DSO 545: Statistical Computing and Data Visualization

Abbass Al Sharif

Fall 2019

Lab 1: Python Variables and Lists

Contents

---

1. Python for Computations

```
1
```

```
## 1
```

```
1 + 5
```

```
## 6
```

```
4/2
```

```
## 2.0
```

```
4**2
```

```
## 16
```

```
12%5
```

```
## 2
```

2. Variables in Python

Variables are containers for storing data values.Unlike other programming languages, Python has no command for declaring a variable.A variable is created the moment you first assign a value to it. (https://www.w3schools.com/python/python_variables.asp)

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character

- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

```
height = 1.80 #in meters
print(height)
```

```
## 1.8
```

```
height
```

```
## 1.8
```

```
weight  = 71 #in Kg
print(weight)
```

```
## 71
```

```
bmi = weight/(height**2)
print(bmi)
```

```
## 21.91358024691358
```

We can see that from the bmi calculations that this person has a normal weight.

Notice that we can use the above code to reproduce the result but for a different weight or height.

```
height = 2 #in meters
weight  = 71 #in Kg
bmi = weight/(height**2)
print(bmi)
```

```
## 17.75
```

The above measures of a person puts them as underweight!

Overriding a variable: Suppose that we have a variable called x

```
x = 5
x
```

```
## 5
```

We can perform computations and save it in x as follows:

```
x = x + 1
x
```

```
## 6
```

Note that the above code could be written as follows in Python:

```
x = 5
x +=1 # similar to x = x + 1
x
```

```
## 6
```

```
x -= 1 #similar to x = x - 1
x
```

```
## 5
```

```
x *=2 #similar to x = x*2
x
```

```
## 10
```

Python performs computations from left to right. In this case, the x+1 portion will be executed first, and its value will be assigned to x on the left hand side. This is called "overriding" the variable, i.e. the value of x will be updated with the new value.

Another interesting feature in Python is that you can assign more than one value to multiple variables on the same line as follows:

```
x, y, z = 45, "Marshall", "DSO 545"

x
```

```
## 45
```

```
y
```

```
## 'Marshall'
```

```
z
```

```
## 'DSO 545'
```

3. Types of Data or Variables

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them. (https://www.tutorialspoint.com/python3/python_variable_types)

Variables do not need to be declared with any particular type and can even change type after they have been set. (https://www.w3schools.com/python/python_variables.asp)

```
type(x)
```

```
## <class 'int'>
```

```
type(bmi)
```

```
## <class 'float'>
```

```
type("DSO 545")
```

```
## <class 'str'>
```

In the above lines of code, we see three different variable/data types. The data type for variable x is "int" or integer, the data type for the variable bmi is "float" or think of it as decimal number, and finally the data type for "DSO 545" is "str" or string of characters.

In addition, there are three other data types that we will talk more about them later in the course: list, tuple, and dictionary.

To conclude, we can think of 7 different data types:

- **int**: integer
- **float**: similar to decimal
- **str**: string of characters
- **bool**: boolean values (i.e. True, False)

- **list**: *to be discussed later!*
- **tuple**: *to be discussed later!*
- **dictionary**: *to be discussed later!*

You can convert from one type to another using functions `int()`, `float()`, and `str()`. A function in python allows you to perfom a certain procedure, and in this case the function `int()` for example, allows you to type cast or change the type of a certain number to integer.

```
x = 2.5
type(x)
```

```
## <class 'float'>
```

```
int(x)
```

```
## 2
```

```
type(x)

# we need to "override" the original content in object x
```

```
## <class 'float'>
```

```
x = int(x)
type(x)
```

```
## <class 'int'>
```

In the following 3 lines of code, we casted/changed the variable x of type integer to a string value. Notice that when we print the value of y now, we can see that the value of 2 is not in between qoutations '2'. A string in Python will always be between a single qoute or a double qoute.

```
type(x)
```

```
## <class 'int'>
```

```
y= str(x)
y
```

```
## '2'
```

```
type(y)
```

```
## <class 'str'>
```

4. Python "List" Data Structure

So far, we have been working with variables that stores only one value at a time. In data science, we will be dealing with thousands of values when we are analyzing our data.

Python provides the capability of handeling large amount of data points using different data structures depending on the type of problem we are solving. In this section, we will focus on our first compound data type called "list".

Lists are just like the arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are also very useful for implementing stacks and queues. Lists are mutable, and hence, they can be altered even after their creation. (https://www.geeksforgeeks.org/python-list/)

A list data type groups values together, and can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type. (https://docs.python.org/2/tutorial/introduction.html#lists)

In the following example, we will create a list of the 7 elements called squares:

```
squares = [1, 4, 9, 25, 36, 49, 64]
squares
```

```
## [1, 4, 9, 25, 36, 49, 64]
```

```
type(squares)
```

```
## <class 'list'>
```

Typically a list holds many items of the same type, i.e. numbers, strings, boolean, etc. However, a list is a versatile data structure in Python that allows you to have more than one type for the items in the list. For example, we can have a list of integers and strings at the same time:

```
mixed_list = ["one", 1, "four", 4, "nine", 9, "twenty five", 25]
mixed_list
```

```
## ['one', 1, 'four', 4, 'nine', 9, 'twenty five', 25]
```

```
type(mixed_list)
```

```
## <class 'list'>
```

Finally, we can have a list of lists! Remember that a list can have its element be of any type, so we can definitly have a list of lists as follows:

```
list_of_lists = [["one", 1], ["four", 4], ["nine", 9], ["twenty five", 25]]
list_of_lists
```

```
## [['one', 1], ['four', 4], ['nine', 9], ['twenty five', 25]]
```

```
type(list_of_lists)
```

```
## <class 'list'>
```

5. Accessing Elements of Lists in Python

Index: We can use the index operator `[]` to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4. **Note that the index should only be an integer.**

In order to get the first element in the `squares` list, we use the index 0 as follows.

```
squares[0] # first element in the squares list
```

```
## 1
```

We can access any element in a given list, but keep in mind that that indexing starts at 0. So to get the third element, we will index at 2 as follows:

```
squares[2] # access the third element
```

```
## 9
```

```
squares[5] # access the sixth element
```

```
## 49
```

Negative Index: Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

For example, to get the last element in the list, we use the index -1 as follows:

```
squares[-1] #access the last element in the list of squares
```

```
## 64
```

```
squares[-2] #access the second to last element in the list of squares
```

```
## 49
```

List Slicing: We can access a range of items in a list by using the "colon" slicing operator (:). The following rules apply to slicing (https://www.geeksforgeeks.org/python-list/):

List Slicing	Code
To print elements from beginning to a range	`[:Index]`
To print elements from specific `Index` till the end	`[Index:]`
To print elements within a range	`[Start Index:End Index]`
To print every other nth element in the list	`[::n]`
To print whole List in reverse order	`[::-1]`

Examples:

To access the first 3 elements in the squares list:

```
squares[:3]
```

```
## [1, 4, 9]
```

To access the all elements in the squares list starting at 5th element:

```
squares[4:]
```

```
## [36, 49, 64]
```

To access the all elements in the squares list starting at 3rd element, and ending at the 5th element:

```
squares[2:5]
```

```
## [9, 25, 36]
```

To print every other 3rd element in the list:

```
squares[::3]
```

```
## [1, 25, 64]
```

To print all elements of the list in reverse order:

```
squares[::-1]
```

```
## [64, 49, 36, 25, 9, 4, 1]
```

To print every other (second) element of the list in reverse order:

```
squares[::-2]
```

```
## [64, 36, 9, 1]
```

** Updating Elements in a List:** List are mutable, meaning, their elements can be changed. We can use assignment operator (=) to change an item or a range of items.

For example, suppose we have the following list, and we want to update the third element to be 95.

```
x = [30, 60, 70, 100, 120]
x
```

```
## [30, 60, 70, 100, 120]
```

```
x[3] = 95
x
```

```
## [30, 60, 70, 95, 120]
```

Note that we can update a range of elements at the same time. For example, we can update the range of elements (2nd to 4th element):

```
x = [30, 60, 70, 100, 120]
x
```

```
## [30, 60, 70, 100, 120]
```

```
x[1:4] = [55,65,95]
x
```

```
## [30, 55, 65, 95, 120]
```

6. Iterating over a Python List

There are multiple ways to iterate over a list in Python. Here we will discuss the three most used methods (https://www.geeksforgeeks.org/iterate-over-a-list-in-python/):

Method 1: Using a For loop

8. Use a for loop to print all the elements in `list = [1, 3, 5, 7, 9]`.

```
list = [1, 3, 5, 7, 9] #define the list

# Using for loop
for i in list:
    print(i)
```

```
## 1
## 3
## 5
## 7
## 9
```

2. Create a list (`degrees = [0, 10, 20, 40, 100]`). Print each of the elements on a seperate line starting with "List element:" followed by the number (e.g. List element: 10).

```
## Source: http://hplgit.github.io/primer.html/doc/pub/looplist/._looplist-bootstrap002.html

degrees = [0, 10, 20, 40, 100]
```

```
for C in degrees:
    print('list element:', C)
```

```
## list element: 0
## list element: 10
## list element: 20
## list element: 40
## list element: 100
```

```
print('The degrees list has', len(degrees), 'elements')
```

```
## The degrees list has 5 elements
```

Method #2: For loop and `range()`

Python `range()` accepts an integer and returns a range object, which is nothing but a sequence of integers.

```
for i in range(6):
    print(i)
```

```
## 0
## 1
## 2
## 3
## 4
## 5
```

In the output, we got integers from 0 to 5. `range()` function doesn't include the last number in the result. The most common use of `range()` function in python is to iterate sequence type (List, string etc) with for loop. (https://pynative.com/python-range-function/)

```
# Python code to iterate over a list
list = [1, 3, 5, 7, 9]

# getting length of list
length = len(list)

# Iterating the index
# same as 'for i in range(len(list))'
for i in range(length):
    print(list[i])
```

```
## 1
## 3
## 5
## 7
## 9
```

Method #3: Using list comprehension

Using list comprehensions will boost the computational time, and it runs computations much faster than a for loop.

```
# Python code to iterate over a list
list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Using list comprehension
```

```
[print(i) for i in list]

# Square the numbers in the list using list comprehension
```

```
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 9
## [None, None, None, None, None, None, None, None, None]
```

```
squared = [i**2 for i in list]
print(squared)

# Multiply every item in the list by 3
```

```
## [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[i*3 for i in list]
```

```
## [3, 6, 9, 12, 15, 18, 21, 24, 27]
```

List comprehensions would could be combined with *if conditional statements* to filter the list restult. For example, if we want to select all even numbers from the list, we can add an `if i%2 ==0` condition as follows:

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

[i for i in list if i%2==0]
```

```
## [2, 4, 6, 8]
```

In the code above, Python runs as follows:

- iterate over all items in the original list
- for each item, check if it is divisble by 2
- if the number is divisible by 2, then add it to our result (new list)
- note that in Python (just like all programming language), there is a difference between (==, comparison) and (=, assignment)

7. Selected Methods for Python Lists

Methods that are available with list object in Python programming are tabulated below (https://www.programiz.com/python-programming/list):

Method	Description
append()	Add an element to the end of the list
extend()	Add all elements of a list to the another list
insert()	Insert an item at the defined index
remove()	Removes an item from the list
clear()	Removes all items from the list
index()	Returns the index of the first matched item

Method	Description
count()	Returns the count of number of items passed as an argument
sort()	Sort items in a list in ascending order
reverse()	Reverse the order of items in the list

We can use the method **append()** to add an element to the list. For example, we can add 150 to the x list:

```
x = [30, 60, 70, 100, 120]
x
```

```
## [30, 60, 70, 100, 120]
```

```
x.append(150)
x
```

```
## [30, 60, 70, 100, 120, 150]
```

We can use the method **extend()** to add several items to the list. For example, we can add [170, 180, 190] to x as follows:

```
x = [30, 60, 70, 100, 120]
x
```

```
## [30, 60, 70, 100, 120]
```

```
x.extend([170, 180, 190])
x
```

```
## [30, 60, 70, 100, 120, 170, 180, 190]
```

We can also insert a specific value at a given index in the list. For example, we can instert 50 to the list to become the second element.

```
x = [30, 60, 70, 100, 120]
x
```

```
## [30, 60, 70, 100, 120]
```

```
x.insert(1, 50) # first argument is index and second argument is the value to insert
x
```

```
## [30, 50, 60, 70, 100, 120]
```

We can remove a certain value from a list using the **remove()** method. Suppose we want to remove the value of 50 from x.

```
x = [30, 50, 60, 70, 100, 120]
x
```

```
## [30, 50, 60, 70, 100, 120]
```

```
x.remove(50) # remove the first occurance of element with value of 50
x
```

```
## [30, 60, 70, 100, 120]
```

A very useful method for lists is the **index()** method which returns the index of a given value. For example, if we want to learn the index of 70, we write the following code:

```
x = [30, 50, 60, 70, 100, 120]
x
```

```
## [30, 50, 60, 70, 100, 120]
```

```
x.index(70) # returns the index of 70 in the list
```

```
## 3
```

```
x # the list is not affected
```

```
## [30, 50, 60, 70, 100, 120]
```

Suppose that we want to count the occurance of a specific value in a list, then we can use the method **count()** as follows:

```
x = [30, 50, 50, 50, 50, 60, 70, 100, 120]
x
```

```
## [30, 50, 50, 50, 50, 60, 70, 100, 120]
```

```
x.count(50) # there are 4 occurances of 50 in the list
```

```
## 4
```

The last two methods **sort()** and **inverse()** would be useful while exploring the data. The first sorts the list in asceding order by default, and the second reverses the order of the elements in a list.

```
x = [60, 20, 90, 35, 15]
x.sort() # sorts in ascending order by default
x
```

```
## [15, 20, 35, 60, 90]
```

```
x = [60, 20, 90, 35, 15]
x.sort(reverse= True) # sorts in descending order
x
```

```
## [90, 60, 35, 20, 15]
```

```
x = [60, 20, 90, 35, 15]
x
```

```
## [60, 20, 90, 35, 15]
```

```
x.reverse() #reverses the order of elements in x
x
```

```
## [15, 35, 90, 20, 60]
```

8. Python "Dictionaries" Data Structure

Python provides another composite data type called a dictionary, which is similar to a list in that it is a collection of objects. (Source– https://realpython.com/python-dicts/)

Dictionaries and lists share the following characteristics:

- Both are mutable. The values of the elements in each could be updated.
- Both are dynamic. They can grow and shrink as needed.
- Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

- List elements are accessed by their position in the list, via indexing.

- Dictionary elements are accessed via keys.

Defining a Dictionary:

A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces (`{}`). A colon (`:`) separates each key from its associated value:

The following defines a dictionary that maps a country to the name of its corresponding capital:

```
capitals = {

'Peru': 'Lima',
'Armenia': 'Yerevan',
'Jordan': 'Amman',
'Norway': 'Oslo'

}

print(capitals)
```

```
## {'Peru': 'Lima', 'Armenia': 'Yerevan', 'Jordan': 'Amman', 'Norway': 'Oslo'}
```

```
type(capitals)
```

```
## <class 'dict'>
```

Or, we can define a dictionary using the built-in function `dict()`, which takes a `list` of tuples:

```
capitals = dict([
             ('Peru','Lima'),
             ('Armenia','Yerevan'),
             ('Jordan', 'Amman'),
             ('Norway', 'Oslo')
             ])

print(capitals)
```

```
## {'Peru': 'Lima', 'Armenia': 'Yerevan', 'Jordan': 'Amman', 'Norway': 'Oslo'}
```

```
type(capitals)
```

```
## <class 'dict'>
```

Note that the first element represents a `key`, and the second element represents a value correspondin to the key.

To access elements in a dictionary, we use the key as an index to get the value. i.e. a value is retrieved from a dictionary by specifying its corresponding key in square brackets ([]):

```
capitals['Peru']
```

```
## 'Lima'
```

```
capitals['Jordan']
```

```
## 'Amman'
```

Note that, you can also build the dictionary incrementally as follows:

```
person = {} #initialize the dictionary
type(person)
```

```
## <class 'dict'>
```

```
person['fname'] = 'Joe'
person['lname'] = 'Fonebone'
person['age'] = 51
person['spouse'] = 'Edna'
person['children'] = ['Ralph', 'Betty', 'Joey']
person['pets'] = {'dog': 'Fido', 'cat': 'Sox'}

print(person)
```

```
## {'fname': 'Joe', 'lname': 'Fonebone', 'age': 51, 'spouse': 'Edna', 'children': ['Ralph', 'Betty', 'J
```

We can access elements as follows:

```
person['fname']
```

```
## 'Joe'
```

```
person['children']
```

```
## ['Ralph', 'Betty', 'Joey']
```

What if we want to acces the third child in the list of children?

```
kids = person['children']
kids[2]

## OR
```

```
## 'Joey'
```

```
person['children'][2]
```

```
## 'Joey'
```

We can also convert two lists to a dictionary as using the `zip()` function:

```
countries = ['Peru','Armenia','Jordan','Norway']
cities = ['Lima', 'Yerevan', 'Amman', 'Oslo']

capitals = dict(zip(countries, cities))
```