Dictionary in Python

Dictionaries are simply another type of collection in Python, but with twist. Rather than having an index associated with each data item, Dictionaries in Python have a "Key" and a **"value of that key"**. That is Python dictionaries are a collection of some **key-value** pairs.

Creating a Dictionary

To create a dictionary, you need to include the key:value pair in curly braces as per following syntax:

```
<dictionary-name> = {<key>:<value>, <key>:<value>, …..}
```

Following is an example dictionary name teacher that store the names of teachers as keys and subject being taught by them as values of respective keys.

```
teachers = {"Dimple": "Computer Science", "Karen":
"Sociology", "Harpreet": "Mathematics", "Sabah":"Legal
Studies" };

dict1 = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,
"June:30"};
```

One thing that you must know is that keys of a dictionary must be of immutable types, such as:

- a Python string,
- a number
- a tuple (containing only immutable entries)

Accessing Elements of a Dictionary

The elements are accessed through the keys defined in the key:value pairs, as per the syntax show below:

<dictionary-name>[<key>]

Thus to access the value for key defined as "Karen" in above declared teachers dictionary, you will write:

```
>>>teachers["Karen"]
```

Similarly, following statement

```
>>> print("Karen teaches", teachers['Karen'])
```

will give output as :

Karen teaches Sociology

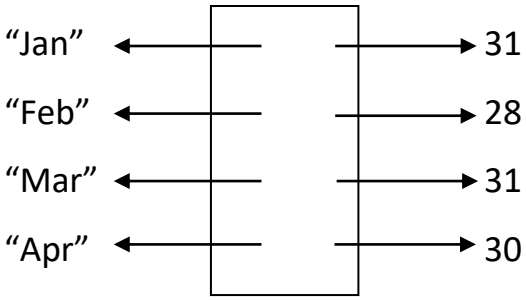
Also, attempting to access a key that doesn't exist causes an error.

d[13]

The above error means that before we can access the value of a particular key using expression such as d[13], we must first ensure the key(13 here) exist in the dictionary.

Memory representation of Dictionary

Like list elements, the key and values of a dictionary are stored through their references. Following figure illustrate the same for dict1



Traversing a Dictionary

Traversal of a collection means accessing and processing each element of it. Thus traversing a dictionary also means the same and same is the tool for it, i.e. the Python loops.

```
for <item> in <Dictionary>:
    process each item here
```

Consider following example : A dictionary namely d1 is defined with three keys – a number, a string, a tuple of integers.

```
d1 = {5 : "number",
    "a": "string",
    (1,2): "tuple"}
```

To traverse the above dictionary you can write for loop as:

```
for key in d1:
    print(key, ":", d1[key])-
```

The above loop will produce the output as show below:

a : String
(1, 2) : tuple
5 : number

How it works

The loop variable key in above loop will be assigned the keys of the Dictionary d1, one at a time. As dictionary elements are unordered, the order of assignment of keys may be different from what you stored.

Using the loop variable, which has been assigned one key at a time, the corresponding value is printed along with the key inside the loop-body using through statement

```
print(key, ":", d1[key])
```

Accessing Keys or Values Simultaneously

To see all the keys in a dictionary in one go, you may write <dictionary>.keys() and to see all the values in one go, you may write <dictionary>.values()

```
d1.keys()
[5, 'a', (1, 2)]
d1.values()
['number', 'string', 'tuple']
```

As you can see that the keys() function return all the keys defined in a dictionary in the form of list type and also the values() function returns all the values defined in that dictionary in the form list type.

Characteristics of a Dictionary

1. Unordered Set
   A dictionary is unordered set of key : value pairs.
2. Not a sequence
   Unlike the string, list and tuple, a dictionary is not a sequence because it is unordered set of elements. The sequences are indexed by a range of ordinal numbers. Hence they are ordered, but a dictionary is an unordered collection.
3. Keys must be unique
   Each of the keys within a dictionary must be unique. Since keys are used to identify values in a dictionary, there cannot be duplicate keys in a dictionary. However, two unique keys can have same values.
   for e.g.
   dayInMonth = {"Jan":31, "Feb":28, "Mar": 31}
   Two different keys "Jan" and "Mar" have same value 31.
4. Mutable
   Like lists, dictionaries are also mutable. We can change the value of a certain key "in place" using the assignment statement as per syntax

   ```
   <dictionary>[key] = <value>
   >>>dayInMonth ["Feb"]
   >>>28
   >>>dayInMonth["Feb"]=29
   >>> dayInMonth["Feb"]
   >>>29
   ```

5. Internally stored as mapping

Internally, the key : value pair of a dictionary are associated with one another with some internal function (called hash-function). This way of linking is called mapping.

Working with Dictionaries

Multiple ways of creating Dictionaries

There are other ways of creating dictionaries too.

1. **Initializing a Dictionary**
   In this method all the key : value pair of a dictionary are written collectively, separated by commas and enclosed in curly braces.

```
Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24 }
```

2. **Adding key : value pairs to an empty dictionary**
   In this method, firstly an empty dictionary is created and then keys and values are added to it one pair at a time.
   To create an empty dictionary, there are two ways:
   i) by giving dictionary contents in empty curly braces

```
Employees = {}
```

   ii) by using dictionary constructor dict()

```
Employees = dict()
```

   Next step is to add key : value pairs, one at a time

```
Employee['desi']='Salesman'
```

3. **Creating a dictionary from name and value p**airs
   Using the dict() constructor of dictionary, you can also create a new dictionary initialized from specified set of keys and values. There are multiple ways to provide keys and values to dict() constructor.
   i) Specify Key : Value pairs as keyword argument to dict(), keys as argument and Values as their values.

```
Employee = dict(name = 'Joshi', salary = 10000, age=24)
>>>Employee
{'name': 'Joshi', 'salary': 10000, 'age': 24}
```

   ii) Specify keys separately and corresponding values separately. In this are enclosed separately in parentheses and are given as argument to zip() function, which is given as argument of dict().

```
>>>Employee = dict(zip(('name', 'salary', 'age'), ('Joshi', 10000, 24)))
>>>Employee
{'name': 'Joshi', 'salary': 10000, 'age': 24}
```

iii) Specify Key: Value pairs separately in form of sequence. In this method, one list or tuple argument is passed to dict().

```
>>>Employee = dict([['name', 'John'], ['salary', 10000],
['age', 24]])
>>>Employee
{'name': 'John', 'salary': 10000, 'age': 24}
```

Adding Element to Dictionary

You can add new element to a dictionary using assignment as per the following syntax. BUT the key being added must not exist in dictionary and must be unique. If the already exists, then this statement will change the value of existing key and not new entry will be added to dictionary.

```
>>>Employee['dept']='Sales'
```

Updating Existing Elements in a Dictionary

Updating an element is similar to what we did just now. That is you can change value of an existing key using assignment.

```
>>>Employee['salary'] = 20000
>>>Employee
{'name': 'John', 'salary': 20000, 'age': 24}
```

Deleting Element from a Dictionary

There are two methods for deleting elements from a dictionary

i) To delete a dictionary element or a dictionary entry i.e. a key : value pair, you can use del command.

```
del <dictionary>[key]
>>>Employee['age']
>>>Employee
{'name': 'John', 'salary': 20000}
```

ii) Another method to delete element from a dictionary is by using pop() method as per following syntax:

```
<dictionary>.pop(<key>)
```

The pop() method will not only delete the key : value pair it also return the corresponding value.

```
>>>Employee.pop('age')
24
>>>Employee
{'name': 'John', 'salary': 20000}
```

Checking for Existence of a key

Usual membership operators in and not in work with dictionaries as well. But they can check for the existence of keys only. To check for whether a value is present in a dictionary, you need to write proper code for that.

```
<key> in <dictionary>
<key> not in <dictionary>
>>>'age' in Employee
True
>>> 'John' in Employee
False
```

Dictionary Function and Methods

1. The len() method

This method return length of the dictionary i.e. the count of elements(key : value) pairs in the dictionary.

```
len(<dictionary>)
```

Takes a dictionary name as argument and returns an integer.

```
>>> len(Employee)
3
```

2. The clear() method

This method removes all items from the dictionary and the dictionary becomes empty dictionary post this method.

```
<dictionary>.clear()
Takes no argument, return no value.
>>> Employee.clear()
>>> Employee
{ }
```

As mentioned, the clear method does not delete the dictionary, it just deletes all the elements inside the dictionary. If, however you want to delete the dictionary itself so that dictionary is also removed and no object remains, then you can use del statement along with dictionary name.

```
>>>del Employee
>>>Employee
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    Employee
NameError: name 'Employee' is not defined
```

3. The get() method

With this method, you can get the item with the given key, similar to **dictionary[key].** If the key not present, Python by default gives error, but you can specify your own message through default argument as per following syntax:

```
   <dictionary>.get(key, [default])
>>> Employee = dict({'name':'John', 'salary':1000,
'age':24})
>>> Employee.get('salary')
```

1000

4. The has_key() method

This method checks for the presence of given key(not value) in the dictionary .

```
   <dictionary>.haskey(key)
```

- Takes key as argument and returns Boolean True or False.

```
>>> employee = {'name':'John', 'salary': 10000, 'age':24}
>>> employee.has_key('dept')
```

False

```
>>>employee.has_key('name')
```

True

5. The items() method

This method returns all of the items in the dictionary as sequence of(key, value) tuples. Note that these are returned in no particular order.

   <dictionary>.items()

- Takes no argument; Returns a sequence of (key, value) pair

```
>>> myList = employee.items()
>>> for x in myList:
       print(x)
('name', 'John')
('salary', 10000)
('age', 24)
```

6. The update() method

This method merge key:value pairs from the new dictionary into original dictionary, adding or replacing as needed. The items in the new dictionary are added to the old one and override any items already there with the same keys.

```
   <dictionary>.update(<other-dictionary>)
>>> employee1 = {'name':'John', 'salary':10000, 'age':24}
>>> employee2 = {'name':'Diya', 'salary':5400, 'dept':
'sales'}
 >>> employee1.update(employee2)
>>> employee1
```

```
{'name': 'Diya', 'salary': 5400, 'age': 24, 'dept':
'sales'}
```

7. The cmp() method

This method is used to compare two dictionaries based on their keys and values.

cmp(dict1, dict2)

- Takes two parameter of dictionary type

This method returns an integer as per these conditions

- returns 0 if both dictionaries are equal
- returns -1 if dict1 < dict2
- returns 1 if dict1 > dict2

the comparison is indeed based on lexicographical order of keys. The smallest key is compared first. If it is same, the next smallest key is compared, and so on.

```
1.dict1 = {'Name': 'C', 'Age': 30, 'Weight': 55}
2.dict2 = {'Name': 'B', 'Age': 34, 'Weight': 50}
3.
4. print "Return Value : %d" %  cmp (dict1, dict2)
```

Return value: -1 suggests that Age takes preference over both Name and Weight (dict1 has bigger Name and Weight).

So, the first thing to be compared is Age, then Name is compared, then Weight. Note that the keys are mentioned in the dictionary in the order Name, Age and then Weight. This suggests that the order of comparison has nothing to do with the order in which the keys are written in the dictionary.