



BELLEVUE
UNIVERSITY

DSC 510

Week 6 strings and lists

Strings

A string is simply a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
```

```
'This is also a string.'
```

In order to use single or double quotes within a string the starting and ending quotes will need to be different than the quotes within the string like this:

```
'I told my friend, "Python is my favorite language!"'
```

```
"The language 'Python' is named after Monty Python, not the snake."
```

```
"One of Python's strengths is its diverse and supportive community."
```

Combining or Concatenating Strings

It's often useful to combine strings. For example, you might want to store a first name and a last name in separate variables, and then combine them when you want to display someone's full name:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name
1 print(full_name)
```

Python uses the plus symbol (+) to combine strings. In this example, we use + to create a full name by combining a `first_name`, a space, and a `last_name` **1**, giving this result:

```
ada lovelace
```

Combining or Concatenating Strings

This method of combining strings is called concatenation. You can use concatenation to compose complete messages using the information you've stored in a variable. Let's look at an example:

```
first_name = "ada"  
last_name = "lovelace"  
full_name = first_name + " " + last_name  
1 print("Hello, " + full_name.title() + "!" )
```

Here, the full name is used at **1** in a sentence that greets the user, and the `title()` method is used to format the name appropriately. This code returns a simple but nicely formatted greeting:

```
Hello, Ada Lovelace!
```

Combining or Concatenating Strings

You can use concatenation to compose a message and then store the entire message in a variable:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name
1 message = "Hello, " + full_name.title() + "!"
2 print(message)
```

This code displays the message "Hello, Ada Lovelace!" as well, but storing the message in a variable at **1** makes the final print statement at **2** much simpler.

String Split()

- Split()
 - The split method will split a string into a list so that each word in the string becomes an element in a list

```
quote = "The joy of coding Python should be in seeing short, concise, readable classes that express " \
        "a lot of action in a small amount of clear code -- not in reams of trivial code that bores the " \
        "reader to death."

split_quote = quote.split()
print(split_quote)
```

Output of the above code:

```
['The', 'joy', 'of', 'coding', 'Python', 'should', 'be', 'in', 'seeing', 'short,', 'concise,', 'readable', 'classes', 'that', 'express', 'a', 'lot', 'of', 'action', 'in', 'a', 'small', 'amount', 'of', 'clear', 'code', '--', 'not', 'in', 'reams', 'of', 'trivial', 'code', 'that', 'bores', 'the', 'reader', 'to', 'death.']
```

String Strip()

- Strip()
 - Return a copy of the string with the leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix or suffix; rather, all combinations of its values are stripped:
 - Notice in the example below how the leading spaces are removed.

```
quote = " The joy of coding Python should be in seeing short, concise, readable classes that express " \
        "a lot of action in a small amount of clear code -- not in reams of trivial code that bores the " \
        "reader to death. "

print(quote)
split_quote = quote.strip()
print(split_quote)
```

Output of the code above:

The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code -- not in reams of trivial code that bores the reader to death.

The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code -- not in reams of trivial code that bores the reader to death.

String Punctuation

- Python has a punctuation operation
 - String of ASCII characters which are considered punctuation characters in the C locale:
 - `!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~.`
 - To use punctuation be sure to **import string** into your Python code.

```
print(string.punctuation)
```

Output of the code above:

`!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~.`

Removing punctuation from a string

- There are several ways to remove punctuation from strings including:
 - Using Regular Expressions
 - Using String `translate()`
 - Using String `replace()`
 - Using String `strip()`

Using string.translate()

```
import string

quote = " The joy of coding Python should be in seeing short, concise, readable classes that express " \
        "a lot of action in a small amount of clear code -- not in reams of trivial code that bores the " \
        "reader to death. "

translated_string = quote.translate(str.maketrans('', '', string.punctuation))
print(quote)
print(translated_string)
```

Output of the code above:

The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code -- not in reams of trivial code that bores the reader to death.

The joy of coding Python should be in seeing short concise readable classes that express a lot of action in a small amount of clear code not in reams of trivial code that bores the reader to death

See next slide for for information

translate() using maketrans()

- `maketrans()` - This static method returns a translation table usable for `str.translate()`
 - `string.maketrans(x, y, z)`
 - `x` Required. If only one parameter is specified, this has to be a dictionary describing how to perform the replace. If two or more parameters are specified, this parameter has to be a string specifying the characters you want to replace.
 - `y` Optional. A string with the same length as parameter `x`. Each character in the first parameter will be replaced with the corresponding character in this string.
 - `z` Optional. A string describing which characters to remove from the original string.
- `translate()` - Return a copy of the string in which each character has been mapped through the given translation table.
 - `string.translate(str.maketrans("", "", string.punctuation))`
 - Call `translate` and pass `str.maketrans` to replace any punctuation with `None`

replace()

```
quote = " The joy of coding Python should be in seeing short, concise, readable classes that express " \
        "a lot of action in a small amount of clear code -- not in reams of trivial code that bores the " \
        "reader to death. "
```

```
print(quote)
for character in string.punctuation:
    quote = quote.replace(character, '')
print(quote)
```

Iterate through the list of punctuation. For each character in `string.punctuation` replace that given character with `None`.

Output of the code above:

The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code -- not in reams of trivial code that bores the reader to death.

The joy of coding Python should be in seeing short concise readable classes that express a lot of action in a small amount of clear code not in reams of trivial code that bores the reader to death

strip()

```
import string

quote = " The joy of coding Python should be in seeing short, concise, readable classes that express " \
        "a lot of action in a small amount of clear code -- not in reams of trivial code that bores the " \
        "reader to death. "

print(quote)
word_list = quote.split()
for word in word_list:
    word = word.strip(string.punctuation)
    print(word)
```

In this example we take the string and split it into a list of words. Then using a for loop we can iterate through the list of words and remove punctuation using the strip() method.

What is a List

A list is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family. You can put anything you want into a list, and the items in your list don't have to be related in any particular way. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as letters, digits, or names.

In Python, square brackets ([]) indicate a list, and individual elements in the list are separated by commas. Here's a simple example of a list that contains a few kinds of bicycles:

bicycles.py

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']print(bicycles)
```

If you ask Python to print a list, Python returns its representation of the list, including the square brackets:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Because this isn't the output you want your users to see, let's learn how to access the individual items in a list.

Related Data

Rather than using individually named variables to represent a group of related data, most programming languages allow you to represent this type of data using a single name. In Python, it is called a *list*. A list is a new data type.

This is a shopping list, which is simple list of strings:

```
'cereal'  
'milk'  
'orange juice'  
'hot dogs'  
'gum'
```

This is a list of test scores:

```
99  
72  
88  
82  
54
```

Elements

There is a special name for each thing in a list called an element.

An *element* is a single member of a list. (It is also known as an *item* in the list.)

Let's look at our shopping list again:

`'cereal'`

`'milk'`

`'orange juice'`

`'hot dogs'`

`'gum'`

Each string in the list is an element of the list: `'cereal'` is an element, `'orange juice'` is an element, etc.

Python Syntax

When we make a list, like a shopping list on paper, we typically write the elements vertically, one per line. In a computer language, we need some special syntax to indicate that we are talking about a list. In Python, we use the square bracket characters, [and]. A list is represented by an open square bracket, the elements separated by commas, and a closing square bracket, as follows:

```
[<element>, <element>, ... <element>]
```

Just like any other type of data, a list is created using an assignment statement. That is, you write single variable name followed by an equals sign, and then you define your list.

```
<myListVariable> = [<element>, <element>, ... <element>]
```

A list can essentially have any number of elements.

The actual number of elements is limited only by the amount of memory in the computer. Here are some examples:

```
shoppingList = ['cereal', 'milk', 'orange juice', 'hot dogs', 'gum']  
scoresList = [24, 33, 22, 45, 56, 33, 45]
```

Mix of Data Types

A list can also be created using a mix of data types:

```
mixedList = [True, 5, 'some string', 123.45]
```

Note that we are showing variable names that represent a list in the form of `<name>List`. This is not required, but a name in this form clearly indicates that the variable represents a list rather than an individual piece of data. We will use this naming convention throughout the rest of this book.

A list is a new data type. To show that a list is a standard data type in Python, let's create a list, print it, and use the type function to find out which data type it is:

```
>>> mixedList = [True, 5, 'some string', 123.45]
>>> print (mixedList) [True, 5, 'some string', 123.45]
>>> print (type(mixedList)) <class 'list'>
>>>
```

Empty List

It is also possible to have an empty list which doesn't have any elements such as:

```
>>> someList = [] # set a list variable to the empty list - no elements
>>> print (someList) []
>>>
```

List Index

It is possible to access individual elements in a list by using the element's index.

- An *index* is the position (or number) of an element in a list. (It is sometimes referred to as a *subscript*.)
- An index is always an integer value. Since each element has an index (number), we can reference any element in the list by using its associated number or position in the list.

To us humans, in our shopping list, cereal is element number 1, milk is element number 2, and orange juice is element number 3.

Sample shoppingList:

Index	Element
1	'cereal'
2	'milk'
3	'orange juice'
4	'hot dogs'
5	'gum'

Index Continued

In python the list index doesn't start with 1 like we saw in the previous human readable form. In Python indexes start at 0:

Sample shoppingList:

Index	Element
0	'cereal'
1	'milk'
2	'orange juice'
3	'hot dogs'
4	'gum'

In this example the elements have an index between 0 and 4

Accessing Elements

In order to access a specific element we use it's index with the following syntax:

`<listVariable>[<index>]`

Using a List of: `numbersList = [20, -34, 486, 3129]`

We can access each element in the `numbersList` as follows:

`numbersList[0]` # would evaluate to 20

`numbersList[1]` # would evaluate to -34

`numbersList[2]` # would evaluate to 486

`numbersList[3]` # would evaluate to 3129

Negative Indices

In addition to indices (starting at 0 and going up to the number of elements minus 1), there is another way to index elements in a list. Python allows you to use negative integers as indices to a list. A negative index means to count backwards from the end; that is, the number of elements in the list. Here are the positive and equivalent negative indices for a list of five elements:

```
0 -5 <element>
1 -4 <element>
2 -3 <element>
3 -2 <element>
4 -1 <element>
```

Let's demonstrate with our shopping list:

```
>>> shoppingList = ['cereal', 'milk', 'orange juice', 'hot dogs', 'gum']
>>> print shoppingList[-1]
gum
>>> print shoppingList[-2]
hot dogs
>>> print shoppingList[-3]
orange juice
```

List Example using the Shell

```
shoppingList = ['cereal', 'milk', 'orange juice', 'hot dogs', 'gum']
>>> print(shoppingList)
['cereal', 'milk', 'orange juice', 'hot dogs', 'gum']
>>> print(shoppingList[2])
orange juice
>>> print(shoppingList[4])
gum
>>> print(shoppingList[0])
Cereal
>>>
```


Accessing a list element using variables

```
>> shoppingList = ['cereal', 'milk', 'orange juice', 'hot dogs', 'gum']
>>> myIndex = input('Enter an index: ')
Enter an index: 3
>>> myIndex = int(myIndex) #convert myIndex to an integer
>>> myElement = shoppingList[myIndex]
>>> print ('The element at index', myIndex, 'is', myElement)
The element at index 3 is hot dogs
>>>
```

Changing an Element in a List

Changing an element in a list can also be done using the index:

```
>>> shoppingList = ['cereal', 'milk', 'orange juice', 'hot dogs', 'gum']
>>> shoppingList[3] = 'apples'
>>> print (shoppingList)
['cereal', 'milk', 'orange juice', 'apples', 'gum']
>>>
```

This changes the value of an element at the given index to a new value. Notice that element 3 was 'hot dogs', but has been changed to 'apples'.

Iterating a List

- There are two fundamental ways of visiting all elements of a list. You can loop over the index values and look up each element, or you can loop over the elements themselves.
- We first look at a loop that traverses all index values. Given the values list that contains 10 elements, we will want to set a variable, say *i*, to 0, 1, 2, and so on, up to 9. A list index must be at least zero and less than the number of elements in the list. An out of range error, which occurs if you supply an invalid list index, can cause your program to terminate. Then the expression `values[i]` yields each element in turn. This loop displays all index values and their corresponding elements in the values list.

```
for i in range(10) :  
    print(i, values[i])
```

- The variable *i* iterates through the integer values 0 through 9, which is appropriate because there is no element corresponding to `values[10]`. Instead of using the literal value 10 for the number of elements in the list, it is a good idea to use the `len` function to create a more reusable loop.

```
for i in range(len(values)) :  
    print(i, values[i])
```

- If you don't need the index values, you can iterate over the individual elements using a for loop in the form:

```
for element in values :  
    print(element)
```

Iterating a List in Action

The most common mechanism for iterating through a list is to use a for loop. The general syntax for a for loop used to iterate through a list is:

```
for <elementVariable> in <list>:  
    <indented statement(s)>
```

Example:

```
>>> cheeses = ['Cheddar', 'Gouda', 'Swiss']  
>>> 'Gouda' in cheeses  
True  
>>> for cheese in cheeses:  
    print(cheese)
```

```
Cheddar  
Gouda  
Swiss  
>>>
```

Adding new Elements to a List

It is often times necessary to define a list without knowing the exact number of elements the list should contain. In this instance you will need to add additional elements to the list using the append method.

Example:

```
Friends = []  
Friends.append('Bill')  
Friends.append('Karla')
```

Inserting an Element into a List

Using our Friends list from the previous slide let's look at another common method which is the insert method. On the left is our code. Notice that in the insert statement we must include the index that we want to use to insert the value. On the right side is our print statements.

```
friends = []  
friends.append('Bill')  
friends.append('Karla')  
print(friends)  
  
friends.insert(2, 'Mike')  
print(friends)
```

```
['Bill', 'Karla']  
['Bill', 'Karla', 'Mike']
```

Common List Methods

- `list.append(elem)` -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- `list.insert(index, elem)` -- inserts the element at the given index, shifting elements to the right.
- `list.extend(list2)` adds the elements in `list2` to the end of the list. Using `+` or `+=` on a list is similar to using `extend()`.
- `list.index(elem)` -- searches for the given element from the start of the list and returns its index. Throws a `ValueError` if the element does not appear (use `"in"` to check without a `ValueError`).
- `list.remove(elem)` -- searches for the first instance of the given element and removes it (throws `ValueError` if not present)
- `list.sort()` -- sorts the list in place (does not return it). (The `sorted()` function shown later is preferred.)
- `list.reverse()` -- reverses the list in place (does not return it)
- `list.pop(index)` -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of `append()`).

List Slicing

Sometimes you want to look at a part of a list. For example, suppose you are given a list of temperatures, one per month: `temperatures = [18, 21, 24, 28, 33, 39, 40, 39, 36, 30, 22, 18]`

You are interested in the temperatures only for the third quarter, with index values 6, 7, and 8. You can use Python's slice operator (`:`) to obtain them. `secondQuarter = temperatures[6 : 9]`

The arguments for the slice operator are the first index to include in the slice, followed by the first index to exclude. This may seem a curious arrangement, but it has a useful property. The length of the slice `temperatures[a : b]` is the difference $b - a$. In our case, the difference $9 - 6 = 3$ is the number of months in a quarter.

Both index values used with the slice operator (6 and 9 here) are optional. If the first index is omitted, all elements from the first element on are included. The slice `temperatures[: 6]` contains all elements up to (but not including) position 6. That's the first half of the year. The slice `temperatures[6 :]` includes all elements from index 6 to the end of the list; that is, the second half of the year. If you omit both index values, `temperatures[:]`, you get a copy of the list.

You can even assign values to a slice. For example, the assignment `temperatures[6 : 9] = [45, 44, 40]` replaces the values for the third quarter. The size of the slice and the replacement don't have to match: `friends[: 2] = ["Peter", "Paul", "Mary"]` replaces the first two elements of `friends` with three new elements, increasing the length of the list. Slices work with all sequences, not just lists. They are particularly useful with strings. A slice of a string is a substring: `greeting = "Hello, World!"` `greeted = greeting[7 : 12]` # The substring "World"