EW

@ENGINEERINGWALLAH

Sinhgad Institutes

UNIT - 2

Decision Control Statements

Sinhgad Institutes

Syllabus

- Decision control statements
- Selection/conditional branching statements :

✓ if, if-else,

✓ nested if,

✓ if-elif-else statements.

- Basic loop Structures/Iterative statements :

✓ while loop, for loop, selecting appropriate loop

✓ Nested loops, break, continue, pass,

✓ else statement used with loops.

- Other data types-Tuples, Lists and Dictionary

Control Statements:

- By default, in all script the statements are executed sequentially from the first to the last.
- If the processing logic requires, then the sequential flow can be altered in different ways.
- A **control statement** is a statement that determines the control flow of a set of instructions, i.e., it decides the sequence in which the instructions in a program are to be executed.
- A **control structure** is a set of statements and the control statements controlling their execution.

Control Statements:

Types of Control Statements —

- **Sequential Control:**

  A Python program is executed sequentially from the first line of the program to its last line.

- **Selection Control or conditional control/Decision control:**

  ✓ To execute only a selected set of statements.

  ✓ In Selection or conditional branching "if, if–else, if–elif–else" statements are used for selection.

- **Iterative Control or Repetitive control or Loop control:**

  ✓ To execute a set of statements repeatedly.

  ✓ While, for, else, pass, break continue statements are used for iterative execution

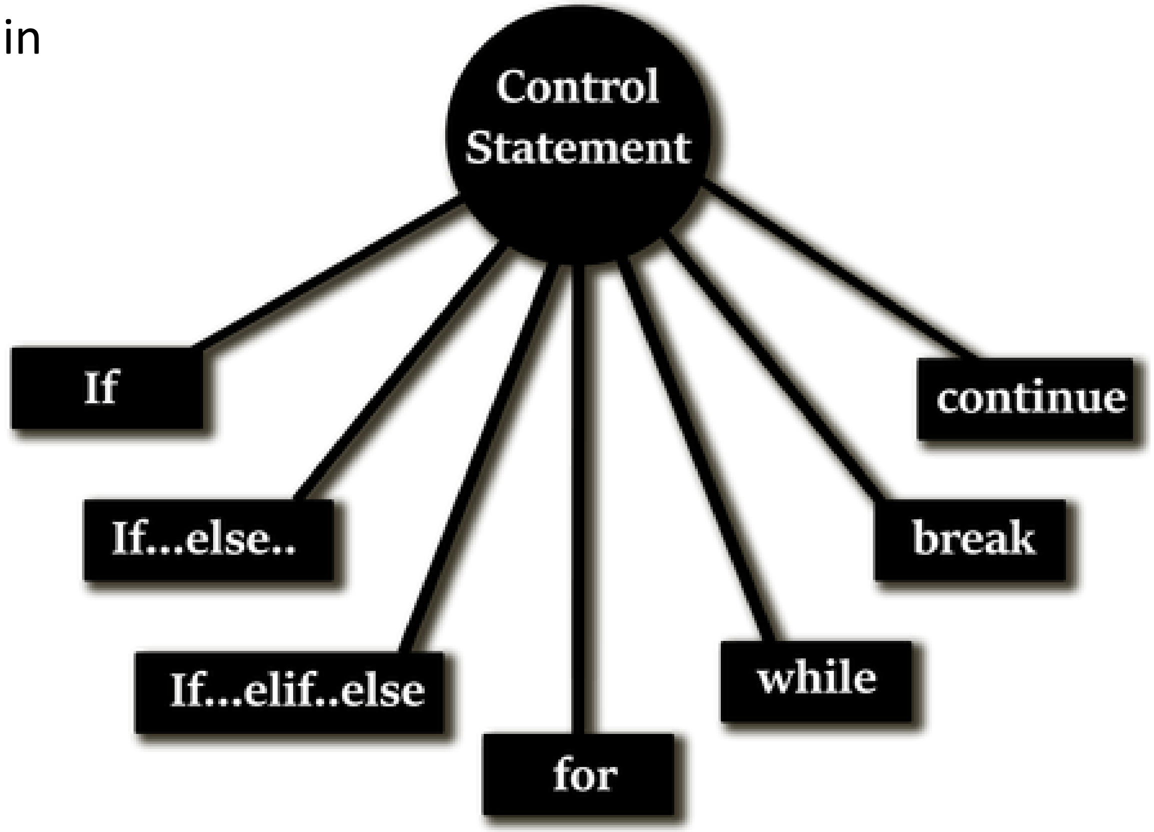
Control Statements

- We have the **three** types of Control statement in Python:
- Sequential Statement
- **Decision Control Statement:**

  If, If..else, if…elif….else.

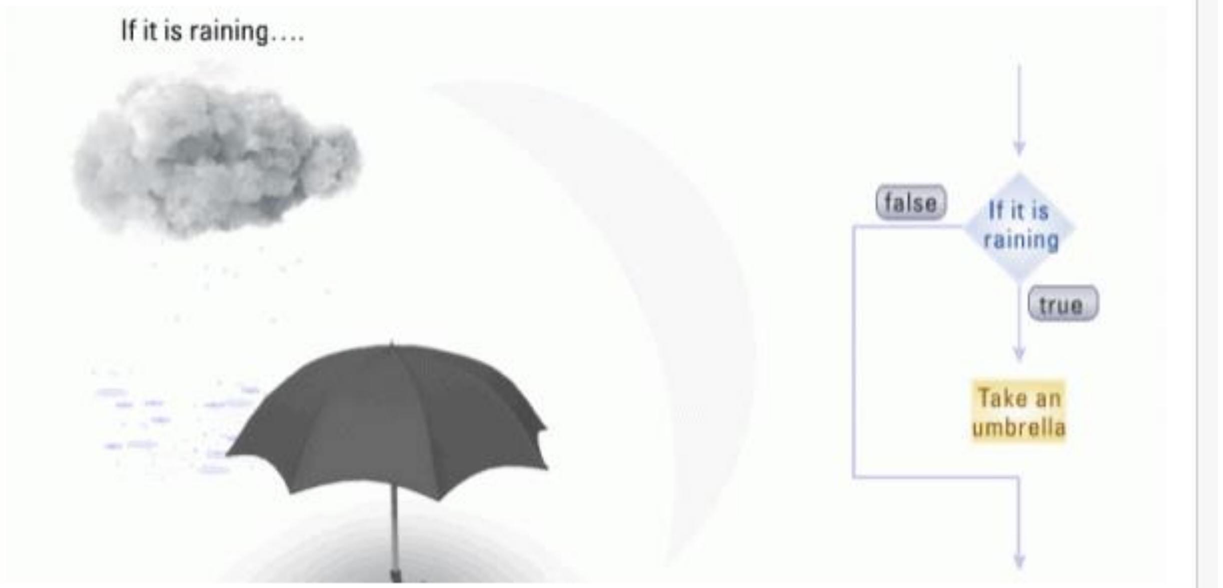
- **Flow Control Statement:**

  for, while break, continue



Decision Control Example

- In everyday life, we make many decisions. For instance, if its raining I will use umbrella so that I don't get wet. This is a type of branching. If one condition is true, that's if raining, I take my umbrella. If the condition is not true but false, then I will not take my umbrella. This type of branching decision making can be implemented in python programming using 'if' statements.

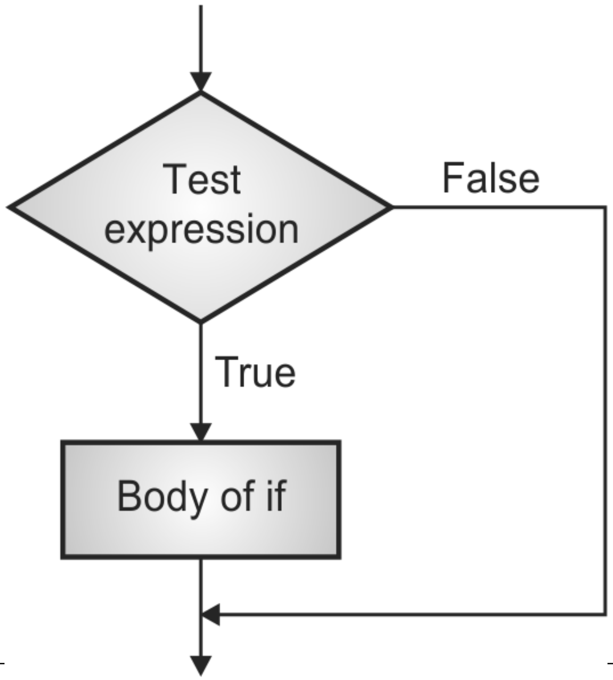


If Statement:

- It is used when we have to take some decision like the comparison between anything or to check the presence and gives us either TRUE or FALSE.
- The Python if statement is used to implement the decision.

```
if <condition>:
        <body>
```

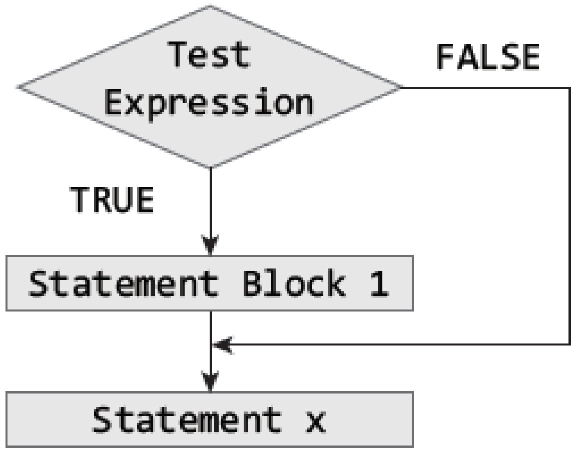
- The body is a sequence of one or more statements indented under the if heading.
- The body is executed if condition is evaluated to True
- The body is skipped if condition is evaluated to False.



If Statement:

SYNTAX OF IF STATEMENT

```
if test_expression:
      statement1
       ………..
     Statement n
statement x;
```



Example: Increment value of x if it greater than 0.

```
x = 10       #Initialize the value of x
if(x>0):     #test the value of x
   x = x+1     #Increment the value of x if it is > 0
print(x)      #Print the value of x
```

OUTPUT

```
x = 11
```

Program to check whether a number is even or not using if statement

- **Program: evenodd.py**

```
print(“Enter the number:”)
number = int(input( ))
if number % 2 == 0:
    print ("Number is even … ")
if number %2 ==1:
    print ("Number is odd …")
```

Output :

```
Enter the number  111
Number is odd …
```

Sinhgad Institutes

If else Statement:

- Two possible paths of execution
  –One is taken if the condition is true, and the other if the condition is false

  Syntax:

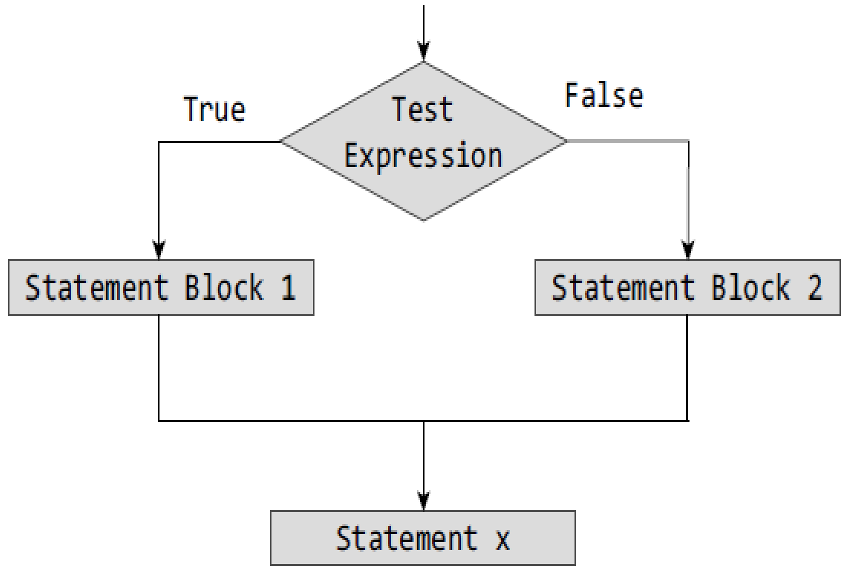
```
if condition:
        statements
else:
        other statements
```

- The else statement can be used with the if statement.
- It usually contains the code which is to be executed at the time when the expression in the if statement returns the FALSE.
- If the condition is true, then the if block is executed. Otherwise, the else block is executed.
- There can only be one else in the program with every single if statement
- It is optional to use the else statement with if statement it depends on your condition.

If Else Statement:

SYNTAX OF IF-ELSE STATEMENT

```
if (test expression):
        statement block 1
else:
       statement block 2
statement x;
```



```
age = int(input("Enter the age : "))
if(age>=18):
   print("You are eligible to vote")
else:
        yrs = 18 - age
        print("You have to wait for another " + str(yrs) +" years to cast your vote")
```

OUTPUT

```
Enter the age : 10
You have to wait for another 8 years to cast your vote
```

Sinhgad Institutes

Program to check whether a number is even or odd using if-else

- **Program: evenodd1.py**

```
print("Enter the number:")
number = int(input( ))
if number % 2 == 0:
    print ("Number is even … ")
else :
    print ("Number is odd …")
```

Output :

```
Enter the number  111
Number is odd …
```

Nested if

- A decision structure can be nested inside another decision structure.
- You can have **if** statements inside if statements, this is called nested if statements
- Commonly needed in programs when you want to check step by step if condition.
- Important to use proper indentation in a nested decision structure
- Rules for writing nested if statements:

    else clause should align with matching if clause

    Statements in each block must be consistently indented

Nested If

Syntax1:
```
if condition:
        statements
        if condition:
            statements
        else:
            statements
else:
            other statements
```

Syntax2:
```
if condition:
        statements
else:
        if condition:
            statements
        else:
            statements
        other statements
```

Program to check whether a number is +ve or -ve using nested-if

- **Program: positive.py**

```
print("Enter the number:")
number = int(input( ))
if number  >= 0:
    if number == 0:
        print("ZERO")
    else:
        print ("Number is positive … ")
else :
    print ("Number is negative …")
```

Output :

```
Enter the number  11
Number is positive
```

The if-elif-else Statement

- if-elif-else statement: special version of a decision structure used when more than two possibilities.
- These possibilities can be expressed using chained conditions. Can include multiple elif statements
- Alignment used with if-elif-else statement: Example of Distinction
  - if, elif, and else clauses are all aligned
  - Conditionally executed blocks are consistently indented
  - Syntax:

```
if condition1:
        statements
elif condition2:
        statements
else:
        statements
```

The if-elif-else Statement (cont'd.)

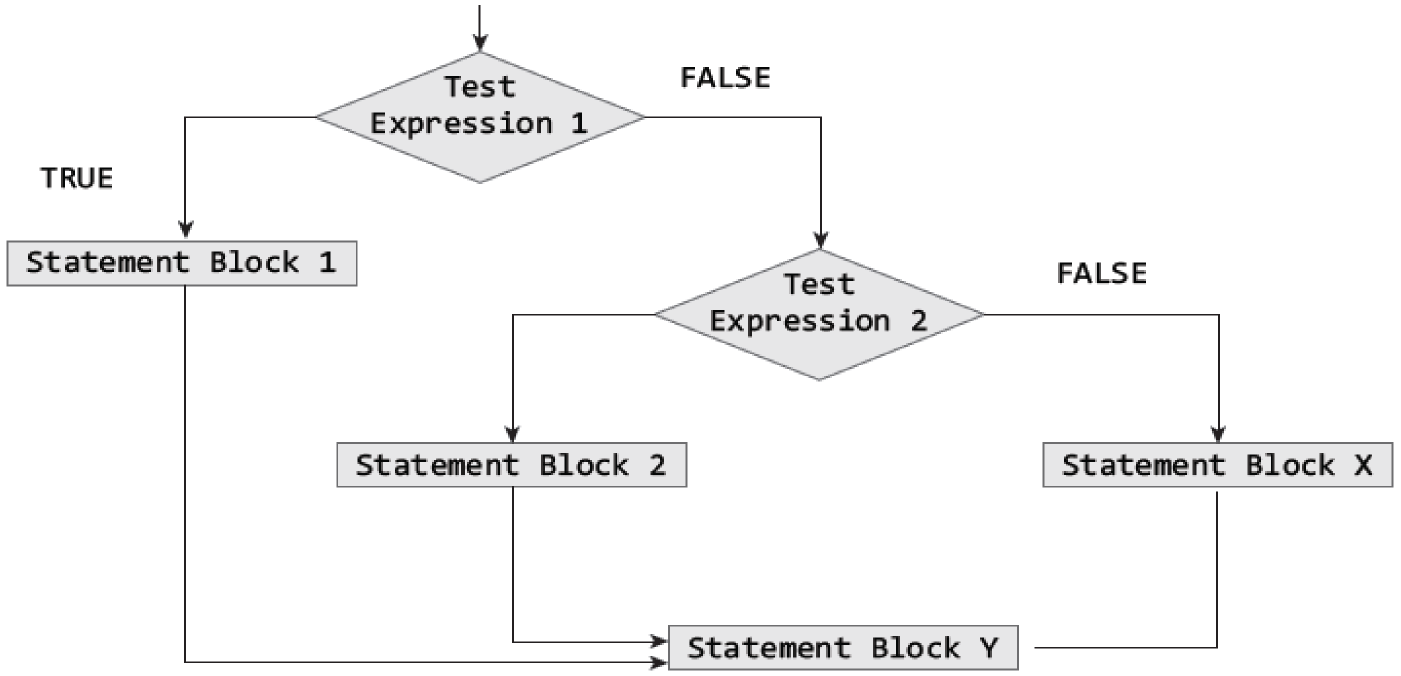
Python supports if-elif-else statements to test additional conditions apart from the initial test expression. The if-elif-else construct works in the same way as a usual if-else statement. If-elif-else construct is also known as nested-if construct.

Example:

```
num = int(input("Enter any number : "))
if(num==0):
    print("The value is equal to zero")
elif(num>0):
    print("The number is positive")
else:
    print("The number is negative")
```

OUTPUT

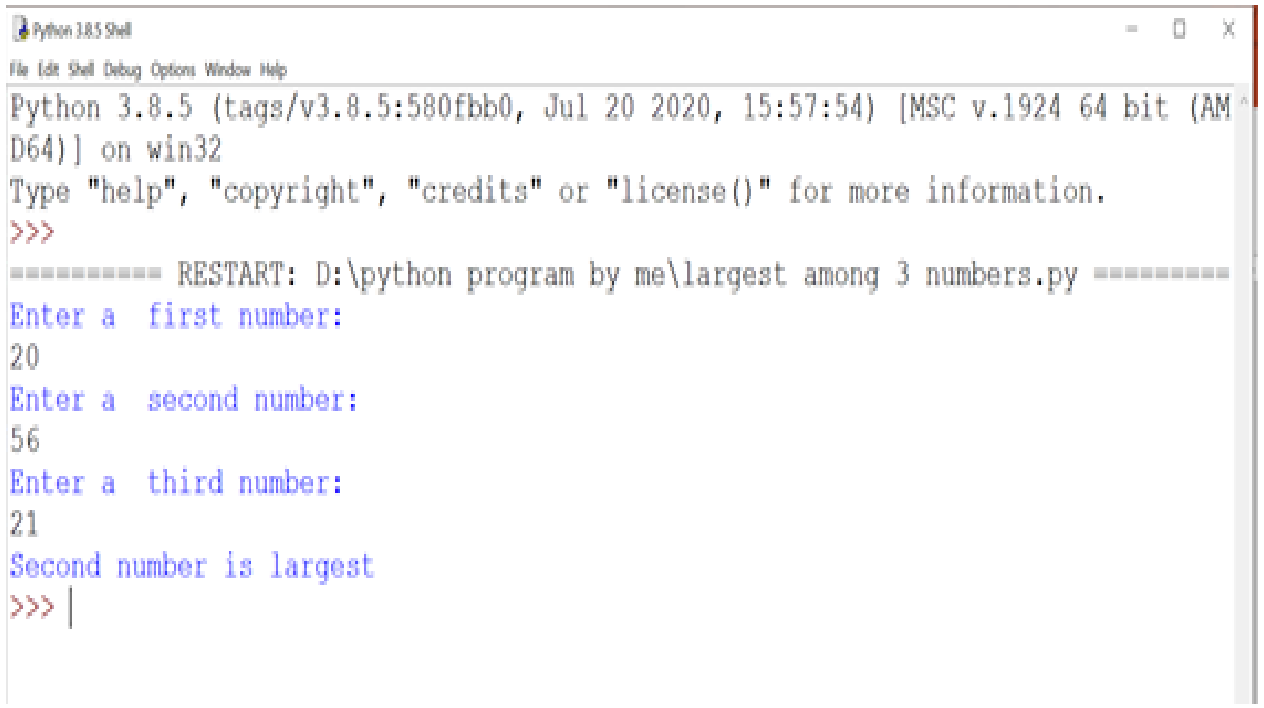
```
Enter any number : -10
The number is negative
```



Program to find largest among the three numbers nested-if

➢ **Program: largest.py**

```
print("Enter a  first number: ")
x = int(input())
print("Enter a  second number: ")
y = int(input())
print("Enter a  third number: ")
z = int(input())
if (x>y) and (x>z):
   print("First number is largest")
elif (y>x) and (y>z):
   print("Second number is largest")
else:
   print("Third number is largest")
```



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: D:\python program by me\largest among 3 numbers.py =========
Enter a  first number:
20
Enter a  second number:
56
Enter a  third number:
21
Second number is largest
>>> |
```

Basic loop structures/ Iterative or repetitive statements

- Loop is a technique that allows to execute a block of statement repeatedly .
- Python provides type of loops to handle looping requirements continuously up to condition fails.
- Two basic loops are

  1. while loop
  2. for loop
- To execute a set of statements repeatedly or iteratively.

  While, for, pass statements are used for iterative execution.

While Loop

- An if statement is run once if its condition evaluates to True, and never if it evaluates to False.

- A **while statement** is similar, except that it can be run more than once. The statements inside it are repeatedly executed, as long as the condition holds. Once it evaluates to False, the next section of code is executed.

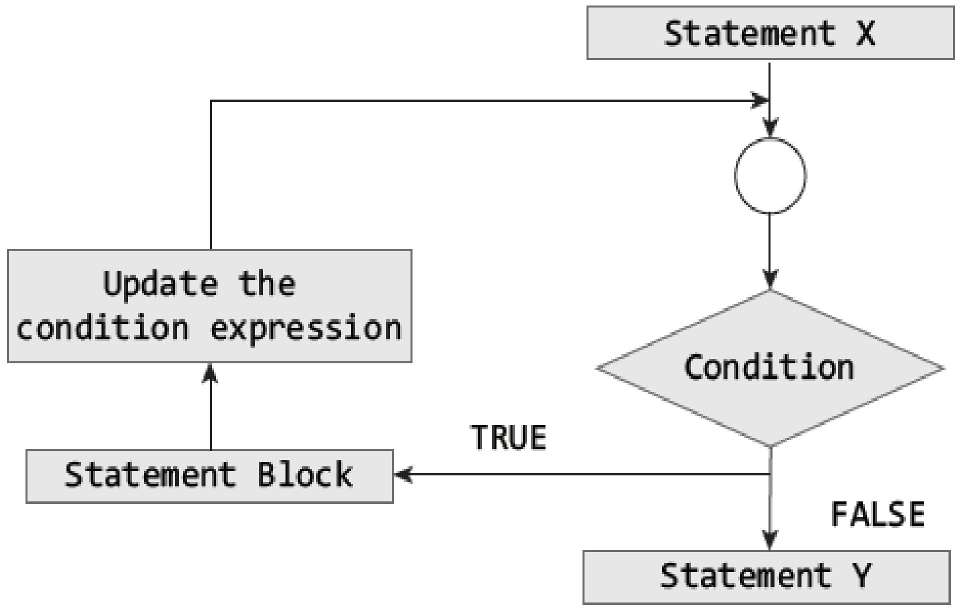
- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

- The while loop is also known as a pre-tested loop.

- A while loop implements indefinite iteration, where the number of times the loop will be executed is not specified explicitly in advance

While Loop

```
SYNTAX OF WHILE LOOP
statement x
while (condition):
            statement block
statement y
```



Flowchart

Example:

```
i = 0
while(i<=10):
    print(i,end=" ")
    i = i+1
```

OUTPUT

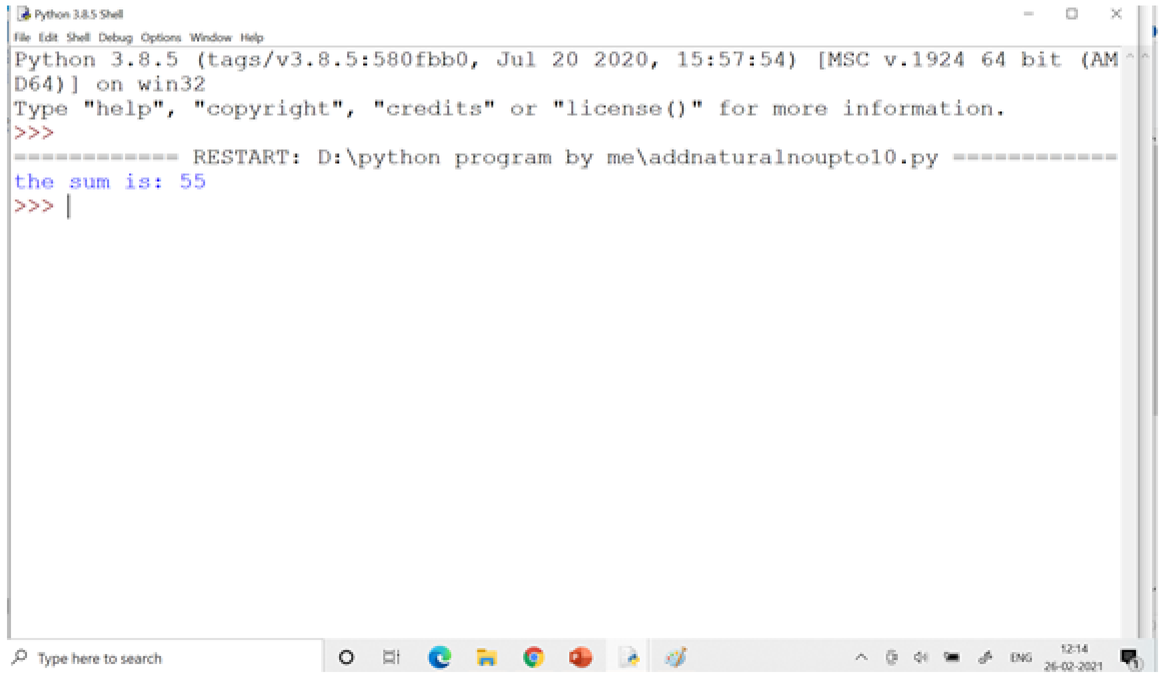
```
0 1 2 3 4 5 6 7 8 9 10
```

```
Example:
i=10
while(i>=0):
        print(i)
        i=i-1
Output:
10 9 8 7 6 5 4 3 2 1  0
```

Program to print addition of natural number from 1 to 10

Create file add.py

```
n=10
sum=0
i=1
while i <= 10:
     sum=sum+i
     i= i+1
print("the sum is:", sum)
```



```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============ RESTART: D:\python program by me\addnaturalnoupto10.py ============
the sum is: 55
>>> |
```

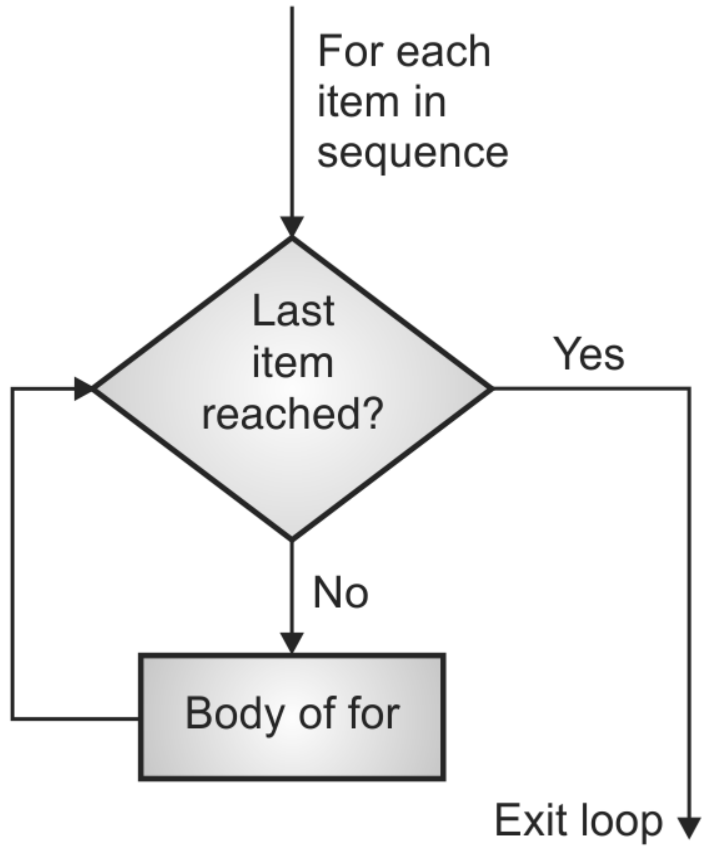
For Loop

- For loop provides a mechanism to repeat a task until a particular condition is True.
- It is usually known as a determinate or definite loop because the programmer knows exactly how many times the loop will repeat.
- The body of the for loop is executed for each member element in the sequence. Hence, it doesn't require explicit verification of Boolean expression controlling the loop (as in the while loop).
- The for…in statement is a looping statement used in Python to iterate over a sequence of objects.
- It is frequently used to iterate or traverse the data structures like list, tuple, or dictionary.

For Loop

Syntax of for Loop

```
for loop_contol_var in sequence:
    statement block
```



Flowchart

Example:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

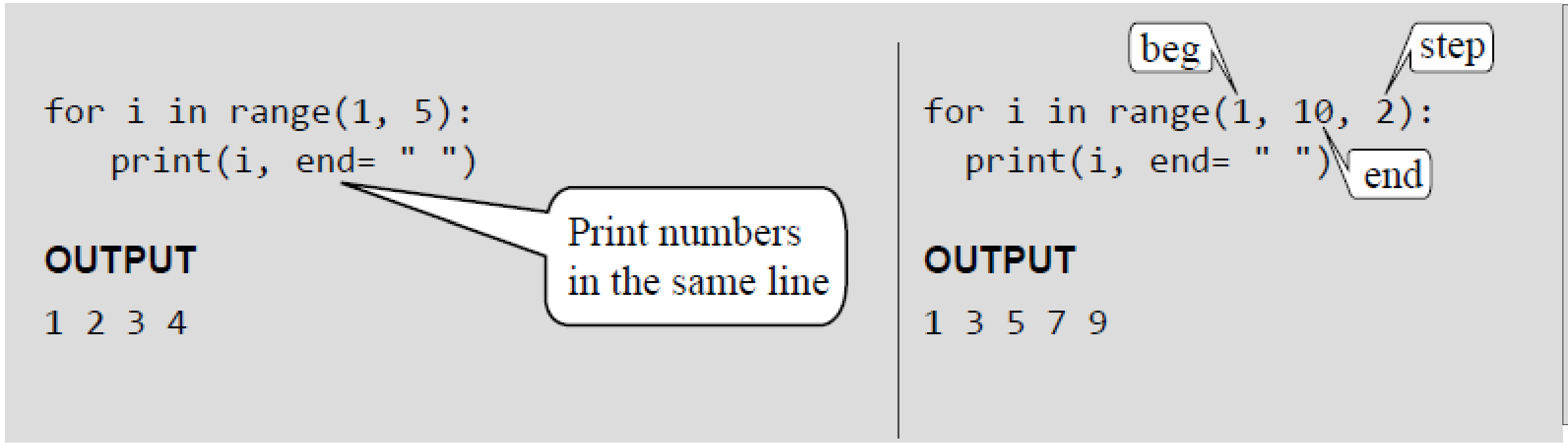
Output:

```
apple
banana
cherry
```

For Loop and Range() Function

- **The range() function is a built-in function in Python that is used to iterate over a sequence of numbers. The syntax of range() is range(beg, end, [step])**
- **The range() produces a sequence of numbers starting with beg (inclusive) and ending with one less than the number end. The step argument is option (that is why it is placed in brackets). By default, every number in the range is incremented by 1 but we can specify a different increment using step. It can be both negative and positive, but not zero.**

Examples:



```
for i in range(1, 5):
    print(i, end= " ")

OUTPUT
1 2 3 4
```

```
for i in range(1, 10, 2):
  print(i, end= " ")

OUTPUT
1 3 5 7 9
```

```
for i in range(10,0,-1):
   print(i)

Output:
10 9 8 7 6 5 4 3 2 1
```

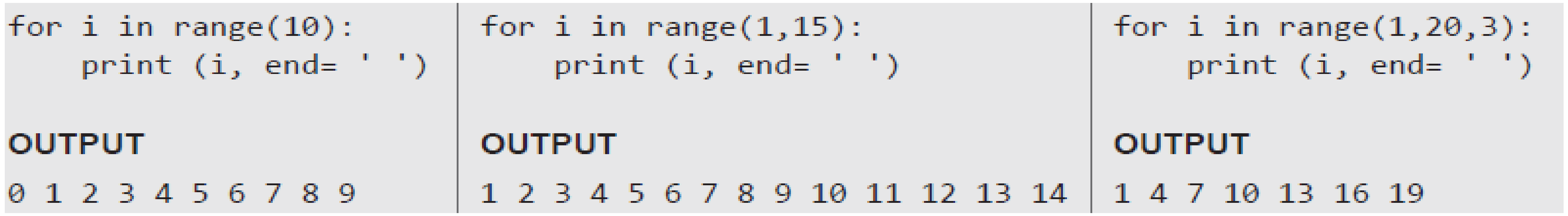
Range() Function

If range() function is given a single argument, it produces an object with values from 0 to argument-1. For example: range(10) is equal to writing range(0, 10).

• If range() is called with two arguments, it produces values from the first to the second. For example, range(0,10).

• If range() has three arguments then the third argument specifies the interval of the sequence produced. In this case, the third argument must be an integer. For example, range(1,20,3).

Examples:



```
for i in range(10):
    print (i, end= ' ')

OUTPUT
0 1 2 3 4 5 6 7 8 9
```

```
for i in range(1,15):
    print (i, end= ' ')

OUTPUT
1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

```
for i in range(1,20,3):
    print (i, end= ' ')

OUTPUT
1 4 7 10 13 16 19
```

Selecting Appropriate Loop

Loop Types:

- Entry-Controlled(pre-test) and Exit-Controlled(Post-test)
- Counter-controlled and condition-controlled(sentinel-controlled)

Entry-Controlled(pre-test) and Exit-Controlled(Post-test)

- Entry-Controlled(Pre-test) loop test the condition before loop start and Exit-Controlled(Post-test) loop test the condition after loop is executed.
- If the condition is not met in entry-controlled loop then loop will never execute. While in case of post-test, body of loop is executed unconditionally for the first time.
- If requirement is to have a pre-test loop, then choose for loop or while loop.

Condition-controlled and Counter-controlled Loops

Attitude	Counter-controlled loop	Condition controlled loop
Number of execution	Used when number of times the loop has to be executed is known in advance.	Used when number of times the loop has to be executed is not known in advance.
Condition variable	In counter-controlled loops, we have a counter variable.	In condition-controlled loops, we use a sentinel variable.
Value and limitation of variable	The value of the counter variable and the condition for loop execution, both are strict.	The value of the counter variable and the condition for loop execution, both are strict.
Example	`i = 0` `while(i<=10):` `    print(i, end = " ")` `    i+=1`	`i = 1` `while(i>0):` `    print(i, end = " ")` `    i+=1` `    if(i==10):` `        break`

Nested Loops

- Python allows its users to have nested loops, that is, loops that can be placed inside other loops. Although this feature will work with any loop like while loop as well as for loop.
- A for loop can be used to control the number of times a particular set of statements will be executed. Another outer loop could be used to control the number of times that a whole loop is repeated.
- Loops should be properly indented to identify which statements are contained within each for statement.

```
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
```

Nested Loops

Example:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]



for x in adj:
    for y in fruits:
        print(x+y)
```

Output:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

The Break Statement

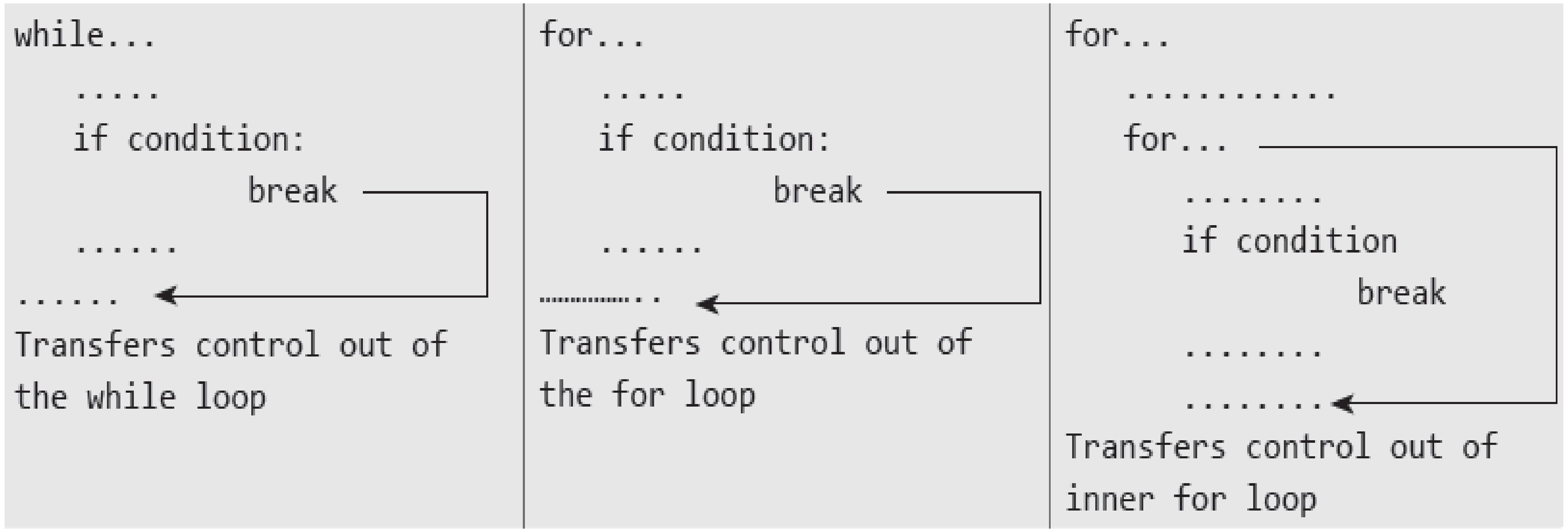
The *break* statement is used to terminate the execution of the nearest enclosing loop in which it appears. The break statement is widely used with for loop and while loop. When interpreter encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears.

Example:

```
i = 1
while i <= 10:
        print(i, end=" ")
        if i==5:
                break
        i = i+1
print("\n Done")
```

OUTPUT

```
1 2 3 4 5
Done
```



```
while...
    .....
    if condition:
            break
    ......
......
```

Transfers control out of the while loop

```
for...
    .....
    if condition:
            break
    ......
..............
```

Transfers control out of the for loop

```
for...
    ...........
    for...
        .......
        if condition
                break
        .......
        .......
```

Transfers control out of inner for loop

The Continue Statement

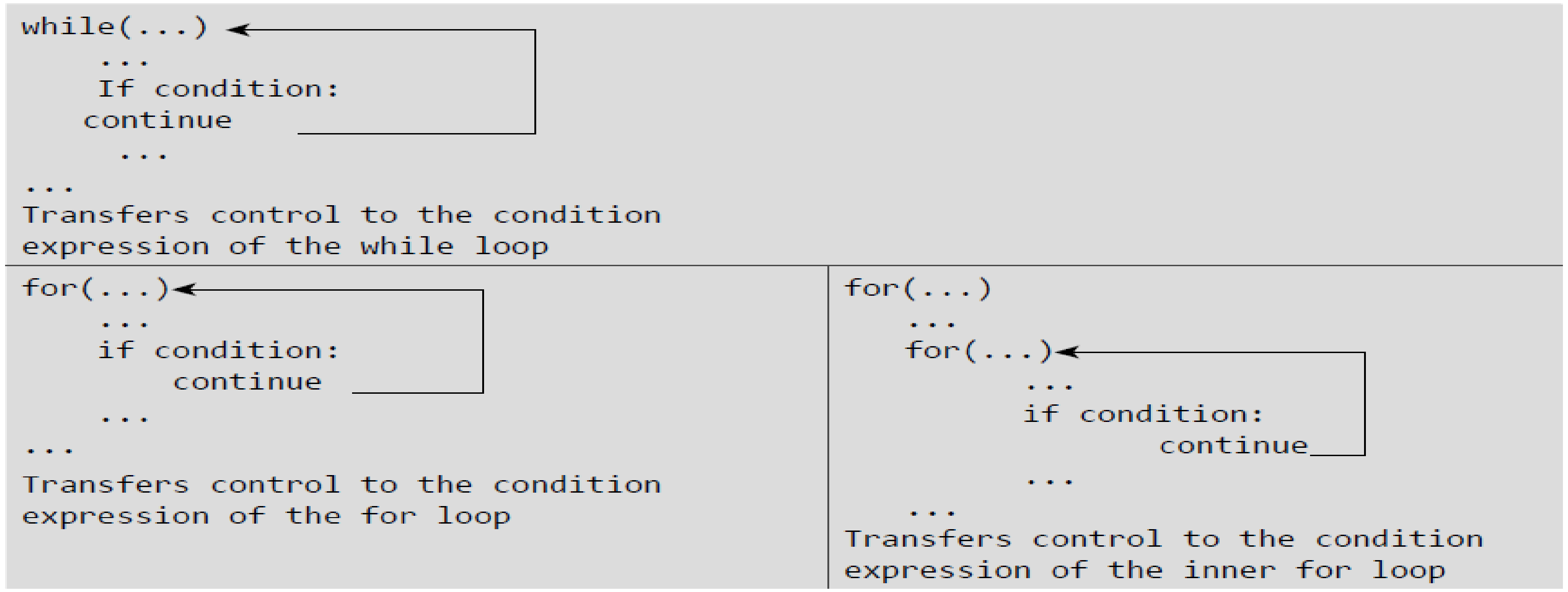
Like the break statement, the continue statement can only appear in the body of a loop. When the interpreter encounters a continue statement then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop.

Example:

```
for i in range(1,11):
    if(i==5):
        continue
    print(i, end=" ")
print("\n Done")
```

OUTPUT

```
1 2 3 4 6 7 8 9 10
Done
```



```
while(...)
    ...
    If condition:
   continue
     ...
...
Transfers control to the condition
expression of the while loop
```

```
for(...)
    ...
    if condition:
        continue
    ...
...
Transfers control to the condition
expression of the for loop
```

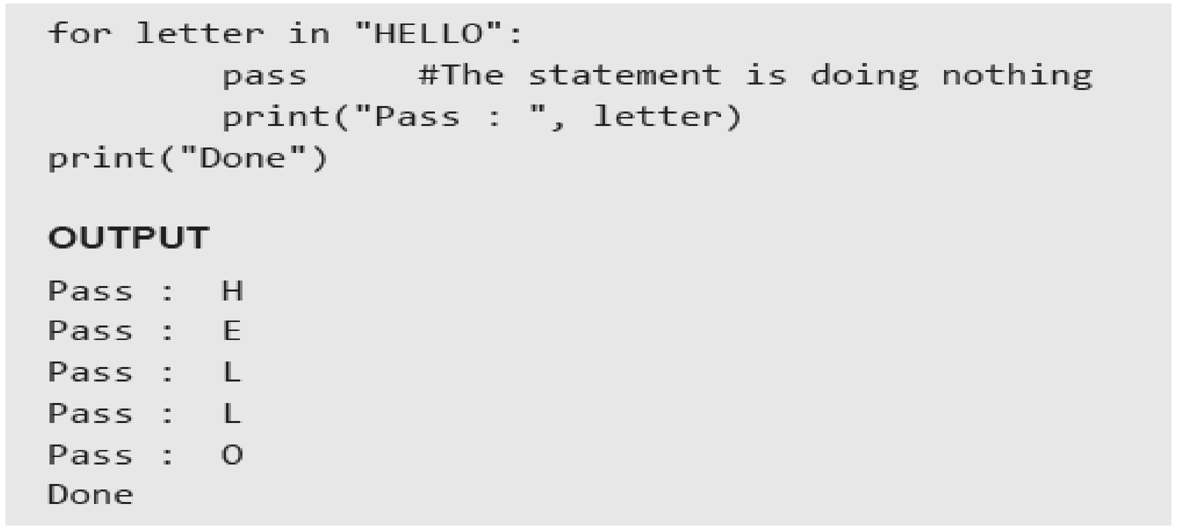
```
for(...)
    ...
    for(...)
        ...
        if condition:
            continue
        ...
    ...
Transfers control to the condition
expression of the inner for loop
```

The Pass Statement

Pass statement is used when a statement is required syntactically but no command or code has to be executed. It specified a *null* operation or simply No Operation (NOP) statement. Nothing happens when the pass statement is executed.

In Python programming, pass is a null statement. The difference between a comment and pass statement is that the interpreter ignores a comment entirely, while pass is not ignored. Comment is not executed but pass statement is executed but nothing happens.

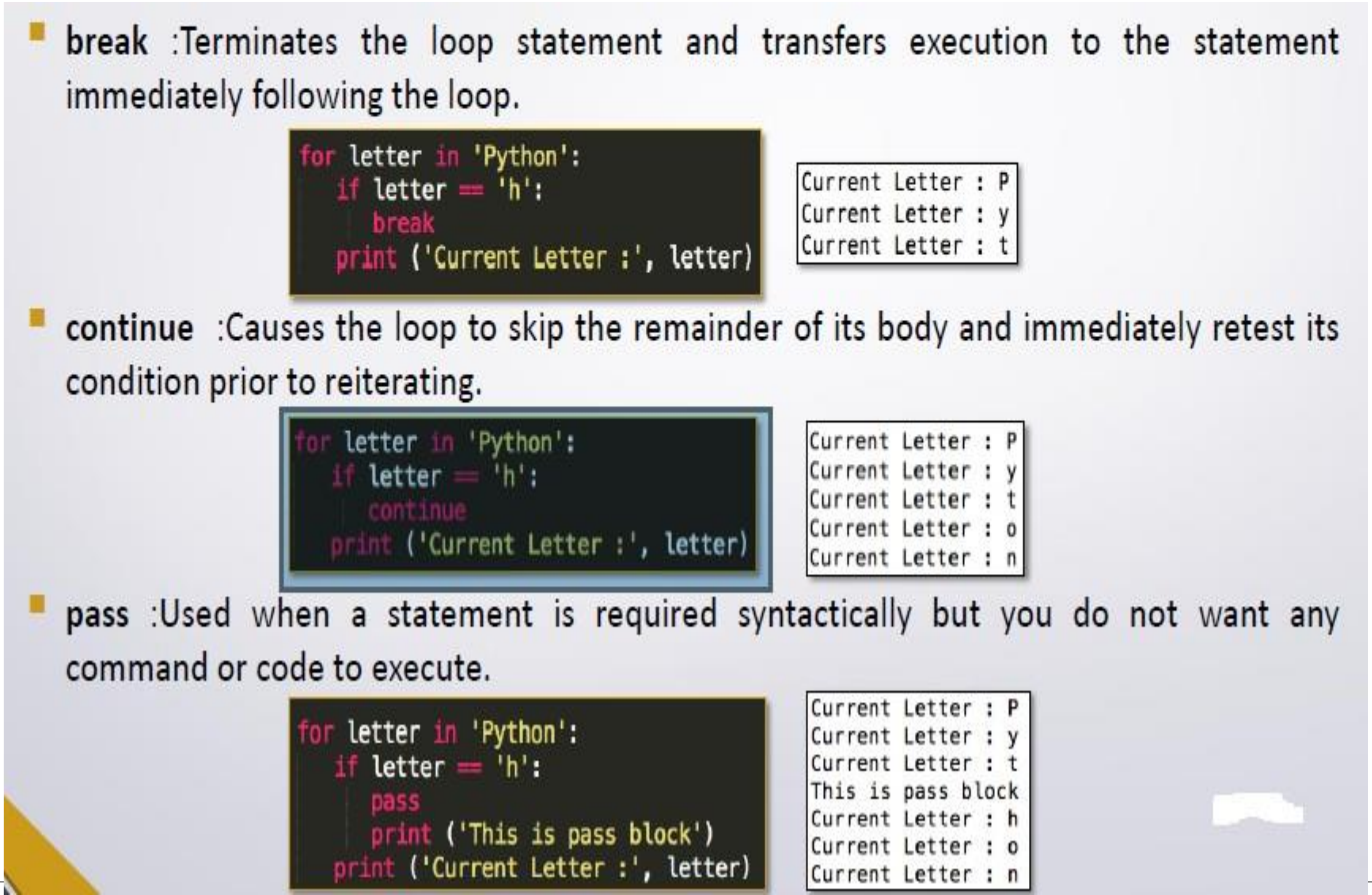
Example:



```
for letter in "HELLO":
        pass     #The statement is doing nothing
        print("Pass : ", letter)
print("Done")
```

OUTPUT

```
Pass :  H
Pass :  E
Pass :  L
Pass :  L
Pass :  O
Done
```



- **break** :Terminates the loop statement and transfers execution to the statement immediately following the loop.

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

```
Current Letter : P
Current Letter : y
Current Letter : t
```

- **continue** :Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

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

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

- **pass** :Used when a statement is required syntactically but you do not want any command or code to execute.

```python
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
```

The Else Statement Used With Loops

Unlike C and C++, in Python you can have the *else* statement associated with a loop statements. If the else statement is used with a *for* loop, the *else* statement is executed when the loop has completed iterating. But when used with the *while* loop, the *else* statement is executed when the condition becomes false.

Examples:

```
for letter in "HELLO":
       print(letter, end=" ")
else:
        print("Done")
```

OUTPUT

```
H E L L O Done
```

```
i = 1
while(i<0):
        print(i)
        i = i - 1
else:
        print(i, "is not negative so
loop did not execute")
```

OUTPUT

```
1 is not negative so loop did not execute
```

Other Data Type

List method and function

Methods:

- **reverse(): reverse the item**
- **Insert(): insert new item or list**
- **sort()**: Sorts the list in ascending order.
- **type(list)**: It returns the class type of an object.
- **append()**: Adds a single element to a list.
- **extend()**: Adds multiple elements to a list.
- **index(value)**: Returns the first appearance of the specified value.
- **pop(),remove(),del(),clear()**: delete item or list
- **clear()**: remove all item from list
- **Count()**:number of occurrences of item
- **copy()**:return copy of list

Function:

max(list): It returns an item from the list with max value.

min(list): It returns an item from the list with min value.

len(list): It gives the total length of the list.

list(seq): Converts a tuple into a list.

sum():sum of all element

List Methods

method	Program	Output

sort()	l= [5,2,3]
	 l.sort()
	 print(l)
		[2,3,5]

append()	l= [5,2,3]
	l.append(6)
	 print(l)
		[5,2,3,6]

extend()	x = [1, 2, 3]
	x.extend([4, 5])
	print(x)
		[1, 2, 3, 4, 5]

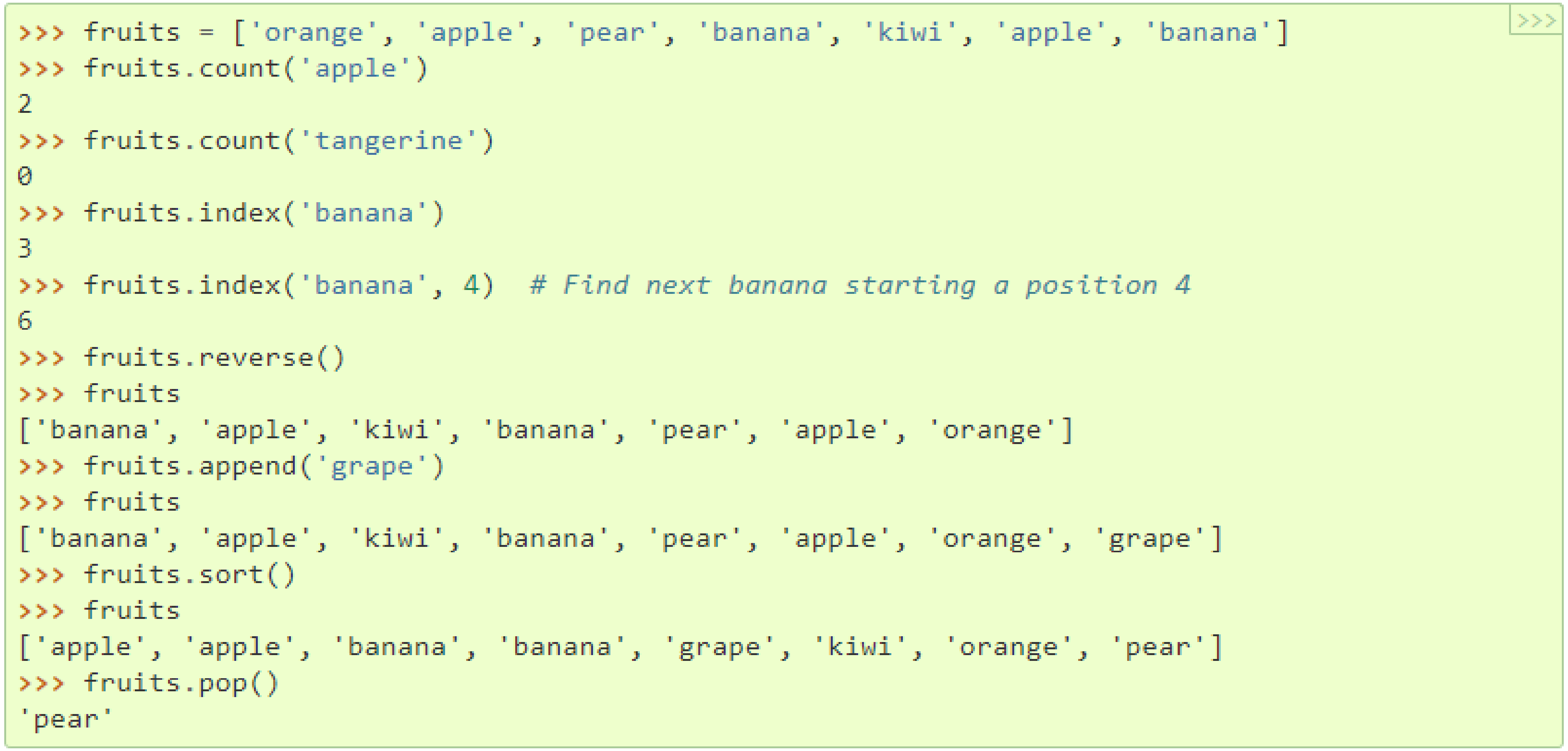
Copy()	x=[1,3,4]
	y=x.copy()
	print(y)
		[1,3,4]

index(item, start, end)	months = ['January', 'February', 'March']
	months.index('February')
		1

reverse()	l= [5,2,3]
	l.reverse()
	 print(l)
		[3,2,5]

Insert(index, value)	l= [5,2,3]
	l.insert(-2,6)
		[5,2,6,3]

List Methods example



```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Remove item from List or List

In python **del** is a keyword and **remove(), pop()** and clear() are in-built methods. The purpose of these are same but the behavior is different **remove()** method delete values or object from the list using value and **del** and **pop()** deletes values or object from the list using an index. Clear method delete all element of List.

Syntax:

```
del list_name[index] # To delete single value
del list_name # To delete whole list
list_name.remove(value)
list_name.pop(index)
List_name.clear()
```

Example :

```
#Program:
numbers = [1, 2, 3, 2, 3, 4, 5]

del numbers[2]
print(numbers)
  #[1,2,2,3,4,5]
numbers. remove(3)
print(numbers)
#[1,2,2,4,5]
numbers. pop(4)
print(numbers)

numbers.clear()
print(numbers)
```

Output:

```
[1,2,2,3,4,5]
[1,2,2,4,5]
[1,2,2,4]
[]
```

List Function

Function	Program	Output

max(list)	l= [5,2,3]
	print(max(l))
		5

min(list)	l= [5,2,3]
	print(min(l))
		2

len(list)	l= [5,2,3,4]
	print(len(l))
		4

list(seq)	t=(4,5,6)
	print(list(t))
		[4,5,6]

sum(list)	l= [5,2,3]
	print(sum(l))
		10

Tuples functions and methods

- **Methods:**
- **Count()**:number of occurrences of item
- **index()**: Returns the first appearance of the specified value.

Function:

- **max(tuple)**: It returns an item from the tuple with max value.
- **min(tuple)**: It returns an item from the tuple with min value.
- **sum()**:sum of all element
- **tuple(sequence):**convert into tuple

Tuples methods

Method	program	Output

Count()	x=(1,3,4,2,1,3)
	print(x.count(1))
		2

index(item,start,end)	months = ('January', 'February', 'March‘)
	months.index('February')
		1

Tuples functions

Function	Program	Output

max(tuple)	l= (5,2,3)
	print(max(l))
		5

min(tuple)	l= (5,2,3)
	print(min(l))
		2

len(tuple)	l= (5,2,3)
	print(len(l))
		3

tuple(seq)	l=[4,5,6]
	print(tuple(l))
		(4,5,6)

sum(tuple)	t= (5,2,3)
	print(sum(t))
		10

Dictionary functions and methods

Functions

- len(): number of items
- sort():sort by key
- cmp(): Compare dictionary
- str():string representation of a dictionary

Methods:

- clear(): remove all element
- copy(): copy into new dictionary
- get(key): get value by key
- keys(): all keys
- values(): all values
- item(): all content
- pop(key): delete value by key

Dictionary Function

Function	Program	Output

cmp(dict1, dict2)
Compares elements of both dict.
This method returns 0 if both dictionaries are equal, -1 if dict1 < dict2 and 1 if dict1 > dic2.
	dict1 = {'Name': 'Zara', 'Age': 7}
	dict2 = {'Name': 'Mahnaz', 'Age': 27}
	dict3 = {'Name': 'Abid', 'Age': 27}
	dict4 = {'Name': 'Zara', 'Age': 7}
	print ("Return Value :”,cmp (dict1, dict2))
	print (Return Value : " ,cmp (dict2, dict3))
	print "Return Value : " cmp (dict1, dict4))
		Return Value : -1
		Return Value : 1
		Return Value : 0

len(dict) Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.	dict = {'Name': 'Zara', 'Age': 7} print ("Length :“, len(dict))	2
str(dict) Produces a printable string representation of a dictionary	dict = {'Name': 'Zara', 'Age': 7} print ("Equivalent String : " str(dict))	Equivalent String : {'Age': 7, 'Name': 'Zara'}
type(variable) Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.	dict = {'Name': 'Zara', 'Age': 7} print ("Variable Type : " ,type(dict))	Variable Type : <type 'dict'>

Method	Program	Output

dict.clear()
Removes all elements of dictionary dict
	dict = {'Name': 'Zara', 'Age': 7}
	print ("Start Len :“,len(dict))
	dict.clear()
	 print ("End Len : " ,len(dict))
		Start Len : 2
		End Len : 0

dict.copy()
Returns a shallow copy of dictionary dict
	dict1 = {'Name': 'Zara', 'Age': 7}
	dict2 = dict1.copy()
	print ("New Dictionary : ”,str(dict2))
		New Dictionary : {'Age': 7, 'Name': 'Zara'}

dict.get(key, default=None) For key, returns value or default if key not in dictionary	dict = {'Name': 'Zabra', 'Age': 7} print ("Value : " ,dict.get('Age')) print ("Value : " ,dict.get('Education', "Never"))	Value : 7 Value : Never
dict.items() Returns a list of *dict*'s (key, value) tuple pairs	dict = {'Name': 'Zara', 'Age': 7} print ("Value : " ,dict.items())	Value : [('Age', 7), ('Name', 'Zara')]
dict.keys() Returns list of dictionary dict's keys	dict = {'Name': 'Zara', 'Age': 7} print ("Value : " , dict.keys())	Value : ['Age', 'Name']

dict.update(dict2)
Adds dictionary *dict2*'s key-values pairs to *dict*
	dict = {'Name': 'Zara', 'Age': 7}
	dict2 = {'Sex': 'female' }
	dict.update(dict2)
	print ("Value :“,dict)
		Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}

dict.values()	dict = {'Name': 'Zara', 'Age': 7}	Value : [7, 'Zara']