

The logo consists of two light blue squares stacked vertically, with a light green horizontal bar passing through the center of both squares.

GLS UNIVERSITY

**0301601 INTRODUCTION TO PYTHON
UNIT- IV**

Data Structure & Classes and Objects

INDEX

- Data Structure
 - List
 - Tuple
 - Dictionary
- Classes & Objects
 - Overview of OOP Technology
 - Creating Classes
 - Creating Objects
 - Accessing Attributes
 - Destroying objects

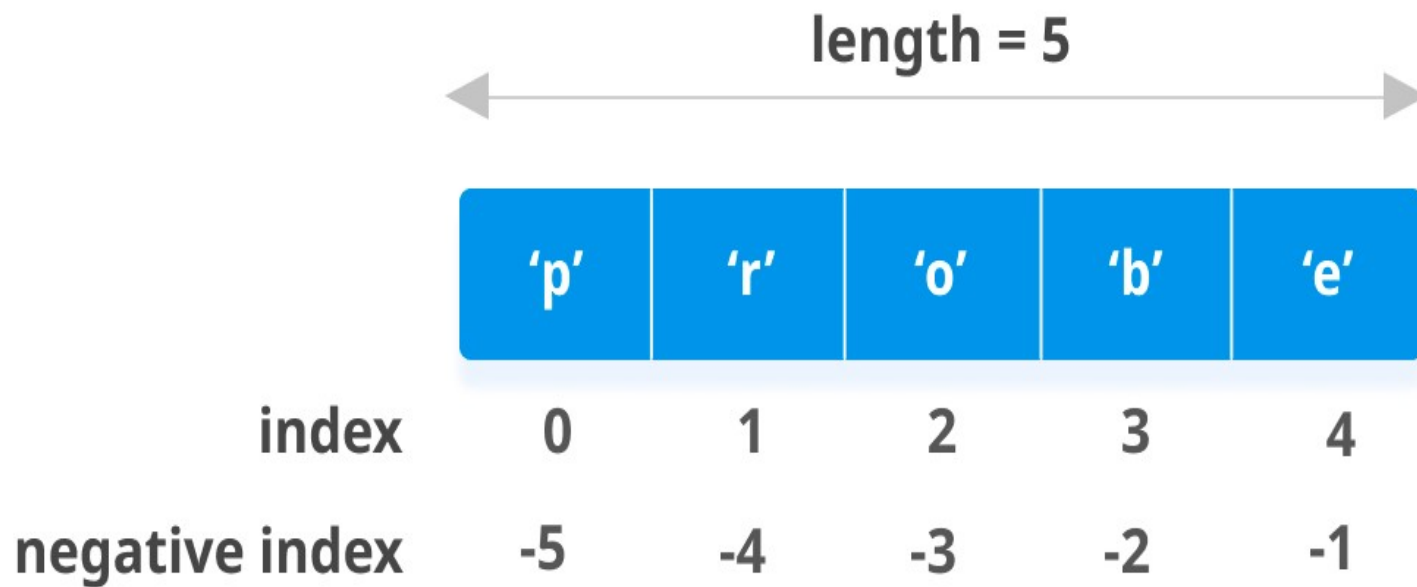
Introduction

- Data structures are basically structures which can hold some data together.
- They are used to store a collection of related data.
- There are built-in data structures in Python:
 - List
 - Tuple
 - Dictionary

List

- A list is a data structure that holds an ordered collection of items.
- A list can be composed by storing a sequence of different type of values separated by commas.
- The elements are stored in the index basis with starting index as 0.
- The list of items should be enclosed in square brackets.
- Once a list is created, user can add, remove or search for items in the list.
- Python lists are mutable i.e., Python will not create a new list if we modify an element in the list.
- **Accessing List:** A list can be created by putting the value inside the square bracket and separated by comma.
- **Syntax:** <list_name>=[value1,value2,value3,...,valuen];

List Index



List

- **Updating Lists:** You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator.

- **Example:**

```
list = ['physics', 'chemistry', 1997, 2000]  
print ("Value available at index 2 : ", list[2])
```

```
list[2] = 2001  
print ("New value available at index 2 : ", list[2])
```


List

- **Delete List Elements**

- To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting.

- **Example:**

```
list = ['physics', 'chemistry', 1997, 2000]  
print (list)
```

```
del list[2]  
print ("After deleting value at index 2 : ", list)
```

Basic List Operations

- Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Basic List Operations

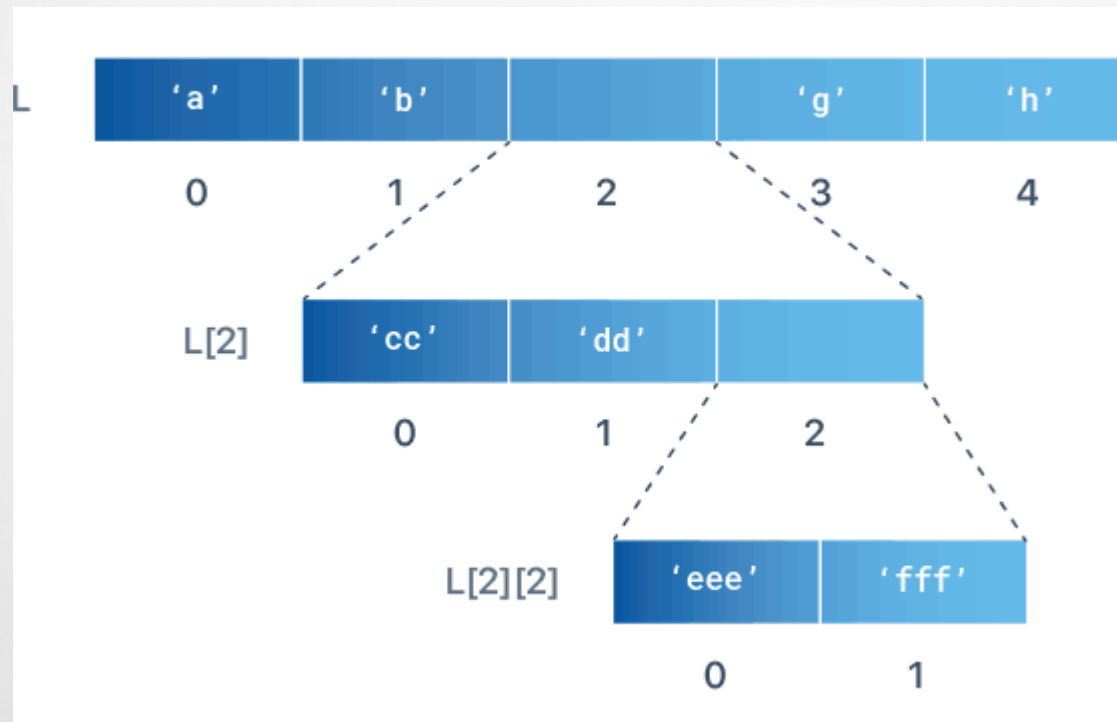
- **Indexing, Slicing and Matrixes**

- Since lists are sequences, indexing and slicing work the same way for lists as they do for strings.
- Assuming the following input –
 - `L = ['C++', 'Java', 'Python']`

Nested List

A nested list is a list that appears as an element in another list

```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
```



List Operations

- `cmp()`
- `len()`
- `max()`
- `min()`
- `list()`
- `append()`
- `extend()`
- `insert()`
- `sum()`
- `remove()`
- `index()`
- `count()`
- `pop()`
- `reverse()`
- `sort()`
- `sorted()`
- `copy()`
- `clear()`

List Operations – cmp()

The cmp() method compares the elements of two lists.

Syntax

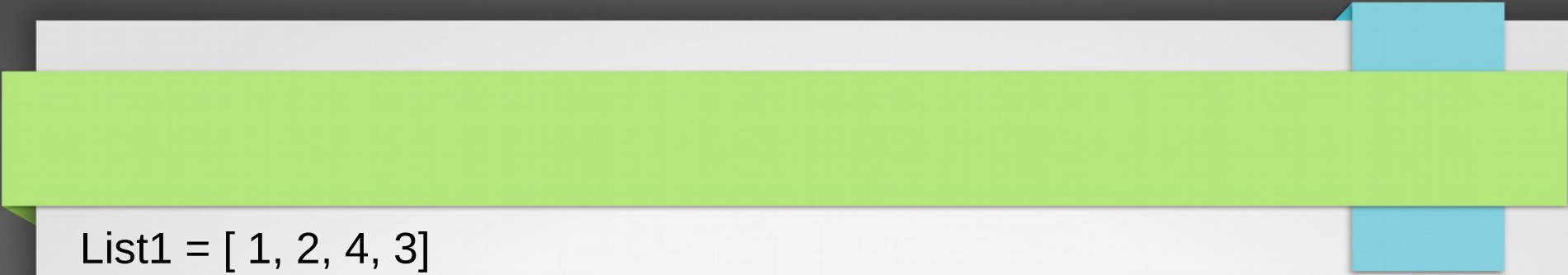
```
cmp(list1, list2)
```

Return value

This function returns 1, if first list is “greater” than second list,
-1 if first list is smaller than the second list
else it returns 0 if both the lists are equal.

Example:

```
list1, list2 = [123, 'xyz'], [456, 'abc']  
print cmp(list1, list2) #-1  
print cmp(list2, list1) #1  
list3 = list2 + [786];  
print cmp(list2, list3) #-1
```



```
List1 = [ 1, 2, 4, 3]
List2 = [ 1, 2, 5, 8]
List3 = [ 1, 2, 5, 8, 10]
list4 = [ 1, 2, 4, 3]
# Comparing lists print "Comparison of list2 with list1 :",
print cmp(list2, list1)
# prints -1, because list3 has larger size than list2
print "Comparison of list2 with list3(larger size) :",
print cmp(list2, list3)
# prints 0 as list1 and list4 are equal
print "Comparison of list4 with list1(equal) :",
print cmp(list4, list1)
```

List Operations – len()

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

Syntax

len(list)

Example:

```
list1 = ['physics', 'chemistry', 'maths']  
print (len(list1)) # 3
```

```
list2 = list(range(5)) #creates list of numbers between 0-4  
print (len(list2)) # 5
```


List Operations – max()

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

Syntax

max(list)

Example:

```
list1, list2 = ['C++', 'Java', 'Python'], [456, 700, 200]
print ("Max value element : ", max(list1)) # Python
print ("Max value element : ", max(list2)) #700
```

List Operations – min()

```
list2 = [456, 700, 200]
```

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

```
min value element : 200
```

List Operations – list()

The list() method takes sequence types and converts them to lists. This is used to convert a given tuple into list.

Syntax

```
list( seq )
```

Example:

```
aTuple = (123, 'C++', 'Java', 'Python')  
list1 = list(aTuple)  
print ("List elements : ", list1)
```

```
str = "Hello World"  
list2 = list(str)  
print ("List elements : ", list2)
```

Output:

```
List elements : [123, 'C++', 'Java', 'Python']  
List elements : ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

List Operations - append()

- The append() method adds an item to the end of the list.
- **Syntax:**
- list.append(item)

Example:

```
List = ['Mathematics', 'chemistry', 1997, 2000]  
List.append(20544)  
print(List)
```

Output:

```
['Mathematics', 'chemistry', 1997, 2000, 20544]
```

List Operations: extend()

- extends the list by adding all items of a list (passed as an argument) to the end.
- **Syntax:**
- `list1.extend(list2)`

- **Example:**

```
List1 = [1, 2, 3]
List2 = [2, 3, 4, 5]
List1.extend(List2) # Add List2 to List1
print(List1)
List2.extend(List1) #Add List1 to List2 now
print(List2)
```

Output:

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

List Operations: insert()

- inserts the element to the list at the given index.
- **Syntax:**
- `list.insert(index, element)`

Example:

```
List = ['Mathematics', 'chemistry', 1997, 2000]  
# Insert at index 2 value 10087  
List.insert(2,10087)  
print(List)
```

Output:

```
['Mathematics', 'chemistry', 10087, 1997, 2000]
```


List Operations: sum()

Calculates sum of all the elements of List.

- **Syntax:**
- `sum(list)`

Example:

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

Output:

15

List Operations: remove()

searches for the given element in the list and removes the first matching element.

Syntax:

```
list.remove(element)
```

Example:

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']  
list1.remove('Biology')  
print ("list now : ", list1)  
list1.remove('maths')  
print ("list now : ", list1)
```

Output:

```
list now : ['physics', 'chemistry', 'maths']  
list now : ['physics', 'chemistry']
```

List Operations: index()

searches an element in the list and returns its index/position.

Syntax:

```
list.index(element)
```

Example:

```
list1 = ['physics', 'chemistry', 'maths']  
print ('Index of chemistry', list1.index('chemistry'))  
print ('Index of C#', list1.index('C#'))
```

Output:

```
Index of chemistry 1
```

```
Traceback (most recent call last):
```

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

```
    print ('Index of C#', list1.index('C#'))
```

```
ValueError: 'C#' is not in list
```

List Operations: count()

returns the number of occurrences of an element in a list.

Syntax:

```
list.count(element)
```

Example:

```
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
```

List Operations: pop()

removes and returns the element at the given index from the list.

Syntax:

```
list.pop(index, element)
```

The parameter passed to the pop() method is optional. If no parameter is passed, the default index -1 is passed as an argument which returns the last element.

Example:

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']  
list1.pop()  
print ("list now : ", list1)  
list1.pop(1)  
print ("list now : ", list1)
```

Output:

```
list now : ['physics', 'Biology', 'chemistry']  
list now : ['physics', 'chemistry']
```

List Operations: reverse()

reverses the elements of a given list. Also it doesn't return any value.

Syntax:

```
list.reverse()
```

Example:

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']  
list1.reverse()  
print ("list now : ", list1)
```

Output:

```
list now : ['maths', 'chemistry', 'Biology', 'physics']
```


List Operations: sort()

sorts the elements of a given list in a specific order - Ascending or Descending.

Syntax:

```
list.sort(key=..., reverse=...)
```

Both parameters are optional:

key: (Optional) A function that extracts a comparison key from each list element while sorting.

reverse: (Optional) If true, the sorted list will be reversed. By default, it is False.

Syntax:

```
sorted(key=..., reverse=...)
```

difference between `list.sort()` sorts the list and save the sorted list, while `sorted(list)` returns a sorted copy of the list, without changing the original list.

List Operations: `sort()` or `sorted()`

There are two ways to sort a list.

We can either use the `sort()` method or the `sorted()` function.

The `sort()` method is a list method and thus can only be used on lists.

The `sorted()` function works on any iterable.

Difference between `sort ()` and `sorted ()` in Python:

The primary difference between the list `sort()` function and the `sorted()` function is that the `sort()` function will modify the list it is called on. The `sorted()` function will create a new list containing a sorted version of the list it is given. ... Once the `sort()` function is called on it, the list is updated.

List Operations: sorted()

The sorted() method sorts the elements of a given iterable in a specific order
- Ascending or Descending.

Syntax:

```
sorted(iterable[, key][, reverse])
```

sorted() Parameters

iterable - sequence (string, tuple, list) or collection (dictionary)

reverse (Optional) - If true, the sorted list is reversed (or sorted in Descending order)

key (Optional) - function that serves as a key for the sort comparison

List Operations: sorted()

Example:

vowels list

```
pyList = ['e', 'a', 'u', 'o', 'i']  
print(sorted(pyList))
```

string

```
pyString = 'Python'  
print(sorted(pyString))
```

vowels tuple

```
pyTuple = ('e', 'a', 'u', 'o', 'i')  
print(sorted(pyTuple))
```

Output:

```
['a', 'e', 'i', 'o', 'u']  
['P', 'h', 'n', 'o', 't', 'y']  
['a', 'e', 'i', 'o', 'u']
```

List Operations: copy()

returns a copy of the list.

- Soft copy
- Deep copy

Soft copy: A list can be copied with = operator but when user modify the new_list, the old_list is also modified.

Deep copy: So copy() method keeps original list unchanged when the new list is modified.

Syntax: new_list = copy.deepcopy(old_list)

A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

List Operations: copy()

```
import copy
```

```
li1 = [1, 2, 3]
```

```
li2 = copy.deepcopy(li1)
```

```
print ("The original elements before deep copying")
```

```
for i in range(0,len(li1)):
```

```
    print (li1[i],end=" ")
```

```
li2[2] = 7
```

```
print ("The new list of elements after deep copying ")
```

```
for i in range(0,len( li1)):
```

```
    print (li2[i],end=" ")
```

```
print ("The original elements after deep copying")
```

```
for i in range(0,len( li1)):
```

```
    print (li1[i],end=" ")
```


List Operations: clear()

- removes all items from the list.

- **Syntax:**

- `list.clear()`

- **Example:**

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

Tuple

- Tuple is immutable. User cannot change elements of a tuple once it is assigned.
- The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
- Tuples are defined by specifying items separated by commas within a pair of parentheses.
- Methods that add items or remove items are not available with tuple.
-

Tuple

- However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:
- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Tuple Operations

- `index()`
- `count()`
- `len()`
- `max()`
- `min()`
- `tuple()`
- `sum()`
- `sorted()`
- `reverse()`
- `any()`
- `all()`

Tuple Operations:

- **Adding Tuples:** Tuple can be added by using the concatenation operator(+) to join two tuple.
- **Replicating tuple:** Replicating means repeating. It can be performed by using '*' operator by a specific number of time.
- **Tuple slicing:** A subpart of a tuple can be retrieved on the basis of index.
- **Count():** returns the number of occurrences of an element in a tuple. It searches the given element in a tuple and returns how many times the element has occurred in it.
 - **Syntax:** tuple.count(element)
- **Index():** searches an element in a tuple and returns its index. If the same element is present more than once, the first/smallest position is returned.
 - **Syntax:** tuple.index(element)

Tuple Operations:

- **max()**: Returns the maximum value from the tuple.
 - **Syntax:** max(tuple)
- **min()**: Returns the minimum value from the tuple.
 - **Syntax:** min(tuple)
- **Len()**: returns the length of a tuple
 - **Syntax:** len(tuple)
- **Sum()**: adds the items of an iterable and returns the sum.
 - **Syntax:** sum(iterable, start)
 - The sum() function adds start and items of the given iterable from left to right. Start is optional. The default value of start is 0.

Operations:

- **all():** returns True when all elements in the given iterable are true.
- If not, it returns False.
 - **Syntax:** all(iterable)
 - iterable - any iterable (list, tuple, dictionary, etc.) which contains the elements
- The all() method returns:
 - True - If all elements in an iterable are true
 - False - If any element in an iterable is false
- In case of dictionaries, if all keys (not values) are true or the dictionary is empty, all() returns True. Else, it returns false for all other cases.

Operations:

Some Python Methods

all() Return Value

all() function returns:

True - If all elements in an iterable are true

False - If any element in an iterable is false

When	Return Value
All values are true	True
All values are false	False
One value is true (others are false)	False
One value is false (others are true)	False
Empty Iterable	True

Operations:

Some Python Methods

- **any():** returns True if any element of an iterable is true. If not, this method returns False.
 - **Syntax:** any(iterable)
 - iterable - any iterable (list, tuple, dictionary, etc.) which contains the elements
- The any() method returns:
 - True if at least one element of an iterable is true
 - False if all elements are false or if an iterable is empty
- In case of dictionaries, if all keys (not values) are false, any() returns False. If at least one key is true, any() returns True.

Any

Some Python Methods

Condition	Return Value
All values are true	True
All values are false	False
One value is true (others are false)	True
One value is false (others are true)	True
Empty Iterable	False

Operations:

Some Python Methods

- **enumerate():** method adds counter to an iterable and returns it (the enumerate object).
 - **Syntax:** enumerate(iterable, start=0)
 - iterable - a sequence, an iterator, or objects that supports iteration
 - start (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.
- User can convert enumerate objects to list and tuple using list() and tuple() method respectively.

Dictionary

- Dictionary is an unordered set of key and value pair.
- It is an container that contains data, enclosed within curly braces.
- The key and the value is separated by a colon(:). This pair is known as item.
- Items are separated from each other by a comma(,).
- Dictionary is mutable i.e., value can be updated.
- Key must be unique and immutable. Value is accessed by key. Value can be updated while key cannot be changed.
- Dictionary is known as **Associative array**.
- **Example:**
 - `stud={1:'Reena', 2:'Seema', 3:'Rahul'}`
 - `print stud`
- **Accessing Values:** As index is not defined, a Dictionaries value can be accessed by their keys.
- **Updation:** New item can be added. The values can be changed.

Dictionary Operations

- `get()`
- `del()`
- `cmp()`
- `keys()`
- `values()`
- `items()`
- `clear()`
- `copy()`
- `update()`
- `has_key()`
- `from_keys()`

Dictionary: Operations: get(key)

get(key): Returns the value of the given key. If key is not present it returns none.

Syntax: dict.get(key, default = None)

The get() method takes maximum of two parameters:

key - key to be searched in the dictionary

value (optional) - Value to be returned if the key is not found. The default value is None.

Return Value from get()

The get() method returns:

- the value for the specified key if key is in dictionary.
- None if the key is not found and value is not specified.
- value if the key is not found and value is specified.

Dictionary: Operations: get(key)

Example:

```
person = {'name': 'Ramesh', 'age': 22}
```

```
print('Name: ', person.get('name'))
```

```
print('Age: ', person.get('age'))
```

```
# value is not provided
```

```
print('Salary: ', person.get('salary'))
```

```
# value is provided
```

```
print('Salary: ', person.get('salary', 0.0))
```

Name: Ramesh

Age: 22

Salary: None

Salary: 0.0

Dictionary: Operations: del()

Deletion: An item can be deleted using **del** statement from a dictionary using the key.

Whole dictionary can also be deleted using del statement.

- **Syntax:** del [key]
- 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
```

Dictionary: Operations: cmp()

cmp(): compares two dictionaries based on key and values.

- **Syntax:** cmp(dict1, dict2)
- returns
 - 0 if both dictionaries are equal,
 - -1 if dict1 < dict2 and
 - 1 if dict1 > dict2.

```
dict1 = {'Name': 'Zara', 'Age': 7};  
dict2 = {'Name': 'Mahnaz', 'Age': 27};  
dict3 = {'Name': 'Abid', 'Age': 27};  
dict4 = {'Name': 'Zara', 'Age': 7};  
print "Return Value : %d" % cmp (dict1, dict2) # -1  
print "Return Value : %d" % cmp (dict2, dict3) #1  
print "Return Value : %d" % cmp (dict1, dict4) #0
```


Dictionary: Operations: keys()

keys(): Return all the keys element of a dictionary.

- **Syntax:** dict.keys()

- Example:

```
dict = {'Name': 'Zara', 'Age': 7}  
print "Value : %s" % dict.keys()
```

Output:

Value : ['Age', 'Name']

Dictionary: Operations: values()

values(): Return all the values element of a dictionary.

- **Syntax:** dict.values()

- Example:

```
dict = {'Name': 'Zara', 'Age': 7}  
print "Value : %s" % dict.values()
```

Output:

Value : ['Zara', 7]

Dictionary: Operations: items()

items(): Return all the items(key-value pair) of a dictionary.

- **Syntax:** dict.items()

Example:

```
dict = {'Name': 'Zara', 'Age': 7}  
print "Value : %s" % dict.items()
```

Output:

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

Dictionary: Operations: clear()

- **clear():** It is used to remove all items of a dictionary. It returns an empty dictionary.
- **Syntax:** dict.clear()

- Example:

```
dict = {'Name': 'Zara', 'Age': 7};  
print "Len : %d" % len(dict)  
dict.clear()  
print "Len : %d" % len(dict)
```

Output:

```
Len : 2  
Len : 0
```

Dictionary: Operations: copy()

- **copy()**: It returns an ordered copy of the data.

- **Syntax:** dict.copy()

- Example:

```
dict1 = {'Name': 'Zara', 'Age': 7};  
dict2 = dict1.copy()  
print "New Dictionary : %s" % str(dict2)
```

Output:

New Dictionary : {'Age': 7, 'Name': 'Zara'}

Dictionary: Operations: update()

- **update(dictionary2):** It is used to add items of dictionary2 to first dictionary.
 - **Syntax:** dict.update(dict2)

```
dict = {'Name': 'Zara', 'Age': 7}  
dict2 = {'Sex': 'female' }
```

```
dict.update(dict2)  
print "Value : %s" % dict
```

Output:

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

Dictionary: Operations: has_key()

- **has_key(key):** It returns a boolean value. True in case if key is present in the dictionary ,else false.
- **Syntax:** dict.has_key(key)

- Example:

```
dict = {'Name': 'Zara', 'Age': 7}  
print "Value : %s" % dict.has_key('Age')  
print "Value : %s" % dict.has_key('Sex')
```

- Output:

```
Value : True  
Value : False
```


Dictionary: Operations: fromkeys()

- **fromkeys(sequence,value1):** creates a new dictionary with keys from seq and values set to value.
- **Syntax:**
 - dict.fromkeys(seq[, value])
 - seq – list of values which would be used for dictionary keys preparation.
 - value – Optional, if provided then value would be set to this value.

Dictionary: Operations: fromkeys()

- Example:

```
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)
```

Output:

```
New Dictionary : {'age': None, 'name': None, 'sex': None}  
New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```