15-110 Principles of Computing – F19

Lecture 20:
Dictionaries 2

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Practice with a given dictionary as input

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

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

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

Practice with a given dictionary as input (from exam)

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). 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 (from exam)

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

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!

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))
```

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!

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')
  ```

Methods for accessing and modifying a dictionary: `update( )`

- Update the dictionary with the elements from the another dictionary object (or from an iterable of key/value pairs (e.g., tuple)): `dict.update([other])`

Takes as input a dictionary (or an iterable of tuples) and use the input to update `dict`

```
some_primes = {13:'p', 17:'p', 23:'p'}
numbers.update(some_primes)
print('Updated dictionary:', numbers)

new_entry= {12:'r'}
numbers.update(new_entry)
print('Dictionary updated with a new single entry')

l = [(10, 'r'), (12, 'r')]
numbers.update(l)
print('Dictionary updated with a list of entries')
```

Methods for accessing and modifying a dictionary: `setdefault()`

- Insert a new (key, value) pair only if key doesn’t already exist, otherwise return the current value: `dict.setdefault(key, <value>)`

The function aims to *update* the dictionary with a new (`key, value`) pair only if the given `key` is not already in the dictionary, otherwise the dictionary (i.e., the existing value of `key`) is *not updated* and the current value associated to the specified `key` is returned instead. If `value` is not passed as input, the default `None` value is used.

```
val = numbers.setdefault(30, 'r')      # key 30 doesn’t exist, pair (30,'r') is inserted in numbers
val = numbers.setdefault(30, 'rr')     # key 30 it exists now, its value isn’t updated, val gets 'r'


new_dict = {}
for i in range(10):
    new_dict.setdefault(i)     # new_dict gets initialized with 10 keys and None values
```

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

- 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’]
```

- Equivalent way, using the `list()` function:

```
keys_to_be_sorted = list(numbers.keys())
keys_to_be_sorted.sort()
```

- 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

- 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()`

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!
```

Creation of dictionary variables: use literals

- Empty dictionary:

```
v = {}
```

- Creation of a dictionary with literals:

```
phone_numbers = {'Ann': 5461, 'Paul': 5472, 'Mark': 3541, 'Liz': 2451}

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

by_name = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
           'F. Dupont': [17623, 'F', 'France'] }

by_country = {'USA': ['J. Smith’, 35672, 'M'], 'Jordan': ['M. Saleh’, 27623, 'M'],
              'France': ['F. Dupont’, 17623, 'F'] }
```

Creation of dictionary variables: use a list of tuples

- Use a list of tuples and the built-in function `dict(key_val_list)` that builds 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)] )


v = 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)

primes = [2, 3, 5, 7, 11, 13]
dict_of_primes = dict.fromkeys(primes, '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 = [1, 2, 3, 4, 5, 6]

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

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

Creation of dictionary variables: cloning and aliasing

- **Cloning**: make a *shallow copy* of the content of a dictionary using method `copy( )`

```
new_dict_same_content = numbers.copy()
new_dict_same_content[36] = 'r'
if 36 not in numbers:
    print("Change in the new dictionary didn't affected previous dictionary")
```

- **Aliasing:** make a copy of the content of a dictionary and establish an alias

```
alias_dict = numbers
alias_dict[36] = 'r'
if 36 in numbers:
    print("Change in the new dictionary affected previous dictionary!")
```

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) }
```