Dictionaries

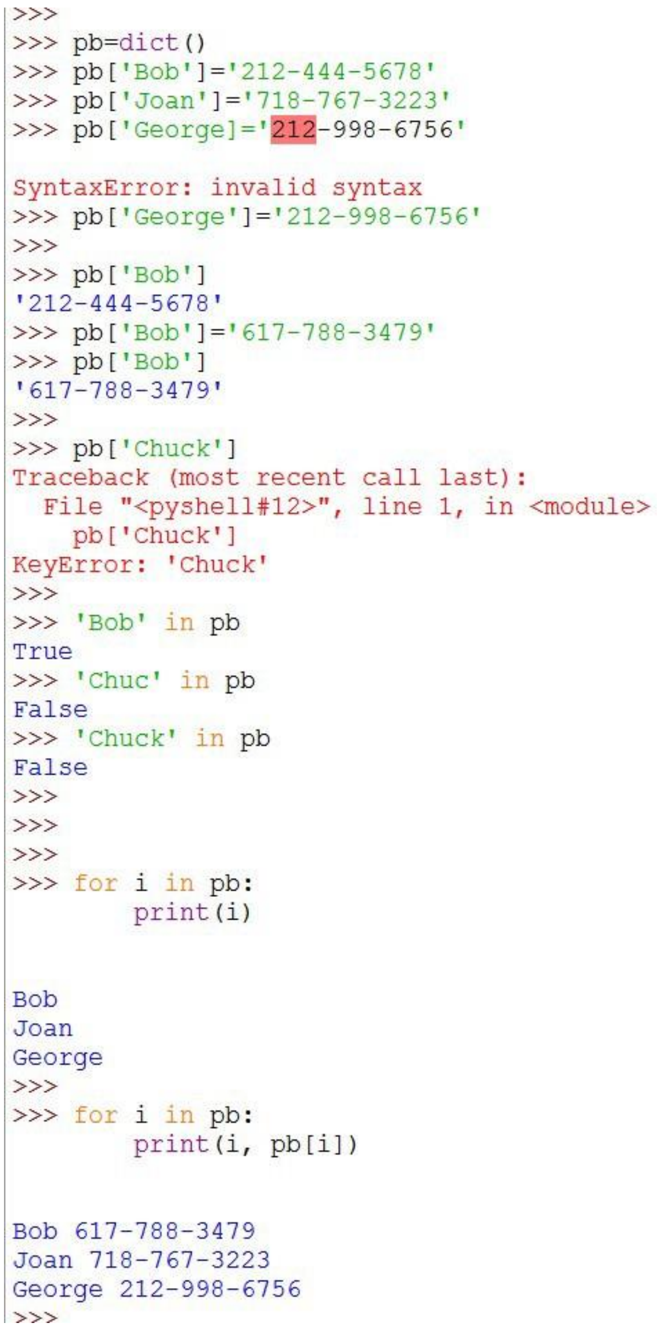
Think of an on-line dictionary. You type in a word and the dictionary returns one or more meanings of the word you entered.

We can think of the dictionary as a “list” that is indexed by the “word” whose definition you seek and the value that is returned is the set of meaning associated with that word.

Or …

Think of an on-line phone book. You time in the name of the person whose phone number you want and the phone book app returns the associated number.

We can model the above in Python using the **dictionary**.



```
>>>
>>> pb=dict()
>>> pb['Bob']='212-444-5678'
>>> pb['Joan']='718-767-3223'
>>> pb['George]='212-998-6756'

SyntaxError: invalid syntax
>>> pb['George']='212-998-6756'
>>>
>>> pb['Bob']
'212-444-5678'
>>> pb['Bob']='617-788-3479'
>>> pb['Bob']
'617-788-3479'
>>>
>>> pb['Chuck']
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    pb['Chuck']
KeyError: 'Chuck'
>>>
>>> 'Bob' in pb
True
>>> 'Chuc' in pb
False
>>> 'Chuck' in pb
False
>>>
>>>
>>>
>>> for i in pb:
        print(i)


Bob
Joan
George
>>>
>>> for i in pb:
        print(i, pb[i])


Bob 617-788-3479
Joan 718-767-3223
George 212-998-6756
>>>
```

Here are some of the dictionary methods:

`s= dict`() also s={}

Create a new dictionary.

`len(d)`

Return the number of items in the dictionary *d*.

`d[key]`

Return the item of *d* with key *key*. Raises a `KeyError` if *key* is not in the map.

`d[key] = value`

Set `d[key]` to *value*.

`del d[key]`

Remove `d[key]` from *d*. Raises a `KeyError` if *key* is not in the map.

`key in d`

Return `True` if *d* has a key *key*, else `False`.

`key not in d`

Equivalent to `not key in d`.

`clear`()

Remove all items from the dictionary.

`copy`()

Return a shallow copy of the dictionary.

Create a new dictionary with keys from *seq* and values set to *value*.

`items`()

Return a new view of the dictionary's items (`(key, value)` pairs).

`keys`()

Return a new view of the dictionary's keys.

`pop`(*key*[, *default*])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

`values`()
Return a new view of the dictionary’s values

There are additional methods. Check out the Python on-line documentation.

Dictionaries in some more detail

A dictionary is a mapping between keys and values. You create a dictionary by enclosing the key-value pairs (key:value) each separated by a colon), in curly braces ({ }), each pair separated by a comma like this:

```
s = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10
   }
```

To access members of a dictionary, use the indexing operator as follows:

```
name = s['name']
cost = s['shares'] * s['price']
```

Inserting or modifying objects works like this:

```
s['shares'] = 75
s['date'] = '2007-06-07'
```

A dictionary is a useful way to define an object that consists of named fields. However, dictionaries are also commonly used as a mapping for performing fast lookups on unordered data.

For example, here’s a dictionary of stock prices:

```
prices = {
   'GOOG' : 490.1,
   'AAPL' : 123.5,
   'IBM' : 91.5,
   'MSFT' : 52.13
}
```

Given such a dictionary, you can look up a price:

```
p = prices['IBM']
```

Dictionary membership is tested with the in operator:

```
if 'IBM' in prices:
   p = prices['IBM']
else:
   p = 0.0
```

This particular sequence of steps can also be performed more compactly using the **get**() method:

```
p = prices.get('IBM', 0.0)    # prices['IBM'] if it exists, else 0.0 # or any other default
```

Use the del statement to remove an element of a dictionary:

```
del prices['GOOG']
```

```
>>> prices = {
    'GOOG' : 490.1,
    'AAPL' : 123.5,
    'IBM' : 91.5,
    'MSFT' : 52.13
}

>>> del prices['TYU']
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    del prices['TYU']
KeyError: 'TYU'
>>> prices.pop('TYU',-1)
-1
>>>
```

Although strings are the most common type of key, you can use many other Python objects, including numbers and tuples.

For example, tuples are often used to construct composite or multipart keys:

```
prices = { }
prices[('IBM', '2015-02-03')] = 91.23
prices['IBM', '2015-02-04'] = 91.42     # Parens omitted
```

Any kind of object can be placed into a dictionary, including other dictionaries.

However, mutable data structures such as lists, sets, and dictionaries **cannot be used as keys.**

Dictionaries are often used as building blocks for various algorithms and data-handling problems.

One such problem is tabulation.

For example, here's how you could count the total number of shares for each stock name in earlier data:

```
portfolio = [
   ('ACME', 50, 92.34),
   ('IBM', 75, 102.25),
   ('PHP', 40, 74.50),
   ('IBM', 50, 124.75)
]

total_shares = { s[0]: 0 for s in portfolio }  # dictionary comprehension
for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = {'IBM': 125, 'ACME': 50, 'PHP': 40}
```

In this example, { s[0]: 0 for s in portfolio } is an example of a dictionary comprehension.

It creates a dictionary of key-value pairs from another collection of data. In this case, it's making an initial dictionary mapping stock names to 0. The for loop that follows iterates over the dictionary and adds up all of the held shares for each stock symbol.

Aside: Many common data processing tasks such as this one have already been implemented by library modules.

For example, the collections module has a Counter object that can be used for this task:

```
from collections import Counter

total_shares = Counter()
for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = Counter({'IBM': 125, 'ACME': 50, 'PHP': 40})
```

Creating an empty dictionary

```
prices = {}        # An empty dict
prices = dict()    # An empty dict
```

It is more idiomatic to use {} for an empty dictionary—although caution is required since it might look like you are trying to create an empty set (use set() instead).

dict() is commonly used to create dictionaries from key-value values. For example:

```
pairs = [('IBM', 125), ('ACME', 50), ('PHP', 40)]
d = dict(pairs)
```

To obtain a list of dictionary keys, convert a dictionary to a list:

```
syms = list(prices)     # syms = ['AAPL', 'MSFT', 'IBM', 'GOOG']
```

Alternatively, you can obtain the keys using dict.keys():

```
syms = prices.keys()
```

The difference between these two methods is that

keys() returns a special "keys **view**" that is attached to the dictionary and actively reflects changes made to the dictionary.

For example:

```
>>> d = { 'x': 2, 'y':3 }
>>> k = d.keys()
>>> k
dict_keys(['x', 'y'])
>>> d['z'] = 4
>>> k
dict_keys(['x', 'y', 'z'])
>>>
```

The "keys()" function is an example of a **view**.

The objects returned from dict.keys(), dict.values(), and dict.items() are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes as seen in the example above.

So:

To obtain the values stored in a dictionary,

use the dict.values() method.

To obtain key-value pairs,

use dict.items().

For example, here's how to iterate over the entire contents of a dictionary as key-value pairs:

```
for sym, price in prices.items():
    print(f'{sym} = {price}')
```

Here are some more examples from the Python documentation.

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

What is the order of elements in the dictionary?

The keys always appear in the same order as the items were initially inserted into the dictionary. The list conversion above will preserve this order.

This can be useful when dicts are used to represent key-value data read from files and other data sources. The dictionary will preserve the input order. This might help readability and debugging. It's also nice if you want to write the data back to a file.

Note: Prior to Python 3.6, however, this ordering was not guaranteed, so you cannot rely upon it if compatibility with older versions of Python is required. Order is also not guaranteed if multiple deletions and insertions have taken place. Its possible (though not very likely) that the ordering might onece again e unpredictable as it was before Python 3.6.

Therefore, if you absolutely need the key value pairs you can use **"collections.OrderedDict":**

collections.OrderedDict is a subclass of the built-in dict class that is guaranteed to maintain the order of the keys in which they were inserted.

For example:

```
import collections

# Create an empty ordered dictionary
od = collections.OrderedDict()

# Add items to the dictionary in a specific order
od['apple'] = 1
od['banana'] = 2
od['orange'] = 3
od['grape'] = 4

# Print the items in the order they were added
for key, value in od.items():
    print(key, value)
```

The output is:

```
apple 1
banana 2
orange 3
grape 4
```

Dictionary comprehensions

A compact way to process all or part of the elements in an iterable and return a dictionary with the results.

`results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`.

Problem

Using a dictionary comprehension create a dictionary with n associated to the nth prime number for the first k primes. K is provided by the user and is not part of the comprehension.

Answer

Copying dictionaries

To copy a dictionary in Python, you can use either the dict() constructor or the dictionary method copy(). Here's an example of how to use dict() to create a shallow copy of a dictionary:

```
original_dict = {'a': 1, 'b': 2, 'c': 3}
copy_dict = dict(original_dict) #constructor call

print(original_dict)  # {'a': 1, 'b': 2, 'c': 3}
print(copy_dict)  # {'a': 1, 'b': 2, 'c': 3}
```

In this example, we create an original_dict with three key-value pairs. We then create a shallow copy of the dictionary using the dict() constructor, and assign it to the copy_dict variable.

When we print both dictionaries, we see that they contain the same key-value pairs, indicating that the dict() constructor created a copy of the original dictionary.

Here's an example of how to use the copy() method to create a shallow copy of a dictionary:

```
original_dict = {'a': 1, 'b': 2, 'c': 3}
copy_dict = original_dict.copy()

print(original_dict)  # {'a': 1, 'b': 2, 'c': 3}
print(copy_dict)  # {'a': 1, 'b': 2, 'c': 3}
```

In this example, we create an original_dict with three key-value pairs. We then create a **shallow copy** of the dictionary using the copy() method, and assign it to the copy_dict variable.

When we print both dictionaries, we see that they contain the same key-value pairs, indicating that the copy() method created a copy of the original dictionary.

Note that both of these methods create a shallow copy of the dictionary, which means that any mutable objects (such as lists or other dictionaries) contained in the original dictionary will still be referenced by both the original and copied dictionaries. I

If you want to create a deep copy of a dictionary that also copies any mutable objects contained within it, you can use the copy module's deepcopy() function instead.

Deep copy

In Python, you can create a deep copy of a dictionary using the **copy.deepcopy()** method from the built-in **copy** module. A deep copy is a new dictionary that is completely independent of the original dictionary. Any changes made to the new dictionary will not affect the original dictionary.

For example:

```
import copy

# Define a dictionary
my_dict = {'a': [1, 2], 'b': [3, 4]}

# Create a deep copy of the dictionary
my_dict_copy = copy.deepcopy(my_dict)

# Modify the copy
my_dict_copy['a'][0] = 5

# Print both dictionaries
print(my_dict)      # Output: {'a': [1, 2], 'b': [3, 4]}
print(my_dict_copy) # Output: {'a': [5, 2], 'b': [3, 4]}
```

In this example, we start by defining a dictionary named **my_dict**. We then create a deep copy of **my_dict** using the **copy.deepcopy()** method and assign it to a variable named **my_dict_copy**.

Next, we modify the value associated with the key 'a' in **my_dict_copy** to **[5, 2]**.

Finally, we print both dictionaries to show that the original dictionary **my_dict** has not been modified, while the copied dictionary **my_dict_copy** has been modified.

Important: Note that **copy.deepcopy()** recursively copies all nested data structures in the dictionary, so if your dictionary contains lists, dictionaries, or other mutable objects, a deep copy is necessary to ensure that these objects are also copied and not simply referenced.

For example:

```
import copy

original_dict = {
    'a': 1,
    'b': {
        'c': 2,
        'd': {
            'e': 3,
            'f': {
                'g': 4
            }
        }
    }
}
```

```
# Create a deep copy of the original dictionary
copy_dict = copy.deepcopy(original_dict)

# Modify the copy to demonstrate that it is independent of the original
copy_dict['b']['d']['f']['g'] = 5

# Print the original and copied dictionaries
print(original_dict)
print(copy_dict)
```

The result is:

```
{'a': 1, 'b': {'c': 2, 'd': {'e': 3, 'f': {'g': 4}}}}
{'a': 1, 'b': {'c': 2, 'd': {'e': 3, 'f': {'g': 5}}}}
```

So the original dictionary has 'g': **4** but the copy has 'g': **5**.

Dictionaries and **kwargs in Python functions

Recall that **kwargs is a syntax used in Python to pass a dictionary of keyword arguments to a function.

The syntax **kwargs in a function parameter list allows the function to accept an arbitrary number of keyword arguments as a dictionary.

For example:

```
def my_func(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

Now, you can call this function with any number of keyword arguments:

```
my_func(apple=3, banana=5, orange=2)
```

and the output will be:

```
apple: 3
banana: 5
orange: 2
```

In this example, the ****kwargs** syntax allows the function to accept any number of keyword arguments and store them in a dictionary named **kwargs**. Inside the function, we can access the keyword arguments as key-value pairs in the **kwargs** dictionary.

So, the relation between ****kwargs** and dictionaries is that ****kwargs** allows you to pass a dictionary of keyword arguments to a function, and the function can then access the keyword arguments as a dictionary.

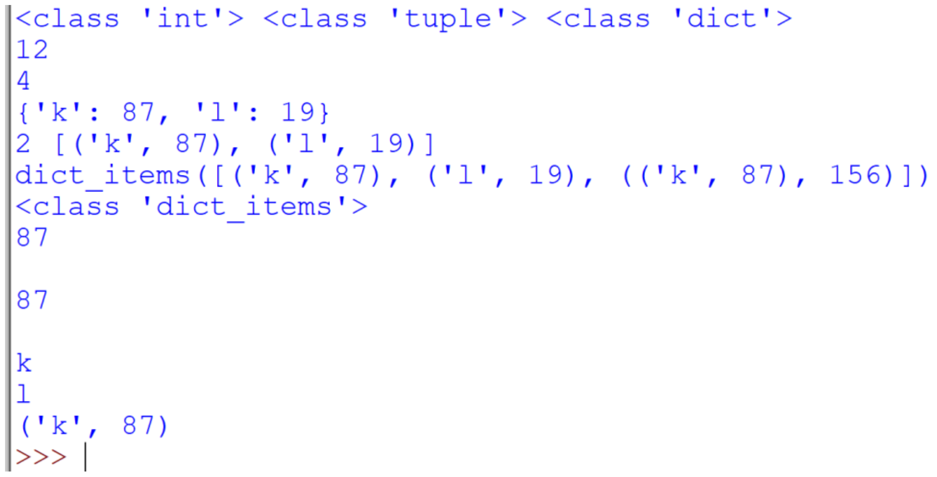
This can be a convenient way to pass a variable number of arguments to a function, especially when you don't know in advance how many arguments will be passed.

Another example:

```
def z(a,*b,**c): # recall the order
    print(type(a),type(b),type(c))
    print(a,b[1],c,sep='\n')
    print(len(c),list(c.items()))
    g=list(c.items())
    c[g[0]]=156
    print(c.items())
    print (type(c.items()))
    d=[i for i in c.keys()]
    print(c[d[0]])
    print()
    z=list(c)
    print(c[list(c)[0]])
    print()
    for i in iter(c):
        print(i)


  z(12,3,4,5,k=87,l=19)
```

The output:



```
<class 'int'> <class 'tuple'> <class 'dict'>
12
4
{'k': 87, 'l': 19}
2 [('k', 87), ('l', 19)]
dict_items([('k', 87), ('l', 19), (('k', 87), 156)])
<class 'dict_items'>
87

87

k
l
('k', 87)
>>> |
```

The technique of using the **kwargs dictionary will allow us to create multiple constructors for Pythn classes.

Let's look at some larger examples of dictionaries. The first will be will be a **scrabble dictionary** and the second will be to create an **inverted index for a file**.
But first, some preliminaries.

Problem:

Write a function, **signature**(n) that and returns a string with same letters in lexicographic order.

For example, **signature**(stop) ➔ opst

Problem:

Scrabble Descrambler – Very Lite

You are in the middle of an intense game of Scrabble, and you find that your tiles have the letters "ACENRT". What can you possibly do with that??

You whip out your smart-phone and run your handy Scrabble Descrambler! You simply enter the letters on the tiles, and PRESTO!!!, all legal Scrabble words with those letters magically appear on your screen. How cool is that?

Your assignment: write the Scrabble Descrambler.

How?

For this problem, we will only deal with six letter words.

1. From main page of the lecture web site copy the contents at the link "Legal Six Letter Scrabble Words" and store it in a file in your Python directory called wordlist.txt.

2. Create a list of tuples (signature,word), where for each word in word list its "signature" is the "word" sorted by its letters. So, for example, if we were dealing with four letter words, the signature of "STOP" is "OPST". Notice that this is also the signature of POTS and TOPS.

3. Sort the list of tuples created in step 2 by the signature.

4. Go through the list created in step 3 and create a Python dictionary where for each entry, the key is the signature and the associated value is a list of all words with the same signature.

5. Finally, in a loop, ask the user for the six letters that they want to look up, and your program will return all valid six-letter Scrabble words matching the rquest.

Pickle

The pickle function is a powerful tool that can be used to save and restore Python objects. It can be used to save objects to a file, send objects over a network, or store objects in a database.

pickle is used to **serialize** and **deserialize** Python objects. Serialization is the process of converting an object into a sequence of bytes, while deserialization is the process of converting a sequence of bytes back into an object.

Think of a complex data structure, say a dictionary where each value is a dictionary each of whose values is a binary tree. How could you store such a structure in a file so that it could be reconstructed later? This is what pickle does.

The pickle function uses a binary format to store objects. This format is efficient and can be used to store any type of Python object. You can unpickle a pickle file to reconstruct the original data structure. The pickle function is also secure, as it can be used to store objects that contain sensitive data.

Here is an example of how to use the pickle function to save a list of numbers to a file:

```
import pickle

numbers = [1, 2, 3, 4, 5]

with open("numbers.pkl", "wb") as f:
    pickle.dump(numbers, f)
```

To restore the list of numbers from the file, you can use the following code:

```
with open("numbers.pkl", "rb") as f:
    restored_numbers = pickle.load(f)

print(restored_numbers)
```

Here is another example.

We create our Scrabble dictionary and then pickle it for later use.

```
def signature(w):
    w1=list(w)
    w1.sort()
    w1=''.join(w1)
    return w1

#create or load?
mode=input("Create or Load C or L: ")
print()
if mode.upper()=='C':
# create a "Scrabble Dictionary"
    d={}
    print('Creating dictionary ... please wait.')
    f = open('C:/python32/six letter words.txt', 'r')
    print()
    sl = f.read()
    z=sl.split(' ')
    print()
    for w in z:
        sig=signature(w)
        if sig not in d:
            d[sig]=[]
            d[sig].append(w)
        else:
            d[sig].append(w)
    f.close()
else:
    print('Unpickling dictionary ... please wait.')
    f=open('slwords','rb')
    d=pickle.load(f)  # this “unpickles”

word=input("Please enter word: ")
print()

while word!='done':
    if len(word)!=6:
        print("word not 6 chars")
    else:
        word=word.upper()
        word=signature(word)

        if word in d:
            print(d[word])
        else:
            print(word,' not found.')

    word=input("Please enter word: ")
    print()

f=open('slwords','wb') print('Pickling ... please
wait.')pickle.dump(d,f) # this pickles
f.close()
```

Another dictionary problem: **Inverted Index**

Problem: Read a text file and create an inverted index for the file.

What is an inverted index of a text file?

An inverted index is a dictionary whose **key** is a “word” in the file and whose **value** is a **list** of tuples **(line number, number of times “word” appears on the line).**

Part 1.

Write a program that reads a text file and creates a simplified version of in inverted index.

Your program will create a dictionary, invertedDict, whose key is the word in the file and whose value is a **list** of integers representing the line numbers in the file where word is found.

What if the word is found k times on some line?

Then the line number will appear k times in the list.

For example, say the word “cat” appears 3 times on line 4, 2 times on line 6, and 4 time on line 8, then the dictionary entry for cat will be:

invertedDict[“cat”] ->[4,4,4,6,6,8,8,8,8].

Part 2.

Write a function squish(x) where x is a list of integers so that squish**(**-[4,4,4,6,6,8,8,8,8]) returns - [(4,3),(6,2),(8,4)].

Part 3.

Modify part 1so that

invertedDict[“cat”] ->[(4,3),(6,2),(8,4)].

Test your program on the text of the Gettysburg Address.