Modules 3 & 4 Agenda

- Python Comments
- Getting User Input
- Strings
- Lists

Chapter 3 and 4 in Python Crash Course: A Hands-on, Project-Based Introduction to Programming

Comments

When you are writing software, you wind up making a lot of decisions about how you approach different problems. Sometimes, your solutions are not exactly apparent and could use some documentation. You may want to explain to the reader (who could be a future version of you, or someone else) why you did something the way you wound up doing it, or how some intricate piece of code works.

Documentation like this, written directly in your code, is called a *comment*. Comments are completely ignored by Python; they are only there for humans. There are three ways to write comments in Python: provide a full-line comment, add a comment after a line of code, or use a multiline comment.

Full-Line Comments

Full-Line Comment
Start a line with the # character, followed by your comment:

```
# This whole line is a comment
# This is another comment line
# All comment lines are ignored by Python
# Even though the next line looks like code, it's just a comment
# x=1
# -----------------------------------
```

Comments after a Line of Code

Add a Comment Alter a Line of Code
You can put a comment at the end of a line of code to explain what is going on inside that line. The following lines are very simple and don't really need comments, but they should serve as a good example of how to add this type of comment:

```
score = 0 # Initializing the  score
priceWithTax = price * 1.09 # add in 9% tax
```

Multiline Comments

Multiline Comment

You can create a comment that spans any number of lines. You do this by having one line with three quote marks (single or double quotes), any number of comment lines, and ending with the same three quote characters (single or double quotes), as follows:

```
'''
A multiline comment starts with a line of three quote
characters(above) This is a long comment block It can be any length
You do not need to use the # character here You end it by entering the
same three quotes you used to start (below)
'''
```

Getting User Input

This is the typical flow of a simple computer program:

1. Input data.
2. Work with data.
3. Do some computation(s).

Output some answer(s).In Python, we can get input from the user using a built-in function called raw_input. Here's how it is used, most typically in an assignment statement:

```
<variable> = input(<prompt string>)
```

On the right side of the equals sign is the call to the raw_input built-in function. When you make the call, you must pass in a *prompt string*, which is any string that you want the user to see. The prompt is a question that you want the user to answer. The raw_input function returns all of the characters that the user types, as a string. Here is an example of how raw_input might be used in a program:

```
favoriteColor = input('What is your favorite color? ')
print 'Your favorite color is', favoriteColor
```

Assignment Statement

When the assignment statement runs, the following steps happen in order:

1. The prompt string is printed to the Shell.
2. The program stops and waits for the user to type a response.
3. The user enters some sequence of characters into the Shell as an answer.
4. When the user presses the Enter key (Windows) or the Return key (Mac), input() returns the characters that the user typed.
5. Typically, input() is used on the right side of an assignment statement. The user's response is stored into the variable on the left-hand side of the equals sign.

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.

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:

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

The most common mechanism for iterating through a list is to use a for loop which will be discussed in more detail in later weeks.

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
>>>
```

References

The Basics of Compiled Languages, Interpreted Languages, and Just-in-Time Compiliers, https://www.upwork.com/hiring/development/the-basics-of-compiled-languages-interpreted- languages-and-just-in-time-compilers/

The End

You may close this Window and return to the course.