UNIT – II

1. Operators:

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators, Bitwise Operators
5. Membership Operators
6. Identity Operators

1.1. Arithmetic Operators

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Example:

```
a = 21
b = 10
c = 0
c = a + b
print("a+b=",c)
c = a - b
print("a-b=" ,c)
c = a * b
print("a*b=" ,c)
c = a / b
print("a/b=" ,c)
c = a % b
print("a%b=", c)
```

```
a = 2
b = 3
c = a**b
print("a pow b=", c)
a = 10
b = 5
c = a//b
print("a//b=" ,c)
```

Output

```
a+b= 31
a-b= 11
a*b= 210
a/b= 2.1
a%b= 1
a pow b= 8
a//b= 2
```

1.2. Relational Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.
Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example:

```
a = 21
```

```
b = 10
if ( a == b ):
        print "Line 1 - a is equal to b"
else:
        print "Line 1 - a is not equal to b"
```

Output:

```
Line 1 - a is not equal to b
```

1.3 Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Example:

```
a = 21
b = 10
c = 0
c = a + b
print "Line 1 - Value of c is ", c
c += a
print "Line 2 - Value of c is ", c
c *= a
print "Line 3 - Value of c is ", c
```

```
c /= a
print "Line 4 - Value of c is ", c
c = 2
c %= a
print "Line 5 - Value of c is ", c
c **= a
print "Line 6 - Value of c is ", c
c //= a
print "Line 7 - Value of c is ", c
```

Output

```
Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864
```

1.4 Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows:

```
a = 0011 1100
b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011
```

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands.	(a & b) = 12 (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right	The left operands value is moved right by the number of bits specified by the	a >> 2 = 15 (means 0000 1111)

Shift	right operand.	

Example:

```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0
c = a & b;        # 12 = 0000 1100
print("a&b=",c)
c = a | b;        # 61 = 0011 1101
print("a|b=",c)
c = a ^ b;        # 49 = 0011 0001
print("a^b=",c)
c = ~a;           # -61 = 1100 0011
print("~a=",c)
c = a << 2;       # 240 = 1111 0000
print("a<<2",c)
c = a >> 2;       # 15 = 0000 1111
print("a>>2",c)
```

Output:

```
#a&b= 12
#a|b= 61
#a^b= 49
#~a= -61
#a<<2 240
#a>>2 15
```

1. 5 Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not (a and b) is false.

1.6 Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below:

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

1.7 Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

2. Expressions and order of evaluations

Python Operators Precedence

The following table lists all operators from highest precedence to lowest

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operator precedence affects how an expression is evaluated.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies 3*2 and then adds into 7. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

```
a = 20
b = 10
c = 15
d = 5
e = 0
e = (a + b) * c / d                #( 30 * 15 ) / 5
print "Value of (a + b) * c / d is ", e
e = ((a + b) * c) / d               # (30 * 15 ) / 5
print "Value of ((a + b) * c) / d is ", e
e = (a + b) * (c / d);              # (30) * (15/5)
print "Value of (a + b) * (c / d) is ", e
e = a + (b * c) / d;                # 20 + (150/5)
print "Value of a + (b * c) / d is ", e
```

When you execute the above program, it produces the following result:

```
Value of (a + b) * c / d is 90
Value of ((a + b) * c) / d is 90
Value of (a + b) * (c / d) is 90
Value of a + (b * c) / d is 50
```

3. Data Structures:

Python language support 4 types of data structures. They are:

1. Lists
2. Tuples
3. Dictionary
4. Sets

3. 1 Lists

Lists are Python's most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, even other lists. Python lists do the work of most of the collection data structures you might have to implement manually in lower-level languages such as C. In terms of some of their main properties, Python lists are:

Ordered collections of arbitrary objects

From a functional view, lists are just a place to collect other objects, so you can treat them as a group. Lists also define a left-to-right positional ordering of the items in the list.

Accessed by offset

Just as with strings, you can fetch a component object out of a list by indexing the list on the object's offset. Since lists are ordered, you can also do such tasks as slicing and concatenation.

Variable length, heterogeneous, arbitrarily nestable

Unlike strings, lists can grow and shrink in place (they're variable length), and may contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can contain other complex objects, lists also support arbitrary nesting; you can create lists of lists of lists, and so on.

Of the category mutable sequence

In fact, sequence operations work the same on lists as on strings. On the other hand, because lists are mutable, they also support other operations strings don't, such as deletion, index assignment, and methods.

Arrays of object references

Technically, Python lists contain zero or more references to other objects. If you've used a language such as C, lists might remind you of arrays of pointers. Fetching an item from a Python lists is about as fast as indexing a C array; in fact, lists really are C arrays inside the Python interpreter. Moreover, references are something like pointers (addresses) in a language such as C, except that you never process a reference by itself; Python always follows a reference to an object whenever the reference is used, so your program only deals with objects. Whenever you stuff an object into a data structure or variable name, Python always stores a reference to the object, not a copy of it (unless you request a copy explicitly). Use the square brackets for slicing along with the index or indices to obtain value available at that index.

Example

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

Output

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

Updating

You can update single or multiple elements of lists

```
list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2];
list[2] = 2001;
print "New value available at index 2 : "
print list[2];
```

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Deleting List Elements

Use the del statement to delete elements.

Ex:

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1;
del list1[2];
```

```
print "After deleting value at index 2 : "
print list1;
```

output:
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]

Basic List Operations

Python Expression	Results
len([1, 2, 3])	3
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']
3 in [1, 2, 3]	True
for x in [1, 2, 3]: print x,	1 2 3

Indexing, Slicing

L = ['spam', 'Spam', 'SPAM!']

Python Expression	Results
L[2]	'SPAM!'
L[-2]	'Spam'
L[1:]	['Spam', 'SPAM!']

Built-in List Functions

1. len()

The method len() returns the number of elements in the list.

```
list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']
print "First list length : ", len(list1);
print "Second list length : ", len(list2);
```

Output:

First list length : 3
Second lsit length : 2

2. max()

The method max returns the elements from the list with maximum value.

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
print "Max value element : ", max(list1);
print "Max value element : ", max(list2);
```

output:

Max value element : zara
Max value element : 700

3. min()

The method min() returns the elements from the list with minimum value.

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
print "min value element : ", min(list1);
```

```
print "min value element : ", min(list2);
```

Output:

```
min value element : 123
min value element : 200
```

4. append()

The method append() appends a passed obj into the existing list

```
aList = [123, 'xyz', 'zara', 'abc'];
aList.append( 2009 );
print "Updated List : ", aList;
```

Output:

```
Updated List : [123, 'xyz', 'zara', 'abc', 2009]
```

5. Count()

The method count() returns count of how many times obj occurs in list.

```
aList = [123, 'xyz', 'zara', 'abc', 123];
print "Count for 123 : ", aList.count(123);
print "Count for zara : ", aList.count('zara');
```

Output:

```
Count for 123 : 2
Count for zara : 1
```

6. Extend()

The method extend() appends the contents of seq to list.

```
aList = [123, 'xyz', 'zara', 'abc', 123];
bList = [2009, 'manni'];
aList.extend(bList)
print "Extended List : ", aList ;
```

Output:

```
Extended List : [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
```

7. index()

The method index() returns the lowest index in list that obj appears.

```
aList = [123, 'xyz', 'zara', 'abc'];
print "Index for xyz : ", aList.index( 'xyz' ) ;
print "Index for zara : ", aList.index( 'zara' ) ;
```

Output:

```
Index for xyz : 1
Index for zara : 2
```

8. insert()

The method insert() inserts object obj into list at offset index.

```
aList = [123, 'xyz', 'zara', 'abc']
aList.insert( 3, 2009)
print "Final List : ", aList
```

Output:
Final List : [123, 'xyz', 'zara', 2009, 'abc']

9. pop()
The method pop() removes and returns last object or obj from the list.
```
aList = [123, 'xyz', 'zara', 'abc'];
print "A List : ", aList.pop();
print "B List : ", aList.pop(2);
```

Output:
A List : abc
B List : zara

10. remove()
This method does not return any value but removes the given object from the list.
```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.remove('xyz');
print "List : ", aList;
aList.remove('abc');
print "List : ", aList;
```

11. reverse()
The method reverse() reverses objects of list in place.
```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.reverse();
print "List : ", aList;
```

Output:
List : ['xyz', 'abc', 'zara', 'xyz', 123]

3.2 Tuples
The next collection type is the Python *tuple*. Tuples construct simple groups of objects. They work exactly like lists, except that tuples can't be changed in place (they're immutable) and are usually written as a series of items in parentheses, not square brackets. Tuples share most of their properties with lists. They are:

Ordered collections of arbitrary objects
Like strings and lists, tuples are an ordered collection of objects; like lists, they can embed any kind of object.

Accessed by offset
Like strings and lists, items in a tuple are accessed by offset (not key); they support all the offset-base access operations we've already seen, such as indexing and slicing.

Of the category immutable sequence
Like strings, tuples are immutable; they don't support any of the in-place change operations we saw applied to lists. Like strings and lists, tuples are sequences; they support many of the same operations.

Fixed-length, heterogeneous, and arbitrarily nestable

Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples can hold any type of object, including other compound objects (e.g., lists, dictionaries, other tuples), and so support arbitrary nesting.

Arrays of object references

Like lists, tuples are best thought of as object reference arrays; tuples store access points to other objects (references), and indexing a tuple is relatively quick. Table highlights common tuple operations. A tuple is written as a series of objects (technically, expressions that generate objects), separated by commas and normally enclosed in parentheses. An empty tuple is just a parentheses pair with nothing inside.

Operation	**Interpretation**
()	An empty tuple
T = (0,)	A one-item tuple (not an expression)
T = (0, 'Ni', 1.2, 3)	A four-item tuple
T = 0, 'Ni', 1.2, 3	Another four-item tuple (same as prior line)

Accessing Values in Tuples

```
tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]

print "tup2[1:5]: ", tup2[1:5]
```

Output:

```
tup1[0]: physics

tup2[1:5]: [2, 3, 4, 5]
```

* **Updating Tuples is immpossible**

* **DeletingTuple Elements is not possible**

Basic Tuples Operations

- Tuples respond to the + and * operators much like strings
- The result is a new tuple

Python Expression	**Results**
len((1, 2, 3))	3
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')
3 in (1, 2, 3)	True
for x in (1, 2, 3): print x,	1 2 3

Indexing, Slicing

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results
L[2]	'SPAM!'
L[-2]	'Spam'
L[1:]	['Spam', 'SPAM!']

Built-in Tuple Functions

1. len()

The method **len() returns the number of elements in the tuple.**

```
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')

print "First tuple length : ", len(tuple1);

print "Second tuple length : ", len(tuple2);
```

Output:

```
First tuple length : 3

Second tuple length : 2
```

2. max()

The method **max() returns the elements from the tuple with maximum value.**

```
tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)

print "Max value element : ", max(tuple1);

print "Max value element : ", max(tuple2);
```

Output:

```
Max value element : zara

Max value element : 700
```

3. min()

The method **min() returns the elements from the tuple with minimum value.**

```
tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)

print "min value element : ", min(tuple1);

print "min value element : ", min(tuple2);
```

Output:

```
min value element : 123

min value element : 200
```

4. tuple()

This method converts and object into tuple.

```
aList = (123, 'xyz', 'zara', 'abc');
```

```
aTuple = tuple(aList)

print "Tuple elements : ", aTuple
```

Output:

```
Tuple elements : (123, 'xyz', 'zara', 'abc')
```

```
>>> a=[2,"abc", 23.7]

>>> at=tuple(a)

>>> at

(2, 'abc', 23.7)
```

3.3. Dictionary

Besides lists, dictionaries are perhaps the most flexible built-in data type in Python. If you think of lists as ordered collections of objects, dictionaries are unordered collections; their chief distinction is that items are stored and fetched in dictionaries by key, instead of offset. As we'll see, built-in dictionaries can replace many of the searching algorithms and data-structures you might have to implement manually in lower-level languages. Dictionaries also sometimes do the work of records and symbol tables used in other languages. In terms of their main properties, dictionaries are:

Accessed by key, not offset

Dictionaries are sometimes called *associative arrays* or *bashes.* They associate a set of values with keys, so that you can fetch an item out of a dictionary using the key that stores it. You use the same indexing operation to get components in a dictionary, but the index takes the form of a key, not a relative offset.

Unordered collections of arbitrary objects

Unlike lists, items stored in a dictionary aren't kept in any particular order; in fact, Python randomizes their order in order to provide quick lookup. Keys provide the symbolic (not physical) location of items in a dictionary.

Variable length, heterogeneous, arbitrarily nestable

Like lists, dictionaries can grow and shrink in place (without making a copy), they can contain objects of any type, and support nesting to any depth (they can contain lists, other dictionaries, and so on).

Of the category mutable mapping

They can be changed in place by assigning to indexes, but don't support the sequence operations we've seen work on strings and lists. In fact, they can't: because dictionaries are unordered collections, operations that depend on a fixed order (e.g., concatenation, slicing) don't make sense. Instead, dictionaries are the only built-in representative of the mapping type category—objects that map keys to values.

Tables of object references (hash tables)

If lists are arrays of object references, dictionaries are unordered tables of object references. Internally, dictionaries are implemented as *hash tables* (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing

algorithms to find keys, so retrieval is very fast. But at the bottom, dictionaries store object references (not copies), just like lists.
Table summarizes some of the most common dictionary. Dictionaries are written as a series of key:value pairs, separated by commas, and enclosed in curly braces. An empty dictionary is an empty set of braces, and dictionaries can be nested by writing one as a value in another dictionary, or an item in a list
(or tuple).
we often build up dictionaries by assigning to new keys at runtime, rather than writing constants. But see the following section on changing dictionaries; lists and dictionaries are grown in different ways. Assignment to new keys works for dictionaries, but fails for lists (lists are grown with append).
Table . Common Dictionary Constants and Operations

Operation	**Interpretation**
d1 = {}	Empty dictionary
d2 = {'spam' : 2, 'eggs' : 3}	Two-item dictionary
d3 = {'food' : {'ham' : 1, 'egg' : 2}}	Nesting
d2['eggs'], d3['food']['ham']	Indexing by key

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.
An empty dictionary without any items is written with just two curly braces, like this: {}.
Ex
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
Keys are unique within a dictionary while values may not be.
The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples .

Accessing Values in Dictionary

Use the square brackets along with the key to obtain its value.

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Name']: ", dict['Name'];

print "dict['Age']: ", dict['Age'];
```

Output:

```
dict['Name']: Zara

dict['Age']: 7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Alice']: ", dict['Alice'];
```

Output

```
dict['Zara']:
```

Traceback (most recent call last):

File "test.py", line 4, in <module>

print "dict['Alice']: ", dict['Alice'];

KeyError: 'Alice'

Updating Dictionary:

You can update a dictionary by

1. adding a new entry or a key-value pair,
2. modifying an existing entry, or
3. deleting an existing entry

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8;                    # update existing entry

dict['School'] = "DPS School";      # Add new entry

print "dict['Age']: ", dict['Age'];

print "dict['School']: ", dict['School'];
```

Output

```
dict['Age']: 8

dict['School']: DPS School
```

Delete Dictionary Elements:

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name'];                   # remove entry with key 'Name'

dict.clear();                       # remove all entries in dict

del dict ;                          # delete entire dictionary

print "dict['Age']: ", dict['Age'];

print "dict['School']: ", dict['School'];
```

Output:

```
dict['Age']:

Traceback (most recent call last):

File "test.py", line 8, in <module>

print "dict['Age']: ", dict['Age'];
```

TypeError: 'type' object is unsubscriptable

More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

print "dict['Name']: ", dict['Name'];
```

Output

```
dict['Name']: Manni
```

Example:

```
dict = {['Name']: 'Zara', 'Age': 7};

print "dict['Name']: ", dict['Name'];
```

When the above code is executed, it produces the following result:

```
Traceback (most recent call last):

File "test.py", line 3, in <module>

dict = {['Name']: 'Zara', 'Age': 7};

TypeError: list objects are unhashable
```

Built-in Dictionary Functions

1. len()

The method len() 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 : %d" % len (dict)
```

Output

```
Length : 2
```

2. str()

The method **str() produces a printable string representation of a dictionary.**

```
dict = {'Name': 'Zara', 'Age': 7};

print "Equivalent String : %s" % str (dict)
```

When we run above program, it produces following result:

```
Equivalent String : {'Age': 7, 'Name': 'Zara'}
```

3. type()
The method type() 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 : %s" % type (dict)
```

Ouput
```
Variable Type : <type 'dict'>
```

4.clear()
The method clear() removes all items from the dictionary.

```
dict = {'Name': 'Zara', 'Age': 7};

print "Start Len : %d" % len(dict)

dict.clear()

print "End Len : %d" % len(dict)
```

Output
```
Start Len : 2

End Len : 0
```

5.copy()
The method copy() returns a shallow copy of the dictionary

```
dict1 = {'Name': 'Zara', 'Age': 7};

dict2 = dict1.copy()

print "New Dictinary : %s" % str(dict2)
```

Output
```
New Dictinary : {'Age': 7, 'Name': 'Zara'}
```

6. fromkeys()
The method fromkeys() creates a new dictionary with keys from seq and values set to value.

```
seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print "New Dictionary : %s" % str(dict)
dict = dict.fromkeys(seq, 10)
print "New Dictionary : %s" % str(dict)
New Dictionary : {'age': None, 'name': None, 'sex': None}
New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```

7. get()
The method get() returns a value for the given key. If key is not available then returns default value None.

```
dict = {'Name': 'Zabra', 'Age': 7}

print "Value : %s" % dict.get('Age')

print "Value : %s" % dict.get('Education‘)
```

Output

```
Value : 7

Value : None
```

8. has_key()

The method has_key() returns true if a given key is available in the dictionary, otherwise it returns a false.

```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" % dict.has_key('Age')

print "Value : %s" % dict.has_key('Sex')
```

Output

```
Value : True

Value : False
```

9. items()

The method items() returns a list of dict's (key, value) tuple pairs

```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" % dict.items()
```

Output:

```
Value : [('Age', 7), ('Name', 'Zara')]
```

10. keys()

The method keys() returns a list of all the available keys in the dictionary.

```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" % dict.keys()
```

Output:

```
Value : ['Age', 'Name']
```

11. setdefault()

The method setdefault() is similar to get(), but will set dict[key]=default if key is not already in dict.

```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" % dict.setdefault('Age', None)
```

```
print "Value : %s" % dict.setdefault('Sex', None)
```

Output

```
Value : 7

Value : None
```

12.update()

The method update() adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

```
dict = {'Name': 'Zara', 'Age': 7}

dict2 = {'Sex': 'female' }

dict.update(dict2)

print "Value : %s" % dict
```

Output

```
Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}
```

13. values()

The method values() returns a list of all the values available in a given dictionary.

```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" % dict.values()
```

Output

```
Value : [7, 'Zara']
```

3.4. SETS

Besides decimals, Python 2.4 also introduced a new collection type, the set—an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory. By definition, an item appears only once in a set, no matter how many times it is added. Accordingly, sets have a variety of applications, especially in numeric and database-focused work.

Because sets are collections of other objects, they share some behavior with objects such as lists and dictionaries that are outside the scope of this chapter. For example, sets are iterable, can grow and shrink on demand, and may contain a variety of object types. As we'll see, a set acts much like the keys of a valueless dictionary, but it supports extra operations.

However, because sets are unordered and do not map keys to values, they are neither sequence nor mapping types; they are a type category unto themselves. Moreover, because sets are fundamentally mathematical in nature we'll explore the basic utility of Python's set objects here.

To make a set object, pass in a sequence or other iterable object to

the built-in set function:

```
>>> x = set('abcde')
```

```
>>> y = set('bdxyz')
```

You get back a set object, which contains all the items in the object passed in

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])                    # Pythons <= 2.6 display format
```

Sets made this way support the common mathematical set operations with expression operators. Note that we can't perform the following operations on plain sequences like strings, lists, and tuples—we must create sets from them by passing them to set in order to apply these tools.

Immutable constraints and frozen sets

lists and dictionaries cannot be embedded in sets, but tuples can if you need to store compound values.
Tuples compare by their full values when used in set operations:
Tuples in a set, for instance, might be used to represent dates, records, IP addresses, and so on. Sets may also contain modules, type objects, and more. Sets themselves are mutable too, and so cannot be nested in other sets directly; if you need to store a set inside another set, the frozenset built-in call works just like set but creates an immutable set that cannot change and thus can be embedded in other sets.

Table : Set Operation *Symbol*

Python Symbol	***Description***
in	Is a member of
not in	Is not a member of
= ==	Is equal to
!=	Is not equal to
<	Is a (strict) subset of
<=	Is a subset of (includes improper subsets)
>	Is a (strict) superset of
>=	Is a superset of (includes improper supersets)
&	Intersection
|	Union
- or \	Difference or relative complement
^	Symmetric difference

Why sets?

Set operations have a variety of common uses, some more practical than mathematical. For example, because items are stored only once in a set, sets can be used to *filter duplicates* out of other collections, though items may be reordered in the process because sets are unordered in general. Simply convert the collection to a set, and then convert it back again:

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
```

```
>>> L = list(set(L))          # Remove duplicates
>>> L
[1, 2, 3, 4, 5]
```

Membership

```
>>> A={1,2,3}
>>> B={3,4,5,6}
>>> 1 in A
True
```

1. add method

Adds new element to the set

```
>>> A={1,2,3}
>>> A.add(4)
>>> A
{1, 2, 3, 4}
```

2. remove()

Removes a member from the set

```
>>> A={2,3,4,5}
>>> A
{2, 3, 4, 5}

>>> A.remove(2)
>>> A
{3, 4, 5}
```

3. union

Union Set of elements in either set A or set B

Operator is |

```
>>> A={1,3,4,5}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 5, 6, 8}
```

4. intersection()

Set of elements in both A and B

```
>>> A={1,3,4,5}
4>>> B={2,4,6,8}
>>> A&B
{4}
```

5. difference

Set of elements in set A, but not in set B

```
>>> A={1,3,4,5}
>>> B={2,4,6,8}

>>> A-B
{1, 3, 5}
```

6. Symmetric difference

Set of elements in set A or set B, but not both

```
>>> A={1,3,4,5}
>>> B={2,4,6,8}

>>> A^B
{1, 2, 3, 5, 6, 8}
```

7. size

Number of elements in the set

```
>>> A={1,3,4,6,7}
>>> len(A)
5
>>>fruit={'apple','banana','mango'}
>>> fruit
{'mango', 'banana', 'apple'}

>>> 'mango' in fruit
True

>>> fruit.add('pineappple')
>>> fruit
{'mango', 'banana', 'apple', 'pineappple'}
```

8. Empty set

```
>>> subject={}
>>> subject
{}
>>> s1=set()
>>> s1
set()

>>> vowelslist=['a','e','i','o','u']
>>> vset=set(vowelslist)
>>> vset
{'a', 'o', 'e', 'i', 'u'}
>>> vowels='aeiou'
>>> vset=set(vowels)
>>> vset
{'a', 'o', 'e', 'i', 'u'}
```

4. Comprehensions

4.1 List Comprehensions:

In addition to sequence operations and list methods, Python includes a more advanced operation known as a list comprehension expression.

[*expr* ***for*** *iter_var* ***in*** *iterable* ***if*** *cond_expr*]

```
>>> M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]                    # Get row 2
[4, 5, 6]
>>> M[1][2]    # Get row 2, then get item 3 within the row
6
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> col2 = [row[1] for row in M]
>>> col2
[2, 5, 8]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- They are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right.
- List comprehensions are coded in square brackets (to tip you off to the fact that they make a list) and are composed of an expression and a looping construct that share a variable name (row, here).
- The preceding list comprehension means basically what it says: "Give me row[1] for each row in matrix M, in a new list." The result is a new list containing column 2 of the matrix.
- List comprehensions can be more complex in practice:

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [row[1] + 1 for row in M]
[3, 6, 9]

>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [row[1] for row in M if row[1] % 2 == 0]
[2, 8]
>>> diag = [M[i][i] for i in [0, 1, 2]]
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'spam']
>>> doubles
['ss', 'pp', 'aa', 'mm']

>>> list(range(5))
[0, 1, 2, 3,4]
>>> list(range(-4,7,2))
[-4, -2, 0, 2, 4, 6]

>>> [[x ** 2, x ** 3] for x in range(4)]
[[0, 0], [1, 1], [4, 8], [9, 27]]
```

4.2 Set comprehensions

- The set comprehension expression is similar in form to the list comprehension , but is coded in curly braces instead of square brackets and run to make a set instead of a list.
- Set comprehensions run a loop and collect the result of an expression on each iteration;
- A loop variable gives access to the current iteration value for use in the collection expression.
- The result is a new set you create by running the code, with all the normal set behavior.

```
>>> {x ** 2 for x in [1, 2, 3, 4]}
{16, 1, 4, 9}
>>> {x for x in 'spam'}
{'m', 'a', 's', 'p'}
>>> {c * 4 for c in 'spam'}
{'aaaa', 'ssss', 'mmmm', 'pppp'}

>>> S = {c * 4 for c in 'spam'}
>>> S | {'mmmm', 'xxxx'}
{'aaaa', 'pppp', 'mmmm', 'ssss', 'xxxx'}

>>> S & {'mmmm', 'xxxx'}
{'mmmm'}
```

4.3 Dictionary Comprehensions:

Dictionaries in 3.X and 2.7 can also be created with dictionary comprehensions. Like the set comprehensions, dictionary comprehensions are available only in 3.X and 2.7. They run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary. A loop variable allows the comprehension to use loop iteration values along the way. To illustrate, a standard way to initialize a dictionary dynamically in both 2.X and 3.X is to combine its keys and values with zip, and pass the result to the dict call. The zip built-in function is the hook that allows us to construct a dictionary from key and value lists this way—if you cannot predict the set of keys and values in your code, you can always build them up as lists and zip them together.

```
>>> D = {x: x*2 for x in range(5)}
>>> D
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8}
>>> D = {c: c * 4 for c in 'SPAM'}
>>> D
{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}

>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'ham': 'HAM!', 'spam': 'SPAM!', 'eggs': 'EGGS!'}
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)
>>> D
{'b': 0, 'c': 0, 'a': 0}
>>> D = {k:0 for k in ['a', 'b', 'c']}
```

```
>>> D
{'b': 0, 'c': 0, 'a': 0}
>>> D = dict.fromkeys('spam')
>>> D
{'m': None, 's': None, 'a': None, 'p': None}
>>> D = {k: None for k in 'spam'}
>>> D
{'m': None, 's': None, 'a': None, 'p': None}
```