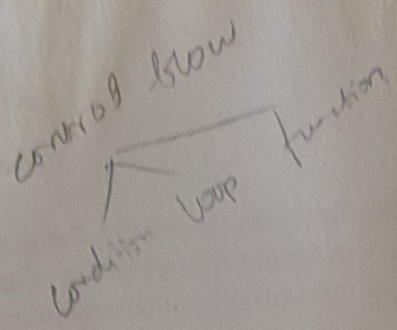


UNIT-II
CONTROL STATEMENTS

CONTROL STATEMENTS

CONTROL FLOW

- A program's **control flow** is the order in which the program code executes.
- The control flow of a Python program is regulated by conditional statements, loops, and function calls.
- Python has *three* types of control structures:
  - Sequential - default mode
  - Selection - used for decisions and branching
  - Repetition - used for looping, i.e., repeating a piece of code multiple times.

1. Sequential

Sequential statements are a set of statements whose execution process happens in a sequence. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

2. Selection / Decision making statements (branching)

In Python, the selection statements are also known as *Decision making statements* or *branching statements*. The selection statement allows a program to test several conditions and execute instructions based on which condition is true. Some Decision Making Statements are:

1. Simple if
2. if-else
3. nested if
4. if-elif-else

3. Repetition (looping)

The repetition statement allows the program code to be executed multiple times till the condition is satisfied. In Python, we have two loops / repetitive statements for loop and while loop

INDENTATION

Python does not use braces { } to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation. The number of spaces in the indentation is different but all statements within the block must be indented the same amount. For example

```
if a > b:
    print ("True")
else:
print ("False")
```

However, the following block generates an error because the print statement with false is not intended properly

```
if  a > b:
    print ("Answer")
    print ("True")
else:
    print "(Answer")
print ("False")
```

IF STATEMENTS

If statements are control flow statements that help us to run a particular code, but only when a certain condition is met or satisfied.

a. Simple if

A *simple if* only has one condition to check. Syntax

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

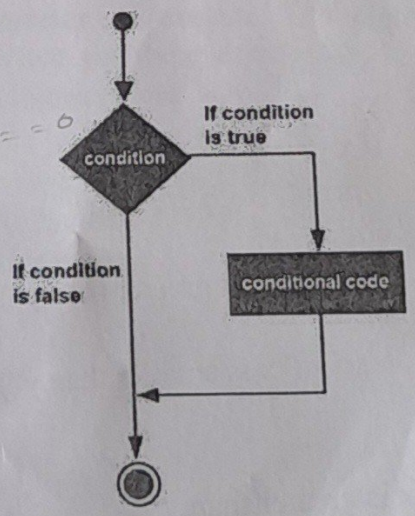
If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.

Example

```
n = 10
if n % 2 == 0:
  print("n is an even number")
```

Output

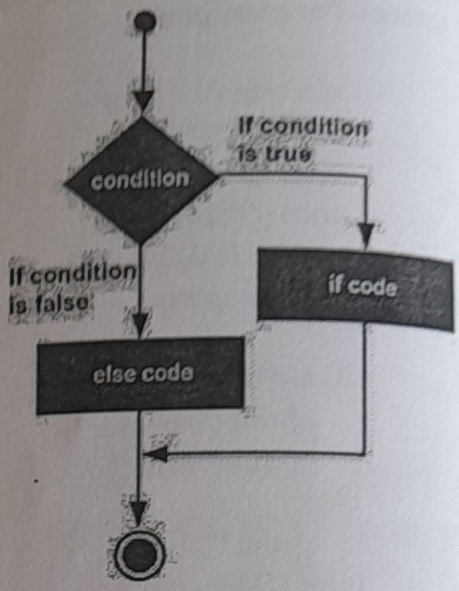
```
n is an even number
```



b. if-else Statement

The *if-else statement* evaluates the condition and if the test condition is True it will execute the body of if, but if the condition is False, then the body of else is executed. Syntax

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



Example

```
n = 5
if n % 2 == 0:
  print("n is even")
else:
  print("n is odd")
```

Output

n is odd

c. if-elif-else Statement

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. Syntax

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

Example

```
x = 15
y = 12
if x == y:
  print("Both are Equal")
elif x > y:
  print("x is greater than y")
else:
  print("x is smaller than y")
```

Output

x is greater than y

d. Nested if

Nested if statements are if statement inside another if statement. Syntax

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

Example

```
a = 5
b = 10
c = 15
if a > b:
   if a > c:
     print("a value is big")
   else:
     print("c value is big")
elif b > c:
    print("b value is big")
else:
   print("c is big")
```

Output

c is big

STATEMENTS AND EXPRESSION ⇒ It does not return anything

A statement is an instruction that the Python interpreter can execute. The user have seen two kinds of statements: **print and assignment.** When you type a statement on the command line, Python executes it and displays the result, if there is one.

The result of a print statement is a value. Assignment statements don't produce a result. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute. For example,

```
print 1
x = 2
print x
```

we should not create variable for print statement

Above code produces the output

```
1
2
```

The assignment statement does not produce an output.

EXPRESSIONS

An expression is a combination of values, variables, and operators. Expression statements are used to compute and write a value, or to call a procedure. Following are a few types of python expressions:

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The above code will get all the number within 10 and put them in a list

```
>>> {x:x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

The above code will get all the numbers within 5 as the keys and will keep the corresponding squares of those numbers as the values.

```
>>> x = "1" if True else "2"
>>> x
'1'
```

The above code is an example of conditional expression. Although expressions contain values, variables, and operators, not every expression contains all of these elements. A value all by itself is considered an expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

When the Python interpreter displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But if you use a print statement, Python displays the contents of the string **without the quotation marks.**

```
>>>message = 'Hello, World!'
>>>message
```

Output `'Hello, World!'`

```
>>> print message
```

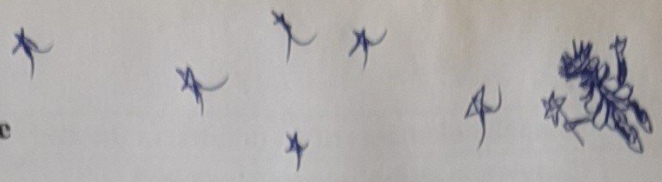
Output `Hello, World!`

STRINGS

Strings in python are surrounded by either single quotation marks, or double quotation marks. 'hello' is the same as "hello". User can display a string literal with the print() function.

Example

```
print("Hello")
print('Hello')
```



Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string. Example

```
a = "Hello"
print(a)
```

User can assign a ***multiline string*** to a variable by using three quotes. Example

```
a = """python is a scripting language
can be divided into client side and server
side scripting language"""
print(a)
```

Output

```
python is a scripting language
can be divided into client side and server
side scripting language
```

Strings Arrays

Like many other popular programming languages strings in Python are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string. Example

```
a = "Hello, World!"
print(a[1])
```

Output

```
e
```

String Length

To get the length of a string, use the len() function. Example

```
a = "Hello, World!"
print(len(a))
```

Output

```
13
```

String Functions

capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
format()	Formats specified values in a string
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
islower()	Returns True if all characters in the string are lower case
isupper()	Returns True if all characters in the string are upper case

join()	Joins the elements of an iterable to the end of the string
lower()	Converts a string into lower case
replace()	Specified value is replaced with a specified value
split()	Splits the string at the specified separator, and returns a list
swapcase()	Swaps cases, lower case becomes upper case and vice versa
upper()	Converts a string into upper case

BOOLEAN EXPRESSION

Booleans represent one of two values True or False. In programming user often need to know if an expression is True or False. User can evaluate any expression in Python, and get one of two answers, True or False. When you compare two values, the expression is evaluated and Python returns the Boolean answer.

Example

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Output

```
True
False
False
```

When you run a condition in an if statement, Python returns True or False. Example

```
a = 200
b = 33
if b > a:
        print("b is greater than a")
else:
        print("b is not greater than a")
```

Output

```
b is not greater than a
```

Evaluate Values and Variables

The bool() function allows you to evaluate any value, and give you True or False in return. Example

```
x = "Hello"
y = 15
print(bool(x))
print(bool(y))
```

output

```
True
True
```

Functions can return a Boolean

User can create functions that returns a Boolean Value:. Example

```
def myFunction()
    return True
print(myFunction())
```

Output

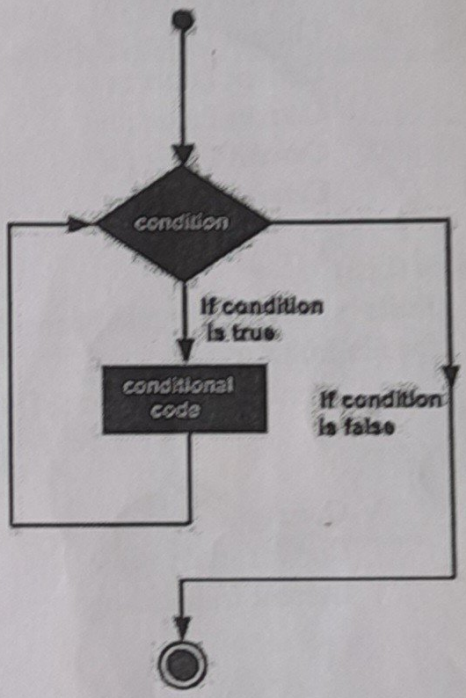
```
True
```

WHILE LOOP

The *while loops* are used to execute a block of statements repeatedly until a given condition is satisfied. Then, the expression is checked again and, if it is still true, the body is executed again. This continues until the expression becomes false. The syntax of a while loop in Python programming language is

Syntax :

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



Example

```
c = 0
while (c < 5):
    print ('The count is:', c)
    c = c + 1
print ("Good")
```

Output

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
Good
```

FOR LOOP

A *for loop* is used to iterate over a sequence that is a list, tuple, dictionary, or a set. User can execute a set of statements once for each item in a list, tuple, or dictionary.

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

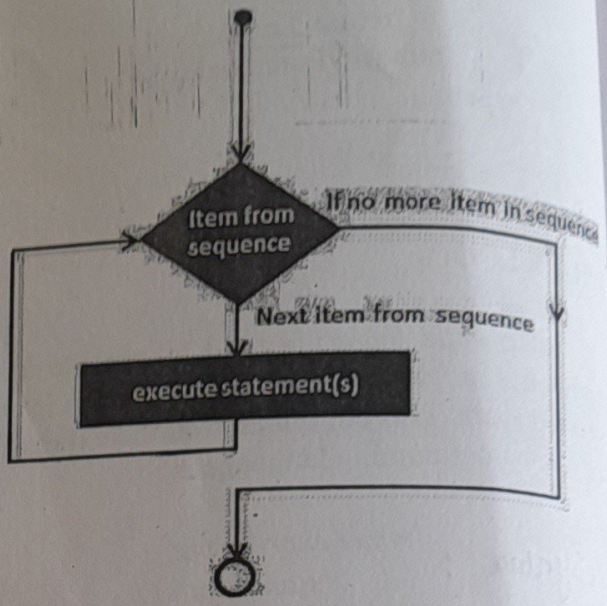
The range() function

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

```
list(range(5))
[0, 1, 2, 3, 4]
for var in list(range(5)):
        print (var)
```

Output

0
1
2
3
4



Example (String)

```
for t in 'Python':
        print ('Current Letter :', t)
```

Output

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

Example (List)

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

Output

Current fruit : banana
Current fruit : apple
Current fruit : mango

BREAK STATEMENT

The break keyword is used to break out of a loop. The break statement takes care of terminating the loop in which it is used. The **break** statement can be used in both *while* and *for* loops

If the break statement is used inside nested loops, the current loop is terminated, and the flow will continue with the code followed that comes after the loop. Flow chart of break statement is given below

Syntax:

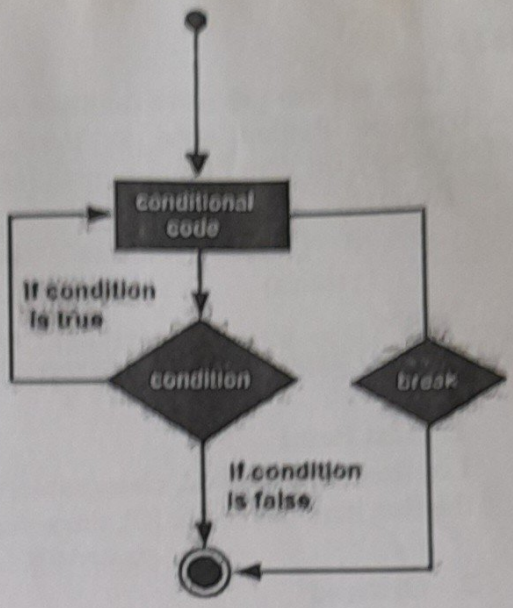
```
break
```

Example (End the loop if i is larger than 3)

```
for i in range(9):
    if i > 3:
        break
    print(i)
```

Output

```
0
1
2
3
```

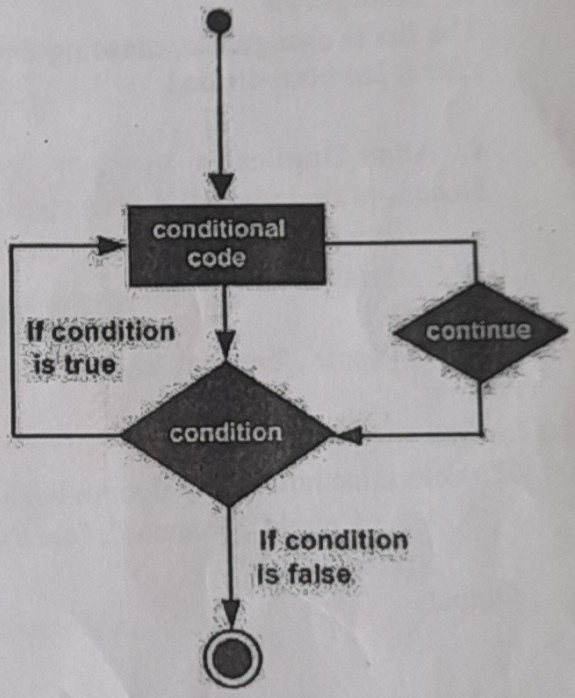


CONTINUE STATEMENT

The continue keyword is used to end the current iteration in a for loop (or a while loop), and continues to the next iteration. The **continue** statement can be used in both *while* and *for* loops

Syntax:

```
continue
```



Example

```
i = 0
while i < 9:
    i += 1
    if i == 3:
        continue
print(i)
```

Output

```
1
2
4
5
6
7
8
9
```

PYTHON COLLECTIONS DATATYPES

add or remove

There are four collection data types in the Python programming language

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered and changeable. No duplicate members.

LISTS

Lists are used to store multiple items in a single variable. Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage. Lists are created using square brackets.

Example

```
a = ["apple", "banana", "cherry"]
print(a)
```

Output

['apple', 'banana', 'cherry']

1. List Items

List items are ordered, changeable, and allow duplicate values. List items are indexed, the first item has index [0], the second item has index [1] etc.

2. Ordered

Lists are ordered means that the items have a defined order and that order will not change. New items to a list will be placed at the end of the list.

3. Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

4. Allow Duplicates

Since lists are indexed, lists can have items with the same value. Example

```
b = ["apple", "banana", "cherry", "apple", "cherry"]
print(b)
```

Output

['apple', 'banana', 'cherry', 'apple', 'cherry']

List Length

To determine how many items a list has, use the len() function: Example

```
a = ["apple", "banana", "cherry"]
print(len(a))
```

Output

3

List Items - Data Types

List items can be of any data type. Example

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

Output

['apple', 'banana', 'cherry']
[1, 5, 7, 9, 3]
[True, False, False]

PYTHON LIST SLICING

To access a range of items in a list, you need to slice a list. One way to do this is to use the simple slicing operator. With this operator you can specify where to start the slicing, where to end and specify the step.

Slicing a List

If L is a list, the expression L[start : stop : step] returns the portion of the list from index start to index stop, at a step size step.

Syntax

```
L[start:stop:step]
```

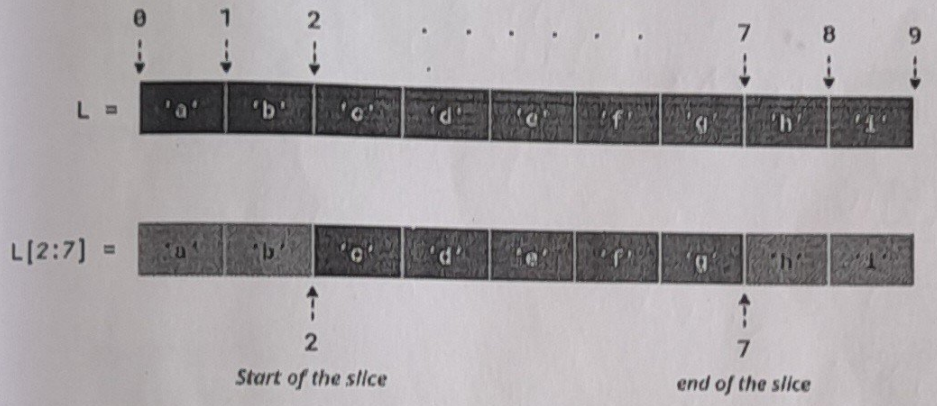
Start position *End position* *The Increment*

Example

```
m = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(m[2:7])
```

Output

```
['c', 'd', 'e', 'f', 'g']
```



Note that the item at index 7 'h' is not included.

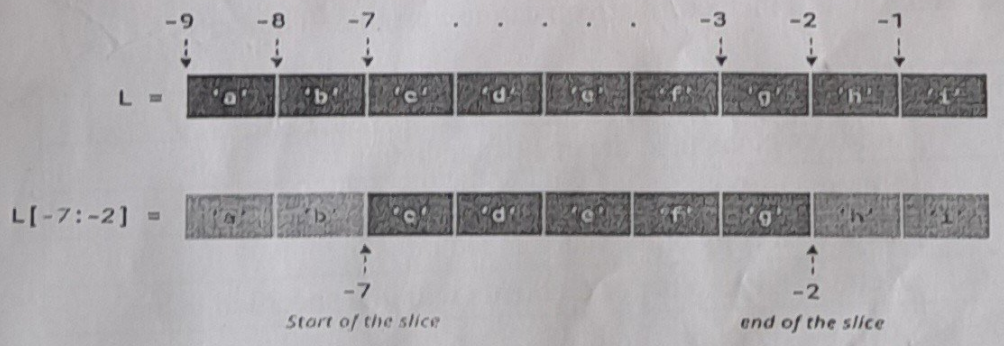
Slice with Negative Indices

You can also specify negative indices while slicing a list.

```
m = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(m[-7:-2])
```

Output

```
['c', 'd', 'e', 'f', 'g']
```



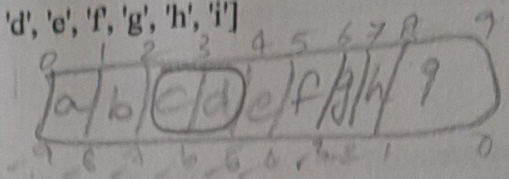
Slice with Positive & Negative Indices

You can specify both positive and negative indices at the same time.

```
m = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(m[2:-5])
```

Output

```
['c', 'd']
```



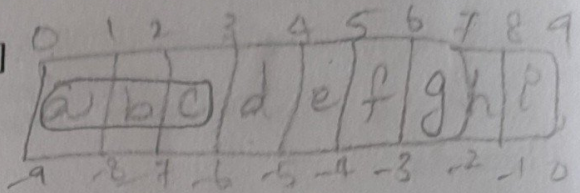
Slice at Beginning & End

Omitting the start index starts the slice from the index 0. Meaning, L[:stop] is equivalent to L[0:stop]

```
m = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(m[:3])
```

Output

```
['a', 'b', 'c']
```

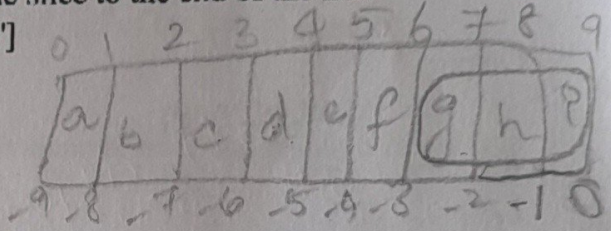


Whereas, omitting the stop index extends the slice to the end of the list.

```
m = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(m[6:])
```

Output

```
['g', 'h', 'i']
```



Insert Multiple List Items

You can insert items into a list without replacing anything. Simply specify **a zero-length slice.**

```
m = ['a', 'b', 'c']
m[:0] = [1, 2, 3]
print(L)
```

Output

```
[1, 2, 3, 'a', 'b', 'c']
```

Insert at the end

```
m = ['a', 'b', 'c']
m[len(L):] = [1, 2, 3]
print(L)
```

Output

```
['a', 'b', 'c', 1, 2, 3]
```

LIST METHODS

Python has a set of built-in methods that you can use on lists. They are

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value

extend()	Add the elements of a list to the end of the current list
index()	Returns the index of the first element with specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

append Method

The append() method adds a single item to the end of the list. This method does not return anything; it modifies the list in place. Syntax

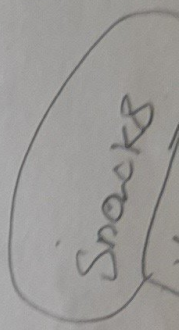
list.append(item)

Example

```
fruits = ['apple', 'banana', 'cherry']
fruits.append('orange')
print(fruits)
```

Output

['apple', 'banana', 'cherry', 'orange']



count Method

The count() method returns the number of elements with the specified value

```
fruits = ['apple', 'banana', 'cherry']
x = fruits.count("cherry")
print(x)
```

Output

1

insert Method

The insert() method inserts the specified value at the specified position

```
fruits = ['apple', 'banana', 'cherry']
fruits.insert(2, "orange")
print(fruits)
```

Output

['apple', 'banana', 'orange', 'cherry']

remove Method

The remove method will remove the item with the specified value

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove("banana")
print(fruits)
```

Output

['apple', 'cherry']

Built-In Functions Used on Lists

Built-In Functions	Description
✓ len()	Returns the numbers of items in a list.
✓ sum()	Returns the sum of numbers in the list.
✓ any()	Returns *True* if any of the Boolean values
✓ all()	Returns *True* if all Boolean values in list are *True*, else *False*
✓ sorted()	Returns a modified copy of the list

Example 1

```
list1 = [123, 'xyz', 'zara']
print "List length : ", len(list1)
```

Output

```
List length : 3
```

Example 2

```
num = [1,2,3,4,5,1,4,5]
s = sum(num)
print(s)
```

Output

```
25
```

MUTABILITY

Lists Are Mutable

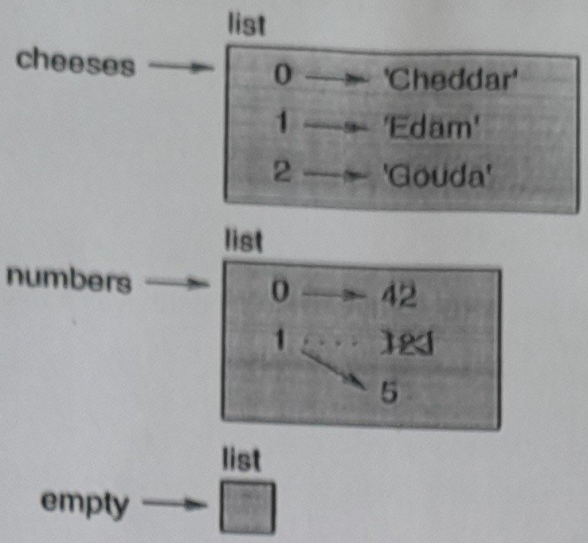
The syntax for accessing the elements of a list is the same as for accessing the characters of a string i.e. the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0. It is possible to add / delete / replace any value in list at any time. Syntax

```
list_name[index]=value
```

Lists are mutable ie when the bracket operator appears on the left side of an assignment it identifies the element of the list that will be assigned

```
>>>numbers = [42, 123]
>>>numbers[1] = 5
>>>numbers
[42, 5]
```

The first element of numbers which used to be 123, is now 5. The following picture shows the state diagram for cheeses, numbers and empty.



Example

```
lis = [2, 3, 4, 1, 32, 4]
print("Old list:" lis)
lis[3] = 100
print("New list:" lis)
```

Output

```
Old list: [2, 3, 4, 1, 32, 4]
New list: [2, 3, 4, 100, 32, 4]
```

ALIASING / CLONING THE LIST

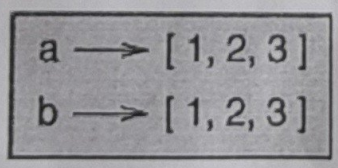
An object with more than one reference has more than one name we say that the object is **aliased.** If **a** refers to an object and you assign b = a, then both variables refer to the same object:

Syntax: new_List = old_List

Example

```
>>> a = [1, 2, 3]
>>> b = a
>>> print a
>>> print b
[1, 2, 3]
[1, 2, 3]
```

Following Figure shows the values stored.



The association of a variable with an object is called a **reference**. In this example, there are two references to the same object. An object with more than one reference has more

than one name, so we say that the object is **aliased**. If the aliased object is mutable, changes made with one alias affect the other:

```
>>>b[0]=20
>>>print(a)
>>>print(b)
[20, 2, 3]
[20, 2, 3]
```

You can use slicing operator to actually copy the list (also known as a shallow copy).

```
aa = ['a', 'b', 'c', 'd', 'e']
bb = aa[:]
print(bb)
print(aa is bb)
```

Output

```
['a', 'b', 'c', 'd', 'e']
False
```

LIST PARAMETERS / PASSING LIST TO FUNCTION

When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. While passing a list to a function the elements of list will change due to function operation because list is mutable. Example

Example

```
def test(num, nums):
    num=100
    nums[0]=500
x=1
y=[1, 2, 3]
print("x is" ,x)
print("y[0] is", y[0])
test(x,y)
print("After parameter passing and calling function")
print("x is" ,x)
print("y[0] is", y[0])
```

Output

```
x is 1
y[0] is 1
After parameter passing and calling function
x is 1
y[0] is 500
```

In the above code after test() function is invoked x value is 1 because number is immutable . but y is changed to 500 because list is mutable.

TUPLE VS LIST

Similarities

1. Length function used in tuple and list are same
2. Indexing and slicing are same
3. Concatenation and Exponentaion are possible
4. In and not in operation are allowed

Differences

1. Tuples are immutable and lists are mutable

TUPLE

A Tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma

```
>>> t1 = 'a',
>>>type(t1)
<class 'tuple'>
```

A value in parentheses is not a tuple

```
>>> t2 = ('a')
>>>type(t2)
<class 'str'>
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>>t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>>t
('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a built-in function, you should avoid using it as a variable name.

TUPLES AS RETURN VALUES

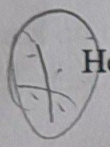
Strictly speaking, a function can only return **one value**, but if the value is a tuple, the effect is the same as returning **multiple values**. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y. It is better to compute them both at the same time.

The built-in function divmod takes two arguments and returns a tuple of two values: the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)
>>>t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>>quot, rem = divmod(7, 3)
>>>quot
2
>>>rem
1
```



Here is an example of a function that returns a tuple:

```
def mm(t):
        return min(t), max(t)
```

max and min are built-in functions that find the largest and smallest elements of a sequence. Mm computes both and returns a tuple of two values.

SET

Sets are used to store multiple items in a single variable. Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage. A set is a collection which is both *unordered* and *unindexed*. Sets are written with curly brackets. Example

```
s={"apple", "banana", "cherry"}
print(s)
```

Output

```
{'banana', 'apple', 'cherry'}
```

Sets are unordered, so you cannot be sure in which order the items will appear.

Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

1. **Unordered**
   Unordered means that the items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
2. **Unchangeable**
   Sets are unchangeable, meaning that we cannot change the items after the set has been created. Once a set is created, you cannot change its items, but you can add new items.
3. **Duplicates Not Allowed**
   Sets cannot have two items with the same value. Example
   Duplicate values will be ignored:

```
s={"apple", "banana", "cherry", "apple"}
print(s)
```

Output

```
{'banana', 'cherry', 'apple'}
```

Get the Length of a Set

To determine how many items a set has, use the len() method. Example

```
s={"apple", "banana", "cherry"}
print(len(s))
```

Output

```
3
```

Set Items - Data Types

Set items can be of any data type. Example

```
s1={"apple", "banana", "cherry"}
s2={1, 5, 7, 9, 3}
s3 = {True, False, False}
```

Output

```
{'cherry', 'apple', 'banana'}
{1, 3, 5, 7, 9}
{False, True}
```

Different Datatype

A set can contain different data types. Example

```
s1 = {"abc", 34, True, 40, "male"}
```

Output

```
{True, 34, 40, 'male', 'abc'}
```

type()

In python sets are defined as objects with the data type 'set'. Example

```
myset={"apple", "banana", "cherry"}
print(type(myset))
```

Output

```
<class 'set'>
```

DICTIONARY

Dictionaries are used to store data values in key:value pairs. A dictionary is a collection which is ordered*, changeable and does not allow duplicates. Dictionaries are written with curly brackets, and have keys and values: Example

```
dic = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
print(dic)
```

Output

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates. Dictionary items are presented in key:value pairs, and can be referred to by using the key name. Example

```
dic = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
print(dic["brand"])
```

Output

```
Ford
```

1. **Ordered or Unordered**
   When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change. Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.
2. **Changeable**
   Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
3. **Duplicates Not Allowed**
   Dictionaries cannot have two items with the same key. Example

```
dic = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964,
 "year": 2020
}
print(dic)
```

Output

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Dictionary Length

To determine how many items a dictionary has, use the len() function. Example

```
print(len(dic))
```

Output

```
3
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type. Example

```
thisdict = {
 "brand": "Ford",
 "electric": False,
 "year": 1964,
 "colors": ["red", "white", "blue"]
}
```

Output print (thisdict)

```
{'brand': 'Ford', 'electric': False, 'year': 1964, 'colors': ['red', 'white', 'blue']}
```

type()

```
<class 'dict'>
```

Example

Print the data type of a dictionary:

```
thisdict = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
print(type(thisdict))
```

Output

```
<class 'dict'>
```