**Dictionaries**

We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

A dictionary is like a list, but more general. In a list, the positions (a.k.a. indices) have to be integers; in a dictionary the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

This section will serve as a brief introduction to dictionaries and consist of:

1.) Constructing a Dictionary

2.) Accessing objects from a dictionary

3.) Nesting Dictionaries

4.) Basic Dictionary Methods

So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

**Constructing a Dictionary**

Let's see how we can construct dictionaries to get a better understanding of how they work!

​

dt **=** {1:"one",2:"Two",3:"Three"}

print(dt[3])

​

dt **=** dict()

dt[4]**=**"Four"

dt[4]**=**"Five"

*# dt.update({k:v})*

print(dt)

Three

{4: 'Five'}

d **=** {}

d **=**{'a':"Asha",'d':"Dheeraj"}

d **=** dict() *#create an empty dictionary and then add the values , else create dict using { } with values*

d['a']**=**"Asha"

d['d']**=**"Dheeraj"

d['1']**=**2

d['2']**=**3

print(d)

print(d['a'])

{'a': 'Asha', 'd': 'Dheeraj', '1': 2, '2': 3}

Asha

*#This output format is also an input format. For example, you can create a new dictionary with three items:*

​

eng2sp **=** {'one': 'uno', 'two': 'dos', 'three': 'tres'}

*#But if you print eng2sp, you might be surprised:*

​

print(eng2sp)

{'one': 'uno', 'two': 'dos', 'three': 'tres'}

*# The order of the key-value pairs is not the same. In fact, if you type the same example on your computer,*

*# you might get a different result. In general, the order of items in a dictionary is unpredictable.*

​

*# But that's not a problem because the elements of a dictionary are never indexed with integer indices.*

*# Instead, you use the keys to look up the corresponding values:*

print (eng2sp['two'])

dos

*# The key 'two' always maps to the value 'dos' so the order of the items doesn't matter.*

​

*# If the key isn't in the dictionary, you get an exception:*

print (eng2sp['four'])

**---------------------------------------------------------------------------**

**KeyError** Traceback (most recent call last)

**<ipython-input-9-c4dfbcc148dd>** in <module>

2

3 **# If the key isn't in the dictionary, you get an exception:**

**----> 4** print **(**eng2sp**['four'])**

**KeyError**: 'four'

**The dict() Constructor Function**

The dict() function can be used to create a new dictionary object from an iterable or a sequence of key:value pairs. The signature of this function is: This is an overloaded function with three version that can take different types of arguments:

• The first option takes a sequence of key:value pairs.

• The second takes a mapping and (optionally) a sequence of key:value pairs.

• The third version takes an iterable of key:value pairs and an optional sequence of key:value pairs.

dict(\*\*kwarg)

dict(mapping, \*\*kwarg)

dict(iteratable, \*\*kwarg)

**def** add(**\*\***kwargs):

**for** keys,values **in** kwargs:

print(keys,values)

**def** add(**\*\***kwargs):

**for** key,item **in** kwargs:

print(key,item)

add(a **=** 10,b**=**20,c**=**30,d**=**50)

add(10,20)

add(23,4,5,6,7)

​

*# print("The name of the animal is{p}".format(p ='Penny'))*

*# a = '10'*

*# #dict(\*\*kwarg) where uk is the name or key and London is value , note : uk is not under quotes as it is named key*

​

styui **=**'10'

​

a **=** 20

​

dict1 **=**dict(uk**=**'London',scotland**=**'edinburg',france **=** 'paris')

print(dict1)

print(type(dict1))

dict2 **=** dict(one **=**'1',two**=**'2',three**=**'3')

print(dict2)

{'uk': 'London', 'scotland': 'edinburg', 'france': 'paris'}

<class 'dict'>

{'one': '1', 'two': '2', 'three': '3'}

*#dict(mapping,optional \*\*kwarg)*

​

​

dict2 **=** dict([('uk','London'),('scotland','edinburg')],india **=**'Delhi') *#List of tuples , here both are in quotes as it is strings, just mapping is provided like uk should be mapped to london*

print(dict2) *#here india = 'delhi is optional kwarg'*

*#or*

dict3 **=** dict((('uk','London'),('scotland','edinburg'))) *#Tuples of tuples*

print(dict3)

{'uk': 'London', 'scotland': 'edinburg', 'india': 'Delhi'}

{'uk': 'London', 'scotland': 'edinburg'}

dict4 **=** dict((['uk','London'],['scotland','edinburg'])) *#Tuples of List*

print(dict4)

​

print(dict4['uk'])

​

*#dict(iterable,optional \*\*kwarg)*

​

lst **=** [1,2,3,4,5,6]

lst1 **=** [3,4,5,6,7] *#2 iterables ziped togather to form key value*

dict5 **=** zip(lst,lst1)

print(dict5)

​

​

dict5 **=** dict(dict5,india **=**'Delhi') *#here india = 'delhi' is optional kwarg*

print(dict5)

{'uk': 'London', 'scotland': 'edinburg'}

London

<zip object at 0x05A57698>

{1: 3, 2: 4, 3: 5, 4: 6, 5: 7, 'india': 'Delhi'}

t **=** [1,2,3]

t **=** t**+**['newitem']

print(t)

[1, 2, 3, 'newitem']

**Accessing Items via Keys**

You can access the values held in a Dictionary using their associated key. This is specified using either the square bracket ('[]') notation (where the key is within the brackets) or the get() method:

print(dict3['uk'])

​

dict3['uk']

print(dict3.get('uk'))

London

London

print(dict3.get('scotland'))

​

​

edinburg

*#Adding new value*

dict3['x']**=**'y'

dict3['india']**=**'delhi'

dict3['karnataka']**=**'bangalore'

print(dict3)

{'uk': 'London', 'scotland': 'edinburg', 'x': 'y', 'india': 'delhi', 'karnataka': 'bangalore'}

*#modifying*

dict3['karnataka']**=**'Bengaluru'

print(dict3)

{'uk': 'London', 'scotland': 'edinburg', 'x': 'y', 'india': 'delhi', 'karnataka': 'Bengaluru'}

​

*# Make a dictionary with {} and : to signify a key and a value*

my\_dict **=** {'key1':'value1','key2':'value2'}

*# Call values by their key*

my\_dict['key1']

Out[47]:

'value1'

#Its important to note that dictionaries are very flexible in the data types they can hold. For example:

my\_dict **=** {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

my\_dict['key2'][0]

Out[10]:

12

*# Let's call items from the dictionary*

my\_dict['key3'][0]

Out[13]:

'item0'

*# Can call an index on that value*

my\_dict['key3'][1]

Out[29]:

'item1'

*# Can then even call methods on that value*

my\_dict['key3'][0].upper()

Out[30]:

'ITEM0'

len(eng2sp)

​

Out[31]:

3

*# The in operator works on dictionaries;*

*# it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).*

​

​

'one' **in** eng2sp.keys()

​

​

Out[13]:

True

'uno' **in** eng2sp.values() *#you can not check with values ,instead use values method*

​

Out[14]:

True

*# To see whether something appears as a value in a dictionary, you can use the method values,*

*# which returns the values as a list, and then use the in operator:*

​

vals **=** eng2sp.values()

'uno' **in** vals

Out[10]:

True

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a hash table that has a remarkable property; the in operator takes about the same amount of time no matter how many items there are in a dictionary.

Type *Markdown* and LaTeX: 𝛼2α2

#We can affect the values of a key as well. For instance:

my\_dict['key1']

Out[57]:

123

*# Subtract 123 from the value*

my\_dict['key1'] **=** my\_dict['key1'] **-** 123

*#Check*

my\_dict['key1']

Out[31]:

-123

Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

*# Set the object equal to itself minus 123*

my\_dict['key1'] **-=** 123

my\_dict['key1']

Out[60]:

-123

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

*# Create a new dictionary*

d **=** {}

*# Create a new key through assignment*

d['animal'] **=** 'Dog'

*# Can do this with any object*

d['answer'] **=** 42

*#Show*

d

Out[16]:

{'animal': 'Dog', 'answer': 42}

​

*# print(help(dict))*

print(dir(dict))

['\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']

dict3['UnitedKingdom']**=None**

print(dict3)

{'uk': 'London', 'scotland': 'edinburg', 'india': 'Delhi', 'karnataka': 'Bengaluru', 'UnitedKingdom': None}

*# dict3['uk']='None'*

*# dict3*

print(dict3.pop('UnitedKingdom'))

dict3

​

*#After poping if you try to access the key , there will be error*

print(dict3['UnitedKingdom'])

None

**---------------------------------------------------------------------------**

**KeyError** Traceback (most recent call last)

**<ipython-input-19-018d5ccfc8ce>** in <module>

5

6 **#After poping if you try to access the key , there will be error**

**----> 7** print**(**dict3**['UnitedKingdom'])**

**KeyError**: 'UnitedKingdom'

**Nesting with Dictionaries**

Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary: The key and value in a dictionary must be an object; however, everything in Python is an object and thus anything can be used as a key or a value. One common pattern is where the value in a dictionary is itself a container such as a List, Tuple, Set or even another Dictionary. The following example uses Tuples to represent the months that make up the seasons

seasons **=** {'spring':('Mar','Apr','May'),

'Summer':('June','july','August'),

'Autumn':('Sep','Oct','Nov'),

'Winter':('Dec','Jan','Feb')}

seasons

Out[32]:

{'spring': ('Mar', 'Apr', 'May'),

'Summer': ('June', 'july', 'August'),

'Autumn': ('Sep', 'Oct', 'Nov'),

'Winter': ('Dec', 'Jan', 'Feb')}

seasons['spring']

Out[33]:

('Mar', 'Apr', 'May')

seasons['spring'][1]

Out[34]:

'Apr'

*# Dictionary nested inside a dictionary nested inside a dictionary*

d **=** {'key1':{'nestkey':{'subnestkey':'value'}}}

print(d['key1']['nestkey']['subnestkey'])

value

print(d['key1']['nestkey']['subnestkey'])

value

print(d['key1']['nestkey']['subnestkey'])

value

Let's see how we can grab that value:

*# Keep calling the keys*

d['key1']['nestkey']['subnestkey']

Out[18]:

'value'

**A few Dictionary Methods**

There are a few methods we can call on a dictionary. Let's get a quick introduction to a few of them:

*# Create a typical dictionary*

d **=** {'key1':1,'key2':2,'key3':3}

*# Method to return a list of all keys*

d.keys()

Out[21]:

dict\_keys(['key1', 'key2', 'key3'])

*# Method to grab all values*

d.values()

Out[22]:

dict\_values([1, 2, 3])

*# Method to return tuples of all items*

d.items()

​

*# d = dict([('key1', 1), ('key2', 2), ('key3', 3)])*

*# print(d)*

Out[23]:

dict\_items([('key1', 1), ('key2', 2), ('key3', 3)])

**using loops**

**Iterating over Keys**

You can loop through a dictionary using the for loop statement. The for loop processes each of the keys in the dictionary in turn. This can be used to access each of the values associated with the keys, for example

**for** countries **in** dict3.keys():

print(countries,end**=**' - ')

print(dict3[countries])

uk - London

scotland - edinburg

india - Delhi

karnataka - Bengaluru

There are three methods that allow you to obtain a view onto the contents of a dictionary, these are values(), keys() and items().

• The values() method returns a view onto the dictionary’s values.

• The keys() method returns a view onto a dictionary’s keys.

• The items() method returns a view onto the dictionary’s items ((key, value) pairs).

A view provides a dynamic window onto the dictionary’s entries, which means that when the dictionary changes, the view reflects these changes.

**for** countries,cities **in** dict3.items():

print(countries,cities)

print("\n")

**for** countries **in** dict3.keys():

print(countries)

​

print("\n")

**for** countries **in** dict3.values():

print(countries)

('uk', 'London')

('scotland', 'edinburg')

('india', 'Delhi')

('karnataka', 'Bengaluru')

uk

scotland

india

karnataka

London

edinburg

Delhi

Bengaluru

dict1 **=**{'key1':1,'key2':2,'key3':3}

dict1['key5'] **=** 5 *#one way of updating*

k1 **=**'key4'

v1 **=** 4

dict1.update({k1:v1}) *#other way is to use update keyword for updating*

dict1.update({'key1':6})

dict1

Out[19]:

{'key1': 6, 'key2': 2, 'key3': 3, 'key5': 5, 'key4': 4}

**for** x **in** dict1.items():

print(x)

('key1', 1)

('key2', 2)

('key3', 3)

('key5', 5)

('key4', 4)

*# convertings lists in to dictionaries using zip*

lst1 **=**[1,2,3]

lst2 **=**[5,6,7]

z**=** zip(lst1,lst2) *# will convert sequences to zip object*

z *#will print zip object , to get it converted to dict need to use dict class*

Out[47]:

<zip at 0x2f96c10>

*# now convert to dictionary object*

d **=** dict(z)

d

Out[82]:

{1: 5, 2: 6, 3: 7}

*# Ordered dictionaries*

*# dictionaries are not ordered so they are not stored in the same order as they are entered, if we want to keep the order of the elements*

*# we can use ordereddict*

**from** collections **import** OrderedDict

d **=** OrderedDict()

d[10]**=**'A'

d[11]**=**'B'

d

​

Out[20]:

OrderedDict([(10, 'A'), (11, 'B')])

Dictionary as a set of counters Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it: You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.

You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item. Each of these options performs the same computation, but each of them implements that computation in a different way.

An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

**def** histogram(s):

d **=** dict()

**for** c **in** s:

**if** c **not** **in** d:

d[c] **=** 1

**else**:

d[c] **=** d[c] **+** 1

**return** d

​

​

*# The name of the function is histogram, which is a statistical term for a set of counters (or frequencies).*

​

​

​

*# The first line of the function creates an empty dictionary.*

*# The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary,*

*# we create a new item with key c and the initial value 1 (since we have seen this letter once).*

*# If c is already in the dictionary we increment d[c].*

​

​

{1,2,3}

​

[0]

d **=**{'a':1,'b':2}

**for** item **in** d.keys():

print(d['a']) **-**1

d['a']

h **=** histogram('brontosaurus')

print(h)

​

*# The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on.*

​

{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}

*# Dictionaries have a method called get that takes a key and a default value.*

*# If the key appears in the dictionary, get returns the corresponding value;*

*# otherwise it returns the default value. For example:*

​

h **=** histogram('a')

print(h)

​

​

{'a': 1}

print(h.get('a', 0))

print(h.get('b', 0))

​

1

0

**Dictionaries and files**

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let's start with a very simple file of words taken from the text of Romeo and Juliet thanks to

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included.

But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief

We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line, and count each word using a dictionary.

You will see that we have two for loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called nested loops because one of the loops is the outer loop and the other loop is the inner loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating ``more quickly'' and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

fname **=** input('Enter the file name: ')

**try**:

fhand **=** open(fname)

**except**:

print ('File cannot be opened:', fname)

exit()

​

counts **=** dict()

**for** line **in** fhand:

words **=** line.split()

**for** word **in** words:

**if** word **not** **in** counts:

counts[word] **=** 1

**else**:

counts[word] **+=** 1

​

print(counts)

*# When we run the program, we see a raw dump of all of the counts in unsorted hash order.*

​

Enter the file name: file1

File cannot be opened:

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-89-bf58b79c9459>** in <module>

7

8 counts **=** dict**()**

**----> 9 for** line **in** fhand**:**

10 words **=** line**.**split**()**

11 **for** word **in** words**:**

**NameError**: name 'fhand' is not defined

python count1.py

Enter the file name: romeo.txt

*# Looping and dictionaries*

*# If you use a dictionary as the sequence in a for statement,*

*# it traverses the keys of the dictionary. For example, print\_hist prints each key and the corresponding value:*

​

**def** print\_hist(h):

**for** c **in** h:

print(c, h[c])

*# Here's what the output looks like:*

*# Again, the keys are in no particular order.*

h **=** histogram('parrot')

print\_hist(h)

​

p 1

a 1

r 2

o 1

t 1

*# If you want to print the keys in alphabetical order,*

*# you first make a list of the keys in the dictionary using the keys method available in dictionary objects,*

*# and then sort that list and loop through the sorted list,*

*# looking up each key printing out key/value pairs in sorted order as follows as follows:*

​

**def** print\_sorted\_hist(h):

lst **=** h.keys()

lst1 **=** sorted(lst)

**for** c **in** lst1:

print(c, h[c])

*# Here's what the output looks like:*

​

h **=** histogram('parrot')

​

print\_sorted\_hist(h)

a 1

o 1

p 1

r 2

t 1

A Note on Dictionary Key Objects A class whose objects are to be used as the key within a dictionary should consider implementing two special methods, these are **hash**() and **eq**(). The hash method is used to generate a hash number that can be used by the dictionary container and the equals method is used to test if two objects are equal. For example: The output from these two lines for an example run is: Python has two rules associated with these methods:

• If two objects are equal, then their hashes should be equal.

• In order for an object to be hashable, it must be immutable.

It also has two properties associated with the hashcodes of an object that should be adhered to: • If two objects have the same hash, then they are likely to be the same object.

• The hash of an object should be cheap to compute.

Why do you need to care about these methods? For built in type you do not need to worry; however for user defined c lasses/ types then if these types are to be used as keys within a dictionary then you should consider implementing these methods.

This is because a Dictionary uses • the hashing method to manage how values are organised and

• the equals method to check to see if a key is already present in the dictionary.

As an aside if you want to make a class something that cannot be used as a key in a dictionary, that is it is not hashable, then you can define this by setting the **hash**() method to None.

**class** NotHashableThing(object):

\_\_hash\_\_ **=** **None**

​

Dictionary Methods ,functions and how they are used in containers such as Dictionary.

clear() Removes all the elements from the dictionary

copy() Returns a copy of the dictionary

fromkeys() Returns a dictionary with the specified keys and values

get() Returns the value of the specified key

items() Returns a list containing the tuple for each key value pair

keys() Returns a list containing the dictionary’s keys

pop() Removes the element with the specified key

popitem() Removes the last inserted key-value pair

setdefault() Returns the value of the specified key. If the key does not exist: insert the key, with the specified value

update() Updates the dictionary with the specified key-value pairs

values() Returns a list of all the values in the dictionary

​