Lecture 9 - Dictionaries

Computer Programming

Robert Vincent and Samia Hilal

Marianopolis College

March 29, 2022

Dictionaries

- Dictionaries are another mutable, iterable type.
- Similar to lists, but not limited to integer indices.
- Provide a general mapping from one set of values (aka keys) to another set of values.

```
Example: d={'x': 2, 'y': 1, 'z': 5}
```

- Each *key* maps to exactly one value.
- A key can be of different types. Details later.
- The Python type is called `dict`.
- Similar types exist in many programming languages.

Python dictionaries

- Dictionaries are written as a comma-separated list of *items* enclosed within curly braces:
- Each item is a *key:value* pair.

```
x = {} # Empty dictionary
y = {'apple':12, 'banana':3, 'peach':1}
```

- You access values by using the key as an index:

```
>>> print(y['apple'])
12
>>> print(y['banana'])
3
```

Constructing dictionaries

- There are different ways to create a dictionary.
- The `dict` constructor expects an iterable sequence of key-value pairs **or** named arguments. Think of a key as a variable name.

```
>>> x = dict() # Empty dictionary
>>> print(x)
{}
>>> y = dict([('x', 2), ('y', 1), ('z', 5)])
>>> print(y)
{'x': 2, 'y': 1, 'z': 5}
>>> z = dict(x=2, y=1, z=5)
>>> print(z)
{'x': 2, 'y': 1, 'z': 5}
```

The zip() function

- A builtin Python function.
- Very Useful in construction of dictionaries.
- `zip()` takes 2 or more iterable arguments.
- It returns a single iterable that combines the arguments.
- Each element in the returned iterable is a tuple consisting of one item taken from each of the original iterable objects.
- The overall length of the result is the same as the *shortest* of the iterable arguments.
- It "zips" lists together like a zipper.

zip() Example 1

```
>>> x = range(10, 13)
>>> y = "abc"
>>> for z in zip(x, y):
        print(z)
(10, 'a')
(11, 'b')
(12, 'c')
```

Example 2: Ignores extra items on one of the lists.

```
>>> x = [5, 4, 3, 8]
>>> y = ["broccoli", "pepper", "tomato"]
>>> for a, b in zip(x, y):
        print(a, b)
5 broccoli
4 pepper
3 tomato
```

zip() Example 3

```
>>> a = (1, 2, 3)
>>> b = [5, 3, 1]
>>> c = [10, 11, 12]
>>> for v in zip(a, b, c):
...     print(v)
...
(1, 5, 10)
(2, 3, 11)
(3, 1, 12)
>>>
```

- Works with any number of iterable arguments.

Using zip() to construct a dictionary

- `zip()` can be used to convert 2 iterables into a dictionary.

```
keys = ['red', 'green', 'blue']
values = (2, 1, 3)
dictionary = dict(zip(keys, values))
print(dictionary)
```

```
{'red': 2, 'green': 1, 'blue': 3}
```

Adding and Changing items in A Dictionary

- It's easy to add an item:

```
>>> y = {'apple':12, 'banana':3,
         'peach':1}
>>> y['apricot'] = 7 # New item.
>>> print(y['apricot'])
7
```

- Values can be changed:

```
>>> y['peach'] += 4  # Change item.
>>> print(y['peach'])
5
```

Iterating over a `dict`

- A `dict` is iterable.
- Normally, the loop will iterate over the keys.

```
>>> y = {'a' : 5, 'b' : 3, 'c' : 3}
>>> for x in y:
...     print(x)
...
a
b
c
```

Iterating, take two:

- If we have the keys, we can access the values:

```
>>> y = {'a' : 5, 'b' : 3, 'c' : 3}
>>> for key in y:
...     print(key, y[key])
a 5
b 3
c 3
```

- Compare to iterating over a list x

```
>>> x = [5, 3, 3]
>>> for i in range(len(x)): # index gives
...     print(i,x[i]) # access to value
>>> 0 5
>>> 1 3
>>> 2 3
```

Accessing nonexistent items

- Reading a nonexistent item will raise an exception:

```
>>> x = {}                  # empty
>>> x['apple'] = 1          # add 1 item.
>>> print(x['apple'])
1
>>> print(x['grape'])
KeyError:  'grape'
```

- The `in` operator checks if a key is present:

```
>>> print('grape' in x)
False
>>> print('apple' in x)
True
```

Why Dictionaries are so useful

- For a finite set of keys, we can represent any function $y = f(x)$ where x is the key and y is the value.
- Useful for "sparse" data where not every key has a value.
- Store position or frequency of words, values, etc.
- Associate a record or object with a particular key for easy access.
- Lookup tables: e.g. translate one set of strings to another.

Dictionary methods

- `clear()` - Remove all items.
- `copy()` - Copy a `dict`.
- `fromkeys()` - Build from a key list and a value.
- `get()` - Access a value.
- `items()` - Get items as an iterable sequence.
- `keys()` - Get keys as an iterable sequence.
- `values()` - Get values as an iterable sequence.
- `pop()` - Removes a key and returns its value.
- `popitem()` - Removes a (key, value) tuple.
- `setdefault()` - Sets a missing key to a value.
- `update()` - Append items to a `dict`.

Method: fromkeys()

- Create a *new* dictionary from a list of keys and a default value.
- First argument is any iterable, used for the keys.
- Second argument is the default value. `None` is used if no value is provided.

```
>>> keys_list = [10.5, 15.0, 14.2]
>>> y = dict.fromkeys(keys_list)
>>> print(y)
{10.5: None, 14.2: None, 15.0: None}
>>> x = dict.fromkeys(["a", "b", "c"], 0)
>>> print(x)
{'a': 0, 'b': 0, 'c': 0}
```

Method: get(key, val)

- Returns the value of item with the specified key.
- Alternative to indexing with square brackets.
- Does not raise `KeyError` for missing keys.
- The second argument *val* is optional. It specifies the default value to be returned if the given key is not present in the dictionary.

```
>>> d = {}
>>> print(d.get('apricot'))
None
>>> print(d.get('plum', 5)) # default value
5
```

Method: items()

- Returns an iterable sequence of key-value pairs as tuples.

```
>>> d = {'cat': 5,'rat': 4,'dog': 2}
>>> print(list(d.items()))
[('cat',5),('rat',4),('dog',2)]
...
>>> for key, val in d.items():
...     print(key, val)

cat 5
rat 4
dog 2
```

Method: keys()

- Returns an iterable sequence of keys.
- The normal behavior to iterate over dictionaries in `for` loops.

```
>>> d = {'cat': 5,'rat': 4,'dog': 2}
>>> print(list(d.keys()))
['cat', 'rat', 'dog']
...
>>> for key in d.keys():
...     print(key)
...
cat
rat
dog
```

Method: values()

- Returns an iterable sequence of values.

```
>>> d = {'cat': 5,'rat': 4,'dog': 2}
>>> print(list(d.values())) # returns a list
[5, 4, 2]
...
>>> for val in d.values(): # values in list
...     print(val)
...
5
4
2
```

Method: pop(key)

- Removes a key and returns its value.

```
>>> d = {'cat': 5,'rat': 4,'dog': 2}
>>> d.pop('bat') # Exception here.
KeyError:  'bat'
>>> d.pop('bat', 0) # OK, default given.
0
>>> d.pop('rat', 0)  # Remove this item.
4
>>> print(d)
{'cat': 5, 'dog': 2}
>>> d.pop('dog')
2
>>> print(d)
{'cat': 5}
```

Method: popitem()

- Removes the last item inserted into the dictionary. In versions before 3.7, the popitem() method removes a *random* item. The item is returned as a tuple: (key, value).

```
>>> d = {'cat': 5,'rat': 4,'dog': 2}
>>> d.popitem()
('dog', 2)
>>> d
{'cat': 5, 'rat': 4}
>>> d.popitem()
('rat', 4)
>>> d.popitem()
('cat', 5)
>>> d.popitem()
KeyError: 'popitem(): dictionary is empty'
```

Method: setdefault(key,default_value)

- Similar to `get()`, *except* if key is missing, it creates the key at the specified default_value.

```
>>> d = {'cat': 5,'rat': 4,'dog': 2}
>>> d.setdefault('rat', 0) # already present
4
>>> d.setdefault('fox', 7) # add to dictionary
7
>>> print(d)
{'fox': 7, 'cat': 5,'rat': 4,'dog': 2}
>>>
```

Method: update()

- Update several items from another `dict` or a `list` of tuples.

```
>>> d = {'cat': 5,'rat': 4,'dog': 2}
>>> d.update({'rat':7,'cat':3}) #from dict
>>> print(d)
{'cat': 3,'rat': 7,'dog': 2}
>>> g={'bat':1}
>>> d.update(g) #from another dict
>>> print(d)
{'cat': 3,'rat': 7,'dog': 2, 'bat': 1}
>>> d.update([('rat',1),('fox',6)])#from list
>>> print(d)
{'cat':3, 'rat':1, 'dog': 2, 'bat':1,'fox':6}
```

More details about dictionaries

- An empty `dict` is `False`.

```
>>> print(bool({}))
False
```

- Use the sorted function to sort the list obtained using any of the dictionary methods: `items()`, `keys()`, or `values()`.
- Any *hashable* type can be used as a key. This includes `int()`, `str()`, and `float()`.
- Immutable data types are hashable and implement the `__hash__()` method.

Sorting a Dictionary??

- `It is not possible` to sort a dictionary. However, we can get a sorted representation or create a new sorted dictionary from original.
- As of python 3.7, dictionaries remember the order of items inserted. So when items are inserted starting from a sorted list of keys or values, the dictionary will keep the order.
- The sorted() function can be used to get a sorted list of keys that can be used to create a new sorted dictionary
- Similarly, a new sorted dictionary can be created based on the sorted values of original.

Sorting & Ordering Examples

```
>>> d={"a":4,"c":8,"z":5,"d":8}
>>> max(d) #returns 'z'
>>> min(d) #returns 'a'
>>> max(d,key=d.get) #key with max value
'c'
>>> sorted(d) #sorted keys
['a','c','d','z']
>>> sorted(d,key=d.get) #keys based on values
['a','z','c','d']
>>> for k in sorted(d):#print in sorted order
        print(k, d[k])
a 4
c 8
d 8
z 5
```

How Does Hashing work

- A *hash table* is a smart way to store & retrieve.
- *Hashing* an item creates an integer "address" that is used to choose its index in the list.
- Most immutable types are (or can be) hashable.
- Dictionary values can be any type.
- Mutable types cannot be used for keys.
- A hash function maps a complex value (such as a string) to a specific integer:

```
>>> print(hash('apple'))
5584453937065830616
>>> print(hash('Apple'))
-5334457415067556667
```

Example application: Counting Words

```
x = 'To be or not to be'
y = {}
for w in x.lower().split():
  if w in y:
    y[w] += 1 # key exists, increment value
  else:
    y[w] = 1 # create key, start at 1
```

After completion:

```
>>> print(y)
{'to': 2, 'or': 1, 'be': 2, 'not': 1}
```

Counting words with `setdefault()`

Remember:

If a key does not exits, `setdefault()` creates the key at the given default value. If key already exists, nothing happens.

```
x = 'To be or not to be'
y = {}
for w in x.lower().split():
    y.setdefault(w, 0) #create key if missing
    y[w] += 1
```

After executing:

```
>>> print(y)
{'to': 2, 'or': 1, 'be': 2, 'not': 1}
```

Counting words with get()

Remember:

If a key exists, `get(key, def_val)` returns the actual key value. If key does not exist, get returns the def_val specified.

```
x = 'To be or not to be'
y = {}
for w in x.lower().split():
# if key not there, create with value=0
  y[w] = y.get(w, 0) + 1
```

After executing:

```
>>> print(y)
{'to': 2, 'or': 1, 'be': 2, 'not': 1}
```

Example: Building a dictionary of lists

```
def word_positions(text):
    '''return a dict: words as keys, value is
    list of indices where a word is found.'''
    result, start = {}, 0
    text = text.lower() # Ignore case.
    for word in text.split():
        wpos = text.index(word, start)
        result.setdefault(word, [])
        result[word].append(wpos)
        start = wpos + len(word)
    return result
```

```
>>> x = "To be or not to be"
>>> print(word_positions(x))
{'or': [6], 'not': [9], 'be': [3, 16],
 'to': [0, 13]}
```

Summary

- Dictionaries in Python associate a set of keys (hashable objects) with a set of values.
- Each key is associated with exactly one value.
- Keys can be any immutable, hashable type.
- The value can be any type, even mutable.
- We cannot sort a dictionary but we can create a new sorted dictionary from original.
- Know how to construct and use dictionaries.
- We can construct a dictionary of lists.
- Know the methods `clear()`, `copy()`, `get()`, `items()`, `values()`, `setdefault()`, and `keys()`.