Dictionary

Dictionary

- A dictionary is like an address-book where you can find the address or contact details of a person by knowing only his/her name.
- We associate keys (name) with values (details). Note that the key must be unique just like you cannot find out the correct information if you have two persons with the exact same name.
- Note that you can use only immutable objects (like strings) for the keys of a dictionary but you can use either immutable or mutable objects for the values of the dictionary.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).
- Pairs of keys and values are specified in a dictionary by using the notation

d = {key1 : value1, key2 : value2 } .

- The dictionaries that are instances/objects of the dict class.

Python Dictionary

- For example −

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two“
Print(dict)    #This will print {'one‘: ‘This is one’, 2:’This is two’}

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

 print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

- This produce the following result −

```
This is one, This is two, {'dept': 'sales', 'code': 6734, 'name': 'john'}, ['dept', 'code', 'name'],
['sales', 6734, 'john']
```

- Dictionaries have no concept of order among elements.

Common Dictionary Constants and Operations

Operation	Interpretation
• d1 = {}	Empty dictionary
• d2 = {'spam' : 2, 'eggs' : 3}	Two-item dictionary
• d3 = {'food' : {'ham' : 1, 'egg' : 2}}	Nesting
• d2['eggs'], d3['food']['ham']	Indexing by key
• d2.has_key('eggs'), d2.keys(), d2.values()	Methods: membership test, keys list, values list, etc.
• len(d1)	Length (number stored entries)
• d2[key] = new, del d2[key]	Adding/changing, deleting

Dictionary Functions

```
d2 = {'spam' : 2, 'ham': 1, 'eggs' : 3}
d2['spam']                # fetch value for key
2
len(d2)                   # number of entries in dictionary
3
d2.has_key('ham')    # key membership test (1 means true)
1
d2.keys()                 # list of my keys ['eggs', 'spam', 'ham']
```

Dictionary Example

- ab = { 'Swaroop'   : 'swaroop@swaroopch.com',
-         'Larry'   : 'larry@wall.org',
-         'Matsumoto' : 'matz@ruby-lang.org',
-         'Spammer'   : 'spammer@hotmail.com‘
- }
- Print("Swaroop's address is", ab['Swaroop'])
- # Deleting a key-value pair
- del ab['Spammer']
- print '\nThere are {} contacts in the address-book\n'.format(len(ab))

Dictionary Example

- for name, address in ab.items():
-     print 'Contact {} at {}'.format(name, address)
- # Adding a key-value pair
- ab['Guido'] = 'guido@python.org'
- if 'Guido' in ab:
-     print "\nGuido's address is", ab['Guido']
- We can check if a key-value pair exists using the in operator.

Example

```
table = {'Python' :  'Guido van Rossum', …         'Perl':    'Larry Wall', …
'Tcl':     'John Ousterhout' } …
language = 'Python‘
creator = table[language]
#creator 'Guido van Rossum‘
for lang in table.keys():
            print lang, '\t', table[lang]
```

Output:

```
• Tcl             John Ousterhout
• Python          Guido van Rossum
• Perl            Larry Wall
```

Histogram and Lookup

- Here is a function that takes a value and returns the first key that maps to that value:
- def lookup(d, v):

```
    for k in d:
        if d[k] == v:
            return k
    Return None
```

- Here is an example of a successful reverse lookup:
- h = histogram('parrot')
- k = lookup(h, 2)
- print (k)

r

Inverting a dictionary

- Here is a function that inverts a dictionary:
- def invert_dict(d):
  - inverse = dict()
    - for key in d:

      ```
      val = d[key]
      if val not in inverse:
          inverse[val] = [key]
      else:
          inverse[val].append(key)
      ```
    - return inverse

Storing fibonnici terms in a dictionary

To keep track of values that have already been computed for a fibonnici series, store them in a dictionary.

A previously computed value that is stored for later use is called a memo. Here is a "memoized" version of fibonacci:

```
known = {0:0, 1:1}
def fibonacci(n):
        if n in known:
                return known[n]
        res = fibonacci(n-1) + fibonacci(n-2)
        known[n] = res
        return res
```

known is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

- Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it.

Dictionary Methods

- It is common to use tuples as keys in dictionaries (primarily because you can’t use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, first and number, we could write:
- directory[last,first] = number The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

Dictionary Methods

- Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair.
- >>> d = {'a':0, 'b':1, 'c':2} >>> t = d.items() >>> print t [('a', 0), ('c', 2), ('b', 1)]
- Going in the other direction, you can use a list of tuples to initialize a new dictionary:
- >>> t = [('a', 0), ('c', 2), ('b', 1)] >>> d = dict(t) >>> print d {'a': 0, 'c': 2, 'b': 1} Combining dict with zip yields a concise way to create a dictionary:
- >>> d = dict(zip('abc', range(3))) >>> print d {'a': 0, 'c': 2, 'b': 1} Thedictionarymethod update alsotakesalistoftuplesandaddsthem,askey-valuepairs, to an existing dictionary.

Dictionary Methods

- subtract takes dictionaries d1 and d2 and returns a new dictionary that contains all the keysfrom d1 thatarenotin d2. Sincewedon’treallycareaboutthevalues,wesetthemall to None.
- def subtract(d1, d2):
  - res = dict()
  - for key in d1:
    - if key not in d2:
      - res[key] = None

  return res

Dictionary Methods

- To find the words in the book that are not in words.txt, we can use process_file to
- build a histogram for words.txt, and then
- subtract: words = process_file('words.txt')
- diff = subtract(hist, words)
- print "The words in the book that aren't in the word list are:" for word in diff.keys(): print word,

Get function

- The method **get()** returns a value for the given key. If key is not available then returns default value None.
- Dict.get(key,default=None)
- Parameters
- **key** − This is the Key to be searched in the dictionary.
- **default** − This is the Value to be returned in case key does not exist.
- This method return a value for the given key. If key is not available, then returns default value None.
- Dic={Name:’Annie’,Age:7}
- Print(“Value : %s” % dict.get(‘Age’))

Sum, max and min in a dictionary

- dict={'a':10,'b':20,'c':30}
- print(sum(dict.values()))
- print(max(dict.values()))
- print(min(dict.values()))

Accessing nested dictionaries

- people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},

```
               2: {'name': 'Marie', 'age': '22', 'sex':'Female'}}
                 print(people[1]['name'])
                 print(people[1]['age'])
                 print(people[1]['sex'])
```

Adding in nested dictionaries

-
  ```
  people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
  2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
  people[3] = {}
  people[3]['name'] = 'Luna‘
  people[3]['age'] = '24‘
  people[3]['sex'] = 'Female‘
  people[3]['married'] = 'No‘
  print(people[3])
  ```

- d3 = {'food' : {'ham' : 1, 'egg' : 2}}
- Print(d3[‘food’][‘ham’]

Sequence Vs Dictionaries

- Sequence operations don't on dictionaries.
- Dictionaries are mappings, not sequences; because there's no notion of ordering among their items, things like concatenation (an ordered joining) and slicing (extracting contiguous section) simply don't apply. In fact, Python raises an error when your code runs, if you try.
- Assigning to new indexes adds entries. Keys can be created either when you write a dictionary constant (in which case they are embedded in the constant itself), or when you assign values to new keys of an existing dictionary object. The end result is the same.
- Keys need not always be strings We've been using strings as keys here, but other immutable objects (not lists) work just as well.
- In fact, you could use integers as keys, which makes a dictionary look much like a list (albeit, without the ordering).

Exercise

Q1 Write a program that reads a word list from a file (see Section 9.1) and prints all the sets of words that are anagrams. Here is an example of what the output might look like:

- ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled'] ['retainers', 'ternaries'] ['generating', 'greatening'] ['resmelts', 'smelters', 'termless']
- Hint: you might want to build a dictionary that maps from a set of letters to a list of words that can be spelled with those letters. The question is, how can you represent the set of letters in a way that can be used as a key?

Q2. Modifythepreviousprogramsothatitprintsthelargestsetofanagramsfirst,followedbythe second largest set, and so on.