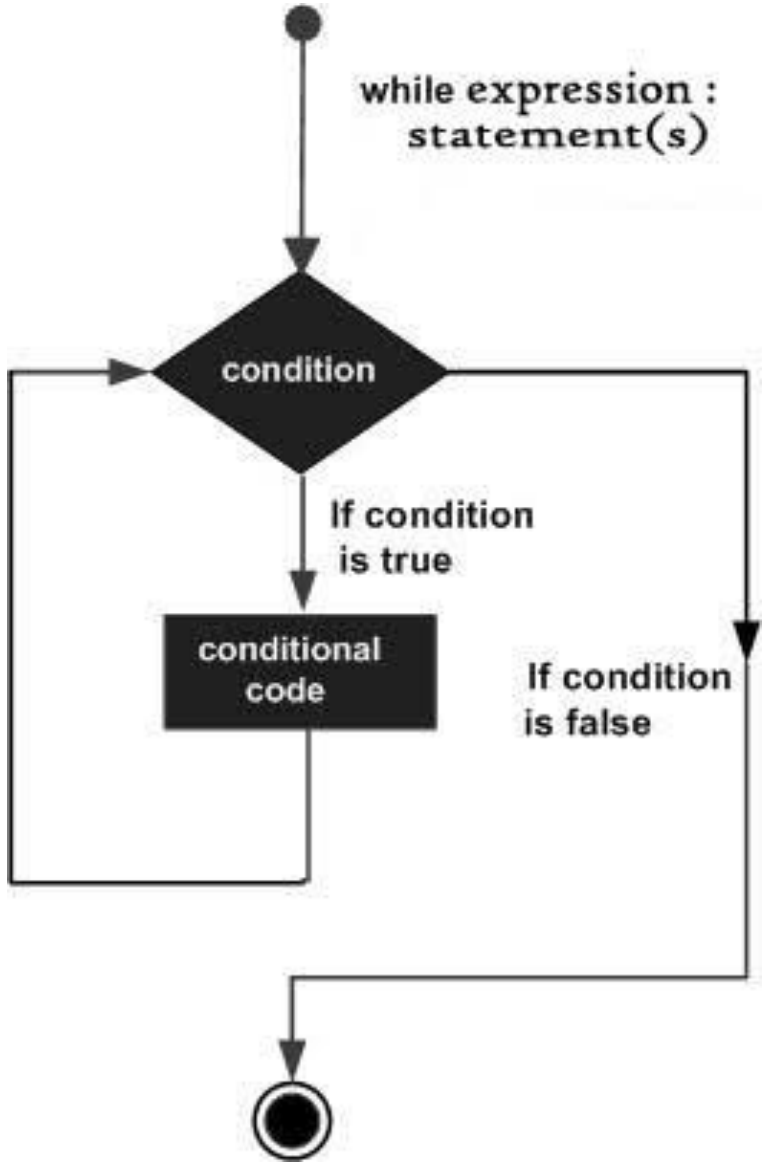
- Repeatedly executes a target statement as long as a given condition is ***true***.
- Syntax

```
while expression :
        statement(s)
```

WHILE Loop Statements (Cont.)

- Statement(s) may be a single statement or a block of statements with uniform indent.
- The condition may be any expression, and true is any non-zero value.
- The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.

WHILE Loop Statements (Cont.)



WHILE Loop Statements (Cont.)

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


The count is :  0
The count is :  1
The count is :  2
The count is :  3
The count is :  4
The count is :  5
The count is :  6
The count is :  7
The count is :  8
>>>
```

The Infinite Loop

- A loop becomes infinite loop if a condition never becomes $\boldsymbol{FALSE}$.
- Must be cautious when using while loops because of the possibility that this condition never resolves to a $\boldsymbol{FALSE}$ value.
- This results in a loop that never ends.
- Such a loop is called an infinite loop.

The Infinite Loop (Cont.)

```
>>> var = 1
>>> while var == 1 :           # This constructs an infinite loop
        num = int (input ("Enter a number : "))
        print ("You entered : ", num)


Enter a number : 5
You entered :  5
Enter a number : 7
You entered :  7
Enter a number : 9
You entered :  9
Enter a number : 11
You entered :  11
Enter a number : 13
You entered :  13
Enter a number : 15
You entered :  15
Enter a number :
```

Using else Statement with Loops

- Python supports having an **`else`** statement associated with a loop statement.
  - If the **`else`** statement is used with a **`for`** loop, the **`else`** statement is executed when the loop has exhausted iterating the list.
  - If the **`else`** statement is used with a **`while`** loop, the **`else`** statement is executed when the condition becomes *false*.

Using else Statement with Loops

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


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
>>>
```

Single Statement Suites

- Similar to the **`if`** statement syntax, if **`while`** clause consists only of a single statement, it may be placed on the same line as the **`while`** header.

```
>>> flag = 1
>>>
>>> while (flag) : print ("Given flag is really true!")

Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
Given flag is really true!
```

while - Lab

- while문을 이용하여 아래와 같이 별(*)을 표시하는 프로그램을 완성하시오.

```
*
**
***
****
******
```

FOR Loop Statements

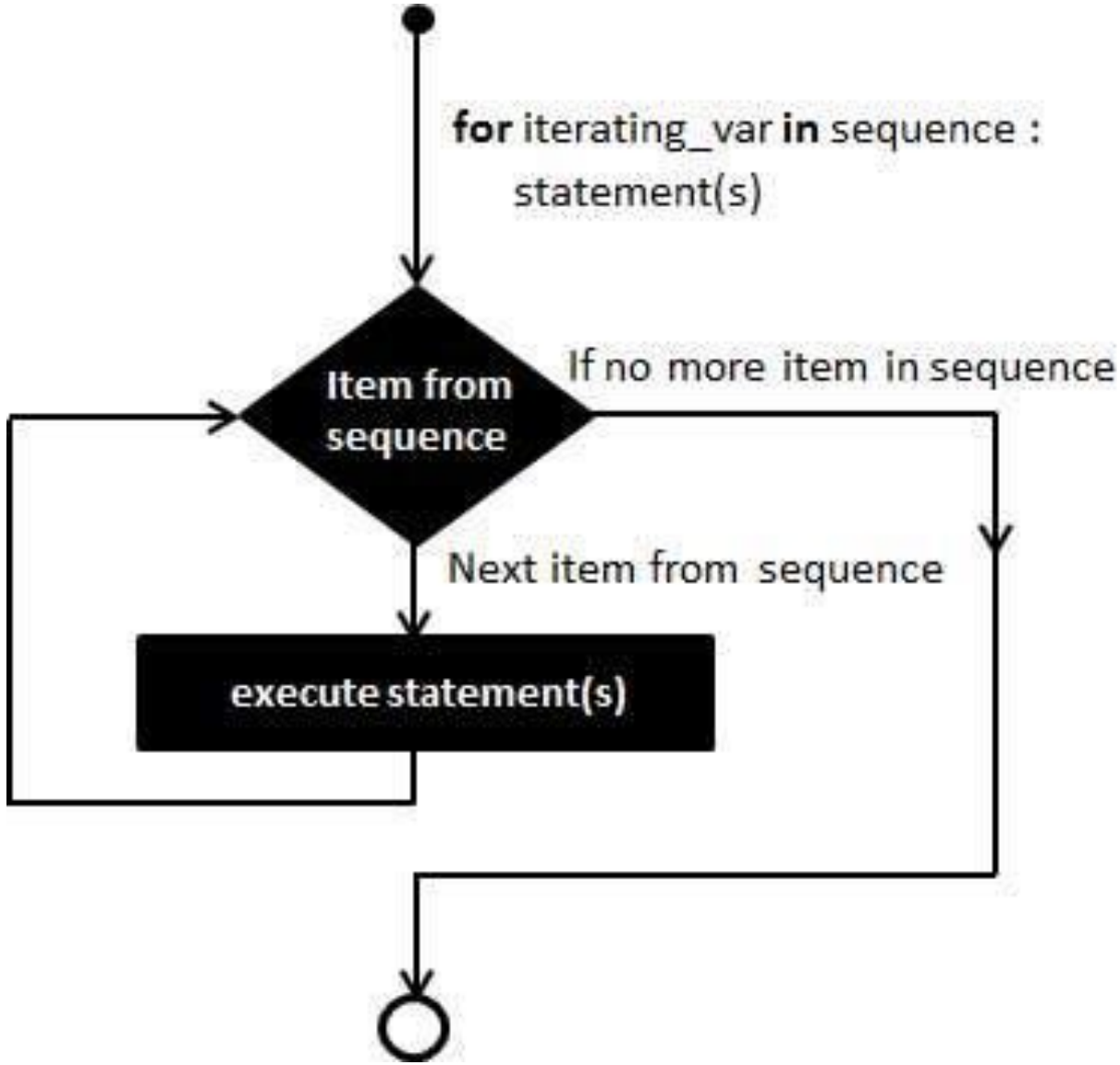
- Has the ability to iterate over the items of any sequence, such as a list or a string.
- Syntax

```
for iterating_var in sequene :
        statement(s)
```

FOR Loop Statements (Cont.)

- If a sequence contains an expression list, it is evaluated first.
- Then, the first item in the sequence is assigned to the iterating variable **`iterating_var`**.
- Next, the statements block is executed.
- Each item in the list is assigned to **`iterating_var`**, and the statement(s) block is executed until the entire sequence is exhausted.

FOR Loop Statements (Cont.)



The range() function

- The built-in function **`range()`** is the right function to iterate over a sequence of numbers.
- It generates an iterator of arithmetic progressions.

```
>>> range(5)
range(0, 5)
>>>
>>> list(range(5))
[0, 1, 2, 3, 4]
>>>
```

The range() function (Cont.)

- **`range()`** generates an iterator to progress integers starting with 0 up to n-1.
- To obtain a list object of the sequence, it is typecasted to **`list()`**.
- Now this list can be iterated using the for statement.

```
>>> for var in list(range(5)) :
        print (var)

0
1
2
3
4
>>>
```

FOR Loop Statements (Cont.)

```
>>> for letter in 'Python' :                    # traversal of a string sequence
        print ("Current Letter : ", letter)


Current Letter :  P
Current Letter :  y
Current Letter :  t
Current Letter :  h
Current Letter :  o
Current Letter :  n
>>>
>>> fruits = ['Banana', 'Apple', 'Mango']
>>> for fruit in fruits :                       # traversal of List sequence
        print ("Current fruit : ", fruit)


Current fruit :  Banana
Current fruit :  Apple
Current fruit :  Mango
>>>
```

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])


Current fruit :  Banana
Current fruit :  Apple
Current fruit :  Mango
>>>
```

Using else Statement with Loops

- Python supports having an **`else`** statement associated with a loop statement.
  - If the **`else`** statement is used with a **`for`** loop, the **`else`** block is executed only if **`for`** loops terminates normally (and not by encountering **`break`** statement).
  - If the **`else`** statement is used with a **`while`** loop, the **`else`** statement is executed when the condition becomes false.

Using else Statement with Loops (Cont.)

```
>>> numbers = [11, 33, 55, 39, 55, 75, 37, 21, 23, 41, 13]
>>>
>>> for num in numbers :
        if num % 2 == 0 :
            print ("The list contains an even number")
            break
else :
        print ("The list does not contain even number")


The list does not contain even number
>>>
```

Nested Loops

- Python programming language allows the usage of one loop inside another loop.
- Syntax

```
for iterating_var in sequence :
        for iterating_var in sequence :
            statement(s)
        statement(s)
```

Nested Loops (Cont.)

- The syntax for a nested **while** loop statement in Python programming language is as follow:
- Syntax

```
while expression :
        while expression :
                statement(s)
        statement(s)
```

Nested Loops (Cont.)

```
>>> import sys
>>> for i in range(1, 11) :
        for j in range(1, 11) :
            k = i * j
            print (k ,end='  ')
        print()


1  2  3  4  5  6  7  8  9  10
2  4  6  8  10  12  14  16  18  20
3  6  9  12  15  18  21  24  27  30
4  8  12  16  20  24  28  32  36  40
5  10  15  20  25  30  35  40  45  50
6  12  18  24  30  36  42  48  54  60
7  14  21  28  35  42  49  56  63  70
8  16  24  32  40  48  56  64  72  80
9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
>>>
```

for - Lab

- A학급에 총 10명의 학생이 있다. 이 학생들의 중간고사 점수는 아래와 같다.

  70, 60, 55, 75, 95, 90, 80, 80, 85, 100

  for 문을 이용하여 A 학급의 평균 점수를 구해 보시오.

Branch(Loop Control) Statements

- Change the execution from its normal sequence.
- When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Statement	Description
`break`	Terminates the loop statement and transfers execution to the statement immediately following the loop.
`continue`	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
`pass`	The `pass` statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

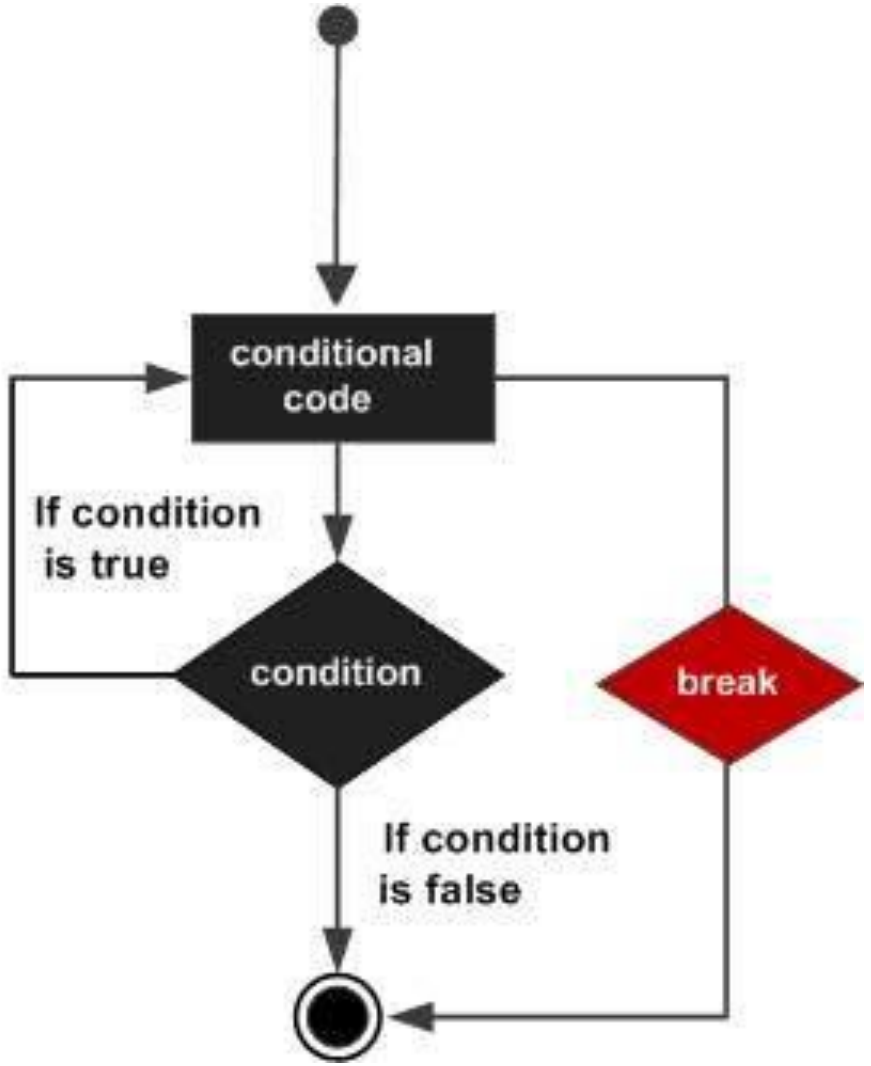
BREAK Statement

- Is used for premature termination of the current loop.
- After abandoning the loop, execution at the next statement is resumed, just like the traditional break statement in C.
- Is when some external condition is triggered requiring a hasty exit from a loop.
- The break statement can be used in both **`while`** and **`for`** loops.
- If you are using nested loops, the **`break`** statement stops the execution of the innermost loop and starts executing the next line of the code after the block.

BREAK Statement (Cont.)

- Syntax

```
break
```



BREAK Statement (Cont.)

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


Current Letter :  P
Current Letter :  y
Current Letter :  t
>>>
>>> var = 10                          # Second example
>>> while var > 0 :
        print ("Current variable value : ", var)
        var = var - 1
        if var == 5 :
                break;


Current variable value :  10
Current variable value :  9
Current variable value :  8
Current variable value :  7
Current variable value :  6
>>>
```

```
2  no = int(input('Any number : '))
3  numbers = [11, 33, 55, 39, 55, 75, 37, 21, 23, 41, 13]
4
5  for num in numbers :
6      if  num == no :
7          print ('Number found in list')
8          break
9  else :
10     print ('Number not found in list')
```

Console

<terminated> Hello.py [/usr/bin/python3.6]

```
Any number : 33
Number found in list
```

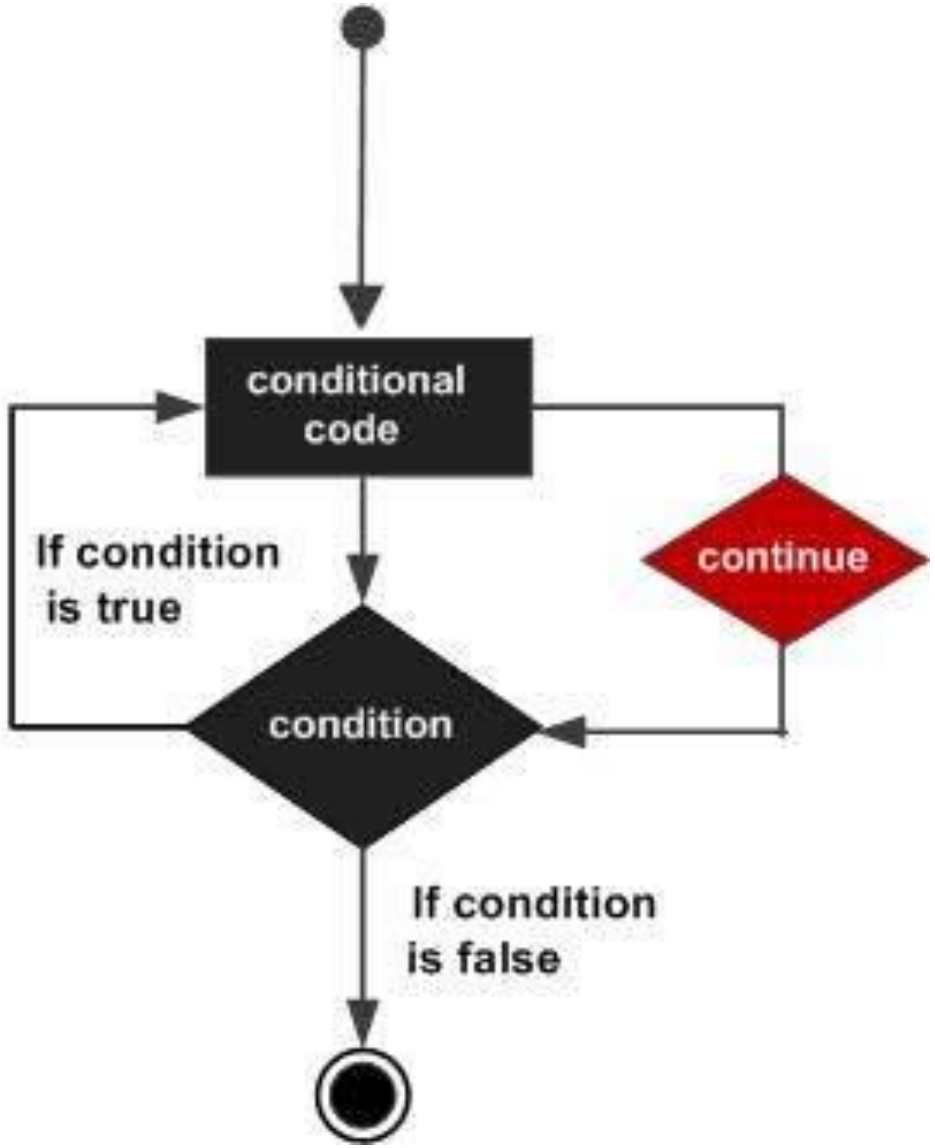
CONTINUE Statement

- Returns the control to the beginning of the current loop.
- When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.
- The continue statement can be used in both `while` and `for` loops.

CONTINUE Statement (Cont.)

- Syntax

```
continue
```



CONTINUE Statement (Cont.)

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


Current Letter :  P
Current Letter :  y
Current Letter :  t
Current Letter :  o
Current Letter :  n
>>>
```

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


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
>>>
```

PASS Statement

- Is used when a statement is required syntactically but you do not want any command or code to execute.
- Is a null operation.
- Nothing happens when it executes.
- Is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs.
- Syntax

```
pass
```

PASS Statement (Cont.)

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


Current Letter :  P
Current Letter :  y
Current Letter :  t
This is pass block
Current Letter :  h
Current Letter :  o
Current Letter :  n
>>>
```

Iterator and Generator

- Iterator is an object which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation.
- In Python, an iterator object implements two methods, `iter()` and `next()`.
- String, List or Tuple objects can be used to create an Iterator.

Iterator and Generator (Cont.)

```
2 list = [1, 2, 3, 4]
3 it = iter(list)          # This builds an iterator object
4 print (next(it))     # Prints next avaiable element in iterator
5
6 # Iterator object can be traversed using regular for statement
7 for x in it :
8     print (x, end = "    ")
```

Console

<terminated> Hello.py [/usr/bin/python3.6]

```
1
2    3    4
```

Iterator and Generator (Cont.)

```
1 import sys
2 list = [1, 2, 3, 4]
3 it = iter(list)          # This builds an iterator object
4
5 while True :
6     try :
7         print (next(it))
8     except StopIteration :
9         sys.exit()
```

Console

<terminated> Hello.py [/usr/bin/python3.6]

```
1
2
3
4
```

Iterator and Generator (Cont.)

- A generator is a function that produces or yields a sequence of values using `yield` method.
- When a generator function is called, it returns a generator object without even beginning execution of the function.
- When the `next()` method is called for the first time, the function starts executing until it reaches the yield statement, which returns the yielded value.
- The yield keeps track i.e. remembers the last execution and the second `next()` call continues from previous value.

Iterator and Generator (Cont.)

```
1  import sys
2  def fibonacci(n):    # Generator function
3      a,b,counter = 0, 1, 0
4      while True :
5          if (counter > n) :
6              return
7          yield a
8          a, b = b, a + b
9          counter += 1
10
11 f = fibonacci(5)     #f is iterator object
12
13 while True :
14     try :
15         print (next(f), end = "    ")
16     except StopIteration :
17         sys.exit()
```

Console

<terminated> Hello.py [/usr/bin/python3.6]

```
0    1    1    2    3    5
```