Lecture 4.1 : Dictionaries

Introduction

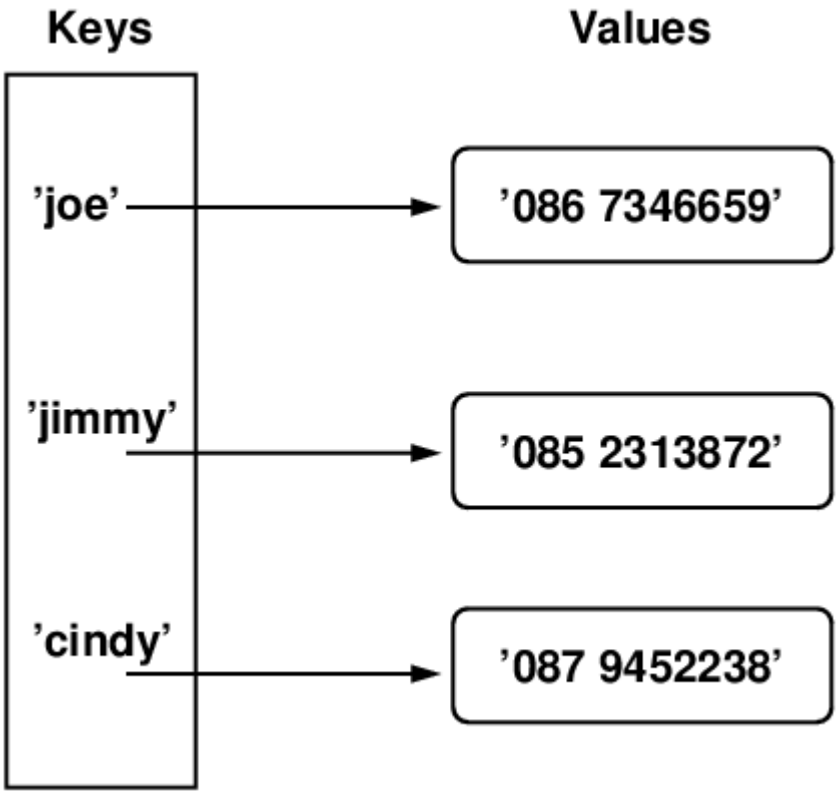
- So far we have met lists, strings and tuples. Each of these is an example of a *data structure*. Here we examine another built-in Python data structure: the *dictionary*. As we will see dictionaries are an extremely useful and powerful data structure. Knowing when and how to use them will make you a better programmer.

Dictionaries

- A dictionary is a *collection type* but it is **not** a *sequence type*. That is, its elements are not ordered as they are in a list, string or tuple. A dictionary is also sometimes referred to as a *map*, *hashmap*, or *associative array*.
- We can think of a dictionary as a collection of pairs of objects. One element in the pair is the *key* and the other is the *value*. A dictionary thus implements a *mapping* from keys to values. When we use a real world dictionary to look up the meaning of a word, the *word* is the *key* and the *meaning of the word* is the *value*.
- A dictionary is designed such that given a *key*, retrieving the associated *value* is a highly efficient operation. We do not know the order in which key-value pairs are stored in a Python dictionary. That information is hidden from us and we should not write programs that rely on it.

Dictionary example

- We can use a dictionary to implement a simple phone book. A phone book is a mapping from names to phone numbers. The dictionary keys are thus names and the dictionary values are phone numbers. Once built, the dictionary can be depicted as follows:



- To find Cindy's phone number we look up the key `'cindy'` in the dictionary. That leads us to the value `'087 9452238'`.

Building dictionaries

- While we use square brackets to create a list, we use curly brackets to create a dictionary. Key-value pairs are separated by a colon. Keys can only be an immutable type e.g. strings or integers but values can be of any type. To create the dictionary in the above example we would write:

```
>>> phone_book = {'joe' : '086 7346659', 'jimmy' : '085 2313872', 'cindy' : '08
>>> type(phone_book)
<class 'dict'>
>>> phone_book
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087 9452238'}
```

Dictionary indexing

- Dictionaries are indexed by keys:

```
>>> phone_book['cindy']
'087 9452238'
>>> phone_book['jimmy']
'085 2313872'
```

- It is an error to index a dictionary with a non-existent key. Specifically, a `KeyError` exception is thrown:

```
>>> phone_book['sally']
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
KeyError: 'sally'
```

- Note dictionaries are *not* sequenced and cannot be indexed by position (as immutable types integers can serve as keys but even then we are indexing by key and not by position):

```
>>> phone_book[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Dictionary assignment

- To add an additional mapping to an existing dictionary we use square brackets to index by the new key and then assign the new value (note how *where* a new entry goes in the dictionary is not predictable):

```
>>> phone_book['louie'] = '087 6551201'
>>> phone_book
{'joe': '086 7346659', 'louie': '087 6551201', 'jimmy': '085 2313872', 'cindy':
```

- If the key is already present in the dictionary its value is updated:

```
>>> phone_book['joe'] = '086 3746659'
>>> phone_book
{'joe': '086 3746659', 'louie': '087 6551201', 'jimmy': '085 2313872', 'cindy':
```

- Note how after a dictionary has been created we can both make changes to the values it contains and continue to add new key-value mappings. Clearly a dictionary is a *mutable* type.

Different key types

- Keys can be any immutable type. Values can be any type (even dictionaries). Here is another dictionary where keys are strings and integers and values are tuples and lists.

```
>>> another_dict = {'jimmy' : ('Dublin', 'Ireland'), 23 : ['a', 'b', 12]}
>>> another_dict['jimmy']
('Dublin', 'Ireland')
>>> another_dict['jimmy'][0]
'Dublin'
>>> another_dict['jimmy'][1]
'Ireland'
>>> another_dict[23]
['a', 'b', 12]
>>> another_dict[23][1]
'b'
```

Dictionary operators

- The `len()` function returns the number of key-value pairs in a dictionary:

```
>>> another_dict = {'jimmy' : ('Dublin', 'Ireland'), 23 : ['a', 'b', 12]}
>>> len(another_dict)
2
```

- The `in` operator can be used to test whether a particular *key* (not *value*) is in a dictionary (we can thus use `in` to avoid indexing on non-existent keys which would cause a `KeyError` exception to be thrown):

```
>>> if 'jimmy' in another_dict:
...     print(another_dict['jimmy'])
...
('Dublin', 'Ireland')
>>> if 'fred' in another_dict:
...     print(another_dict['fred'])
... else:
...     print('No such key.')
...
No such key.
```

- Dictionaries are *iterable*. We can use a `for` loop to cycle through each of a dictionary's keys (and from each key we can map to the corresponding value if required):

```
>>> for k in another_dict:
...     print('Key: {} and value: {}'.format(k, another_dict[k]))
...
Key: jimmy and value: ('Dublin', 'Ireland')
Key: 23 and value: ['a', 'b', 12]
```

Dictionary methods

- We can retrieve a list of all of a dictionary's keys using the `keys()` method.
- We can retrieve a list of all of a dictionary's values using the `values()` method.

```
>>> for k in another_dict.keys(): print(k)
...
jimmy
23
>>> for v in another_dict.values(): print(v)
...
('Dublin', 'Ireland')
['a', 'b', 12]
```

- Keys and values returned by the `keys()` and `values()` methods are in corresponding order.
- We can retrieve a list of tuples of key-value pairs from a dictionary using the `items()` method. We can then use a `for` loop over the items that uses multiple assignment to handily access every key and corresponding value:

```
>>> another_dict.items()
dict_items([('jimmy', ('Dublin', 'Ireland')), (23, ['a', 'b', 12])])
```

```
>>> for (k, v) in another_dict.items():
...     print('Key: {} and value: {}'.format(k, v))
...
Key: jimmy and value: ('Dublin', 'Ireland')
Key: 23 and value: ['a', 'b', 12]
```

- There are more dictionary methods. You can look them up using `help` or the `pydoc` command.

Sorting dictionary items on values

- Suppose we want to print the contents of a dictionary sorted not on keys but on values. How can we do that? Let's try it:

```
>>> d = {'dogs' : 12, 'cats' : 20, 'ponies' : 15}
>>> for (k,v) in sorted(d.items()):
...     print('{} : {}'.format(k, v))
...
cats : 20
dogs : 12
ponies : 15
```

- Hmm. That is not what we required. `d.items()` is the list of tuples: `[('cats', 20), ('ponies', 15), ('dogs', 12)]`. If we pass this list to the `sorted()` function it will sort the list's tuples on the first item in each tuple but that is not what we want. We wish to sort on the second item in each tuple. How can we do that?

- We can sort on an arbitrary data member of an object by specifying a custom `key` function when we invoke the `sorted()` function. It is the job of this `key` function to return the item we wish to sort on. In the above example we wish to sort on the second item in a tuple so our `key` function should be defined as follows:

```
>>> def sorter(t):
...     return t[1]
```

- To have the `sorted()` function sort dictionary items on our new `key` we simply pass our new `sorter()` function as an argument when we invoke `sorted()` as follows:

```
>>> for (k,v) in sorted(d.items(), key=sorter):
...     print('{} : {}'.format(k, v))
...
dogs : 12
ponies : 15
cats : 20
```

- Each tuple in the list `[('cats', 20), ('ponies', 15), ('dogs', 12)]` is successively passed to the `sorter()` function which will return the integers `20`, `15` and `12` which are the keys we wish to sort on.

- Should we wish sort from highest to lowest instead then we pass another argument, `reverse=True`, to the `sorted()` function:

```
>>> for (k,v) in sorted(d.items(), key=sorter, reverse=True):
...     print('{} : {}'.format(k, v))
...
cats : 20
ponies : 15
dogs : 12
```

Tabulating dictionary keys and values

- Suppose we want to both sort and neatly print dictionary keys and corresponding values. How can we do that?
- Firstly we need to work out the width of the widest value in our dictionary. We can do so as follows:

```
>>> d = {'dogs' : 12, 'cats' : 20, 'ponies' : 15, 'aardvarks' : 100}
>>> max(d.values())
100
>>> len(str(max(d.values())))
3
>>> max_v_width = len(str(max(d.values())))
```

- Next we need to work out the width of the widest key in our dictionary. We can do so as follows:

```
>>> max(d.keys())
'ponies'
>>> max(d.keys(), key=len)
'aardvarks'
>>> len(max(d.keys(), key=len))
9
>>> max_k_width = len(max(d.keys(), key=len))
```

- We now format our output using the above widths:

```
>>> def sorter(t):
...     return(t[1])
...
>>> for (k,v) in sorted(d.items(), key=sorter, reverse=True):
...     print('{:>{}s} : {:{}d}'.format(k, max_k_width, v, max_v_width))
...
aardvarks : 100
     cats :  20
   ponies :  15
     dogs :  12
```