DICTIONARY-A

A dictionary is an unordered collection of zero or more key–value pairs whose keys are object references that refer to hashable objects, and whose values are object references referring to objects of any type. **Dictionaries are mutable**, so we can easily add or remove items, but since they are unordered they have no notion of index position and so cannot be sliced or strided.

A ***mapping*** type is one that supports the membership operator (in) and the size function (len()), and is iterable. Mappings are collections of key–value items and provide methods for accessing items and their keys and values. When iterated, unordered mapping types provide their items in an arbitrary order.

Only **hashable objects** may be used as dictionary **keys**, so immutable data types such as float, frozenset, int, str, and tuple can be used as dictionary keys, but mutable types such as dict, list, and set cannot. On the other hand, each key's associated **value** can be an object reference referring to an object of any type, including numbers, strings, lists, sets, dictionaries, functions, and so on.

Dictionary types can be compared using the standard equality comparison operators (== and !=), with the comparisons being applied item by item (and recursively for nested items such as tuples or dictionaries inside dictionaries).Comparisons using the other comparison operators (<, <=, >=, >) are not supported since they don't make sense for unordered collections such as dictionaries.

Let's say you have a list of people. What if you wanted to create a little database where you could store the telephone numbers of these people**:**

```
>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
>>> numbers = ['2341', '9102', '3158', '0142', '5551']
```

Once you've created these lists, you can look up Cecil's telephone number as follows:

```
>>> numbers[names.index('Cecil')]
3158
```

It works, but it's a bit impractical. What you really would want to do is something like the following:

```
>>> phonebook['Cecil']
3158
```

Guess what? If phonebook is a dictionary, you can do just that.

```
>>>phonebook = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

DICTIONARY-A

DICTIONARY SYNTAX

```
>>> phonebook = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
>>>days = {'January':31, 'February':28, 'March':31, 'April':30, 'May':31, 'June':30, 'July':31,
'August':31, 'September':30, 'October':31, 'November':30, 'December':31}
```

Properties of Dictionary Keys

There are two important points to remember about dictionary keys-

- **More than one entry per key is not allowed**. This means no duplicate key is allowed.

When duplicate keys are encountered during assignment, the last assignment wins. For example-

```
>>>d= {'Name': 'Zara', 'Age': 7, 'Name': 'Mani'}
>>>print ("dict['Name']: ", d['Name'])
dict['Name']: Mani
```

- **Keys must be immutable**. This means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example-

```
>>>d = {['Name']: 'Zara', 'Age': 7}
>>>print ("dict['Name']: ", d['Name'])
Traceback (most recent call last):
File "test.py", line 3, in <module>
d = {['Name']: 'Zara', 'Age': 7}
TypeError: list objects are unhashable
```

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects.

DICTIONARY VALUES ARE NOT STORED IN SORTED ORDER.

- Unlike in a list, items stored in a dictionary aren't kept in any particular order.
- Unlike Python lists or tuples, the key and value pairs in dictionary objects are not in any particular order.
- Although the key-value pairs are in a certain order in the instantiation statement, by calling the list method on it (which will create a list from its keys) we can easily see they aren't stored in that order:

CREATING A DICTIONARY

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict([('two', 2), ('one', 1), ('three', 3)])
>>> d = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d
True
```

Here are some examples to illustrate the various syntaxes—they all produce the same dictionary:

```
>>>d1 = dict({"id": 1948, "name": "Washer", "size": 3})
>>>d2 = dict(id=1948, name="Washer", size=3)
>>>d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
>>>d4 = {"id": 1948, "name": "Washer", "size": 3}
```

Dictionary d1 is created using a dictionary literal.
Dictionary d2 is created using keyword arguments.
Dictionaries d3 are created from sequences,
Dictionary d4 is created by assignment

ASSIGNMENT:

Creating dictionaries simply involves *assigning a dictionary to a variable*, regardless of whether the dictionary has elements or not:

Empty braces: Dictionaries can also be created using braces—empty braces, {}, create an empty dictionary; nonempty braces must contain one or more commas separated items, each of which consists of a key, a literal colon, and a value.

```
>>> dict1 = {}
>>> dict2 = {'name': 'earth', 'port': 80}
```

Keyword arguments: a sequence of two objects, the first of which is used as a key and the second of which is used as a value. Alternatively, for dictionaries where the keys are valid Python identifiers, keyword arguments can be used, with the key as the keyword and the value as the key’s value.

```
>>> dict3 = {'name'= 'mars', 'port'= 82}
>>> dict1, dict2, dict3
({}, {'port': 80, 'name': 'earth'}, {'port': 82, 'name': 'mars'})
```

TYPE CONVERSION:

Dictionaries may also be created using the factory function dict(). You can use the dict function to construct dictionaries from other mappings (for example, other dictionaries) or from sequences of (key, value) pairs:

```
>>> items = [('name', 'Gumby'), ('age', 42)]
>>> d = dict(items)
>>> d
{'age': 42, 'name': 'Gumby'}
>>> d['name']
'Gumby'
>>> items = ['name', 'Gumby', 'age', 42]
>>> e = dict(items)
Traceback (most recent call last):
File "<pyshell#4>", line 1, in <module>
e = dict(items)
ValueError: dictionary update sequence element #0 has length 4; 2 is required
>>>
>>> items = [['name', 'Gumby'], ['age', 42]]
>>> e = dict(items)
>>> e
{'name': 'Gumby', 'age': 42}
```

It can also be used with *keyword arguments*, as follows:

```
>>> d = dict(name='Gumby', age=42)
>>> d
{'age': 42, 'name': 'Gumby'}
>>> fdict = dict((['x', 1], ['y', 2]))
>>> fdict
{'y': 2, 'x': 1}
```

Converting a Dictionary to a list will give a list of keys.

```
>>> numbers = {'first': 1, 'second': 2, 'third': 3, 'Fourth': 4}
>>> numbers
{'first': 1, 'second': 2, 'third': 3, 'Fourth': 4}
>>> list(numbers)
['first', 'second', 'third', 'Fourth']
```

FROM KEYS:

Dictionaries may also be created using a very convenient built-in method for creating a "default" dictionary whose elements all have the same value (defaulting to None if not given), fromkeys():

```
>>> ddict = {}.fromkeys(('x', 'y'), -1)
>>> ddict
{'y': -1, 'x': -1}
>>>
>>> edict = {}.fromkeys(('foo', 'bar'))
>>> edict
{'foo': None, 'bar': None}
```

Provided all the *key's values are the same initially*, you can also create a dictionary with this special form—simply pass in a list of keys and an initial value for all of the values (the default is None):

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

EXAMPLE OF CREATING A DICTIONARY:

```
>>>
> > > menu_specials = {}
> > > menu_specials['breakfast'] = 'Canadian ham'
> > > menu_specials['lunch'] = 'tuna surprise'
> > > menu_specials['dinner'] = 'Cheeseburger Deluxe'
> > > print(menu_specials)
{'breakfast': 'Canadian ham', 'lunch': 'tuna surprise', 'dinner': 'Cheeseburger Deluxe'}
> > > menu_specials = {'breakfast' : 'sausage and eggs',
'lunch' : 'split pea soup and garlic bread',
'dinner': '2 hot dogs and onion rings'}
>>>
> > > print(menu_specials)
{'breakfast': 'sausage and eggs', 'lunch': 'split pea soup and garlic bread', 'dinner': '2 hot dogs and
onion rings'}
>>>
> > > print("%s" % menu_specials["breakfast"])
sausage and eggs
>>>
>>>
```

DICTIONARY KEYS ARE UNIQUE:

So if we add a key–value item whose key is the same as an existing key, the effect is to replace that key's value with a new value.

```
>>>
>>>d = {'Name': 'Zara', 'Age': 7, 'Name': 'Mani'}
>>>print ("dict['Name']: ", d['Name'])
dict['Name']: Mani
>>>
```

ACCESSING VALUES IN DICTIONARY

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Following is a simple example.

```
>>>
>>>d = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
>>>print ("dict['Name']: ", d['Name'])
dict['Name']Zara
>>>
>>>print ("dict['Age']: ", d['Age'])
>>>
dict['Age']7
>>>
```

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

```
>>>d = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
>>>print( "dict['Alice']: ", d['Alice'] )
Traceback (most recent call last):
print( "dict['Alice']: ", dict['Alice']);
KeyError: 'Alice'
```

AVOIDING MISSING-KEY ERRORS

Errors for nonexistent key fetches are common in sparse matrixes, but you probably won't want them to shut down your program. There are at least three ways to fill in a default value instead of getting such an error message

- you can test for keys ahead of time in if statements,
- use a try statement to catch and recover from the exception explicitly
- use the dictionary get method shown earlier to provide a default for keys *that do not exist*.

Consider the first two of these previews for statement syntax:

```
>>> Matrix={}
>>> Matrix={(2, 3, 5) : 'T',(2, 4, 8) : 'F'}
>>>
>>> if (2, 3, 6) in Matrix:                    # Check for key before fetch
        print(Matrix[(2, 3, 6)])
else:
        print(0)
>>>
0
>>> try:
        print(Matrix[(2, 3, 6)])               # Try to index
except KeyError:                               # Catch and recover
        print(0)
>>>
0
>>>
>>> Matrix[(2, 3, 4)]
Traceback (most recent call last):
File "<pyshell#5>", line 1, in <module>
Matrix[(2, 3, 4)]
KeyError: (2, 3, 4)
>>>
>>> Matrix.get((2, 3, 4), 0)
0
>>>
>>> Matrix.get((2, 4, 8), 0)
'F'
>>>
```

Of these, the get method is the most concise in terms of coding requirements, but the if and try statements are much more general in scope.

NESTING IN DICTIONARIES.

The following, for example, fills out a dictionary describing a hypothetical person, by assigning to new keys over time.

```
>>> rec = {}
>>> rec['name'] = 'Bob'
>>> rec['age'] = 40.5
>>> rec['job'] = 'developer/manager'
>>>
>>> print(rec['name'])
Bob
```

Especially when nested, Python's built-in data types allow us to easily represent structured information.

The following again uses a dictionary to capture object properties, but it codes it all at once (rather than assigning to each key separately) and nests a list and a dictionary to represent structured property values:

```
>>> rec = {'name': 'Bob',
           'jobs': ['developer', 'manager'],
           'web': 'www.bobs.org/˜Bob',
           'home': {'state': 'Overworked', 'zip': 12345}}
```

To fetch components of nested objects, simply string together indexing operations:

```
>>> rec['name']
'Bob'
>>> rec['jobs']
['developer', 'manager']
>>> rec['jobs'][1]
'manager'
>>> rec['home']['zip']
12345
```

Also notice that we've focused on a single "record" with nested data here.

There's no reason we couldn't nest the record itself in a larger, enclosing database collection coded as a list or dictionary, though an external file or formal database interface often plays the role of top-level container in realistic programs:

```
>>> db = []
>>> db.append(rec)                              # A list "database"
>>> db.append(other)
>>> db[0]['jobs']
>>>
>>> db = {}
>>> db['bob'] = rec                             # A dictionary "database"
>>> db['sue'] = other
>>> db['bob']['jobs']
```

Notice also the nesting of a list inside a dictionary in this example (the value of the key 'ham'). All collection data types in Python *can nest inside each other arbitrarily*:

```
>>> D={'eggs': 3, 'spam': 2, 'ham': 1}
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
>>> D['ham'] = ['grill', 'bake', 'fry']            # Change entry (value=list)
>>> D
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> del D['eggs']                                  # Delete entry
>>> D
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> D['brunch'] = 'Bacon'                          # Add new entry
>>> D
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

DICTIONARY-A

NESTED DICTIONARIES.

For example, here's a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic. The totalBrought() function can read this data structure and calculate the total number of an item being brought by all the guests.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}
             }

def totalBrought(guests, item):
    numBrought = 0
    for k, v in guests.items():
        numBrought = numBrought + v.get(item, 0)
        return numBrought

print('Number of things being brought:')
print(' - Apples ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies ' + str(totalBrought(allGuests, 'apple pies')))
```

- Inside the totalBrought() function, the for loop iterates over the key-value pairs in guests.
- Inside the loop, the string of the guest's name is assigned to k, and the dictionary of picnic items they're bringing is assigned to v.
- If the item parameter exists as a key in this dictionary, it's value (the quantity) is added to numBrought v.
- If it does not exist as a key, the get() method returns 0 to be added to numBrought.

The output of this program looks like this:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies 1
```

This same totalBrought() function could easily handle a dictionary that contains thousands of guests, each bringing thousands of different picnic items.Then having this information in a data structure along with the totalBrought() function would save you a lot of time!

MIXED-VALUE DICTIONARIES

Dictionary values can be any datatype, including integers, Booleans, arbitrary objects, or even other dictionaries. And within a single dictionary, the values don’t all need to be the same type; you can mix and match as needed. Dictionary keys are more restricted, but they can be strings, integers, and a few other types. You can also mix and match key datatypes within a dictionary.

```
>>>
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
                1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>>
>>> len(SUFFIXES)
2
>>>
>>> 1000 in SUFFIXES
True
>>> SUFFIXES[1000]
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>>
>>> SUFFIXES[1024]
['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
>>>
>>> SUFFIXES[1000][3]
'TB'
>>>
```

- The len() function gives you the number of keys in a dictionary.
- You can use the in operator to test whether a specific key is defined in a dictionary.
- 1000 is a key in the SUFFIXES dictionary; its value is a list of eight items (eight strings, to be precise).
- Similarly, 1024 is a key in the SUFFIXES dictionary; its value is also a list of eight items.
- Since SUFFIXES[1000] is a list, you can address individual items in the list by their 0-based index.

CHANGING DICTIONARIES IN PLACE

Dictionaries, like lists, are mutable, so you can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry.

UPDATING DICTIONARY

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

```
>>>
>>>d = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
>>>d['Age'] = 8;                               # update existing entry
>>>d['School'] = "DPS School"                   # Add new entry
>>>
>>>print ("dict['Age']: ", d['Age'])
>>>print ("dict['School']: ", d['School'])
dict['Age']: 8
dict['School']: DPS School
>>>
```

Dictionaries do not have any predefined size limit. You can add new key-value pairs to a dictionary at any time, or you can modify the value of an existing key.

```
>>>
>>> a_dict={'server': 'db.python3.org', 'database': 'mysql'}
>>> a_dict
{'server': 'db.python3.org', 'database': 'mysql'}
>>>
>>> a_dict['database'] = 'blog'
>>> a_dict
{'server': 'db.python3.org', 'database': 'blog'}
>>>
>>> a_dict['user'] = 'mark'
>>> a_dict
{'server': 'db.python3.org', 'user': 'mark', 'database': 'blog'}
>>>
>>> a_dict['user'] = 'dora'
>>> a_dict
{'server': 'db.python3.org', 'user': 'dora', 'database': 'blog'}
>>>
>>> a_dict['User'] = 'mark'
>>> a_dict
{'User': 'mark', 'server': 'db.python3.org', 'user': 'dora', 'database': 'blog'}
>>>
```

DICTIONARY-A

1. You cannot have duplicate keys in a dictionary. Assigning a value to an existing key will wipe out the old value.

2. You can add new key-value pairs at any time. This syntax is identical to modifying existing values.

3. The new dictionary item (key 'user', value 'mark') appears to be in the middle. In fact, it was just a coincidence that the items appeared to be in order in the first example; it is just as much a coincidence that they appear to be out of order now.

4. Assigning a value to an existing dictionary key simply replaces the old value with the new one.

5. Dictionary keys are case-sensitive, so this statement is creating a new key-value pair, not overwriting an existing one. It may look similar to you, but as far as Python is concerned, it’s completely different.

DELETE DICTIONARY ELEMENTS

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation. To explicitly remove an entire dictionary, just use the del statement. del deletes the entry associated with the key specified as an index.

Following is a simple example-

```
>>>d = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
>>>del d['Name']                    # remove entry with key 'Name'
>>>d.clear()                                # remove all entries in dict
>>>del d                                     # delete entire dictionary
>>>
>>>print ("dict['Age']: ", d['Age'])
>>>print ("dict['School']: ", d['School'])
```

This produces the following result. Note: An exception is raised because after **del** dict, the dictionary does not exist anymore.

```
dict['Age']:
Traceback (most recent call last):
File "test.py", line 8, in <module>
print "dict['Age']: ", d['Age'];
TypeError: 'type' object is unsubscriptable
```

DICTIONARIES IN A BOOLEAN CONTEXT

You can also use a dictionary in a Boolean context, such as an **if** statement.

```
>>>
>>> d= {}
>>>
>>> if d:
	print("true")
else:
	print('false')
false
>>>
>>> d={1:'a'}
>>>
>>> if d:
	print("true")
else:
	print('false')
true
>>>
>>> if not d: print('D is empty')
>>>
>>> if d: print(d)
{1: 'a'}
>>>
```

1. In a Boolean context, an empty dictionary is **false**.
2. Any dictionary with at least one key-value pair is **true**.

DICTIONARY-A

DICTIONARIES VS. LISTS

- Unlike lists, items in dictionaries are unordered.
- The first item in a list named spam would be spam[0]. But there is no “first” item in a dictionary.
- While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

```
>>>
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>>
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
>>>
```

- Because dictionaries are not ordered, they can’t be sliced like lists.
- Trying to access a key that does not exist in a dictionary will result in a KeyError error message, much like a list’s “out-of-range” *IndexError* error message.

Enter the following into the interactive shell, and notice the error message that shows up because there is no 'color' key:

```
>>>
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
spam['color']
KeyError: 'color'
>>>
```

DICTIONARY-A

EXAMPLE: You can use a dictionary with the names as keys and the birthdays as values. Enter the following code.

```
birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}
while True:
        print('Enter a name: (blank to quit)')
        name = input()
        if name == '':
                break
        if name in birthdays:
                print(birthdays[name] + ' is the birthday of ' + name)
        else:
                print('I do not have birthday information for ' + name)
                print('What is their birthday?')
                bday = input()
                birthdays[name] = bday
                print('Birthday database updated.')
```

- You create an initial dictionary and store it in birthday.
- You can see if the entered name exists as a key in the dictionary with the *in* keyword, just as you did for lists.
- If the name is in the dictionary, you access the associated value using square brackets;
- If not, you can add it using the same square bracket syntax combined with the assignment operator.
- When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions-

Function	Description
cmp(dict1, dict2)	No longer available in Python 3
len(dict)	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
str(dict)	Produces a printable string representation of a dictionary
type(variable)	Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type

Dictionary len() Method

Description

The method len() gives the total length of the dictionary. This would be equal to the number of items in the dictionary.

Example

The following example shows the usage of len() method.

```
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Length : %d" % len (dict))
```

When we run the above program, it produces the following result-

```
Length : 3
```

Dictionary str() Method

Description

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

Example

The following example shows the usage of str() method.

```
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Equivalent String : %s" % str (dict))
```

When we run the above program, it produces the following result-

```
Equivalent String : {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
```

DICTIONARY-A

Dictionary type() Method

Description

The method type() returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.

Example

The following example shows the usage of type() method.

```
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Variable Type : %s" % type (dict))
```

When we run the above program, it produces the following result-

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

PYTHON INCLUDES THE FOLLOWING DICTIONARY METHODS.

Methods	Description
dict.clear()	Removes all elements of dictionary dict
dict.copy()	Returns a shallow copy of dictionary dict
dict.fromkeys()	Create a new dictionary with keys from seq and values set to value
dict.get(key, default=None)	For key key, returns value or default if key not in dictionary
dict.has_key(key)	Removed, use the in operation instead
dict.items()	Returns a list of dict's (key, value) tuple pairs
dict.keys()	Returns list of dictionary dict's keys
dict.setdefault(key, default=None)	Similar to get(), but will set dict[key]=default if key is not already in dict
dict.update(dict2)	Adds dictionary dict2's key-values pairs to dict
dict.values()	Returns list of dictionary dict's values

Dictionary clear() Method

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

Example
The following example shows the usage of clear() method.

```
dict1 = {'Name': 'Zara', 'Age': 7}
print ("Start Len : %d" % len(dict1))
dict1.clear()
print ("End Len : %d" % len(dict1))
```

When we run the above program, it produces the following result-

```
Start Len : 2
End Len : 0
```

Dictionary copy() Method

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

Example
The following example shows the usage of copy() method.

```
dict1 = {'Name': 'Mani', 'Age': 7, 'Class': 'First'}
dict2 = dict1.copy()
print ("New Dictionary : ",dict2)
```

When we run the above program, it produces following result-

```
New dictionary : {'Name': 'Mani', 'Age': 7, 'Class': 'First'}
```

Dictionary fromkeys() Method

Description

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

Example

The following example shows the usage of fromkeys() method.

```
seq = ('name', 'age', 'sex')
dict1 = dict1.fromkeys(seq)
print ("New Dictionary : %s" % str(dict1))
dict2 = dict2.fromkeys(seq, 10)
print ("New Dictionary : %s" % str(dict2))
```

When we run the above program, it produces the following result-

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

Dictionary get() Method

Description

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

Example

The following example shows the usage of get() method.

```
dict1 = {'Name': 'Zara', 'Age': 27}
print ("Value : %s" % dict1.get('Age'))
print ("Value : %s" % dict1.get('Sex', "NA"))
```

When we run the above program, it produces the following result-

```
Value : 27
Value : NA
```

Dictionary items() Method

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

Example
The following example shows the usage of items() method.

```
dict1 = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" % dict1.items())
```

When we run the above program, it produces the following result-

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

Dictionary keys() Method

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

Example
The following example shows the usage of keys() method.

```
dict1 = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" % dict1.keys())
```

When we run the above program, it produces the following result-

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

Dictionary setdefault() Method

Description

Similar to get(), but will set dict[key]=default if the key is not already in dict.

Example

The following example shows the usage of setdefault() method.

```
dict1 = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" % dict1.setdefault('Age', None))
print ("Value : %s" % dict1.setdefault('Sex', None))
print (dict1)
```

When we run the above program, it produces the following result-

```
Value : 7
Value : None
{'Name': 'Zara', 'Sex': None, 'Age': 7}
```

Dictionary update() Method

Description

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

Example

The following example shows the usage of update() method.

```
dict1 = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict1.update(dict2)
print ("updated dict : ", dict1)
```

When we run the above program, it produces the following result updated

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

Dictionary values() Method

Description

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

Example

The following example shows the usage of values() method.

```
dict = {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
print ("Values : ", list(dict.values()))
```

When we run above program, it produces following result-

```
Values : ['female', 7, 'Zara']
```

DICTIONARY-A

For reference and preview again, the Table summarizes some of the most common and representative dictionary operations, and is relatively complete as of Python 3.3.

As usual, though, see the library manual or run a dir(dict) or help(dict) call for a complete list

Operation	Interpretation
D = {}	Empty dictionary
D = {'name': 'Bob', 'age': 40}	Two-item dictionary
E = {'cto': {'name': 'Bob', 'age': 40}}	Nesting

D = dict(name='Bob', age=40)
D = dict([('name', 'Bob'), ('age', 40)])
D = dict(zip(keyslist, valueslist))
D = dict.fromkeys(['name', 'age'])
	Alternative construction techniques:
	keywords,
	key/value pairs,
	zipped key/value pairs,
	key lists

D['name'] E['cto']['age']	Indexing by key 'age' in D Membership: key present test

D.keys()
D.values()
D.items()
D.copy()
D.clear()
D.update(D2)
D.get(key, default?)
D.pop(key, default?)
D.setdefault(key, default?)
D.popitem()
	Methods:
	all keys,
	all values,
	all key+value tuples,
	copy (top-level),
	clear (remove all items),
	merge by keys,
	fetch by key, if absent default (or None),
	remove by key, if absent default (or error)
	fetch by key, if absent set default (or None),
	remove/return any (key, value) pair; etc.

len(D)	Length: number of stored entries
D[key] = 42	Adding/changing keys
del D[key]	Deleting entries by key

list(D.keys())
D1.keys()
D2.keys()
	Dictionary views (Python 3.X)

D.viewkeys(), D.viewvalues()	Dictionary views (Python 2.7)
D = {x: x*2 for x in range(10)}	Dictionary comprehensions (Python 3.X, 2.7)