15-110 Principles of Computing – F21

Lecture 18: Dictionaries 2

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Useful operations on key and value sets: `sorted()`, `sort()`

```
numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
```

- Get the **sorted list of keys** from the dictionary items:

```
sorted_keys = sorted(numbers) → [1, 2, 3, 4, 5, 6]
sorted_keys = sorted(numbers.keys())
```

- Get the **sorted list of values** from the dictionary items:

```
sorted_values = sorted(numbers.values()) →  ['p', 'p', 'p', 'p', 'r', 'r']
```

- Get the **sorted list of keys, paired with their associated values** :

```
sorted_dict_list = sorted(numbers.items())
                 → [(1, 'p'), (2, 'p'), (3, 'p'), (4, 'r'), (5, 'p'), (6, 'r')]
```

Useful operations on key and value sets: `min()`, `max()`, `sum()`

- Find **min / max** of key/values from the dictionary items:

```
max_key_val = max(numbers)          → 6
min_key_val = min(numbers)          → 1
max_key_val = max(numbers.keys())       → 6
min_key_val = min(numbers.keys())       → 1

max_values = max(numbers.values())    → r
min_values = min(numbers.values())    → p
```

- Find **sum** of key/values from the dictionary items:

```
key_sum = sum(numbers) → 34
key_sum = sum(numbers.keys()) → 34
values_sum = sum(numbers.values()) → Error, sum not defined over strings!
```

Useful operations on key and value sets

- Watch out! The `sorted()` function and the `sort()` method could have been used without a comparison function given that in these example all keys / values are homogeneous (`int` or `str`) and python knows how to perform comparisons among these homogeneous data types

- In the general case, the use of sort functions/methods might require the additional definition of a comparison function, based on the characteristics of the keys / values to sort

- This applies also to `min()`, `max()`, `sum()`

Creation of dictionary variables: empty dictionaries

- Empty dictionary:

```
v = {}
```

- Empty dictionary:

```
v = dict()
```

Creation of dictionary variables: use a list of tuples

- Use a **list of tuples** and the built-in function `dict(key_val_list)` to build a dictionary directly from *sequences* of input (key-value) pairs:

```
word_list = [ ('Hello', 5), ('this', 4), ('is', 2), ('a', 1), ('list', 4) ]
word_dict = dict(word_list)

parabola = dict( [(0,0), (0.5, 0.25), (1,1), (1.5, 2.25)] )

Accounts_by_country = dict( [ ('USA', [35672, 'M', 'J. Smith']),
                              ('Jordan', [27623, 'M', 'M. Saleh']),
                              ('France', [17623, 'F', 'F. Dupont']) ] )
```

Creation of dictionary variables: use list of keys with default values

- Use **a list of keys** and assign a common (optional) value to the keys by using the method `fromkeys(key_list, <value>)`

```
list_of_words = ["This", "is", "a", "list", "of", "key", "strings"]
dict_of_words = dict.fromkeys(list_of_words, 0)
```

```
{'This': 0, 'is': 0, 'a': 0, 'list': 0, 'of': 0, 'key': 0, 'strings': 0}
```

```
primes = [2, 3, 5, 7, 11, 13]
dict_of_primes = dict.fromkeys(primes, 'p')
```

```
{2: 'p', 3: 'p', 5: 'p', 7: 'p', 11: 'p', 13: 'p'}
```

Creation of dictionary variables: use of two lists

- Use **two lists of the same length**, one containing the keys and the other the values, and pair them using the function `zip(key_list, value_list)`

```
list_of_keys = [4, 1, 3, 2, 6, 5]

list_of_values = ['r', 'r', 'p', 'p', 'r', 'p' ]

numbers = dict(zip(list_of_keys, list_of_values))
```

```
list( zip(list_of_keys, list_of_values)) ) →

                    [(1, 'r'), (2, 'p'), (3, 'p'), (4, 'r'), (5, 'p'), (6, 'r')]
```

Practice

Implement the function `count(L)` that takes a list and returns a dictionary where the keys are elements of the list and the values are the number of times that element occurred in the list.

For example, count(['a','b','b','a','c','b'])
should return the dictionary: {'a': 2, 'b': 3, 'c': 1}.

```
def count(L):
    d = {}
    for e in L:
        if e in d:
            d[e] += 1
        else:
            d[e] = 1
    return d
```

```
def count(L):
    d = dict.fromkeys(L, 0)
    for k in d:
        d[k] = L.count(k)
    return d
```

Practice

Implement the function `get_middle(d)` that takes a dictionary and returns value of the middle key (if the dictionary was sorted).

For example, `get_middle({'b': 5, 'a': 3, 'c': 1})` should return 5.

```
def get_middle(d):
    items = d.items()
    items = sorted(items)
    middle = items[len(items)//2]
    return middle[1]
```

```
def get_middle(d):
    sorted_keys = sorted(d)
    middle_key = sorted_keys[ len(sorted_keys) // 2]
    return d[middle_key]
```

Practice with a given dictionary as input

Implement the function `compute_avg(d)` that takes a dictionary where keys are strings (e.g., student names) and values are lists of numbers (e.g., student grades), and returns another dictionary where each key (e.g., student) is associated with its average value (e.g., the average grade), written with two decimal digits. The input dictionary must be unchanged.

The function also prints out a multi-line string that reports about maximum, minimum, and median values of the average (e.g., of student grades).

```
d = {'beth': [7.0, 9.6, 8.5, 10.0],
     'jerry': [6.0, 5.4, 3.8, 10.0],
     'morty': [8.0, 7.5, 10.0, 9.0, 7.6],
     'rick': [10.0, 10.0, 10.0, 9.7, 8.7],
     'summer': [10.0, 9.5, 8.5, 5.0, 7.2, 8.0] }
```

```
Max avg grade: 9.68
Min avg grade: 6.3
Median avg: 8.78

{'rick': 9.68, 'morty': 8.42, 'beth': 8.78, 'jerry': 6.3, 'summer': 8.03}
```

Practice with a given dictionary as input

```
def compute_avg(d):
    avg_d = {}
    for k in d:
        grades = d[k]
        avg = round(sum(grades) / len(grades), 2)
        avg_d[k] = avg

    v = list(avg_d.values())
    max_v = max(v)
    min_v = min(v)
    median_v = sorted(v)[len(v)//2]
    print('Max avg grade:', max_v, '\nMin avg grade:', min_v,
          '\nMedian avg:', median_v )
    return avg_d
```

Practice

A dog may be categorized for the breed based on weight (in grams), height (in cm), and width (in cm). We want to build a few data structures for implementing a dog classifier. At this aim we need to associate triples of numeric attributes for weight, height, and width to a string label that represents the corresponding breed. For instance, the triple 107, 95, 134 could be associated to the breed with label ``poodle''.

Implement the function `classifier(data)` that takes as input a list `data` of quadruples where the first three elements of each quadruple are the three integer attributes above (weight, height, width) and the fourth is the string label with the breed / category.

The function uses the input data for creating a dictionary `breeds_dict` that maps a dog breed to a triple of numeric attributes representing the available measures of weight, height, and width for that breed.

```
 dogs = [ [1230, 35, 72, 'poodle'], [1710, 72, 35, 'beagle'],
          [27110, 123, 78, 'labrador'], [11710, 78, 123, 'setter'],
          [9720, 112, 78, 'bulldog'], [9759, 108, 72, 'bulldog'] ]
```

```
{ 'poodle': (230, 35, 72),  'beagle': (1710, 72, 35),  'labrador': (27110, 123, 78),
 'setter': (11710, 78, 123),  'bulldog': (9720, 112, 78), (9759, 108, 72) }
```

(Optional) Methods for accessing and modifying a dictionary: `.pop()`

```
numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
```

- Remove and return dictionary element associated to passed key: `dict.pop(key, <value>)`

  `key` is the key to be searched, and `value` is the value to return if the specified key is not found in the dictionary. If `value` is not passed, an error is thrown in the case `key` is not in the dictionary

```
key = 3
x = numbers.pop(key, None)
if x != None:
   print('Removed pair (', key, ':', x, ')')
```

- Advantage over the use of `del` and `[]` operators:

```
key = 11
del numbers[key]
```

→ Throws an Error since the selected key is not in the dictionary!

(Optional) Methods for accessing and modifying a dictionary: `.popitem()`

- Remove and return the last inserted dictionary element: `dict.popitem()`

  A pair (key, value) is removed from the dictionary following a LIFO order (last-in, first-out). The removed pair is returned as a tuple

```
x = numbers.popitem()
if len(numbers) > 0:
    print('Removed the last inserted key-value pair (', x, ')')
    print('New size of the dictionary:', len(numbers))
```

(Optional) Methods for accessing and modifying a dictionary: `.get()`

- Get the value for a specified key if key is in dictionary: `dict.get(key, <value>)`

  `key` is the key to be searched, and `value` is the value to return if the specified key is not found in the dictionary. The `value` parameter is optional. If `value` is not passed, `None` is returned.

```
key = 3
x = numbers.get(key)
if x != None:
    print('Value associated to key', key, 'is:', x)
```

- Advantage over the use of the `[]` operator:

```
x = numbers[key]
```

→ Throws an Error if key is not in the dictionary!

(Optional) Methods for accessing and modifying a dictionary: `.clear()`

- Remove *all* elements from a dictionary element: `dict.clear()`

  All elements are removed, no values are returned, after the call `dict` is equivalent to `{}`

```
numbers.clear()
print('Removed all elements')
```