6.2. Strings and Lists

Throughout the first chapters of this book we have used strings to represent words or phrases that we wanted to print out. Our definition was simple: a string is simply some characters inside quotes. In this chapter we explore strings in much more detail.

Additionally, we explore lists, which are very much like strings but can hold different types.

6.2.1. Strings

Strings can be defined as sequential collections of characters. This means that the individual characters that make up a string are in a particular order from left to right.

A string that contains no characters, often referred to as the **empty string**, is still considered to be a string. It is simply a sequence of zero characters and is represented by ‘’ or “” (two single or two double quotes with nothing in between).

6.2.2. Lists

A **list** is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

There are several ways to create a new list. The simplest is to enclose the elements in square brackets ( [ and ]).

[10, 20, 30, 40]

["spam", "bungee", "swallow"]

The first example is a list of four integers. The second is a list of three strings. As we said above, the elements of a list don’t have to be the same type. The following list contains a string, a float, an integer, and another list.

["hello", 2.0, 5, [10, 20]]

**Note**

WP: Don’t Mix Types!

You’ll likely see us do this in the textbook to give you odd combinations, but when you create lists you should generally not mix types together. A list of just strings or just integers or just floats is generally easier to deal with.

6.2.3. Tuples

A **tuple**, like a list, is a sequence of items of any type. The printed representation of a tuple is a comma-separated sequence of values, enclosed in parentheses. In other words, the representation is just like lists, except with parentheses () instead of square brackets [].

One way to create a tuple is to write an expression, enclosed in parentheses, that consists of multiple other expressions, separated by commas.

julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")

The key difference between lists and tuples is that a tuple is immutable, meaning that its contents can’t be changed after the tuple is created. We will examine the mutability of lists in detail in the chapter on [Mutability](https://fopp.umsi.education/runestone/static/fopp/TransformingSequences/Mutability.html#mutability).

To create a tuple with a single element (but you’re probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the (5) below as an integer in parentheses:

t = (5,)

print(type(t))

​

x = (5)

print(type(x))

​

**Check your understanding**

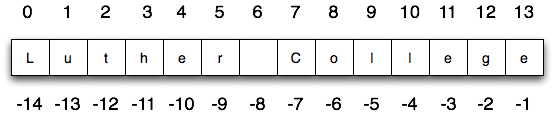
sequences-2-1: A list can contain only integer items.

Top of Form

A. False  
B. True

# 6.3. Index Operator: Working with the Characters of a String

The **indexing operator** (Python uses square brackets to enclose the index) selects a single character from a string. The characters are accessed by their position or index value. For example, in the string shown below, the 14 characters are indexed left to right from postion 0 to position 13.



It is also the case that the positions are named from right to left using negative numbers where -1 is the rightmost index and so on. Note that the character at index 6 (or -8) is the blank character.

school = "Luther College"

m = school[2]

print(m)

​

lastchar = school[-1]

print(lastchar)

​

The expression school[2] selects the character at index 2 from school, and creates a new string containing just this one character. The variable m refers to the result.

The letter at index zero of "Luther College" is L. So at position [2] we have the letter t.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets. Give it a try.

The expression in brackets is called an **index**. An index specifies a member of an ordered collection. In this case the collection of characters in the string. The index indicates which character you want. It can be any integer expression so long as it evaluates to a valid index value.

Note that indexing returns a string — Python has no special type for a single character. It is just a string of length 1.

## 6.3.1. Index Operator: Accessing Elements of a List or Tuple

The syntax for accessing the elements of a list or tuple is the same as the syntax for accessing the characters of a string. We use the index operator ( [] – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0. Any integer expression can be used as an index and as with strings, negative index values will locate items from the right instead of from the left.

Try to predict what will be printed out by the following code, and then run it to check your prediction. (Actually, it’s a good idea to always do that with the code examples. You will learn much more if you force yourself to make a prediction before you see the output.)

numbers = [17, 123, 87, 34, 66, 8398, 44]

print(numbers[2])

print(numbers[9-8])

print(numbers[-2])

​

prices = (1.99, 2.00, 5.50, 20.95, 100.98)

print(prices[0])

print(prices[-1])

print(prices[3-5])

​

**Check your understanding**

sequences-3-1: What is printed by the following statements?

s = "python rocks"

**print**(s[3])

Top of Form

A. t  
B. h  
C. c  
D. Error, you cannot use the [ ] operator with a string.

Bottom of Form

sequences-3-2: What is printed by the following statements?

s = "python rocks"

**print**(s[2] + s[-4])

Top of Form

A. tr  
B. to  
C. ps  
D. nn  
E. Error, you cannot use the [ ] operator with the + operator.

Bottom of Form

sequences-3-3: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

**print**(alist[5])

Top of Form

A. [ ]  
B. 3.14  
C. False  
D. "dog"

# 6.4. Introduction: Dictionaries

To provide an example of this new kind of datatype, we will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings and the values will also be strings.

One way to create a dictionary is to start with the empty dictionary and add **key-value pairs**. The empty dictionary is denoted {}.

**Check your understanding**

dictionaries-1-1: A dictionary is an unordered collection of key-value pairs.

Top of Form

A. False  
B. True

Bottom of Form

dictionaries-1-2: What is printed by the following statements?

mydict = {"cat":12, "dog":6, "elephant":23}

**print**(mydict["dog"])

Top of Form

A. 12  
B. 6  
C. 23  
D. Error, you cannot use the index operator with a dictionary

Assignment:

1. Create a dictionary that keeps track of the USA’s Olympic medal count. Each key of the dictionary should be the type of medal (gold, silver, or bronze) and each key’s value should be the number of that type of medal the USA’s won. Currently, the USA has 33 gold medals, 17 silver, and 12 bronze. Create a dictionary saved in the variable medals that reflects this information
2. You are keeping track of olympic medals for Italy in the 2016 Rio Summer Olympics! At the moment, Italy has 7 gold medals, 8 silver metals, and 6 bronze medals. Create a dictionary called olympics where the keys are the types of medals, and the values are the number of that type of medals that Italy has won so far. Bottom of Form

Bottom of Form

Bottom of Form

## 6.4.1. Dictionary methods

Dictionaries have a number of useful built-in methods. The following table provides a summary and more details can be found in the [Python Documentation](http://docs.python.org/py3k/library/stdtypes.html#mapping-types-dict).

| **Method** | **Parameters** | **Description** |
| --- | --- | --- |
| keys | none | Returns a view of the keys in the dictionary |
| values | none | Returns a view of the values in the dictionary |
| items | none | Returns a view of the key-value pairs in the dictionary |

As we saw earlier with strings and lists, dictionary methods use dot notation, which specifies the name of the method to the right of the dot and the name of the object on which to apply the method immediately to the left of the dot. The empty parentheses in the case of keys indicate that this method takes no parameters. If x is a variable whose value is a dictionary, x.keys is the method object, and x.keys() invokes the method, returning a view of the value.

The keys method returns the keys, not necessarily in the same order they were added to the dictionary or any other particular order.

inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

​

for akey in inventory.keys(): # the order in which we get the keys is not defined

print("Got key", akey, "which maps to value", inventory[akey])

​

ks = list(inventory.keys())

print(ks)

​

It’s so common to iterate over the keys in a dictionary that you can omit the keys method call in the for loop — iterating over a dictionary implicitly iterates over its keys.

inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

​

for k in inventory:

print("Got key", k)

​

The values and items methods are similar to keys. They return the objects which can be iterated over. Note that the item objects are tuples containing the key and the associated value.

inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

​

print(list(inventory.values()))

print(list(inventory.items()))

​

for k in inventory:

print("Got",k,"that maps to",inventory[k])

​

**Note**

Technically, .keys(), .values(), and .items() don’t return actual lists. Like the range function described previously, in python 3 they return objects that produce the items one at a time, rather than producing and storing all of them in advance as a list. Unless the dictionary has a whole lot of keys, this won’t make a difference for performance. In any case, as with the range function, it is safe for you to think of them as returning lists, for most purposes. For the python interpreter built into this textbook, they actually do produce lists. In a native python interpreter, if you print out type(inventory.keys()), you will find that it is something other than an actual list. If you want to get the first key, inventory.keys()[0] works in the online textbook, but in a real python interpreter, you need to make the collection of keys into a real list before using [0] to index into it: list(inventory.keys())[0].

The in and not in operators can test if a key is in the dictionary:

inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

print('apples' in inventory)

print('cherries' in inventory)

​

if 'bananas' in inventory:

print(inventory['bananas'])

else:

print("We have no bananas")

​

# 6.5. Length

The len function, when applied to a string, returns the number of characters in a string.

fruit = "Banana"

print(len(fruit))

​

To get the last letter of a string, you might be tempted to try something like this:

fruit = "Banana"

sz = len(fruit)

last = fruit[sz] # ERROR!

print(last)

​

That won’t work. It causes the runtime error IndexError: string index out of range. The reason is that there is no letter at index position 6 in "Banana". Since we started counting at zero, the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from the length. Give it a try in the example above.

fruit = "Banana"

sz = len(fruit)

lastch = fruit[sz-1]

print(lastch)

​

Typically, a Python programmer will access the last character by combining the two lines of code from above.

lastch = fruit[len(fruit)-1]

As with strings, the function len returns the length of a list (the number of items in the list). However, since lists can have items which are themselves sequences (e.g., strings), it important to note that len only returns the top-most length.

alist = ["hello", 2.0, 5]

print(len(alist))

print(len(alist[0]))

​

Note that alist[0] is the string "hello", which has length 5.

**Check your understanding**

sequences-5-1: What is printed by the following statements?

s = "python rocks"

**print**(len(s))

Top of Form

A. 11  
B. 12

Bottom of Form

sequences-5-2: What is printed by the following statements?

alist = [3, 67, "cat", 3.14, False]

**print**(len(alist))

Top of Form

A. 4  
B. 5

Bottom of Form

sequences-5-3: What is printed by the following statements?

L = [0.34, '6', 'SI106', 'Python', -2]

**print**(len(L[1:-1]))

Top of Form

A. 2  
B. 3  
C. 4  
D. 5

# 6.6. The Slice Operator

A substring of a string is called a **slice**. Selecting a slice is similar to selecting a character:

singers = "Peter, Paul, and Mary"

print(singers[0:5])

print(singers[7:11])

print(singers[17:21])

​

The slice operator [n:m] returns the part of the string from the n’th character to the m’th character, including the first but excluding the last. In other words, start with the character at index n and go up to but do not include the character at index m.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string.

fruit = "banana"

print(fruit[:3])

print(fruit[3:])

​

What do you think fruit[:] means?

## 6.6.1. List Slices

The slice operation we saw with strings also work on lists. Remember that the first index is the starting point for the slice and the second number is one index past the end of the slice (up to but not including that element). Recall also that if you omit the first index (before the colon), the slice starts at the beginning of the sequence. If you omit the second index, the slice goes to the end of the sequence.

a\_list = ['a', 'b', 'c', 'd', 'e', 'f']

print(a\_list[1:3])

print(a\_list[:4])

print(a\_list[3:])

print(a\_list[:])

​

## 6.6.2. Tuple Slices

We can’t modify the elements of a tuple, but we can make a variable reference a new tuple holding different information. Thankfully we can also use the slice operation on tuples as well as strings and lists. To construct the new tuple, we can slice parts of the old tuple and join up the bits to make the new tuple. So julia has a new recent film, and we might want to change her tuple. We can easily slice off the parts we want and concatenate them with the new tuple.

julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")

print(julia[2])

print(julia[2:6])

​

print(len(julia))

​

julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]

print(julia)

​

**Check your understanding**

sequences-6-1: What is printed by the following statements?

s = "python rocks"

**print**(s[3:8])

Top of Form

A. python  
B. rocks  
C. hon r  
D. Error, you cannot have two numbers inside the [ ].

Bottom of Form

sequences-6-2: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

**print**(alist[4:])

Top of Form

A. [ [ ], 3.14, False]  
B. [ [ ], 3.14]  
C. [ [56, 57, "dog"], [ ], 3.14, False]

# 6.7. Concatenation and Repetition

Again, as with strings, the + operator concatenates lists. Similarly, the \* operator repeats the items in a list a given number of times.

fruit = ["apple","orange","banana","cherry"]

print([1,2] + [3,4])

print(fruit+[6,7,8,9])

​

print([0] \* 4)

​

It is important to see that these operators create new lists from the elements of the operand lists. If you concatenate a list with 2 items and a list with 4 items, you will get a new list with 6 items (not a list with two sublists). Similarly, repetition of a list of 2 items 4 times will give a list with 8 items.

**Check your understanding**

sequences-7-1: What is printed by the following statements?

alist = [1,3,5]

blist = [2,4,6]

**print**(alist + blist)

Top of Form

A. 6  
B. [1,2,3,4,5,6]  
C. [1,3,5,2,4,6]  
D. [3,7,11]

Bottom of Form

sequences-7-2: What is printed by the following statements?

alist = [1,3,5]

**print**(alist \* 3)

Top of Form

A. 9  
B. [1,1,1,3,3,3,5,5,5]  
C. [1,3,5,1,3,5,1,3,5]  
D. [3,9,15]

# 6.8. Count and Index

As you create more complex programs, you will find that some tasks are commonly done. Python has some built-in functions and methods to help you with these tasks. This page will cover two helpful methods for both strings and lists: count and index.

## 6.8.1. Count

The first method we’ll talk about is called count. It requires that you provide one argument, which is what you would like to count. The method then returns the number of times that the argument occured in the string/list the method was used on. There are some differences between count for strings and count for lists. When you use count on a string, the argument can only be a string. You can’t count how many times the integer 2 appears in a string, though you can count how many times the string “2” appears in a string. For lists, the argument is not restricted to just strings.

a = "I have had an apple on my desk before!"

print(a.count("e"))

print(a.count("ha"))

​

The activecode window above demonstrates the use of count on a string. Just like with the turtle module when we had to specify which turtle was changing color or moving, we have to specify which string we are using count on.

z = ['atoms', 4, 'neutron', 6, 'proton', 4, 'electron', 4, 'electron', 'atoms']

print(z.count("4"))

print(z.count(4))

print(z.count("a"))

print(z.count("electron"))

​

When you run the activecode window above, you’ll see how count with a list works. Notice how “4” has a count of zero but 4 has a count of three? This is because the list z only contains the integer 4. There are never any strings that are 4. Additionally, when we check the count of “a”, we see that the program returns zero. Though some of the words in the list contain the letter “a”, the program is looking for items in the list that are just the letter “a”.

## 6.8.2. Index

The other method that can be helpful for both strings and lists is the index method. The index method requires one argument, and, like the count method, it takes only strings when index is used on strings, and any type when it is used on lists. For both strings and lists, index returns the leftmost index where the argument is found. If it is unable to find the argument in the string or list, then an error will occur.

music = "Pull out your music and dancing can begin"

bio = ["Metatarsal", "Metatarsal", "Fibula", [], "Tibia", "Tibia", 43, "Femur", "Occipital", "Metatarsal"]

​

print(music.index("m"))

print(music.index("your"))

​

print(bio.index("Metatarsal"))

print(bio.index([]))

print(bio.index(43))

​

sequences-8-1: What will be stored in the variable ty below?

qu = "wow, welcome week!"

ty = qu.index("we")

Top of Form

A. 5  
B. 6  
C. 13  
D. 14  
E. There is an error.

Bottom of Form

sequences-8-2: What will be stored in the variable ty below?

qu = "wow, welcome week! Were you wanting to go?"

ty = qu.count("we")

Top of Form

A. 0  
B. 2  
C. 3  
D. There is an error.

Bottom of Form

sequences-8-3: What will be stored in the variable ht below?

rooms = ['bathroom', 'kitchen', 'living room', 'bedroom', 'closet', "foyer"]

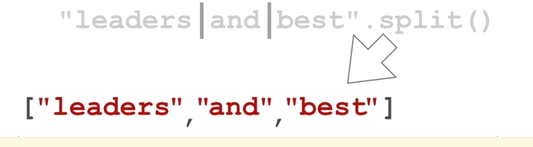
ht = rooms.index("garden")

Top of Form

A. 0  
B. -1  
C. There is an error.

# 6.9. Splitting and Joining Strings

Two of the most useful methods on strings involve lists of strings. The split method breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary.



song = "The rain in Spain..."

wds = song.split()

print(wds)

​

An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries.



The following example uses the string ai as the delimiter:

ong = "The rain in Spain..."

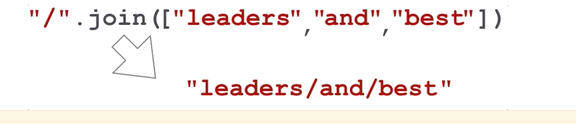
wds = song.split('ai')

print(wds)

​

Notice that the delimiter doesn’t appear in the result.

The inverse of the split method is join. You choose a desired **separator** string, (often called the glue) and join the list with the glue between each of the elements.



wds = ["red", "blue", "green"]

glue = ';'

s = glue.join(wds)

print(s)

print(wds)

​

print("\*\*\*".join(wds))

print("".join(wds))

​

The list that you glue together (wds in this example) is not modified. Also, you can use empty glue or multi-character strings as glue.

**Check your understanding**

Create a new list of the 6th through 13th elements of lst (eight items in all) and assign it to the variable output.

lst = ["swimming", 2, "water bottle", 44, "lollipop", "shine", "marsh", "winter", "donkey", "rain", ["Rio", "Beijing", "London"], [1,2,3], "gold", "bronze", "silver", "mathematician", "scientist", "actor", "actress", "win", "cell phone", "leg", "running", "horse", "socket", "plug", ["Phelps", "le Clos", "Lochte"], "drink", 22, "happyfeet", "penguins"]

​

​

Create a variable output and assign it to a list whose elements are the words in the string str1.

str1 = "OH THE PLACES YOU'LL GO"

​

​

# 6.10. Tuple Packing and Unpacking

Wherever python expects a single value, if multiple expressions are provided, separated by commas, they are automatically **packed** into a tuple. For example, we could have omitted the parentheses when first assigning a tuple to the variable julia.

julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")

# or equivalently

julia = "Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia"

print(julia[4])

​

**Check your understanding**

tuples-2-1: Which of the following statements will output Atlanta, Georgia

Top of Form

A. print(julia['city'])  
B. print(julia[-1])  
C. print(julia(-1))  
D. print(julia(6))  
E. print(julia[7])

Bottom of Form

**2.** Create a tuple called practice that has four elements: ‘y’, ‘h’, ‘z’, and ‘x’.

**Unpacking:**

Python has a very powerful **tuple assignment** feature that allows a tuple of variable names on the left of an assignment statement to be assigned values from a tuple on the right of the assignment. Another way to think of this is that the tuple of values is **unpacked** into the variable names.

julia = "Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia"

​

name, surname, birth\_year, movie, movie\_year, profession, birth\_place = julia

​

This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b:

a = 1

b = 2

temp = a

a = b

b = temp

print(a, b, temp)

​

Tuple assignment solves this problem neatly:

a = 1

b = 2

(a, b) = (b, a)

print(a, b)

​

# 6.11. Chapter Assessment

**Check your understanding**

sequences-10-1: What will the output be for the following code?

let = "z"

let\_two = "p"

c = let\_two + let

m = c\*5

**print**(m)

Top of Form

A. zpzpzpzpzp  
B. zzzzzppppp  
C. pzpzpzpzpz  
D. pppppzzzzz  
E. None of the above, an error will occur.

Bottom of Form

Write a program that extracts the last three items in the list sports and assigns it to the variable last. Make sure to write your code so that it works no matter how many items are in the list.

sports = ['cricket', 'football', 'volleyball', 'baseball', 'softball', 'track and field', 'curling', 'ping pong', 'hockey']

​

​

​

Write code that combines the following variables so that the sentence “You are doing a great job, keep it up!” is assigned to the variable message. Do not edit the values assigned to by, az, io, or qy.

by = "You are"

az = "doing a great "

io = "job"

qy = "keep it up!"

​

​

​

sequences-10-2: What will the output be for the following code?

ls = ['run', 'world', 'travel', 'lights', 'moon', 'baseball', 'sea']

new = ls[2:4]

**print**(new)

Top of Form

A. ['travel', 'lights', 'moon']  
B. ['world', 'travel', 'lights']  
C. ['travel', 'lights']  
D. ['world', 'travel']

Bottom of Form

sequences-10-3: What is the type of m?

l = ['w', '7', 0, 9]

m = l[1:2]

Top of Form

A. string  
B. integer  
C. float  
D. list

Bottom of Form

sequences-10-4: What is the type of m?

l = ['w', '7', 0, 9]

m = l[1]

Top of Form

A. string  
B. integer  
C. float  
D. list

Bottom of Form

sequences-10-5: What is the type of x?

b = "My, what a lovely day"

x = b.split(',')

Top of Form

A. string  
B. integer  
C. float  
D. list

Bottom of Form

sequences-10-6: What is the type of a?

b = "My, what a lovely day"

x = b.split(',')

z = "".join(x)

y = z.split()

a = "".join(y)

Top of Form

A. string  
B. integer  
C. float  
D. list

Bottom of Form

Write code to determine how many 9’s are in the list nums and assign that value to the variable how\_many. Do not use a for loop to do this.

nums = [4, 2, 23.4, 9, 545, 9, 1, 234.001, 5, 49, 8, 9 , 34, 52, 1, -2, 9.1, 4]

​

​

​

Write code to get rid of the the second 8 so that here are only two 8’s in the list nums.

nums = [4, 2, 8, 23.4, 8, 9, 545, 9, 1, 234.001, 5, 49, 8, 9 , 34, 52, 1, -2, 9.1, 4]

​

​

​

Assign the last element of lst to the variable end\_elem. Do this so that it works no matter how long lst is.

lst = ["hi", "goodbye", "python", "106", "506", 91, ['all', 'Paul', 'Jackie', "UMSI", 1, "Stephen", 4.5], 109, "chair", "pizza", "wolverine", 2017, 3.92, 1817, "account", "readings", "papers", 12, "facebook", "twitter", 193.2, "snapchat", "leaders and the best", "social", "1986", 9, 29, "holiday", ["women", "olympics", "gold", "rio", 21, "2016", "men"], "26trombones"]

​

​

Assign the number of elements in lst to the variable num\_lst.

lst = ["hi", "goodbye", "python", "106", "506", 91, ['all', 'Paul', 'Jackie', "UMSI", 1, "Stephen", 4.5], 109, "chair", "pizza", "wolverine", 2017, 3.92, 1817, "account", "readings", "papers", 12, "facebook", "twitter", 193.2, "snapchat", "leaders and the best", "social", "1986", 9, 29, "holiday", ["women", "olympics", "gold", "rio", 21, "2016", "men"], "26trombones"]

​

​

Create a variable called wrds and assign to it a list whose elements are the words in the string sent. Do not worry about punctuation.

sent = "The bicentennial for our university is in 2017!"

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form