Chapter 3: Strings, Lists, Dictionaries, and Tuples

Python uses strings, lists, dictionaries, and tuples as collections of elements to accomplish certain tasks. In this chapter, you'll learn these four types of collections one by one. You'll also see examples of their uses.

Strings

Even though strings are variables, not collections of elements, we can treat strings as a sequence of characters and there are similarities between strings and lists, dictionaries, and tuples.

We will discuss how elements in a string are indexed, how to slice them, and how to join multiple strings together to form a new string.

String Indexing

The characters in string are indexed from left to right as 0, 1, 2, and so on. Note that Python is using 0 indexing, so the first element is indexed as 0 instead of 1. The characters in the string can be letters, numbers, white space, or special characters.

You can access characters in a string using the bracket operator. Consider these two lines of code:

```
msg= "hello"
print(msg[1])
```

and the output is

```
e
```

since *e* is the second character in the string "hello".

Python also uses negative indexing. The last character in the string `msg` can be accessed by `msg[-1]`, and the second last one by `msg[-2]` and so on. This is useful when you have a long string and you don't know the position counting from left, but you know the position counting backwards. In such cases, it's easier if you try to locate characters at the end of the string.

For example, if you want to find out the third to last character of the string `msg`, you can use the following line of code:

```
print(msg[-3])
```

and the output is

```
l
```

String Slicing

You can slice the string by taking out a subset of characters using the square bracket operator.

For example, if you run the following two lines of code

```
msg= "hello"
print(msg[0:3])
```

you'll see an output of

```
hel
```

The code `msg[a:b]` gives you the substring from position *a* to position *b* in the string *msg*, where the character in position *a* is included in the substring but the character in position *b* is not. Therefore, `msg[0:3]` produces a substring of with the first three characters in the string *msg*.

NOTE: If you omit the starting position when slicing a string, the default starting position is the first character. If you omit the ending position, the default is the last character. Therefore, `msg[:3]` *gives you the same result as* `msg[0:3]`*, and* `msg[0:]` *is equivalent to* `msg[:]` *(both produce the original string).*

String Methods

Several string methods are worth mentioning since they are commonly used in Python. You'll use them quite often later in this course.

The `replace()` Method

The `replace()` method replaces certain characters or substrings in the string with other characters. It takes two arguments: the first argument is the character that you want to be replaced, and the second the character to replace it.

For example, here is a program that involves the method:

```
inp = "University of Kentucky"
inp1 = inp.replace(' ', '+')
print(inp1)
```

The program uses the `replace()` method to replace all empty space with the plus sign. The output from the above program is

```
University+of+Kentucky
```

The `lower()` Method

The `lower()` method converts all upper case letters in the string to lower case ones. Since Python strings are case sensitive, converting all letters to lower cases is convenient for matching strings so that we will not get error messages because one string uses upper cases and the other uses lower cases in the same places.

Imaging you want to find the term “department of education” in a large document. However, in some places the term appears as “Department of Education” while in other places “department of education”. You can use the `lower()` method to convert the term to an all lower case string to avoid mismatch.

The program below shows how to match two strings using the `lower()` method:

```
inp = "Department of Education"
inp1 = "department of education"
print(inp.lower() == inp1.lower())
```

The program tests if the two strings *inp* and *inp1* are the same if we convert both into all lower cases. The output from the above program is

```
True
```

which means the two strings are the same if we ignore capitalization.

The `find()` Method

You can use the string method `find()` to locate the position of a character in a string. The method returns the index of the character in the string.

Enter the following lines of code into the Spyder editor and save it as *ExtractLastname.py* and run it. The program is also available on Canvas:

```
email= "John.Smith@uky.edu"
    pos1=email.find(".")     #❶
print(pos1)
    pos2=email.find("@")     #❷
print(pos2)
    LastName=email[(1+pos1):pos2]   #❸
print(LastName)
```

The string variable *email* has a patten: it consists of the first name, the dot ("."), and the last name, followed by "@uky.edu." We can use this pattern to locate the positions of the dot and the at sign ("@"). Then we can retrieve the last name based on those two positions using string slicing.

Line ❶ produces the position of "." in the email and define it as a variable *pos1*. At ❷, we find the position of "@" and define it as *pos2*. Line ❸ slices the string and takes characters between the two positions. The returned substring is saved as a variable *LastName*.

If you run the program, you'll have the following output:

```
4
10
Smith
```

The output says that the indexes of "." and "@" in the email are 4 and 10, respectively, and the last name is "Smith."

You can also use the string method `find()` to locate a substring in a string. The method returns the starting position of the substring in the original string (that is, the position of the first character in the substring).

For example, if you run the following lines of code:

```
email= "John.Smith@uky.edu"
pos=email.find("uky.edu")
print(pos)
```

You'll have the following output:

```
11
```

The output says that the substring "uky.edu" starts in the 12th character in the email.

NOTE: If a character or a substring is not in a string, the output is -1 instead of an error message. For example, `email.find("$")` *will give you an output of -1. You can use this feature to identify cases where something is not in a string.*

The `split()` Method

The string method `split()` is useful when you parse text. It splits a string into multiple substrings using the specified separator.

Enter the following code in Spyder and run it:

```
msg="Please think of an integer"
words=msg.split()
print(words)
```

and the output is

```
['Please', 'think', 'of', 'an', 'integer']
```

The default delimiter (a fancy name for separator) is a white space (' '). You can also specify the delimiter when you use the `split()` method. Let's revisit the example of extracting the last name from an email address, and name the new program *SplitString.py*:

```
email="John.Smith@uky.edu"
names=email.split("@")  #①
print(names)
firstlast=names[0].split(".")   #②
print("last name is", firstlast[1])   #③
```

The above program first splits the email into two parts using "@" as the delimiter ❶. As a result, *names* is a list of two substrings: *'John.Smith'* and *'uky.edu'*. The program then splits the first part *'John.Smith'* into the first name and the last name using *'.'* as the delimiter, and save them in a list *firstlast* ❷. Finally, we print out the second element in the list *firstlast* as the last name ❸.

The output is

```
['John.Smith', 'uky.edu']
last name is Smith
```

The `join()` Method

Now that you know how to separate a string into substrings using the `split()` method, you may wonder if you can do the opposite: combine several strings into a larger string.

It turns out you can: the string method `join()` will help you with that. The program *JoinString.py* below uses the method:

```
mylink = ('&')  #❶
strlist = ['University', 'of', 'Kentucky']  #❷
JoinedString = mylink.join(strlist)  #❸
print(JoinedString)
```

We first define the ampersand ("&") as a variable *mylink*, to be used as our delimiter ❶. The *strlist* is a list of the three words that we want to join together ❷. We then use the *join()* method to combined the three words into one single string ❸. Note that you need to put the *join()* method after the delimiter. Finally, we print out the joined string.

The output from the program is:

```
University&of&Kentucky
```

Lists

A *list* is a collection of values separated by commas (","). The values in a list are called elements or items, and they can be either a value, a variable, or another list.

Create A List

To create a new list, you can simply put the elements in square brackets. The following is an example:

```
lst = [1, "a", "hello"]
```

We define the list name as *lst*, which has three elements: an integer number 1, and two strings: `"a"` and `"hello"`.

Note that `list()` is a built-in function in Python and you cannot use *list* as a variable name or list name.

You can also create an empty list by using a pair of square brackets with nothing in it.

```
lst1=[]
```

Another way to create an empty list is by using the `list()` function

```
lst2=list()
```

Access the Elements in A List

You can access the elements of a list using the bracket operator. For example,

```
lst=[1, "a", "hello"]
print(lst[2])
```

will produce

```
hello
```

In the program, `lst[2]` refers to the third element in the list, which is "hello".

You can traverse the elements of a list using a loop:

```
for x in range(len(lst)): print(lst[x])
```

leads to the following output:

```
1
a
hello
```

Here we use the built-in function `len()` to return the length of the list, which is 3 in this case. The built-in function `range()` returns values 0, 1, and 2 here. In the above program, you can either put `print(lst[x])` in the same line as the `for` statement or put it in the next line. If you put it in the next line, you need to indent it, as follows:

```
for x in range(len(lst)):
    print(lst[x])
```

If you have more than one line of code inside the `for` loop, you don't have a choice but to put them in the following lines and indent them. The idea is for the program to know which commands are in the `for` loop.

A List of Lists

A list can use lists as its elements.

A useful feature about a list of lists is that you can map element positions to coordinates on a 2-dimensional space. Here is one example:

```
llst=[[1,2,3,5],
          [2,2,6,8],
          [2,3,5,9],
          [3,5,4,7],
          [1,3,5,0]]
print('the value of llst[1][2] is ', llst[1][2])
print('the value of llst[3][2] is ', llst[3][2])
print('the value of llst[1][3] is ', llst[1][3])
```

The output from the above program is

```
the value of llst[1][2] is  6
the value of llst[3][2] is  4
the value of llst[1][3] is  8
```

To find out the value of `llst[1][2]`, we fist try to find out what is `llst[1]`. Since `llst` is a list of five lists, with the second element as `[2, 2, 6, 8]`, we find out `llst[1] = [2, 2, 6, 8]`. The third element of `[2, 2, 6, 8]` is 6, hence `llst[1][2] = 6`.

Now let us draw a corresponding picture on a 2-dimentional space as in Figure 1.

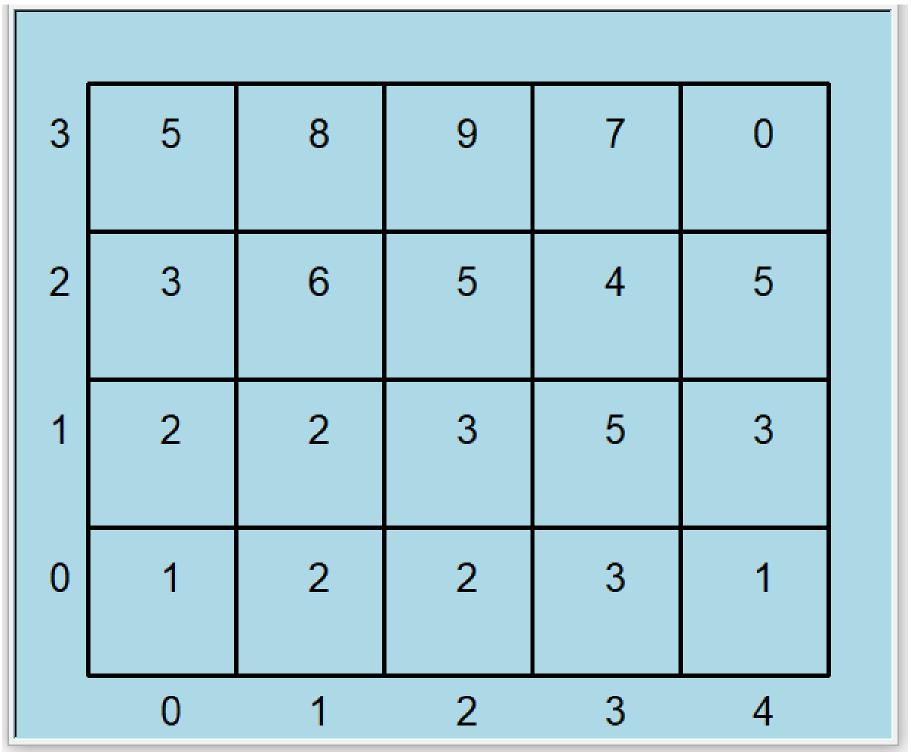

Figure 1: Map a list of lists to a 2-dimensional space

We can easily map the value of `llst[x][y]` to the value in the *x-th* column and *y-th* row. The column numbers (i.e., the x values) are 0, 1, 2, 3, and 4 at the bottom of the screen, while the row numbers (i.e., the y values) are 0, 1, 2, and 3 at the left of the screen.

It is straightforward to locate the elements in the list of lists based on the coordinates on the screen. For example, to find `llst[3][2]`, we go to the column that is marked "3" and look at the corresponding row value that is marked "2", which is 4. Similarly, to find `llst[1][3]`, we go to the column that is marked "1" and look at the corresponding row value that is marked "3", and find out that the answer is 8.

Therefore, a list of lists is useful in graphics and animation videos to help us locate a point on the two-dimensional screen.

Add or Multiple Lists

You can use the plus (+) or multiplication (*) operator on lists, but not in the mathematical sense.

For example, if you run the following lines of code:

```
lst=[1, "a", "hello"]
print(lst + lst)
print(lst * 3)
```

You will see the following output:

```
 [1, "a", "hello", 1, "a", "hello"]
[1, "a", "hello", 1, "a", "hello", 1, "a", "hello"]
```

The plus operator joins two lists together into a larger list. The multiplication operator repeats the elements in the list. If you multiply a list by 3, the elements in the list will appear three times as a result.

List Methods

You can use list methods to operate on a list. We will introduce several useful list methods here so that you can use them for complicated tasks.

Specifically, you'll learn these six list methods: `enumerate()`, `append()`, `remove()`, `index()`, `count()`, and `sort()`.

The `enumerate()` Method

You can use the `enumerate()` method to print out all elements in a list and their corresponding indexes.

Assume we have the following list, *names*:

```
names=['Adam','Mike','Kate']
```

which contains three names. The following line of code

```
for x, name in enumerate(names): print(x, name)
```

will generate an output as follows:

```
0 Adam
1 Mike
2 Kate
```

The program prints out the index and the value of each element in the list, based on 0 indexing. The program tells you that the first element in the list *names* is Adam, the second is Mike, and so on.

If you want the start value to be 1 instead of 0, you can put `start=1` as an option inside the `enumerate()` function so that the first element is indexed as 1.

Enter the following into the Spyder editor and run it:

```
names=['Adam','Mike','Kate']
for x, name in enumerate(names, start=1):
    print(x, name)
```

The output is as follows:

```
1 Adam
2 Mike
3 kate
```

As you can see, the first element is indexed as 1, and the second 2, and so on.

The `append()` method

You can append an element to the end of a list by using the `append()` method.

Consider the program *ListAppend.py* below:

```
lst=[1, "a", "hello"]
lst.append(2) #❶
print(lst)
```

The above program is appending another element, 2, to the existing list, *lst*. The program will produce the following output:

```
[1, "a", "hello", 2]
```

As you can see, the new list *lst* now has four elements, with newly added element at the end of the list.

You can append only one element at a time. Appending two elements will lead to an error message. However, if you put the two elements inside square brackets as a list, you can append that list as an element to the original list.

If you change line ❶ in the program *ListAppend.py* to the following line:

```
lst.append(2, 3)
```

You'll lead to the following error message:

```
TypeError: append() takes exactly one argument (2 given)
```

On the other hand, if you add square brackets around the two number as follows:

```
lst.append([2, 3])
```

You'll get the following output:

```
[1, "a", "hello", [2, 3]]
```

The new list has four elements.

To add two or more elements to the existing list, you should use the plus operator. For example, to add 2 and 3 as two separate elements to the list, you can use the following line of code instead:

```
lst + [2, 3]
```

The `remove()` method

You can remove an element from a list by using the `remove()` method. Here is an example:

```
lst=[1, "a", "hello", 2]
lst.remove("a")
print(lst)
```

We are removing the element “a” from the list. After running the program you’ll have the following output:

```
[1, "hello", 2]
```

The new list has no element “a” any more.

You can remove only one element at a time.

The `index()` method

You can find the position of an element in a list by using the `index()` method. Here is an example:

```
lst=[1, "a", "hello", 2]
print(lst.index("a"))
```

The above program will produce the following output:

```
1
```

The result tells you that the element “a” has an index of 1 in the list. That is, it is the second element in the list.

The `count()` method

You can count how many times an element appears in a list by using the `count()` method. Here is an example:

```
lst=[1, "a", "hello", 2, 1]
print(lst.count(1))
print(lst.count("a"))
```

After running the program, you’ll see the following output:

```
2
1
```

The `count()` method tells us that the element 1 has appeared in the list twice, while the element “a” has appeared in the list once.

The `sort()` method

You can sort the elements in a list by using the `sort()` method.

The elements in the list must be the same type (or at least convertible to the same type). For example, if you have both integers and strings in a list, trying to sort the list will lead to the following error message:

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

Numbers are sorted from the smallest to the largest. However, putting `reverse=True` inside the method as an option will reverse the ordering. Here is an example:

```
lst=[5, 47, 12, 9, 4, -1]
lst.sort()
print(lst)
lst.sort(reverse=True)
print(lst)
```

The above program will produce the following output:

```
[-1, 4, 5, 9, 12, 47]
[47, 12, 9, 5, 4, -1]
```

Letters are sorted by the alphabetic order, and they come after numbers. Consider this example:

```
lst=["a", "hello", "ba", "ahello", "2", "-1"]
lst.sort()
print(lst)
```

and the output is:

```
['-1', '2', 'a', 'ahello', 'ba', 'hello']
```

Use built-in functions on lists

Several Python built-in functions can be used on lists directly, including `min()`, `max()`, `sum()`, and `len()`. They produce the minimum value, the maximum value, the total sum, and the length of the list, respectively.

Here is an example that uses all the above four functions:

```
lst=[5, 47, 12, 9, 4, -1]
print("the range of the numbers is", max(lst)-min(lst))
print("the mean of the numbers is", sum(lst)/len(lst))
```

and the output is:

```
the range of the numbers is 48
the mean of the numbers is 12.666666666666666
```

The list() function

You can use the *list()* function to convert a string to a list of characters:

Here is an example with these functions:

```
msg="hello"
letters=list(msg)
print(letters)
```

and the output is:

```
['h', 'e', 'l', 'l', 'o']
```

Dictionaries

What Is A Dictionary?

A dictionary is a list of key-value pairs. The elements are put inside curly brackets, {}. Here is one example:

```
scores = {'blue':10, 'white':12}
```

In this example, the dictionary *scores* has two elements, separated by a comma: the first element is `'blue':10` and the second `'white':12`. Further, `'blue'` and `'white'` are keys, while `10` and `12` are the corresponding values. Each element is a key-value pair, connected by a colon (“:”).

To create an empty dictionary, you can use `dict()` function or simply use a pair of curly brackets with nothing in it.

```
Dict1=dict()
Dict2={}
```

You can add a new element to the existing dictionary as follows:

```
Dict3={}
Dict3['yellow'] = 6
Print(Dict3)
```

The line `Dict3['yellow'] = 6` assigns a value of 6 to the key `'yellow'`. As a result, the new `Dict3` contains an element `'yellow':6`.

Access Values in A Dictionary

You can access values in a dictionary using the bracket operator. The key value in each pair are the indexes.

For example, we can access the values in the dictionary *scores* as follows:

```
print(scores['blue'])
print(scores['white'])
```

The above commands will give you the following results:

```
10
12
```

Another way to access the values in a dictionary is to use the `get()` method. The advantages of using the `get()` method instead of the bracket operator are: if the key is not in the dictionary, there is no error message, and you can assign a default value in such cases.

Consider the following program, *GetMethod.py*:

```
scores = {'blue':10, 'white':12}
print(scores['blue'])
print(scores['white'])
print(scores.get('yellow'))
print(scores.get('yellow',0))
```

The above program produces the following results:

```
10
12
None
0
```

Since the key `'yellow'` is not in the dictionary *scores*, the method `get('yellow')` returns a value of `None`. Further, when you put the option 0 in the method, `get('yellow', 0)` returns a value of 0.

Dictionary methods

You can use the `keys()` method to produce a list of all keys in a dictionary:

```
scores = {'blue':10, 'white':12}
teams=list(scores.keys())
print(teams)
```

The above program will lead to this output:

```
['blue', 'white']
```

Similarly, we can use the `values()` method to produce a list of all values in a dictionary:

```
points=list(scores.values())
print(points)
```

The output from above is:

```
[10, 12]
```

We can also use the `items()` method to get the list of key-value pairs.

```
print(list(Scores.items()))
```

produces the following result:

```
[('blue', 10), ('white', 12)]
```

How to Use Dictionaries

The values in a dictionary can be any type of variables, a list, or even another dictionary.

Here is an example of a dictionary that uses lists as values:

```
scores2 = {'blue':[5, 5, 10], 'white':[5, 7, 12]}
```

The value for each key is a 3-element list. Imagine the three values are points in the first half, second half, and total scores, respectively. To find out the points of the white team in the second half, you can simply do this:

```
print(scores2['white'][1])
```

The advantage of a dictionary is that its key can be any value, not necessarily an integer. This makes dictionaries useful in many situations. Here is an example to use a dictionary as a word counter, *MostFreqWord.py*:

```
news=(
'''The fastest growing segment of the U.S. electorate is seniors.
They supported President Trump in 2016 but aren’t squarely
in his camp as the 2020 campaign picks up.
In 2016, despite polling showing an advantage for Democrat Hillary Clinton,
 voters over 65 backed Mr. Trump in the presidential election by a margin,
 according to exit polls.
''')          #❶
wdcnt=dict()  #❷
wd=news.split()  #❸
for w in wd:     #❹
    wdcnt[w]=wdcnt.get(w,0)+1
print(wdcnt)
for w in list(wdcnt.keys()):   #❺
    if wdcnt[w]==max(list(wdcnt.values())):
        print(w)
```

The program first define *news* as a string variable with a short news story ❶. We then create an empty dictionary *wdcnt* ❷. At ❸, we split the string into a list of separate words. We then count the frequency of each word and store the information in the dictionary, with word as the key and the word count as the value ❹. In particular, we use the method `get()`. If a word is not already in the dictionary as a key, the second argument in the `get()` method assigns a value of 0 to the word.

At ❺, the program prints out the words that have the highest frequency. The result of the above program is as follows:

```
344
{'The': 1, 'fastest': 1, 'growing': 1, 'segment': 1, 'of': 1, 'the': 3,
'U.S.': 1, 'electorate': 1, 'is': 1, 'seniors.': 1, 'They': 1, 'supported':
1, 'President': 1, 'Trump': 2, 'in': 3, '2016': 1, 'but': 1, 'aren’t': 1,
'squarely': 1, 'his': 1, 'camp': 1, 'as': 1, '2020': 1, 'campaign': 1,
'picks': 1, 'up.': 1, 'In': 1, '2016,': 1, 'despite': 1, 'polling': 1,
'showing': 1, 'an': 1, 'advantage': 1, 'for': 1, 'Democrat': 1, 'Hillary': 1,
'Clinton,': 1, 'voters': 1, 'over': 1, '65': 1, 'backed': 1, 'Mr.': 1,
'presidential': 1, 'election': 1, 'by': 1, 'a': 1, 'margin,': 1, 'according':
1, 'to': 1, 'exit': 1, 'polls.': 1}
the
in
```

It turns out that the most frequent words in the above news are *the* and *in*, and each is used 3 times.

How to Switch Keys and Values in A Dictionary?

Sometimes you may want to switch the positions of keys and values in a dictionary.

Now let us take the term “dictionary” literally and suppose you have the following English to Spanish dictionary that uses the English word as the key and the Spanish translation as the value:

```
spanish={'one':"uno", 'two':"dos", 'three':"tres"}
```

You want to create a new dictionary that uses the Spanish word as the key and the English translation as the value. You can accomplish this by using the following line of code:

```
english={y:x for x,y in spanish.items()}
```

The command `x,y in spanish.items()` retrieves all the key-value pairs in the dictionary *spanish*. The command `y:x for x,y` switches the positions of the keys and values. You must put curly brackets around it so that the program treats it as a dictionary.

To verify, enter

```
print(english)
```

in the Spyder editor and run it, and you will have the following output:

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

How to Combine Two Dictionaries?

If you wish to combine two dictionaries into one large dictionary, you can accomplish it with the following line of code:

```
spanishenglish=(**spanish, **english)
```

The result is a new dictionary called *spanishenglish* with 6 elements in it: 3 pairs of words from the dictionary *spanish* and 3 pairs from the dictionary *english*.

Tuples

A *tuple* is a collection of values separated by commas, similar to a list. However, elements of a tuple cannot be changed (that is, they are immutable). We usually put elements of a tuple inside parentheses instead of square brackets to distinguish it from a list.

The following is an example of a tuple and the attempt to modify it:

```
tpl=(1,2,3,9,0)
tpl.append(4)
print(tpl)
```

The above code produces the following error message

```
AttributeError: 'tuple' object has no attribute 'append'
```

Because tuples are immutable, we cannot use methods such as `append()` or `remove()` on them. We cannot sort the elements in a tuple either.

The elements of a tuple are indexed by integers, and we can access them using the bracket operator. The following code

```
tpl=(1,2,3,9,0)
print(tpl[3])
print(tpl[1:4])
```

produces the output below:

```
9
(2, 3, 9)
```

You can, however, compare two tuples. The comparison of two tuples is determined by their first elements. If the first elements are the same, we go to the second elements to break the tie. If the second elements are also the same, we go to the third elements, and so on, until we find a difference.

Run the following lines of code in your Spyder editor:

```
lt=[(1,2),(3,9),(0,7),(1,0)]
lt.sort()
print(lt)
```

And you'll see the following output:

```
 [(0, 7), (1, 0), (1, 2), (3, 9)]
```

We can use the comparability of tuples for various purposes. Let's revisit the most frequent words example and use tuples to help us.

Run the following program in your Spyder editor:

```
news=(
'''The fastest growing segment of the U.S. electorate is seniors.
They supported President Trump in 2016 but aren’t squarely
in his camp as the 2020 campaign picks up.
In 2016, despite polling showing an advantage for Democrat Hillary Clinton,
voters over 65 backed Mr. Trump in the presidential election by a margin,
according to exit polls.
''')
wdcnt=dict()  #❶
words=news.split()
for word in words:
    wdcnt[word]=wdcnt.get(word,0)+1
lst=list()  #❷
for wd, cnt in list(wdcnt.items()):  #❸
    lst.append((cnt,wd))
lst.sort(reverse=True)   #❹
print(lst)
```

We first put the news story as a large string variable *news*. Starting at ❶, we count the frequency of each word and put them in the dictionary *wdcnt*. We then create an empty list *lst* ❷.

At ❸, we create a tuple for each word and frequency pair, with the frequency being the first element of the tuple. We sort the tuples in the list in reverse order ❹. As a result, the tuple with the most frequent count appears first. The program prints out the sorted list, which is shown below:

```
[(3, 'the'), (3, 'in'), (2, 'Trump'), (1, 'voters'), (1, 'up.'), (1, 'to'),
(1, 'supported'), (1, 'squarely'), (1, 'showing'), (1, 'seniors.'), (1,
'segment'), (1, 'presidential'), (1, 'polls.'), (1, 'polling'), (1, 'picks'),
(1, 'over'), (1, 'of'), (1, 'margin,'), (1, 'is'), (1, 'his'), (1,
'growing'), (1, 'for'), (1, 'fastest'), (1, 'exit'), (1, 'electorate'), (1,
'election'), (1, 'despite'), (1, 'campaign'), (1, 'camp'), (1, 'by'), (1,
'but'), (1, 'backed'), (1, 'as'), (1, 'aren't'), (1, 'an'), (1, 'advantage'),
(1, 'according'), (1, 'a'), (1, 'U.S.'), (1, 'They'), (1, 'The'), (1,
'President'), (1, 'Mr.'), (1, 'In'), (1, 'Hillary'), (1, 'Democrat'), (1,
'Clinton,'), (1, '65'), (1, '2020'), (1, '2016,'), (1, '2016')]
```

We can clearly tell that the most frequent words are *the* and *in*, with each appearing three times in the news story.

Summary

In this chapter, you learned four types of collections of elements in Python, namely, strings, lists, dictionaries, and tuples. You learned their properties and how to use them to accomplish certain tasks.

You'll these tools in later chapters for various tasks.

End of Chapter Exercises

1. The grades for the mid-term project of the 8 groups in a class are put in a list as Midterm = [95, 78, 77, 86, 90, 88, 81, 66]. Use Python built-in functions on the list to calculate the range (difference between the maximum and minimum values) and the average of the grades.
2. Assume inp= "University of Kentucky", find out inp[5:10], inp[-1], inp[:10], and inp[5:].
3. If email= John.Smith@uky.edu, what is email.find("y")?

4. llst=[[1,2,3,5],[2,2,6,8],[2,3,5,9],[3,5,4,7],[1,3,5,0]], what are the values of llst[2], llst[2][2], and llst[3][0]?
5. [1, "a", "hello", 2].remove(1)=?
6. [1, "a", "hello", 2].append(“hi”)=?
7. Scores2 = {'blue':[5, 5, 10], 'white':[5, 7, 12]}, what is Scores2['blue'][2]
8. tpl=(1,2,3,9,0), what is tpl[3:4]?
9. You have a list lst=[1, "a", "hello", 2]. Create a dictionary with four key-value pair: the key is the position of each element in *lst*, and the value is the element at the position.