Chapter 7 - Dictionary & Sets

7 Dictionaries & Sets

7.1 Dictionaries

Dictionaries uses the key-value mapping structure to store data. A key-value mapping structure is where a **unique** *key* is used to associate with a *value*. This *key* is generally a string type but it can actually be any of the Python's immutable object types, such as integer, float, frozenset and many more. The characteristics of a dictionary are:

- Mutable
- Dynamic. Meaning that they can grow and shrink in size as needed
- Can be nested. Meaning that there can be 1 more more dictionaries in 1 dictionary. It can also contain Lists.

7.1.1 Creation

The syntax of a Dictionary is as follows.

```
# how a key-value pair looks like
d = {key: value}
```

Therefore to create a dictionary we can either use the curly brackets `{}` or the name of its object.

```
# empty dictionary
d = {}
d = dict()

# dictionary with elements
d = {'Name': 'John', 'MatrNo': '002558', 'Profession': 'Student'}

# converting a list/tuple to a dictionary
list_var = [ ('a', 'apple'), ('c', 'cat'), ('b', 'batman') ]
dict(list_var)
# output: {'a': 'apple', 'c': 'cat', 'b': 'batman'}
```

7.1.2 Accessing Keys or Values

There many ways to access either a key or a value or both key-value pairs from a dictionary.

- Accessing a particular value

```
d = {'a': 'apple', 'c': 'catwoman', 'b': 'batman', 'o': 'orange',
     'p': 'penguin'}

# method 1: using the square brackets [] operator
print(d['c'])

# method 2: using the dictionary function 'get'
# function 'get' has a 2nd argument where you can specify a return value
# if the element is not found in the dictionary
print(d.get('c', None))

# method 3: check for the key before using the 'get' function
# using the membership 'in' operator
```

```
if 'c' in d:
    print(d.get('c'))
```

- Accessing all keys. Using the `keys()` function returns an object called `dict_keys()` is an iterable view object of all the keys in the dictionary. This is especially useful for large dictionaries where there is memory constrains on the system. If there is a need to work with the dictionary keys, convert the `dict_keys()` object into a list.

```
# converting the returned dict_keys object to a list
list(d.keys())
```

- Accessing all values. Similar to the `keys()` function, the `values()` function returns a `dict_values()` object and it can be converted to a list for processing.

```
# converting the returned dict_values object to a list
list(d.values())
```

- Accessing key-value pairs. Similar to the functions before, the `items()` function returns `dict_items()` object and it can be converted to a list for processing.

```
# converting the returned dict_items object to a list
list(d.items())
```

- Getting the length of a dictionary. Unlike the other iterable objects where the `len()` returns the number of elements, for dictionaries, `len()` returns the number of *key-values* pairs of a dictionary.

```
len(d)  # output: 5
```

7.1.3 Adding or Changing Items

Adding or changing items in a dictionary is done via keys.

- Adding new key-value pairs

```
# original dictionary
d = {'a': 'apple', 'c': 'catwoman', 'b': 'batman', 'o': 'orange',
     'p': 'penguin'}
# adding a new key-value pair
d['t'] = 'two-face'
```

- Changing a value of a key

```
d['b'] = 'bane'
```

7.1.4 Combining Dictionaries

We can combine 2 dictionaries using the `update()` dictionary function. This function does not return any values but does the combining then replaces the dictionary object that called it. Beware that if there are any duplicated keys in both dictionaries, the values in the second dictionary wins.

```
d1 = {'a': 'apple', 'c': 'catwoman', 'b': 'batman', 'o': 'orange',
     'p': 'penguin'}
d2 = {'a': 'apocalypse', 'p': 'phoenix', 'w': 'wolverine'}

d1.update(d2)
print(d1)
# output: {'a': 'apocalypse', 'c': 'catwoman', 'b': 'batman', 'o': 'orange',
#          'p': 'phoenix', 'w': 'wolverine'}
```

7.1.5 Iterating

Iterating a dictionary is done by using a `for` loop and there are 3 ways to do it with each showing a different result.

- iterating over the values

```
for val in d1.values():
    print(val)
```

- iterating over the keys

```
for k in d1.keys():
    print(k)
```

- iterating over the items

```
for k, v in d1.items():
    print(f"key: {k}, value: {v}")
```

7.1.6 Sorting

As it was mention in the chapter on lists and tuples that the `sorted()` function is able to sort any Python iterable object, it must also be able to accept a custom function that returns a key such that the `sorted` function can be used to sort the object.

```
d = {'o': 'orange', 'c': 'catwoman', 'a': 'apocalypse', 'p': 'phoenix',
     'b': 'batman', 'w': 'wolverine'}

def sort_via_values(item):
    return item[1]

sorted(d.items(), key=sort_via_values)
# output: [('a', 'apocalypse'),
#          ('b', 'batman'),
#          ('c', 'catwoman'),
#          ('o', 'orange'),
#          ('p', 'phoenix'),
#          ('w', 'wolverine')]
```

7.1.7 Deleting

Deleting or removing key-value items from a dictionary can be done using several ways. Most delete or removal functions/statement does their operations in place.

- using the `del()` built-in function

  ```
  d1 = {'a': 'apple', 'c': 'catwoman', 'b': 'batman', 'o': 'orange',
       'p': 'penguin'}
  del d1['a']
  ```

- using the `pop(<key>, <default>)` function returns the popped value corresponding to the *key*. This requires a *key* to be given. If the *key* is not found, an error is raised. To prevent an error being raised, a *default* value can be provided.

  ```
  d1.pop('p')  # output: penguin
  d1.pop('w', None) # output: None
  ```

- Clearing the whole dictionary

  ```
  # using the 'clear' function
  d1.clear()
  # reassign an empty dictionary
  d1 = {}
  ```

7.2 Sets

You may recall Sets and Set Theory from math in school. The ones using the Venn Diagrams. Refer to figure 1 below.

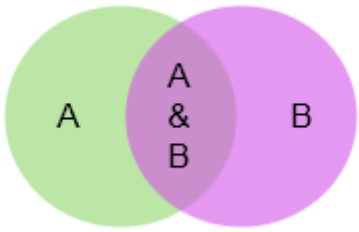

Figure 1: Venn Diagram.

In Python, think of Sets as dictionary but with out the *values* thus only the *keys*. Therefore each value in a set has to be **unique**. A set is also **unordered** and the elements in it **must consist** of immutable objects.

7.2.1 Creation

As all the brackets have been used up for creation of lists, tuples and dictionaries, to create a set we have to use its object name

```
# creation from a single string
set('run')  # output: {'n', 'r', 'u'}

# creation from list
set(['rogue', 'phoenix', 'gambit', 'storm', 'galactus'])
# output: {'galactus', 'gambit', 'phoenix', 'rogue', 'storm'}

# creation from tuple
set(('rogue', 'phoenix', 'gambit', 'storm', 'galactus'))
# output: {'galactus', 'gambit', 'phoenix', 'rogue', 'storm'}

# creation from dictionary, it only uses the keys
set({'c': 'catwoman', 'b': 'batman', 'o': 'orange', 'p': 'penguin'})
# output: {'b', 'c', 'o', 'p'}
```

7.2.2 Manipulating and Iterating

Manipulating and iterating through set is similar to list and dictionary in the sense that they uses object functions.

- adding an element to a set

```
s = set(('rogue', 'phoenix', 'gambit', 'storm', 'galactus'))
s.add('angel')
```

- checking the length of a set object

```
len(s)
```

- checking if an item is in a set object

```
'gambit' in s  # output: True
```

- removing/clearing an element from the set

```
# this function will raise an error if the element is not found
s.remove('phoenix')

# this function will do nothing if the element is not found
s.discard('phoenix')

# this function will remove any element from the set, it will raise an
# error if the set is empty
s.pop()

# clear all elements in the set
s.clear()
```

- iterating a set

```
for elem in s:
    print(elem)
```

7.2.3 Operators

Since most of the operations are similar to iterable objects, what is so special about the `Set` object? It is the operators! imagine that you have a dictionary of bank loans with the keys containing the type of loan and the values containing a `set` of loan requirements.

```
loans = {
    'property': {'50000 min income', '5% per annum', '30 yrs'},
    'study': {'0 min income', '3% per annum', '60 yrs'},
    'personal': {'10000 min income', '10% per annum', '10 yrs'},
    'fixed': {'10000 min income', '3% per annum', '5 yrs'},
    'debt': {'0 min income', '5% per annum', '60 yrs'},
}
```

Let's extract out the requirements for a study and fixed loan

```
study = loans['study']
fixed = loans['fixed']
```

A young person comes to the bank and needs help with choosing a loan for their future studies. So the young person wants to know what is the similarities between a fixed and study loan. Using sets we can find the *intersection* of the 2 loans.

```
study.intersection(fixed)
# output: {'3% per annum'}
```

Both loans has an interest rate of 3% per annum. Let's say that the bank has ran out of pamphlets for the details of the study loan thus this young person was given a pamphlet of the fixed loan thus his next question is what is the difference between the study and fixed loan?

```
study.difference(fixed)
# output: {'0 min income', '60 yrs'}
```

Wow, the study loan has a 60 yrs repayment period. Let's view both loans data together.

```
study.union(fixed)
# output: {'0 min income', '10000 min income', '3% per annum', '5 yrs', '60 yrs'}
```

Viewing the loans together can be cumbersome if we want to see only the differences between the 2 loans and not a mixed of similarities and differences.

```
study.symmetric_difference(fixed)
# output: {'0 min income', '10000 min income', '5 yrs', '60 yrs'}
```

Now we are able to see the just the differences only. There are equivalent operators for those functions (refer to the table below).

Function Name	Equivalent Operator	Example
union	| (pipe symbol)	study | fixed
intersection	& (ampersand symbol)	study & fixed
difference	- (subtract sign)	study - fixed
symmetric_difference	^ (caret symbol)	study ^ fixed

7.2.4 Frozenset

There is an immutable version of a set called `Frozenset`. Frozensets are used when there is a need to create an immutable set. No modifying operations can be done on frozensets.

```
# Frozensets
a1 = frozenset(set(['foo']))
a2 = frozenset(['bar'])
a = {a1, a2}

# output: {frozenset({'bar'}), frozenset({'foo'})}
```

7.3 References

1. Built-in Functions, https://docs.python.org/3/library/functions.html#sorted
2. Mapping Types — dict, https://docs.python.org/3/library/stdtypes.html#mapping-types-dict
3. Set Types — set, frozenset, https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
4. Lubanovic, 2019, 'Chapter 8. Dictionaries and Sets' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.