- Dictionary Object,
- Dictionary Comprehension
- Comparative study of Dictionary and List

Features of Dictionary

- **Dictionary** is an **unordered** set of **key and value pair**
- **Mutable** i.e., value can be updated.
- **Key** must be **unique** and immutable, such as numbers, strings
- V**alues** of a dictionary may **be any data type**
- **key and value** is known as **item**
- **Container** that contains data, enclosed within **curly braces**.

Creating Dictionary

- **Dictionary** enclosed within **curly braces**.
- **The key and the value is separated by a colon (:), pair is known as item**
- **Items are separated from each other by a comma (,)**
- **Different items are enclosed within a curly brace and this forms Dictionary**

Creating dictionaries example

```
dict1 = {'Name': 'Ajay', 'Age':30, 'Profession’ : 'Programmer'}
print(dict1)


dict2 = {}
print(type(dict2))
```

Accessing dictionary Items

- Dictionaries value can be accessed by their keys

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession': 'Programmer'}

print(dict1)


no = dict1['ID']
print(no)


age = dict1['Age']
print(age)


name = dict1['Name']
print(name)
```

Note: if the key is not available returns Error

```
#des = dict1.['Description']

#print(des)
```

Accessing values using get()

- **Dictionary elements also be accessed with get()**

  syn: get(“key”)

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession’: 'Programmer'}
print(dict1)

job2 = dict1.get('Profession')
print(job2)

des = dict1.get('Description')
print(des)
# Key
```

Dictionary Mutability Updating dictionary values

- **Dictionary is mutable**
  - **new items added** or **existing items can be changed**
  - If the key is already present, value gets updated, else {key: value} pair is added to the dictionary

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession': 'Programmer'}

# update value
dict1['Name'] = "Aditya"
dict1
```

```
# add item
dict1['Description'] = "Python Programming"
dict1
```

Updating dictionary values using update()

- **update() :** updates the dictionary with the elements from another dictionary object

  or

  from an iterable of key/value pairs.

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession':'Programmer'}
dict2 ={"Area":"Machine Learning"}
dict1.update(dict2)
print (dict1)
```

Deleting values from dictionaries using del

- **del** statement is used for performing deletion operation

  - Item can be deleted from a dictionary using the key

    Syntax: del [key]

  - Whole dictionary can be deleted using the **del** statement

Note: For deleting all items of dictionary

del dict1

Note: For deleting specific item using Key

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession': 'Programmer'}
del dict1['ID']
dict1
```

Deleting values from dictionaries using pop

- **pop:** removes as item with the provided key and returns the value

  - remove an item in a dictionary

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession’: 'Programmer'}

dict1.pop ('ID')

dict1
```

Deleting values from dictionaries using clear

- clear(): Remove all items form the dictionary.

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession':'Programmer'}

dict1.clear()
```

Dictionary Iteration

- **keys() :** displays a list of all the keys in the dictionary
- **values**() : Return dictionary's values
- **Items**(): Return (key, value) in tuple pairs

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession’: 'Programmer’}

print (dict1.keys())

print (dict1.values())

print (dict1.items())
```

Iterating dictionary elements using keys()

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession':'Programmer'}

for k in dict1.keys():

    print (k, dict1[k])
```

Iterating dictionary elements using items()

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession':'Programmer'}
for k,v in dict1.items():
    print (k, v)
```

Iterating dictionary elements using values()

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30,

'Profession':'Programmer'}

values = dict1.values()

values

                                    for value in dict1.values():

                                        print(value)
```

Dictionary len(), copy()

- **len**() : Return number of items in the dictionary

- **copy**() : Return a copy of the dictionary.

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30, 'Profession':'Programmer'}

print (len(dict1))
```

```
dict1 = {'ID': '100', 'Name': 'Shashank ', 'Age':30,

'Profession':'Programmer'}

dict2 = dict1.copy()

print(dict2)
```

fromkeys()

- **fromkeys**() : creates a **new dictionary** from the given **sequence** of elements

dict.fromkeys(*keys*, *value*)

Dictionary all(), any()

- **all**(): returns **True** if all keys of the dictionary are true

  - or if the dictionary is empty

- **any** () return **True** if any key of the dictionary is true.

  - If the dictionary is empty, returns “False”.

PYTHON PROGRAMMING

Dictionary Comprehensions

Dictionary comprehension is an elegant and concise way to create dictionaries.

Dr Venkat Raju

Dictionary Comprehension

- Dictionary comprehensions are used when the input is in the form of a dictionary

  or a **Key : Value pair**

  dict_variable = { key: value for var in iterables}

- is a powerful concept and can be used to substitute for **loops and lambda functions**.

Dictionary Comprehension

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

double_dict1 = {k:v*2 for (k,v) in dict1.items()}

print(double_dict1)
```

Dict Vs List

- Imagine that you are organizing a data science conference. You are making a list of attendees. Later you want to look up a name in this attendee list. How much time does it take to find a name if you store the data as a list, and as a dictionary?
- If 100 people are attending your conference, you don't have to think about lookup speed. You can keep your data in lists or dictionaries. You can even build an Excel table and use INDEX and MATCH keys to find the names you want.
- What if you are storing billions of names? Do you think it is a good idea to store too many elements in a list? Can dictionaries do a better job in finding a certain item in a collection of too many elements?

Dict Vs List

In general, accessing an element in a dictionary is faster than accessing an element in a list in Python, especially when the collection size is large.

The reason for this is that dictionaries use a hash table to store their data, which allows for constant-time access to elements. When a value is looked up in a dictionary by its key, Python computes a hash value for the key and uses it to find the corresponding value in the hash table quickly. On the other hand, in a list, Python must traverse the list sequentially to find the desired element, which takes time proportional to the size of the list.

Dict Vs List

- Notice how dictionaries are significantly faster, and how search runtime increases as input increases on our lists.
- This is because dictionaries in Python are implemented as hash tables, something we listed as a critical difference between these two data types in our previous section.
- In a nutshell, hash tables are made up of hash functions, which map data of any length to a fixed-length value (hashes).
- It doesn’t matter what your input is (10, 1000, or 10000); the result will always be a predetermined, fixed-length hash.
- Since the same string will always produce an exact hash, they’re incredibly fast to compute, especially compared to lists

Dict Vs List

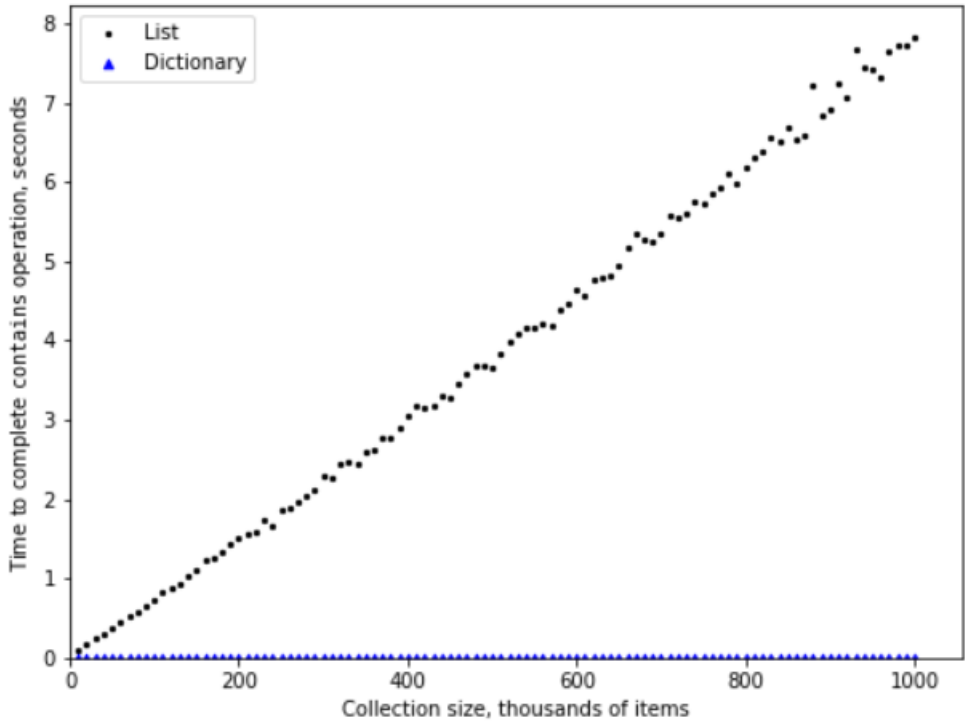

Figure 4: Comparing the `in` Operator for Python Lists and Dictionaries

https://runestone.academy/ns/books/published/pythonds3/AlgorithmAnalysis/Dictionaries.html