Programming with Python(PCCCS(AIML)393) Dictionaries

Miss CHITTABARNI SARKAR

(Assistant Professor,Dept of CSE,FIT)

Dictionaries

Creating a dictionary is as simple as placing items inside curly braces **{}** separated by commas.
An item has a key and a corresponding value that is expressed as a pair (**key: value**).
While the values can be of any data type, keys must be of **mutable type (string, number or tuple with immutable elements) and must be unique.**

Empty dictionary

dict1 = {}

Dictionary with integer keys

dict2 = {1: ‘Kolkata', 2: ‘Delhi'}

Dictionary with mixed keys

dict3 = {‘city': ‘Kolkata', 1: [2, 4, 3]}

Using dict() function

dict4 = dict({1:'Kolkata', 2:'Delhi'})

# from sequence having each item as a pair

dict5 = dict([(1,‘Kolkata'), (2,'Delhi')])

Continued

Characteristics of Dictionaries

- Items are ordered.
- Items can be changeable
- It does not allow duplicate elements
- It supports several types of data elements
- Dictionaries are written with curly brackets, and have keys and values:

Note: As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Accessing Values in Dictionary

Other data types use indexing to access values but a dictionary uses keys. Keys can be used either **inside square brackets [] or with the get() method.**

Note: If an user uses the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

get & [] for retrieving elements

dict1 = {‘Subject': ‘Computer', ‘Mark': 86}

print(dict1[' Subject']) Output: Computer
print(dict1.get(' Mark')) Output: 86

Try to access keys which doesn't exist throws error
print(dict1.get(‘city')) Output None
print(dict1[‘city']) KeyError

Continued

Example

```
thisdict = {
“Name": “A",
  “Subject": “Math",
  “Roll": “10"
}
```

get() method

```
x = thisdict.get(“Roll“)
print(x)
```
Output: 10

- Get keys

The keys() method will return a list of all the keys in the dictionary.

```
x = dict1.keys()
print(x)
```
Output:dict_keys([‘Name', ‘Subject‘,’Roll’])

Continued

Get Values

The values() method will return a list of all the values in the dictionary.

Example:

x = thisdict.values()

print(x)

Output: dict_keys([‘A', ‘Math‘,10])

Get Items

The items() method will return each item in a dictionary, as tuples in a list.

x = thisdict.items()

print(x)

Output:dict_items([(‘Name', ‘A'), (‘Subject', 'Math'), (‘Roll', 10)])

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword:

if “Roll" in thisdict:

  print("Yes, ‘Roll is one of the keys in the thisdict dictionary")

Output:Yes, ‘Roll is one of the keys in the thisdict dictionary

Changing and Adding Dictionary elements

Dictionaries are mutable. User can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

Changing and adding Dictionary Elements

```
thisdict = {
"Name": "A",
  "Subject": "Math",
  "Roll": "10"
}
thisdict["city"]=["kolkata"]

print(thiscity) Ans. {'Name': 'A', 'Subject': 'Math', 'Roll': 10, 'city': 'kolkata'}
```

# update value

```
dict1['city1'] = Mumbai

Or

thisdict.update({"color": "Delhi"})

print(dict1) or print(thisdict) Ans. {'Name': 'A', 'Subject': 'Math', 'Roll': 10, 'city': 'Delhi'}
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword:

```
dict1 = {city1: 'Kolkata', city2: 'Delhi'}

# Yes, 'city2' is one of the keys in the dict1 dictionary

if " city2 " in dict1:
  print("Yes, 'city2' is one of the keys in the dict1 dictionary")
```

Deleting elements from Dictionary

Methods & Keywords

User can delete a particular item in a dictionary by using

- pop() method:-This method deletes an item with the provided key and returns the value.

- popitem() method can be used to delete and return an arbitrary (key, value) item pair from the dictionary.

- clear:-All the items can be deleted at once, using the clear() method.
- del:-We can also use the del keyword to delete individual items or the entire dictionary itself.

Examples

- **pop()**

thisdict.pop(“Subject”)
print(thisdict) Ans. {‘Name': ‘A', ‘Roll': 10}

- **popitem()**

thisdict.popitem()

print(thisdict) Ans{‘Name': ‘A', ‘Roll': 10} It can select random value

- **del-keyword** removes the item with the ***specified key name*** and he del keyword can also delete the ***dictionary completely***:

del thisdict[“Roll"]

print(thisdict) Ans. {‘Name': ‘A', ‘Subject': ‘Math’}

del thisdict

print(thisdict)

Ans. this will cause an error because "thisdict" no longer exists.
NameError: name 'thisdict' is not defined

clear():The clear() method empties the dictionary:

thisdict.clear()

print(thisdict) **Ans.{}**

Loop Through a Dictionary

User can loop through a dictionary by using a for loop.
When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Examples

```
dict1 = {city1: ‘Kolkata', city2: ‘Delhi'}
for x in dict1:
  print(x)
#print all values in the dictionary, one by one:
Output:
Kolkata
Delhi
for x in dict1:
  print(dict1[x])
Output: Kolkata
        Delhi
#User can also use the values() method to return values of a dictionary:
for x in dict1.values():
  print(x)
#User can use the keys() method to return the keys of a dictionary:
city1
city2
for x in dict1.keys():
  print(x)
#Loop through both keys and values, by using the items() method:
for x, y in dict1.items():
  print(x, y) #city1 Kolkata
              #city2 Delhi
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries

Create a dictionary that contain three dictionaries:

```
students = {
 “student1" : {
   "name" : "Emil",
   "year" : 2004
 },
 “student2" : {
   "name" : "Tobias",
   "year" : 2007
 },
 “student3" : {
   "name" : "Linus",
   "year" : 2011
 }
}
print(students)
```

Output

```
{‘student1': {'name': 'Emil', 'year': 2004}, ‘student2': {'name': 'Tobias',
'year': 2007}, ‘student3': {'name': 'Linus', 'year': 2011}}
```

```
student1 = {
  "name" : "Emil",
  "year" : 2004
}
student2 = {
  "name" : "Tobias",
  "year" : 2007
}
student3 = {
  "name" : "Linus",
  "year" : 2011
}
students= {
  “student1" : student1,
  “student2" : student2,
  “student3" : student3
}
print(students)
```

Output

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias',
'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}
```

Copy a Dictionary

User **cannot** copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a *reference* to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

```
# {city1: 'Kolkata', city2: 'Delhi'}
dict1 = {city1: 'Kolkata', city2: 'Delhi'}
dict2 = dict1.copy()
print(dict2) Ans. {city1: 'Kolkata', city2: 'Delhi'}
```

Another way to make a copy is to use the built-in function dict().

```
# {city1: 'Kolkata', city2: 'Delhi'}
dict2 = dict(dict1)
print(dict2) Ans. {city1: 'Kolkata', city2: 'Delhi'}
```

Functions

Function Names	OUTPUTS
cmp(dict1, dict2)	Compares elements of both dict.
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.
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*.

Continued

Function Names	OUTPUTS
dict.get(key, default=None)	For key, returns value or default if key not in dictionary
dict.has_key(key)	Returns true if key in dictionary dict, false otherwise
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

Continued

Function Names	OUTPUTS
fromkeys(seq[, v])	Returns a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Returns the value of the key. If the key does not exist, returns d (defaults to None).
pop(key[,d])	Removes the item with the key and returns its value or d if key is not found. If d is not provided and the key is not found, it raises KeyError.
popitem()	Removes and returns an arbitrary item (**key, value**). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of d and returns d (defaults to None).

Properties of Dictionary Keys

- Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.
- There are two important points to remember about dictionary keys –

(a) 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.

For example –

```
dict = {'Name': ‘Rita', ‘class': 7, 'Name': ‘Ram'}
print "dict['Name']: ", dict['Name']
```

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

```
dict['Name']: Ram
```

(b) Keys must be immutable. Which means user can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example –

Live Demo

```
#!/usr/bin/python dict = {['Name']: ‘Rita', ‘class': 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: unhashable type: 'list'
```

Dictionary Membership Test

User can test if a key in the present inside a dictionary or not. This test can be performed only on the key of a dictionary and not the value. The membership test is done using the **in** keyword. When user checks the key in the dictionary using the **in** keyword, the expression returns true if the key is present and false if not.

```
my_dict = {"username": "XYZ", "email": "xyz@gmail.com", "location":"Mumbai"}
print("email" in my_dict)
 print("location" in my_dict)
print("test" in my_dict)
```

Output:

```
True
True
False
```

Difference between Dictionaries and Lists

Dictionary	Lists
Defined with a curly brace({}) enclosing one or more comma-separated key-value pairs where a key and its associated value is separated by a colon(:).	Defined with a square bracket([]) where each item in the list is separated by a comma(,), with the first item at index 0.
Elements are accessed using keys.	Elements are accessed by their position in the list, using indexing.

Assignments

1.Find the mean of all values in Dictionary

```
test_dict = {"zero" : 4, "one" : 7, "two" : 8, "three" : 6, "four" : 10}
# printing original dictionary
print("The original dictionary is : " + str(test_dict))
# loop to sum all values
res = 0
for val in test_dict.values():
    res += val
# using len() to get total keys for mean computation
res = res / len(test_dict)
# printing result
print("The computed mean : " + str(res))
```

<u>Output:</u>

```
The original dictionary is : {'zero': 4, 'one': 7, 'two': 8, 'three': 6, 'four': 10}
The computed mean : 7.0
```

Dictionary Comprehensive

Just like list comprehensive dictionary comprehension is an elegant way. We can create dictionaries using simple expressions. It is basically mathematical statements in one line. It consists of Output expression,Input sequence,A variable representing a member of the input sequence and an optional predicate part. Such as

lst = [x ** 2 for x in range (1, 11) if x % 2 == 1]

here, x ** 2 is output expression, range (1, 11) is input sequence, x is variable and if x % 2 == 1 is predicate part.

A dictionary comprehension takes the form {key: value for (key, value) in iterable}

Example

keys = ['a','b','c','d','e']

values = [1,2,3,4,5]

# but this line shows dict comprehension here

myDict = { k:v for (k,v) in zip(keys, values)}

# We can use below too

myDict = dict(zip(keys, values))

print (myDict) Ans. {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

Continued

2.Maximum record value key in dictionary

```
test_dict = {'gfg' : {'Manjeet' : 5, 'Himani' : 10},
             'is' : {'Manjeet' : 8, 'Himani' : 9},
             'best' : {'Manjeet' : 10, 'Himani' : 15}}
# printing original dictionary
print("The original dictionary is : " + str(test_dict))
# initializing search key
key = 'Himani'
# Maximum record value key in dictionary # Using loop
res = None
res_max = 0
for sub in test_dict:
    if test_dict[sub][key] > res_max: #test_dict[sub][key] is 15
        res_max = test_dict[sub][key]
        res = sub  #best
# printing result
print("The required key is : " + str(res))
```

Output

```
he original dictionary is : {'gfg': {'Manjeet': 5, 'Himani': 10}, 'is': {'Manjeet': 8, 'Himani': 9}, 'best': {'Manjeet': 10, 'Himani': 15}}
The required key is : best
```

Continued

3. Group Similar items to Dictionary Values List

```
from collections import Counter
# initializing list
test_list = [4, 6, 6, 4, 2, 2, 4, 4, 8, 5, 8]
# printing original list
print("The original list : " + str(test_list))
# using * operator to perform multiplication
res = {key : [key] * val for key, val in Counter(test_list).items()}
# printing result
print("Similar grouped dictionary : " + str(res))
```

Output:

```
The original list : [4, 6, 6, 4, 2, 2, 4, 4, 8, 5, 8]
Similar grouped dictionary : {4: [4, 4, 4, 4], 6: [6, 6], 2: [2, 2], 8: [8, 8], 5: [5]}
```

Note:Counter is a sub-class that is used to count hashable objects. It implicitly creates a hash table of an iterable when invoked.

Continued

4. **Counting the frequencies in a list using dictionary in Python**

```
def CountFrequency(my_list):
    # Creating an empty dictionary
    freq = {}
    for item in my_list:
        if (item in freq):
            freq[item] += 1
        else:
            freq[item] = 1
    for key, value in freq.items():
        print ("% d : % d"%(key, value))
# Driver function
if __name__ == "__main__":
    my_list =[1, 1, 1, 5, 5, 3, 1, 3, 3, 1, 4, 4, 4, 2, 2, 2, 2]
    CountFrequency(my_list)
```

Output:

```
1 :  5
 5 :  2
 3 :  3
 4 :  3
 2 :  4
```