Python Programming
Module 6: Dictionaries

Learning objectives

1. Defining a dictionary and how to work with the information stored in a dictionary.
2. Accessing and modify individual elements in a dictionary, and how to loop through all of the information in a dictionary.
3. Loop through a dictionary’s key-value pairs, its keys, and its values.
4. Nest multiple dictionaries in a list, nest lists in a dictionary, and nest a dictionary inside a dictionary.

Dictionaries

A Simple Dictionary

Consider a game featuring aliens that can have different colours and point values. This simple dictionary stores information about a particular alien:

```
alien_0 = {'colour': 'green', 'points': 5}

print(alien_0['colour'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's colour and point value. The two print statements access and display that information, as shown here:

```
green
5
```

Working with dictionaries

A dictionary in Python is a collection of key-value pairs. Each key is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.

In Python, a dictionary is wrapped in braces, {}, with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'colour': 'green', 'points': 5}
```

A key-value pair is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.

The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'colour': 'green'}
```

This dictionary stores one piece of information about `alien_0`, namely the alien's colour. The string `'colour'` is a key in this dictionary, and its associated value is `'green'`.

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

```
alien_0 = {'colour': 'green'}
print(alien_0['colour'])
```

This returns the value associated with the key `'colour'` from the dictionary `alien_0: green`

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original `alien_0` dictionary with two key-value pairs:

```
  alien_0 = {'colour': 'green', 'points': 5}
```

Now you can access either the colour or the point value of `alien_0`. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
  alien_0 = {'colour': 'green', 'points': 5}

1 new_points = alien_0['points']
2 print("You just earned " + str(new_points) + " points!")
```

Once the dictionary has been defined, the code at `1` pulls the value associated with the key `'points'` from the dictionary. This value is then stored in the variable `new_points`. The line at `2` converts this integer value to a string and prints a statement about how many points the player just earned:

```
  You just earned 5 points!
```

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

To add two new pieces of information to the `alien_0` dictionary: the alien's x and y-coordinates, which will help us display the alien in a particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting its y-coordinate to positive 25, as shown here:

```
  alien_0 = {'colour': 'green', 'points': 5}
  print(alien_0)

1 alien_0['x_position'] = 0
2 alien_0['y_position'] = 25
  print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. At `1`, we add a new key-value pair to the dictionary: key `'x_position'` and value 0. We do the same for key `'y_position'` at `2`. When we print the modified dictionary, we see the two-additional key-value pairs:

```
  {'colour': 'green', 'points': 5}
  {'colour': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

The final version of the dictionary contains four key-value pairs. The original two specify colour and point value, and two more specify the alien's position. Notice that the order of the key-value pairs does not match the order in which we added them. Python doesn't care about the order in which you store each key-value pair; it cares only about the connection between each key and its value.

Starting with an Empty Dictionary

To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

```
alien_0 = {}

alien_0['colour'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Here we define an empty `alien_0` dictionary, and then add colour and point values to it. The result is the dictionary we've been using in previous examples:

```
{'colour': 'green', 'points': 5}3e
```

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

```
alien_0 = {'colour': 'green'}
print("The alien is " + alien_0['colour'] + ".")

alien_0['colour'] = 'yellow'
print("The alien is now " + alien_0['colour'] + ".")
```

We first define a dictionary for `alien_0` that contains only the alien's colour; then we change the value associated with the key `'colour'` to `'yellow'`. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the `del` statement to completely remove a key-value pair. All `del` needs is the name of the dictionary and the key that you want to remove.
For example, let's remove the key `'points'` from the `alien_0` dictionary along with its value:

```
  alien_0 = {'color': 'green', 'points': 5}
  print(alien_0)

1 del alien_0['points']
  print(alien_0)
```

The line at `1` tells Python to delete the key `'points'` from the dictionary `alien_0` and to remove the value associated with that key as well. The output shows that the key `'points'` and its value of 5 are deleted from the dictionary, but the rest of the dictionary is unaffected:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

You can also use a dictionary to store one kind of information about many objects. For example, say you want to poll a number of people and ask them what their favourite programming language is. A dictionary is useful for storing the results of a simple poll, like this:

```
favourite_languages = {
     'jen': 'python',
     'sarah': 'c',
     'edward': 'ruby',
     'phil': 'python',
     }
```

As you can see, we've broken a larger dictionary into several lines. Each key is the name of a person who responded to the poll, and each value is their language choice. When you know you'll need more than one line to define a dictionary, press enter after the opening brace. Then indent the next line one level (four spaces), and write the first key-value pair, followed by a comma. From this point forward when you press enter, your text editor should automatically indent all subsequent key-value pairs to match the first key-value pair.

Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair and indent it one level so it aligns with the keys in the dictionary. It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.

To use this dictionary, given the name of a person who took the poll, you can easily look up their favourite language:

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
       }

1 print("Sarah's favourite language is " +
2    favourite_languages['sarah'].title() +
3    ".")
```

To see which language Sarah chose, we ask for the value at:

```
favourite_languages['sarah']
```

This syntax is used in the print statement at 2, and the output shows Sarah's favourite language:

```
Sarah's favorite language is C.
```

Looping through a dictionary
A single Python dictionary can contain just a few key-value pairs or millions of pairs. Because a dictionary can contain large amounts of data, Python lets you loop through a dictionary. Dictionaries can be used to store information in a variety of ways; therefore, several different ways exist to loop through them. You can loop through all of a dictionary's key-value pairs, through its keys, or through its values.

Looping Through All Key-Value Pairs
Before we explore the different approaches to looping, let's consider a new dictionary designed to store information about a user on a website. The following dictionary would store one person's username, first name, and last name:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

What if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a for loop:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }

1 for key, value in user_0.items():
2     print("\nKey: " + key)
3     print("Value: " + value)
```

As shown at 1, to write a for loop for a dictionary, you create names for the two variables that will hold the key and value in each key-value pair. This code would work just as well if you had used abbreviations for the variable names, like this:

```
for k,
2 in user_0.items()
```

The second half of the for statement at 1 includes the name of the dictionary followed by the method `items()`, which returns a list of key-value pairs. The for loop then stores each of these pairs in the two variables provided. In the preceding example, we use the variables to print each key 2, followed by the associated value 3. The `"\n"` in the first print statement ensures that a blank line is inserted before each key-value pair in the output:

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

Notice again that the key-value pairs are not returned in the order in which they were stored, even when looping through a dictionary. Python doesn't care about the order in which key-value pairs are stored; it tracks only the connections between individual keys and their values.

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
     }
1 for name, language in favourite_languages.items():
2    print(name.title() + "'s favourite language is " +
           language.title() + ".")
```

The code at 1 tells Python to loop through each key-value pair in the dictionary. As it works through each pair the key is stored in the variable name, and the value is stored in the variable language. These descriptive names make it much easier to see what the print statement at 2 is doing.
Now, in just a few lines of code, we can display all of the information from the poll:

```
Jen's favourite language is Python.
Sarah's favourite language is C.
Phil's favourite language is Python.
Edward's favourite language is Ruby.
```

This type of looping would work just as well if our dictionary stored the results from polling a thousand or even a million people.

Looping Through All the Keys in a Dictionary

The `keys()` method is useful when you don’t need to work with all of the values in a dictionary. Let’s loop through the `favourite_languages` dictionary and print the names of everyone who took the poll:

```
  favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

1 for name in favourite_languages.keys():
     print(name.title())
```

The line at `1` tells Python to pull all the keys from the dictionary `favourite_languages` and store them one at a time in the variable name. The output shows the names of everyone who took the poll:

```
Jen
Sarah
Phil
Edward
```

Looping through the keys is actually the default behaviour when looping through a dictionary, so this code would have exactly the same output if you wrote . . .

```
for name in favourite_languages:
```

rather than . . .

```
for name in favourite_languages.keys():
```

You can choose to use the `keys()` method explicitly if it makes your code easier to read, or you can omit it if you wish.
You can access the value associated with any key you care about inside the loop by using the current key. Let’s print a message to a couple of friends about the languages they chose. We’ll loop through the names in the dictionary as we did previously, but when the name matches one of our friends, we’ll display a message about their favourite language:

```
favourite_languages = {
     'jen': 'python',
     'sarah': 'c',
     'edward': 'ruby',
     'phil': 'python',
        }

 1 friends = ['phil', 'sarah']
   for name in favourite_languages.keys():
       print(name.title())

 2    if name in friends:
            print(" Hi " + name.title() +
               ", I see your favorite language is " +
 3              favorite_languages[name].title() + "!")
```

At 1, we make a list of friends that we want to print a message to. Inside the loop, we print each person's name. Then at 2, we check to see whether the name we are working with is in the list friends. If it is, we print a special greeting, including a reference to their language choice. To access the favourite language at 3, we use the name of the dictionary and the current value of name as the key. Everyone's name is printed, but our friends receive a special message:

```
Edward
Phil
    Hi Phil, I see your favourite language is Python!
Sarah
    Hi Sarah, I see your favourite language is C!
Jen
```

You can also use the `keys()` method to find out if a particular person was polled. This time, let's find out if `Erin` took the poll:

```
  favourite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'edward': 'ruby',
       'phil': 'python',
       }

1 if 'erin' not in favourite_languages.keys():
      print("Erin, please take our poll!")
```

The `keys()` method isn't just for looping: It actually returns a list of all the keys, and the line at 1 simply checks if `'erin'` is in this list. Because she's not, a message is printed inviting her to take the poll:

```
Erin, please take our poll!
```

Looping Through a Dictionary's Keys in Order
One way to return items in a certain order is to sort the keys as they're returned in the for loop. You can use the `sorted()` function to get a copy of the keys in order:

```
favourite_languages = {
     'jen': 'python',
     'sarah': 'c',
     'edward': 'ruby',
     'phil': 'python',
     }

for name in sorted(favourite_languages.keys()):
   print(name.title() + ", thank you for taking the poll.")
```

This `for` statement is like other for statements except that we've wrapped the `sorted()` function around the `dictionary.keys()` method. This tells Python to list all keys in the dictionary and sort that list before looping through it. The output shows everyone who took the poll with the names displayed in order:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Looping Through All Values in a Dictionary
If you are primarily interested in the values that a dictionary contains, you can use the `values()` method to return a list of values without any keys. For example, say we simply want a list of all languages chosen in our programming language poll without the name of the person who chose each language:

```
favourite_languages = {
  'jen': 'python',
  'sarah': 'c',
  'edward': 'ruby',
  'phil': 'python',
  }

print("The following languages have been mentioned:")
for language in favourite_languages.values():
   print(language.title())
```

The `for` statement here pulls each value from the dictionary and stores it in the variable language. When these values are printed, we get a list of all chosen languages:

```
The following languages have been mentioned:
Python
```

```
  C
  Python
  Ruby
```

This approach pulls all the values from the dictionary without checking for repeats. That might work fine with a small number of values, but in a poll with a large number of respondents, this would result in a very repetitive list. To see each language chosen without repetition, we can use a `set`. A set is similar to a list except that each item in the set must be unique:

```
  favourite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
     }

  print("The following languages have been mentioned:")
1 for language in set(favourite_languages.values()):
     print(language.title())
```

When you wrap `set()` around a list that contains duplicate items, Python identifies the unique items in the list and builds a set from those items. At 1, we use `set()` to pull out the unique languages in `favourite_languages.values()`.
The result is a non-repetitive list of languages that have been mentioned by people taking the poll:
The following languages have been mentioned:

```
  Python
  C
  Ruby
```

Nesting

A List of Dictionaries

The following code builds a list of three aliens:

```
  alien_0 = {'colour': 'green', 'points': 5}
  alien_1 = {'colour': 'yellow', 'points': 10}
  alien_2 = {'colour': 'red', 'points': 15}

1 aliens = [alien_0, alien_1, alien_2]

  for alien in aliens:
     print(alien)
```

We first create three dictionaries, each representing a different alien. At 1 we pack each of these dictionaries into a list called aliens. Finally, we loop through the list and print out each alien:

```
  {'colour': 'green', 'points': 5}
  {'colour': 'yellow', 'points': 10}
  {'colour': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example we use `range()` to create a fleet of 30 aliens:

```
  # Make an empty list for storing aliens.
  aliens = []

  # Make 30 green aliens.
1 for alien_number in range(30):
2    new_alien = { 'colour': 'green', 'points': 5, 'speed': 'slow' }
3    aliens.append(new_alien)

  # Show the first 5 aliens:
4 for alien in aliens[:5]:
      print(alien)
  print("...")

  # Show how many aliens have been created.
5 print("Total number of aliens: " + str(len(aliens)))
```

This example begins with an empty list to hold all of the aliens that will be created. At `1`, `range()` returns a set of numbers, which just tells Python how many times we want the loop to repeat. Each time the loop runs we create a new alien `2` and then append each new alien to the list aliens `3`. At `4`, we use a slice to print the first five aliens, and then at `5` we print the length of the list to prove we've actually generated the full fleet of 30 aliens:

```
  {'speed': 'slow', 'colour': 'green', 'points': 5}
  {'speed': 'slow', 'colour': 'green', 'points': 5}
  {'speed': 'slow', 'colour': 'green', 'points': 5}
  {'speed': 'slow', 'colour': 'green', 'points': 5}
  {'speed': 'slow', 'colour': 'green', 'points': 5}
  ...

  Total number of aliens: 30
```

These aliens all have the same characteristics, but Python considers each one a separate object, which allows us to modify each alien individually.
How might you work with a set of aliens like this? Imagine that one aspect of a game has some aliens changing colour and moving faster as the game progresses. When it's time to change colours, we can use

a for loop and an if statement to change the colour of aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range (0,30):
    new_alien = {'colour': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[0:3]:
    if alien['colour'] == 'green':
         alien['colour'] = 'yellow'
         alien['speed'] = 'medium'
         alien['points'] = 10

# Show the first 5 aliens:
for alien in aliens[0:5]:
    print(alien)
print("...")
```

Because we want to modify the first three aliens, we loop through a slice that includes only the first three aliens. All of the aliens are green now but that won't always be the case, so we write an if statement to make sure we're only modifying green aliens. If the alien is green, we change the colour to `'yellow'`, the speed to `'medium'`, and the point value to 10, as shown in the following output:

```
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'medium', 'colour': 'yellow', 'points': 10}
{'speed': 'slow', 'colour': 'green', 'points': 5}
{'speed': 'slow', 'colour': 'green', 'points': 5}
...
```

You could expand this loop by adding an elif block that turns yellow aliens into red, fast-moving ones worth 15 points each. Without showing the entire program again, that loop would look like this:

```
for alien in aliens[0:3]:
      if alien['colour'] == 'green':
           alien['colour'] = 'yellow'
           alien['speed'] = 'medium'
           alien['points'] = 10
      elif alien['colour'] == 'yellow':
           alien['colour'] = 'red'
           alien['speed'] = 'fast'
```

```
            alien['points'] = 15
```

It's common to store a number of dictionaries in a list when each dictionary contains many kinds of information about one object.

A List in a Dictionary

Rather than putting a dictionary inside a list, it's sometimes useful to put a list inside a dictionary. For example, consider how you might describe a pizza that someone is ordering. If you were to use only a list, all you could really store is a list of the pizza's toppings. With a dictionary, a list of toppings can be just one aspect of the pizza you're describing.

In the following example, two kinds of information are stored for each pizza: a type of crust and a list of toppings. The list of toppings is a value associated with the key `'toppings'`. To use the items in the list, we give the name of the dictionary and the key `'toppings'`, as we would any value in the dictionary. Instead of returning a single value, we get a list of toppings:

```
  # Store information about a pizza being ordered.
1 pizza = {
       'crust': 'thick',
       'toppings': ['mushrooms', 'extra cheese'],
       }

  # Summarize the order.
2 print("You ordered a " + pizza['crust'] + "-crust pizza "
     + "with the following toppings:")

3 for topping in pizza['toppings']:
     print("\t" + topping)
```

We begin at `1` with a dictionary that holds information about a pizza that has been ordered. One key in the dictionary is `'crust'`, and the associated value is the string `'thick'`. The next key, `'toppings'`, has a list as its value that stores all requested toppings. At `2`, we summarize the order before building the pizza. To print the toppings, we write a for loop `3`. To access the list of toppings, we use the key `'toppings'`, and Python grabs the list of toppings from the dictionary.

The following output summarizes the pizza that we plan to build:

```
  You ordered a thick-crust pizza with the following toppings:
     mushrooms
     extra cheese
```

In the earlier example of favourite programming languages, if we were to store each person's responses in a list, people could choose more than one favourite language. When we loop through the dictionary, the value associated with each person would be a list of languages rather than a single language. Inside the dictionary's for loop, we use another for loop to run through the list of languages associated with each person:

```
1 favourite_languages = {
        'jen': ['python', 'ruby'],
        'sarah': ['c'],
        'edward': ['ruby', 'go'],
        'phil': ['python', 'haskell'],
        }

2 for name, languages in favourite_languages.items():
      print("\n" + name.title() + "'s favourite languages are:")
3     for language in languages:
           print("\t" + language.title())
```

As you can see at 1 the value associated with each name is now a list. When we loop through the dictionary at 2, we use the variable name languages to hold each value from the dictionary, because we know that each value will be a list. Inside the main dictionary loop, we use another for loop 3 to run through each person's list of favourite languages:

```
Jen's favourite languages are:
   Python
   Ruby
Sarah's favourite languages are:
   C

Phil's favourite languages are:
   Python
   Haskell

Edward's favourite languages are:
   Ruby
   Go
```

You should not nest lists and dictionaries too deeply. If you're nesting items much deeper than what you see in the preceding examples or you're working with someone else's code with significant levels of nesting, most likely a simpler way to solve the problem exists.

A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the user names as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username. In the following listing, we store three pieces of information about each user: their first name, last name, and location. We'll access this information by looping through the usernames and the dictionary of information associated with each username:

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },
    'mcurie': {
           'first': 'marie',
           'last': 'curie',
           'location': 'paris',
           },

     }

1 for username, user_info in users.items():
2    print("\nUsername: " + username)
3    full_name = user_info['first'] + " " + user_info['last']
         location = user_info['location']

4    print("\tFull name: " + full_name.title())
     print("\tLocation: " + location.title())
```

We first define a dictionary called users with two keys: one each for the usernames `'aeinstein'` and `'mcurie'`. The value associated with each key is a dictionary that includes each user's first name, last name, and location. At `1` we loop through the users dictionary. Python stores each key in the variable username, and the dictionary associated with each username goes into the variable `user_info`. Once inside the main dictionary loop, we print the username at `2`.
At `3` we start accessing the inner dictionary. The variable `user_info`, which contains the dictionary of user information, has three keys: `'first'`, `'last'`, and `'location'`. We use each key to generate a neatly formatted full name and location for each person, and then print a summary of what we know about each user `4`:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton
Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Notice that the structure of each user's dictionary is identical. Although not required by Python, this structure makes nested dictionaries easier to work with. If each user's dictionary had different keys, the code inside the for loop would be more complicated.

Exercises

1. Write a Python script to sort (ascending and descending) a dictionary by value.

2. Write a Python script to add a key to a dictionary.
   Sample Dictionary : {0: 10, 1: 20}
   Expected Result : {0: 10, 1: 20, 2: 30}

3. Write a Python script to check if a given key already exists in a dictionary.

4. Write a Python program to iterate over dictionaries using for loops.

5. Write a Python script to generate and print a dictionary that contains a number (between 1 and n) in the form (x, x*x).
   Sample Dictionary : (n = 5) :
   Expected Output : {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

6. Write a Python program to remove a key from a dictionary.

7. Write a Python program to map two lists into a dictionary.

8. Write a Python program to sort a dictionary by key.

9. Write a Python program to get the maximum and minimum value in a dictionary.

10. Write a Python program to remove duplicates from Dictionary.

11. Write a Python program to check a dictionary is empty or not.

12. Write a Python program to print all unique values in a dictionary.
    Sample Data : `[{"V":"S001"}, {"V": "S002"}, {"VI": "S001"}, {"VI": "S005"}, {"VII":"S005"}, {"V":"S009"}, {"VIII":"S007"}]`
    Expected Output : `Unique Values: {'S005', 'S002', 'S007', 'S001', 'S009'}`

13. Write a Python program to create and display all combinations of letters, selecting each letter from a different key in a dictionary.
    Sample data : `{'1':['a','b'], '2':['c','d']}`
    Expected Output :

```
ac
ad
bc
bd
```

14. Write a Python program to create a dictionary from a string.
    Note : Track the count of the letters from the string.
    Sample string : `'okarango'`
    Expected output : `{'o': 2, 'k': 1, 'a': 2, 'r': 1, 'n': 1, 'g': 1}`

15. Write a Python program to print a dictionary in table format.

16. Write a Python program to count the values associated with key in a dictionary.
    Sample data : `= [{'id': 1, 'success': True, 'name': 'Lary'}, {'id': 2, 'success': False, 'name': 'Rabi'}, {'id': 3, 'success': True, 'name': 'Alex'}]`
    Expected result : Count of how many dictionaries have success as `True`.

17. Write a Python program to count number of items in a dictionary value that is a list.