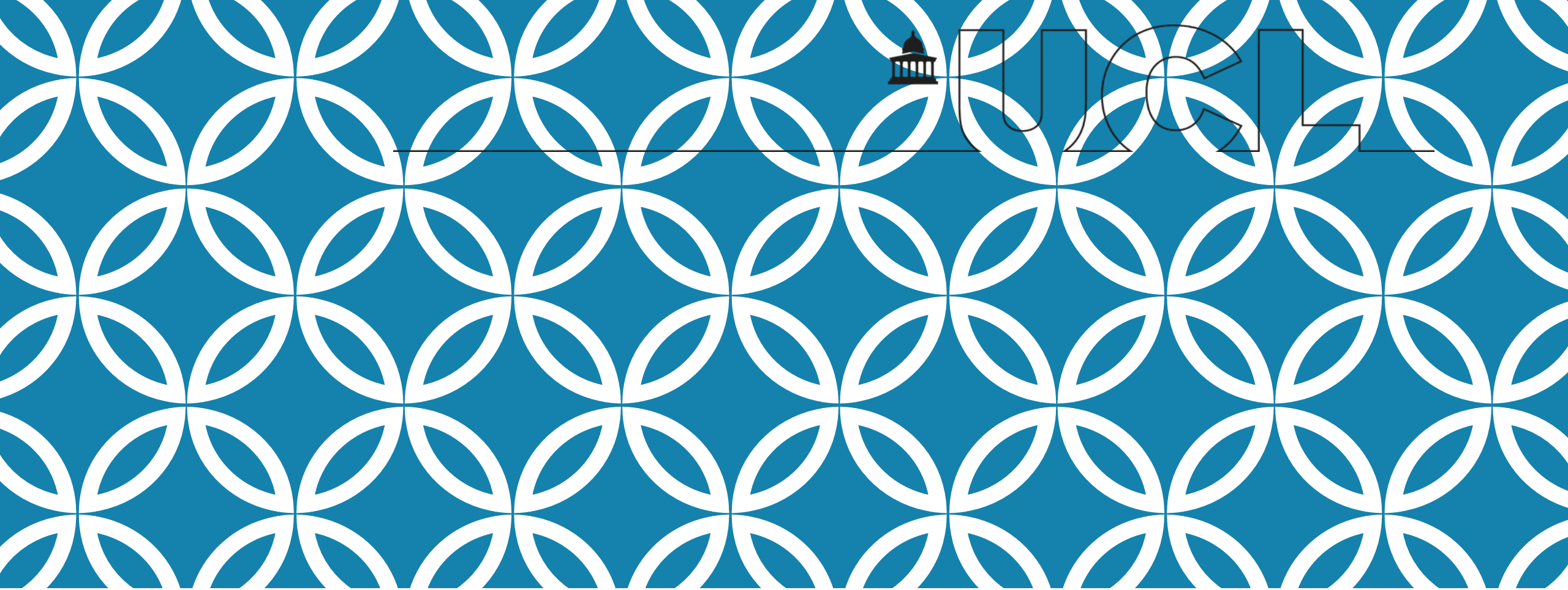

INST0002 PROGRAMMING 1 & INST0091 INTRODUCTION TO PROGRAMMING

LECTURE 06 – LISTS

Dr Daniel Onah
Lecturer
Module Leader

Copyright Note:

Copyright Licence of this lecture resources is with the Module Lecturer named in the front page of the slides. If this lecture draws upon work by third parties (e.g. Case Study publishers) such third parties also hold copyright. It must not be copied, reproduced, transferred, distributed, leased, licensed or shared with any other individual(s) and/or organisations, including web-based organisations, online platforms without permission of the copyright holder(s) at any point in time.

Recap

- Strings and various types of operations
- Built-in string functions
- Use of string formatting operator
- Use of string comparing and iteration
- String immutable

Re: Cap☺

Outline/Learning outcomes

1. Introduction to lists
2. Creating a list
3. Accessing list items
4. Functions or methods in list
5. List mutable and updating values
6. Loops (for-loops and While-loops) and lists
7. Searching and sorting in list
8. Summary

Introduction to Lists

Introduction to Lists

The list datatype in Python is one of the most powerful datatype, and it can be written as a list of comma-separated values between square brackets. A list is a container of items or elements that are organized in order from first to last. List is like arrays in other programming languages such as *Java* , *Ruby* etc.

Introduction to Lists

A List is a kind of collection that allows us to store many elements or values in a single 'variable'. *A list collection allows us to hold so many values in one convenient package.*

Let's see examples to help us understand this better.

Names = ['Danny' , 'Elaine', 'Jane']

bag = ['socks', 't-shirt', 'perfume']

Important Facts about Lists

The following are some important points that you should know about lists and creation.

- *List items are ordered, and that order will not change except for some list methods or functions.*
- *List allows duplicate element entry or values.*
- *Lists are changeable, which means we can update and add to the values and delete items from a list after it has been created and assigned. Unlike strings that we cannot update or add to the string.*
- *The first element in the list order has an index[0], and the index increases up to the length of the list.*

Creating Lists

A list may contain any number of items of various kinds, such as **integers**, **floats**, **strings** and so on. Let's see a few example to understand more about list creation.

```
# empty list
element = [ ]
print(element)
# list of integers
number = [1, 2, 3, 4, 5]
print(number)
# list with mixed data types
multi_items = [3, "Danny Onah", " Daniel Onah", 99.9]
print(multi_items)
```

Output:

```
[ ]
[1, 2, 3, 4, 5]
[3, 'Danny Onah', 'Daniel Onah', 99.9]
```

Basic List Examples

Let's look at an example to help us understand this better.

```
#!/usr/bin/env python
"""
A Program to demonstrate basic list examples
"""
# Basics of lists creation
name = ['Danny', 'Onah', 31, 32]
number = [1, 2, 3, 4, 5, 6, 7]
letter = ["a", "b", "c", "d", "e", "f"]
print("The name elements are: ", name)
print("The number elements are: ", number)
print("The letter elements are:", letter)
# list accept duplicate values
names = ["Danny", "Onah", "Daniel", "Onah", "Daniel", "Danny", "Dan"]
print("The names are: ", names)
```

Basic List Outcomes

```
The name elements are:  ['Danny', 'Onah', 31, 32]
The number elements are:  [1, 2, 3, 4, 5, 6, 7]
The letter elements are: ['a', 'b', 'c', 'd', 'e', 'f']
The names are:  ['Danny', 'Onah', 'Daniel', 'Onah', 'Daniel', 'Danny', 'Dan']
```

Basic List Example

Let's see example of how to check for the class type of a list and using list as part of the expression for creating and initializing a list.

```
#!/usr/bin/env python

"""
A program to show the list type and using list in the expression
"""
# We can use list as part of the expression for creating and initialising a list

name_number = ['Danny', 'Onah', 31, 32]
print(type(name_number)) # checking the list type

listExpression = list(('Danny', 'Onah', 31, 32))
print(listExpression)
```

Output:

```
<class 'list'>
['Danny', 'Onah', 31, 32]
```

Collection?

All variables that we create hold a value. When we store a new value in the variable, the old value is overwritten.

```
var = 5
var = 8
print(var)
```

Output: 8

List Constants

List have constants which are surrounded by square brackets and the elements in the list are separated *by commas*. A list element can be represented as any Python object, even another list. A list can also be empty (this is called empty list).

```
print([1, 76, 89])
print(['green', 'brown', 'blue'])
print(['red', 23, 76.9])
print([2, [4, 5], 8])
print([ ])
```

Output:

```
[ 1, 76, 89]
['green', 'brown', 'blue']
['red', 23, 76.9]
[2, [4, 5], 8]
[ ]
```

Accessing Values in a List

All elements in a list is indexed, and they can be accessed by referring to their index number. There are several ways to access a list of elements. We can access a list object using the index operator [ ]. As you know the index start from *0*. For example, suppose we have a list with seven elements, then this will start from *0 to 6*. If we try to access out of index value, this will raise an *indexError* (Index_Out_Of_Range_Error). Note that an index or list indices must only be an integer value, therefore float and other data types are not used for the index value. If we try to use another data type apart from integer in the index, this will raise *TypeError*.

Accessing Elements in Lists

Like String, we can also access the elements in a list. We can use the *index* of the list specified in square brackets to get any single element in a list.

Danny	Elaine	Jane
0	1	2

Let's see an example to help us understand this.

Accessing Values in a List - Positive

```
#!/usr/bin/env python

"""
A program to access the elements in the list using the index value
"""
# List indexing
name = ['d', 'a', 'n', 'i', 'e', 'l']
print(name[0])
print(name[1])
print(name[2])
print(name[3])
print(name[4])
print(name[5])
```

Output:

```
d
a
n
i
e
l
```

Accessing Elements in Lists

Like String, we can also access the elements in a list. We can use the *index* of the list specified in square brackets to get any single element in a list.

Let's see an example to help us understand this.

```
#!/usr/bin/env python

"""
A program to access the elements in a list
"""

friends = ['Danny', 'Elaine', 'Jane'] # a list of names
print(friends[2]) # selecting the third name in index 2
```

Output:

Jane

Negative Indexing

Negative indexing is also allowed for its sequences. In negative indexing, we refer to the last element using -1, the second to the last element can be accessed using -2, and -3 can be used to access the third to the last element , and so on. Let's illustrate this with 6 elements in a list (length = 6).

	'd'	'a'	'n'	'i'	'e'	'l'
Positive index →	0	1	2	3	4	5
Negative index →	-6	-5	-4	-3	-2	-1

Negative Indexing

Negative indexing is also allowed for its sequences. In negative indexing, we refer to the last element using -1, the second to the last element can be accessed using -2, and -3 can be used to access the third to the last element , and so on. Let's illustrate this with 6 elements in a list (length = 6).

	D	A	N	I	E	L
Positive index →	0	1	2	3	4	5
Negative index →	-6	-5	-4	-3	-2	-1

Negative Indexing Example

Let's look at an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to access the elements of element from the last item
"""
# Negative list indexing
name = ['d', 'a', 'n', 'i', 'e', 'l']

print(name[-1]) # Last index
print(name[-2])
print(name[-3])
print(name[-4])
print(name[-5])
print(name[-6]) # First index
```

Output:

```
l
e
i
n
a
d
```

Negative Indexing Example

Let's look at an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to access the elements of element from the last item
"""
# Negative list indexing
multi_list = ['Danny', 'Onah', 31, 32]
print(multi_list[-1])
print(multi_list[-2])
print(multi_list[-3])
print(multi_list[-4])
```

Output:

```
32
31
Onah
Danny
```

Mutable Lists

Lists are Mutable!

As you can remember from previous lecture on **Strings (Lecture-5)**. We mentioned that Strings are "immutable", and this simply means *we cannot change the contents of a string after they are assigned*. We must always create or make a new string to make any changes. Let's refresh and revisit this with an example.

```
# Let's look at an example of String immutable
fruit = 'Apple'
fruit[0] = 'a'
```

```
line 3, in <module>
    fruit[0] = 'a'
TypeError: 'str' object does not support item assignment
```

```
# String converted to lower case
fruit = 'Apple'
fruitLower = fruit.lower()
print(fruitLower)
```

We can convert Strings from capital case to lower case! But this does not change the nature of the string.

Output:

apple

Lists are Mutable!

Lists are "mutable", this simply means unlike String, the elements in a list can be changed using the index operator. We can update and modify a list even after they are created and assigned values. Let's look at an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to demonstrate list mutable
"""
numbers = [ 16, 7, 18, 2, 1, 73] # list of numbers
print("Original numbers are: ",numbers)

# now let's perform list mutability - change one of the numbers in index 3
numbers[3] = 91
print("Updated numbers are:",numbers) # displaying the new updated numbers
```

```
Original numbers are:  [16, 7, 18, 2, 1, 73]
Updated numbers are: [16, 7, 18, 91, 1, 73]
```

Updating List Values: Examples

We can update single or multiple lists. Updating and modifying a list can be done using the assignment operator (=) , append() method and we can also use the extend() method to accomplish a list update task. Let's see an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to add or update a list
"""


number = [1, 2, 3, 4, 5]
number[0] = 9 # change the first item
print(number)

number[1:4] = [29, 30, 31] # change 2nd to 4th items
print(number)
```

Updating List Values: Examples

We can update single or multiple lists. Updating and modifying a list can be done using the assignment operator (=) , append() method and we can also use the extend() method to accomplish a list update task. Let's see an example to help us understand this better.

```
# appending and extending list in Python using the append
number = [13, 14, 15]
number.append(45) # appending or adding 45 to the list
print(number)
# Add several items using the extend() method
number.extend([50, 60, 70])
print(number)
# combining two list
number_extend = (number + [100, 101, 102])# add (+) operator to combine or concatenate two lists
print(number_extend)
```

Updating List Values: Outcomes

```
[9, 2, 3, 4, 5]
[9, 29, 30, 31, 5]
[13, 14, 15, 45]
[13, 14, 15, 45, 50, 60, 70]
[13, 14, 15, 45, 50, 60, 70, 100, 101, 102]
```

Nested List

A nested list is a list inside a list. This simply means a list created inside or within another list. This can also be said to be a list that contains a sub-list of elements.

For example, suppose we have a list called names and within the list creation expression (or initialization) we also have another list and sub-list. A nested list is typically made up of sub-lists separated by commas.

Let's look at an example to help us understand this better.

Nested List: Examples

```
#!/usr/bin/env python
"""
A program to demonstrate nested list
"""
name = ['danny', ['onah', ['dan', 'daniel'], 'mike', 'joe'], 'a', 'p']
# display the sub-list element in index 1 (i.e. '['onah', ['dan', 'daniel'], 'mike', 'joe']')
print(name[1])
print(name[2]) # display the element in index 2 (i.e. 'a')
print(name[1][1]) # display the second element in the sub-list index[1]
print(name[1][1][0]) # display the first element in the sub-list index[1][1]
```

Output:

```
['onah', ['dan', 'daniel'], 'mike', 'joe']
a
['dan', 'daniel']
dan
```

Functions with Nested List

We can use functions together with nested list. We can certainly add and remove items from the nested list using *append()*, *insert()*, *extend()*, *pop()*, *del*, *remove()* methods. We can also find the length (*len()*) and iterate through a nested list by using *for-loop*.

Let's look at an example to help us understand this better.

Functions with Nested List: Examples

```
#!/usr/bin/env python
"""
A program to use functions with nested list
"""
element = ['d', ['an', 'ie'], 'l']
element[1][1] = 5 # element in index 1 in the sub-list was replace by integer 5
print(element)
element[1].append('DD') # DD was added into the sub-list
print(element)
element[1].insert(0, 'KK') # KK is inserted as the first element in the list
print(element)
# extending the list with new values at the specify index of the list
element[1].extend([85, 98, 109]) # add the new list at the end of index [1] of the sub-list
print(element)
```

Functions with Nested List: Outcomes

```
['d', ['an', 5], 'l']
['d', ['an', 5, 'DD'], 'l']
['d', ['KK', 'an', 5, 'DD'], 'l']
['d', ['KK', 'an', 5, 'DD', 85, 98, 109], 'l']
```

Aliasing

Aliasing simply means when we assign one variable object to another variable. The both variables will then point to the same object. The object not referenced will then be garbage collected from the memory. Variables are storing objects, and when we assign one to another, all assigned variables will refer to the same object in memory. *When two identifiers refer to the same variable, this is called* **aliasing**.

Let's look at an example to help us understand this concept better.

List Aliasing

```
#!/usr/bin/env python
"""
A program to perform aliasing in list
"""
number1 = [23, 56, 97]
number2 = number1 # this is know as aliasing(both pointing to the same list object)
# check whether this is 'True' with a print statement
print(number1 is number2)
print(number1)
print(number2)
```

Output:

```
True
[23, 56, 97]
[23, 56, 97]
```

List Aliasing

In the above example, you notice that the same list has two different names, number1 and number2. When we try to execute an assignment statement, we can see that *number1* and *number2* relate to the same list of elements. This simply indicate that it is aliasing. When updating the order of the elements in *number1*, this will also update the equivalent element in *number2* vice versa.

Let's look at another example to help us understand this better.

Aliasing: Example

```
#!/usr/bin/env python
"""
A program to compare two aliasing list variables
"""
# before the aliasing
number1 = [23, 56, 97]
number2 = [23, 56, 97]

print(number1)
print(number2)

print(number1 == number2)
print(number1 is number2)
```

Output:

```
[23, 56, 97]
[23, 56, 97]
True
False
```

Aliasing: Example

```
# now with aliasing
number1 = number2
print(number1 == number2)
print(number1 is number2)

# update both variables by updating one
# This is common with aliasing objects, a change in one will update the other
#number1[0] = 30
number2[0] = 30
print(number1)
print(number2)
```

Output:

```
True
True
[30, 56, 97]
[30, 56, 97]
```

List Function Aliasing

In function aliasing, we create a new variable and assign the function reference to an existing function to a variable. We can verify that both the objects pointing to the same memory location using **id()** built-in function mechanism in Python.

Let's look at an example to help us understand this better.

List Function Aliasing: Example

```
def greetings(name):
print("Hello {}, a very good morning!!".format(name))

greeting = greetings
print(f"Check the memory_id of greetings(): {id(greetings)} ")
print(f"Check memory_id of greeting(): {id(greeting)} ")
greetings("Danny")
greeting("Danny")

name1 = ["Danny", "Daniel", "Onah"]
name2 = name1
print(f'Memory_id name1: {id(name1)}')
print(f'Memory_id name2: {id(name2)}')
```

List Function Aliasing: Outcomes

Output:

Check the memory_id of greetings(): 4368318128

Check memory_id of greeting(): 4368318128

Hello Danny, a very good morning!!

Hello Danny, a very good morning!!

Memory_id name1: 4368659200

Memory_id name2: 4368659200

Cloning Lists

Cloning Lists

In python cloning a list is the process of copying the list. *If we want to change a list and at the same time keep a copy of the original list, we must copy the whole list and not just only the reference to the list.* We would use a slicing method to allow us to easily clone the list. The slicing process is the simplest and quickest way to clone a list. In cloning, when updating one list, the original list is unchanged. Cloning works differently from aliasing. *As you know in aliasing, when one of the lists is updated, both lists are updated as well.*

Let's look at an example to help us understand this concept better.

Cloning Lists: Example

```
#!/usr/bin/env python

"""
A program to clone a list
"""

number1 = [333, 444, 555]
number2 = number1[:] # make a clone using the slice
print(number1 == number2) # comparing if both equal
print(number1 is number2) # checking if both is same

# update the value of the second list
number2[0] = 666 # updating the value of one list does not have any effect on the other list
print(number1)
print(number2)
```

Output:

```
True
False
[333, 444, 555]
[666, 444, 555]
```

Cloning : Using Slicing

This is the most straightforward method of cloning a list. We can clone a list and keep the original copy using this approach. This approach uses a function to perform this task.

Let's look at an example to help us understand this concept better.

Cloning : Using Slicing

```
#!/usr/bin/env python
"""
A program to clone a list using a slice method and function
"""
def cloning_List(item):
    number = item[:] # slice operator
    return number
number1 = [35, 45, 55, 65, 75]
number2 = cloning_List(number1)
print("Original List: ", number1)
print("After Cloning: ", number2)
```

Output:

Original List: [35, 45, 55, 65, 75]

After Cloning: [35, 45, 55, 65, 75]

Cloning: Using Extend() Method

The *extend()* function can be used to copy the element of a list into a new list. Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python
"""
A program to clone a list using extend()function
"""
def cloning_List(item):
        number = [] # creating an empty list
        number.extend(item)
        return number
number1 = [69, 79, 89, 99]
number2 = cloning_List(number1)
print("Original List: ", number1)
print("After Cloning: ", number2)
```

Output:

```
Original List:  [69, 79, 89, 99]
After Cloning:  [69, 79, 89, 99]
```

Cloning: Using *list()* Methods

Using the built-in function *list()* is also another simple way to clone a list. Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python
"""
A program to clone a list using a list() function
"""
def cloning_List(item):
        number = list(item)
        return number
number1 = [11, 22, 33, 44, 55]
number2 = cloning_List(number1)
print("Original List: ", number1)
print("After Cloning: ", number2)
```

Output:

```
Original List:  [11, 22, 33, 44, 55]
After Cloning:  [11, 22, 33, 44, 55]
```

Cloning: Using List Comprehension

We use the *list comprehension approach to copy all the elements from one list to another*. Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python
"""
A program cloning a list using list comprehension approach
"""
def cloning_List(item):
        number = [ i for i in item] # list comprehension approach
        return number
number1 = [40, 50, 60, 70, 80]
number2 = cloning_List(number1)
print("Original List: ", number1)
print("After Cloning: ", number2)
```

Output:

```
Original List:  [40, 50, 60, 70, 80]
After Cloning:  [40, 50, 60, 70, 80]
```

Cloning: Using the *append()* Method

The *append()* method can be used to append or add elements to the list or copy them to a new list. It's used to add elements to the list's last location or position.

Let's look at an example to help us understand this concept better.

Cloning: Using the *append()* Method

```
#!/usr/bin/env python
"""
A program to append to a list using function
"""
def cloning_List(item):
        number = [] # empty list
        for i in item: number.append(i)
        return number1

number1 = [34, 56, 78, 98]
number2 = cloning_List(number1)
print("Original List: ",number1)
print("After Cloning: ", number2)
```

Output:

Original List: [34, 56, 78, 98]

After Cloning: [34, 56, 78, 98]

Cloning: Using *copy()* Method

We can also copy all the elements of a list to anther list using the copy() method. Let's look at an example to understand this concept better.

```
#!/usr/bin/env python
"""
A program to copy a list using a copy() method
"""
def cloning_List(item):
        number = [] # empty list

        number = item.copy()
        return number

number1 = [12, 13, 14, 15]
number2 = cloning_List(number1)
print("Original List: ", number1)
print("After Cloning: ", number2)
```

Output:

Original List: [12, 13, 14, 15]
After Cloning: [12, 13, 14, 15]

Lists and built-in Functions

List Parameters

The list() constructor takes a single argument such as a string, tuples, sets, dictionary or an iterative object. We can also pass a list as an argument.

Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python

name_list = ['Danny', 'Onah', 'Daniel', 'Dan']
print(list(name_list))

def list_name(item):
    print(item)
name_list = ['Danny', 'Onah', 'Daniel', 'Dan']
list_name(name_list)
```

Output:

['Danny', 'Onah', 'Daniel', 'Dan']
['Danny', 'Onah', 'Daniel', 'Dan']

List Operations

List generally supports + and * operators for list concatenation and list repetition. But there are other *list methods* that can be used to perform the same task. In the previous examples, we used **slice**, **append** , and **extend** method to perform an operation on a list.

Let's look at and example to help us understand this concept better.

List Operations: Examples

```
#!/usr/bin/env python
"""
A program to demonstrate basic list operations
"""
name_letter = ['d', 'a', 'n', 'i', 'e', 'l']

print(name_letter + ['x', 'y', 'z'])# A '+' operator used to combine two lists together
print(2 * name_letter) # A * operator for repetition.

print(name_letter[1:3]) # display element from 1st to 3rd

# Appending and extending lists in Python using the append()
name_letter.append('Onah')
print(name_letter)
# add several items using extend() method
name_letter.extend(['a','b', 'c'])
print(name_letter)
```

List Operations: Outcomes

```
['d', 'a', 'n', 'i', 'e', 'l', 'x', 'y', 'z']
['d', 'a', 'n', 'i', 'e', 'l', 'd', 'a', 'n', 'i', 'e', 'l']
['a', 'n']
['d', 'a', 'n', 'i', 'e', 'l', 'Onah']
['d', 'a', 'n', 'i', 'e', 'l', 'Onah', 'a', 'b', 'c']
```

len() Function!

Just like String, we can also use the *len()* function to find the length of the list. The *len()* function takes a list as a parameter and returns the number of elements in that list. Actually, *len()* tells us about the number of elements of any set or sequence (such as a string etc.)

Recall, to get the length of a string we do the following:

```
#!/usr/bin/env python
"""
A program to find the length of a string
"""
greet = 'Hello Danny' # define a string
# display the length inclusive of the space
print("The length of the string is: ", len(greet))
```

Output: The length of the string is: **11**

len() Function!

Just like String, we can also use the *len()* function to find the length of the list. The *len()* function takes a list as a parameter and returns the number of elements in that list. Actually, *len()* tells us about the number of elements of any set or sequence (such as a string etc.)

While string include spaces in the computation of the length, *list does not automatically include spaces as part of the length*. To get the length of a list of items or elements we do the following:

```
#!/usr/bin/env python
"""
A program to compute the length of a list
"""
# list of elements, combination of numbers and strings
listItems = [3, 20, 'danny', 100.5]
print("The length of the list is: ", len(listItems)) # display the length of the list
```

Output: The length of the list is: 4

range() Function

The *range()* function *returns a list of numbers* that range from (or between) zero to one less than the parameter. In the range() function, we can construct an index loop using *for-loop* and use an integer for the *iteration*.

Let's look at an example to help us understand this better.

range() Function

```
#!/usr/bin/env python

"""
A program to use the range() function of a list
"""
# use the range function to print the number in a for loop
for i in range(4):
        print(i)
        print(range(4))

# finding the length of a list
friends = ['Danny', 'Elaine', 'Jane']
print(len(friends))

# finding the length of a range of list
print(range(len(friends)))
# Run this code using: python FileName.py (e.g.- list_range.py)
```

Output:

```
0
1
2
3
[0, 1, 2, 3]
3
[0, 1, 2]
```

Two *for-loops*...

We can also use two for-loops to display the list of items in the list.

```
#!/usr/bin/env python

"""
A program using two loops in a list
"""
friends = ['Danny', 'Elaine', 'Jane']
# first the usual for-loop
for friend in friends: # this auto compute the range
        print('First - Happy New Year: ', friend)
# second for-loop using the length as parameter
for i in range(len(friends)): # here we define the range
        friend = friends[i]
        print('Second - Happy New Year: ', friend)
```

Output:

```
First - Happy New Year:  Danny
First - Happy New Year:  Elaine
First - Happy New Year:  Jane
Second - Happy New Year:  Danny
Second - Happy New Year:  Elaine
Second - Happy New Year:  Jane
```

List Concatenation

We can concatenate two or more lists together using the plus (+) symbol like strings. We can *create a new list by adding two existing lists together*. Let's see an example to help us understand this better.

```
#!/usr/bin/env python

number1 = [0, 1, 2, 3, 4] # defining the first list
number2 = [5, 6, 7, 8, 9] # defining the second list
numbers = number1 + number2 # concatenate the two variables together
print(numbers) # display output
```

Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

List Slice

We can also slice a list just as we did with a string. Here note that the second number is "up-to but not to inclusive" i.e. exclusive of the stop position.

```
#!/usr/bin/env python

# A program to perform list slice
number = [90, 13, 1, 3,43, 23, 71]
print(number[1:4]) # slice from index 1 through index 3
print(number[:5]) # slice from index 0 through index 4
print(number[2:]) # slice from index 2 through the list index 6
print(number[:]) # slice through all elements in the list
```

```
[13, 1, 3]
[90, 13, 1, 3, 43]
[1, 3, 43, 23, 71]
[90, 13, 1, 3, 43, 23, 71]
```

Methods in Lists

List Methods

We can easily used built-in methods to perform our tasks successfully. We can accomplish our task by directly using these list methods.

Built-in list methods	Description
list.append(obj)	This method appends object **obj** to the list
list.count(obj)	This method returns a count of how many times **obj** occurs in the list
list.extend(seq)	This method appends the contents of **seq** to the list
list.index(obj)	This method returns the lowest index in the list that **obj** appears
list.insert(index, obj)	This method inserts object **obj** into the list at offset index
list.pop(obj=list[-1])	This method removes and returns the last object or **obj** from the list
list.remove(obj)	This method removes object **obj** from the list
list.reverse()	This method reverse objects of the list in place
list.sort([func])	This method sorts objects of the list

List Methods

```
>>> x = list()
>>> type(x)
<type 'list'>
>>> dir(x)
['append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

http://docs.python.org/tutorial/datastructures.html

Review List Creation

We can create and build a list from scratch. This can be done using the following steps.

- We can create an empty list and then add elements using the append method or function.
- The list stays in order and new elements are added at the end of the list.

Let's look at an example to help us understand this better.

List creation & update

```
#!/usr/bin/env python
"""
A program to append items to a list
"""
listItem = list() # creating an empty list
listItem.append('watch') # add first item to the list
listItem.append(1555) # add second item to the list
print(listItem) # display the items added
listItem.append('coke') # add a third item to the list
print(listItem) # print the full list of items.
```

Output:

['watch', 1555]

['watch', 1555, 'coke']

Viewing a List

In Python we can search whether an item is in a list. Python provides two operators that let you check if an item is in a list. These are logical operators that return True or False. They do not modify the list, they are used to check the Boolean condition of True or False.

```
#!/usr/bin/env python
"""
A program to check if an item is in the list
"""
number = [4, 65, 7, 11]# define a list
print(7 in number) # check whether 7 is in the list
print(3 in number) # check whether 3 is in the list
print(20 not in number) # check whether 20 is not in the list
```

Output:

True
False
True

Lists Order

As you know, list are collection of items that are in order. A *list* can store many items and keeps those items in the order they are entered or stored until we decided to do something with it to change the order. A *list* can be sorted (i.e. change its order).

```
#!/usr/bin/env python

"""
A program to sort a list of items
"""
friends = ['Elaine', 'Jane', 'Danny'] # declare a list
friends.sort() # sort the list in alphabetical order
print(friends)# display the items/friends in the sorted list
print(friends[1])# display the item in index 1 from the sorted list
```

```
['Danny', 'Elaine', 'Jane']
Elaine
```

Using Empty List

We can append input values to an *empty list*. Let's look at a program that would take number inputs and store them in an empty list and then perform the average of the numbers.

Let's see an example to help us understand this better.

Example

Let's look at an example of a program to calculate the average of certain numbers.

```
#!/usr/bin/env python

"""
A program to calculate the average
"""
total = 0
count = 0
while True:
        number = input('Enter a number: ‘)
        if number == 'finish’: # stop prompting for input, display result
                break
        value = float(number)
        total = total + value
        count = count + 1
average = total/count # compute the average of the numbers
print('Average', round(average,2))
```

Output:

```
Enter a number: 32
Enter a number: 78
Enter a number: 90
Enter a number: 57
Enter a number: finish
Average 64.25
```

Using Empty List

```
#!/usr/bin/env python

"""
A program to calculate the average of numbers and appending a value to the list
"""
numberList = list() # create an empty list to hold the values
while True:
        number = input('Enter a number: ‘)
        if number == 'finish': # when finish is entered, the program stops taking input then display the result
                break
        value = float(number) # convert the input to floating-point number
        numberList.append(value) # add the numbers to the empty list (numberList)
# compute the average of the numbers
average = sum(numberList)/len(numberList)
print(('Average: ', round(average,2)))
```

Output:

```
Enter a number: 32
Enter a number: 78
Enter a number: 90
Enter a number: 57
Enter a number: finish
Average:  64.25
```

Strings and List

Strings and List operations are similar. Some operations you can perform with a string can also be done using a list. Using the *Split()* function breaks a string into parts and produces a list of strings. We think of these as words. We can access a particular word or loop through all the words.

Let's look at an example to help us understand this better.

Strings and List: *split()*

```
#!/usr/bin/env python
"""
A program to demonstrate using split() function create list of words
"""
sentence = 'Programming in Python is interesting!'
sentProcess = sentence.split()
print(sentProcess)
# compute the length of the sentence
print("The length of the sentence is: ", len(sentProcess))
for w in sentence:
        print(w, end='')
print("\n")
```

Output:

['Programming', 'in', 'Python', 'is', 'interesting!']
The length of the sentence is: 5
Programming in Python is interesting!

Delimiter and *split()* Function

Delimiter is very important for strings and list. When you do not specify a delimiter, *multiple spaces are treated like one delimiter*.

Let's look at an example to see how this works.

Delimiter and Split() Function

```
#!/usr/bin/env python

"""
A program to compute delimiter and space in string
"""
sentence = 'There is a lot of spaces here!'
sentSplit = sentence.split()
print(sentSplit)
print(len(sentSplit))

sentence = 'There-is-a-lot;of-space;here!’
# splitting the sentence with the delimiter
sentDelimeter = sentence.split()

print(sentDelimeter)
print(len(sentDelimeter))
# using the delimiter to print the length
sentSplit = sentence.split(';’)
print(sentSplit)
print(len(sentSplit))
```

Output:

```
['There', 'is', 'a', 'lot', 'of', 'spaces', 'here!']
7
['There-is-a-lot;of-space;here!']
1
['There-is-a-lot', 'of-space', 'here!']
3
```

Navigating List: Operators & Methods

We can manipulate the elements in the list using some operators and methods. A list element can be access through a range of items in a list by using the slicing operator. Which is represented by a colon (:). It is possible to add and change an element in a list by using the assignment operator (=).

del: The keyword *del* can be used to remove one or more items from a list. It can completely delete the list.

delete(): The *delete()* method can be used to remove a specific object from the list.

pop(): The *pop()* method can be used to remove an item at a specific index. It is used to remove the last element from the list.

clear(): The *clear()* method is used to empty a list.

Navigating List: Operators & Methods

Let's look at an example to help us understand how the operators and methods are used:

```
#!/usr/bin/env python
"""
A program to manipulate and navigate through the elements in a list
"""
name = ['d', 'a', 'n', 'i', 'e', 'l']
print(name[2:4]) # elements in position 3rd to 4th (index[3] inclusive)
print(name[:-5]) # element from the beginning to the 5th negative index
print(name[3:]) # from the 4th element on the list to the end
print(name[:]) # all elements from the beginning to the end
```

Navigating List: Outcomes

```
['n', 'i']
['d']
['i', 'e', 'l']
['d', 'a', 'n', 'i', 'e', 'l']
```

Deleting Item From a List: *del()*

Let's look at an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to delete items from list
"""
name = ['d', 'a', 'n', 'i', 'e', 'l']

# deleting elements
del name[2] # deleting the item in index 2
print(name) # display the current element in the list

# deleting multiple items from a list
del name[1:5] # deleting multiple items from index 1 to the last index
print(name) # display the element
```

Deleting Item From a List: Outcomes

```
['d', 'a', 'i', 'e', 'l']
['d']
```

remove() Method - List

The *remove()* method allows us to remove items from the list. Let's see an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to remove item from a list
"""
# this program uses the remove() method to remove item from a list

name = ['d', 'a', 'n', 'i', 'e', 'l']
name.remove('n')
print(name)
```

Output:

```
['d', 'a', 'i', 'e', 'l']
```

pop() Method in List

Let's see an example of how pop() method is used to help us understand this better.

```
#!/usr/bin/env python
"""
A program to demonstrate the pop() method
"""
name = ['d', 'a', 'n', 'i', 'e', 'l']
print(name.pop(1))# pop and display the element in index 1.
print(name.pop()) # pop and display the last element in the list
```

Output:

a

l

clear() Method In List

We can use the *clear()* method to empty the list and remove all elements from the list. Let's see an example to help us understand how this is done.

```
#!/usr/bin/env python
"""
A program to clear the list
"""
name = ['d', 'a', 'n', 'i', 'e', 'l']
name.clear()# this will clear the list item - empty the list
print(name) # this will print empty list
```

Output:

[ ]

del Entire List

Let's see what happens when we try to delete entire list and display the outcome. If we try to print a list that is already deleted, this will display error , list not define.

```
#!/usr/bin/env python
"""
A program to delete the entire list and print the outcome
"""
name = ['d', 'a', 'n', 'i', 'e', 'l']
del name # deleting the entire list
print(name) # trying to print the deleted list
```

```
Traceback (most recent call last): line 5, in <module>
    print(name) # trying to print the deleted list
NameError: name 'name' is not defined
```

Loops and Lists

For-loop and Lists

We should now be able to work on much more interesting programs using for-loop and lists. Using for-loop in a list, we are instructing our program to perform repetitive actions during the execution. We will see a few examples on how to use for-loop to display various list items or elements.

Note that before we can use a for-loop successfully, we should first think of where to store the result of the loops. The best way we can do this is with lists. As you know that *a list is a storage container for holding elements and organizing them in order from the first to the last element*.

Let's see an example of how to make a list to help us understand better.

```
hair = ['brown', 'blond', 'dark grey']
eyes = ['brown', 'blue', 'green']
weights = [67.8, 80.1, 54.9]
```

List Initialization

In declaring or initializing a list, we start the list with the left square bracket ('[') which opens the list. Then we put each element of the list separated by commas, this process is similar to function arguments. Lastly, end or close the list with a right square bracket (']') to indicate that the list initialization or creation expression is finish.

Python interpreter then takes the list and all its elements and assigns them to the variable.

We've Seen Lists!

In the previous lectures we have seen lists and how this could be used with strings and with for-loop etc. Let's see a simple example on how this is used with a loop.

```
for i in [1, 2, 3, 4]:
    print(i)
```

Output:

```
4
3
2
1
```

Lists and Definite Loops

We can use loops to iterate over elements in the list. The looping is done from one element to another in the list and these are displayed separately after the loop has completed its circle. Let's see an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to wish so many friends in a list happy new year using for loop.
"""
friends = ['Danny', 'Elaine', 'Jane'] # list of friends
for friend in friends: # for-loop to iterate over the list of friends
    print('Happy New Year: ', friend) # display the greetings and the friends
```

Output:

```
Happy New Year:  Danny
Happy New Year:  Elaine
Happy New Year:  Jane
```

For-loops & Lists Examples

Now let's see how we could build some lists using some for-loop constructs and display the outcome.

```
#!/usr/bin/env python

"""
A program to build some lists using for-loops and display the outcomes.
"""
count = [1, 2, 3, 4, 5, 6, 7]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# this first kind of for-loop goes through a list
for number in count:
    print(f'This is count: {number}')
```

For-loops & Lists Examples

Now let's see how we could build some lists using some for-loop constructs and display the outcome.

```
# This for loop will be same as above goes through the list

for fruit in fruits:
        print(f"A fruit of type: {fruit}")

# Here we also go through a mixed lists too
# notice we have to use {} because we don't know what's in it

for i in change:
        print(f"I got: {i}")
```

For-loops & Lists Examples

Now let's see how we could build some lists using some for-loop constructs and display the outcome.

```
# We can also build list, lets first start with an empty list or empty one
items = [] # this is an empty list

# then use the range function to iterate from 0 to 5 counts
for i in range(0,6):
        print(f"Adding {i} to the list.")
# append is a function that lists understand to add elements to it
        items.append(i)

# now we can display or print the newly added items out too
for i in items:
        print(f"The item was: {i}")
```

For-loops & Lists Outcomes

Now let's see how we could build some lists using some for-loop constructs and display the outcome.

```
This is count: 1
This is count: 2
This is count: 3
This is count: 4
This is count: 5
This is count: 6
This is count: 7
```

```
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
```

```
I got: 1
I got: pennies
I got: 2
I got: dimes
I got: 3
I got: quarters
```

```
Adding 0 to the list.
Adding 1 to the list.
Adding 2 to the list.
Adding 3 to the list.
Adding 4 to the list.
Adding 5 to the list.
```

```
The item was: 0
The item was: 1
The item was: 2
The item was: 3
The item was: 4
The item was: 5
```

While Loops & List

A *while-loop* will continue executing the code block under it as long as a Boolean expression is True. Remember that when we write our code construct (e.g., *if-statement*, *for-loop*, *functions* and *while-loop* etc.) and end this with a colon (:) this tells Python to start a new block of code. This in turn will indent a new code body for us to type the next instruction to help us structure our program so that Python knows how to execute the code.

While loops also perform a task like if-statement, *but instead of running the code block once, while loop execution allows the program to loop to the 'top' where the while loop is and repeat*. A wile loop runs until the expression is False.

As you know while loops sometimes do not stop. This is great if your intention is to just to allow your program to keep looping.

While-loop & List Example

Let's look at an example to help us understand this better.

```
#!/usr/bin/env python

"""
A program to use while-loop with the list
"""
i = 0 # initialising variable i by 1
numbers = [] # an empty list to store the numbers
while i < 6:
        print(f"At the top of i is {i}")
        numbers.append(i) # add the values in the empty list

        i = i + 1 # very important to avoid infinite loop
        print("Numbers now are: ", numbers) # printing numbers stored in the empty list
        print(f"At the bottom i is {i}") # displaying the iterated-loop number
print("The numbers: ")
for number in numbers:
        print(number)
```

While-loop & Lists Outcomes

```
At the top of i is 0
Numbers now are:  [0]
At the bottom i is 1
At the top of i is 1
Numbers now are:  [0, 1]
At the bottom i is 2
At the top of i is 2
Numbers now are:  [0, 1, 2]
At the bottom i is 3
At the top of i is 3
Numbers now are:  [0, 1, 2, 3]
At the bottom i is 4
At the top of i is 4
Numbers now are:  [0, 1, 2, 3, 4]
At the bottom i is 5
At the top of i is 5
Numbers now are:  [0, 1, 2, 3, 4, 5]
```

```
At the bottom i is 6
The stored numbers are:
0
1
2
3
4
5
```

Searching and Sorting List

Searching and Sorting

Searching in List

Searching list means we want to check whether the element is present in the list or not. We have done a little bit of searching in the previous sessions using the *in* keyword. Searching a list simply checks if a list contains particular item. We use the (*in*) operator to accomplish the searching mechanism in list.

Let's look at an example to help us understand this concept better.

Searching in List:Example

```
#!/usr/bin/env python
"""
A program to search for item in a list
"""
names = ['Daniel', 'Danny', 'Dan', 'Onah']

# search whether the name is in the list
if 'Dan' in names:
    print('Yes, Dan is in the list of: ', names)

# searching the index of an element in the list
dan_pos_index = names.index('Dan')
print("The index position of Dan is: ",dan_pos_index)
```

Output:

Yes, Dan is in the list of: ['Daniel', 'Danny', 'Dan', 'Onah']

The index position of Dan is: 2

Searching in List: Example

```
#!/usr/bin/env python
"""
A program to compute a flower list search
"""
while True:
        flowerList = ['cactus','lotus', 'rose', 'queen of the  night','sunflower','hibiscus', 'lily', 'bluebell’]
        flowerName = input("Enter a flower name: ") # taking input from user
        if flowerName == 'done’:
                print("You are exiting. Goodbye!")
                break
        if flowerName.lower() in flowerList:
                print("%s is found in the list"%(flowerName))
        else:
                print("%s is not found in the list"%(flowerName))
```

Searching in List: Outcome

```
Enter a flower name: lily
lily is found in the list
Enter a flower name: queen of the night
queen of the night is found in the list
Enter a flower name: rose flower
rose flower is not found in the list
Enter a flower name: lotus
lotus is found in the list
Enter a flower name: tulip
tulip is not found in the list
Enter a flower name: done
You are exiting. Goodbye!
```

Sequential Search

Sequential search is also called linear search. This is the process of searching an element sequentially in the order in which they appear in the list. For example, let's look at the key factors to consider when conducting a linear or sequential search:

- Begin with the list's leftmost element and compare **x** to each of the list's elements one after the other.
- Return True if **x** matches an element
- Return False if **x** does not match any of the elements

Let's look at an example to help us understand this concept better.

Sequential Search: Example

```
#!/usr/bin/env python

"""
A program to perform linear search on a list
"""
def linear_search(list, item):
        for i in range(len(list)):
                if list[i] == item:
                        return True
        return False
list_item = [59, 69, 'Danny', 33.3, 'Onah', 156]
name = 'Danny'
if linear_search(list_item, name):
        print("%s is Found!"%(name))
else:
        print("%s is Not Found!"%(name))
```

Output:

Danny is Found!

Binary Search

The binary search technique is to find or search elements from a list of elements. Below are the properties of a binary search:

- The first searching process starts by comparing the search element with the middle element in the list.
- If the search element is smaller than the middle element, it will search the left side of the list and repeat it.
- If the search element is greater than the middle element, it will search the right side of the list and repeat it.

Binary Search

- The binary search process is implement using recursion.
- If an element is search in the middle, left, and right part of the list, then it displays the element found.
- The worst-case time complexity is log n. It is efficient than linear search algorithm.

Let's look at an example to help us understand this concept better.

Binary Search: Example

```
#!/usr/bin/env python
"""
A program to perform binary search on a list
"""
# A program for iterative Binary Search Function
def binary_search(item, var):
        start = 0
        end = len(item) - 1
        middle = 0
```

Binary Search: Example

```
while start <= end:
        middle = (start + end)// 2 # floor division that would    produce only integer value
        if (item[middle] < var):
                start = middle + 1
        elif item[middle] > var:
                end = middle - 1
        else:
                 return middle
return -1
```

Binary Search: Example

```
# Testing the list
number = [1, 56,89, 98, 115]
#var = 115
var = int(input("Enter the number: "))
# calling the function
result = binary_search(number, var)
if result != -1:
        print("Item %d is present at index number[%s]"%((var), str(result)))
        #print("Item %d is present at index"%(var), str(result))# we display     this way as well
else:
         print("Item is Not present in the list")
```

Output:

Enter the number: 89

Item 89 is present at index number[2]

Binary Search Recursion: Example

```
#!/usr/bin/env python
"""
A program to compute binary search using recursive function
"""
# A program using recursive function in binary search
def binary_search(item, low, high, var):
        # check the base case
        if high >= low:
                middle = (high + low)//2 # floor division
                if item[middle] == var:
                        return middle
                elif item[middle] > var:
                         return binary_search(item, low, middle - 1, var)# recursive                                function
call
                else:
                        return binary_search(item, middle + 1, high, var)
        else:
                return -1 # means element is not present in the list here
```

Binary Search Recursion: Example

```
#Testing list

number = [1, 56,89, 98, 115]
#var = 56
var = int(input("Enter the number: "))

# function call or invocation
result = binary_search(number, 0, len(number) -1, var)
if result != -1:
        print("Item is present at index", str(result))
else:
        print("Item is not present in list")
```

Output:

Enter the number: 56

Item is present at index 1

Passing List to Function

Passing a list to function means to pass a list variable to a function, and the function copy the value of the list variable to the function parameters. This becomes very easy to display all the values of the list through the function parameters. Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python
"""
A program passing list to a function
"""
def my_name(names):
        for name in names:
            print(name)

name_list = ["Danny", "Onah", "Daniel", "Dan"]
my_name(name_list)# passing a list into a function call
```

Output:

Danny
Onah
Daniel
Dan

Returning List from a Function

Returning list to function means to return the value of the list to the calling function. What this simply means is that we are assigning a calling function to another variable to hold the values of the operation. This value is then displayed from the variable storing the result from the calling function.

Let's look at an example to help us understand this concept more.

Returning List from a Function

```
#!/usr/bin/env python
"""
A program to return a list from a function
"""

def my_name(names):
        for name in names:
            return(name)

name_list = ["Danny", "Onah", "Daniel", "Dan"]

for name in name_list:
        my_name(name)
        print(name)
```

Output:

Danny
Onah
Daniel
Dan

filter() Function

A *filter()* function filters an iterable by using a function to check if each element in the iterable list is valid or not. A *filter()* function returns an iterator that has been passed and part of the iterable list through the function check.

Filter() function Syntax:

```
filter(function, iterable)
```

filter() Function

There are two parameters passed into the *filter()* function. These are the function name and iterable list.

- **function name**: It returns true if elements are iterable (are in the list), false otherwise.
- **iterable**: It means that the iterable to be filtered could be concepts from sets, list, tuples, to containers of any iterators.

Let's look at an example to help us understand this concept better.

filter() Function: Example

Let's write a program to filter a list of numbers and return a new list with only the values that met the condition set within the function.

```
#!/usr/bin/env python
"""
A program to use filter function with a list.
This program should return a new list with only values equal to or above 21.
"""
number = [12, 21, 34, 16, 90, 20, 56]
def my_number(number):
        if number < 21: # checking the condition
                return False
        else:
                return True
numbers = filter(my_number, number)
for n in numbers:
         print(n)
```

Output:

21
34
90
56

filter() Function: Example

```
#!/usr/bin/env python
"""
A program to use the filter()function to compute a vowel list
"""
# A function that filters vowels
def vowel_funct(n):
        vow = ['a', 'e', 'i', 'o', 'u']
        if(n in vow):
                return True
        else:
                return False
# Let's create a sequence
sequence = ['d','a','n','i','e','l','o','n', 'a', 'h']
filter_vowels = filter(vowel_funct, sequence)
print('The filtered vowel letters are: ')
for letter in filter_vowels:
        print(letter)
```

Output:

The filtered vowel letters are:

a
i
e
o
a

Sorting Lists

Sorting Lists

Sorting is the process of arranging the list data in either ascending or descending order. The primary task of sorting algorithms is to organize data elements in a particular order. *If we sort our data properly, then this will be very easy to search the element fast*. Sorting is used to maintain data structure and could also be used to display list or any data in a more readable format. Sorting could be used in different *real-life context*, for example in sorting a *telephone directory*, *dictionary*, and so on.

In Python, the *sort()* method is used for sorting elements or items of a given list in a specific order either in ascending or descending order.

Sorting Lists

The syntax for the *sort()* method is:

```
list.sort(key= …, reverse = …)

sorted(list, key = …, reverse = …)
```

A very important point to note, which we must always remember is the basic difference between *sort()* and *sorted()* methods. The *sort()* methods changes the list directly as they are passed to it and does not return any value. While the *sorted()* method does not change the list and it only just returns the result of the sorted list.

Within the sort() and sorted(), there are two parameter lists which are known as *reverse* and *key*. The sorted list is **reversed** if the *reverse* parameter is *True*, while the *key* parameter is used as a sort comparison key.

Types of Sorting

The following are the various types of sorting:

- ***In-place sorting:*** *In this sorting technique, the elements are sorted within the list. There is no extra space needed for sorting an element.*
- ***Not-in-place sorting:*** *In this sorting technique, the elements are sorted, but a different list is required to complete this. There is extra space needed for sorting the elements.*
- ***Stable sorting:*** *In a sorting algorithm, stable sorting occurs when the contents do not change the sequence of the related content in which they appear after sorting.*
- ***Unstable sorting:*** *After sorting, if the contents change the sequence of similar content in which they appear, it is called unstable sorting in a sorting algorithm.*

Sorting Lists: Example

```
#!/usr/bin/env python
"""
A program to perform list sorting
"""


# Sort the list in ascending order
vowels_list = ['i','a', 'u', 'o', 'e'] # vowels
vowels_list.sort()# sort the vowels
print('Sorted list in ascending order: ', vowels_list)# print vowels

# Sort the list in descending order
vowels_list = ['i','a', 'u', 'o', 'e'] # vowels
vowels_list.sort(reverse=True)# sort the list in descending order
print("Sorted list in descending order: ", vowels_list)
```

Sorting Lists: Example

```
# Custom sorting with key
name = ['Daniel', 'Dan', 'Onah', 'Danny']
# sorting based on the number of characters or length of the string
print("Sorted list based on length of string: ", sorted(name, key=len))

# using a function to sort the list by the length of the values or strings
def sort_funct(n):
        return len(n)

name = ['Daniel', 'DanielOnah', 'Onah', 'Danny']
name.sort(key=sort_funct)
print("Sorted list based on the length of the string: ",name)
```

Sorting Lists: Output

```
Sorted list in ascending order:  ['a', 'e', 'i', 'o', 'u']
Sorted list in descending order:  ['u', 'o', 'i', 'e', 'a']
Sorted list based on length of string:  ['Dan', 'Onah', 'Danny', 'Daniel']
Sorted list based on the length of the string:  ['Onah', 'Danny', 'Daniel', 'DanielOnah']
```

Bubble Sort

Bubble sort is a comparison-based algorithm that compare each pair of adjacent elements and **swaps** them if they are out of the ordered list. Since the average and worst-case complexity of this algorithm are O(n2), where n is the number of items, it is not suitable for large data sets. The bubble sorting procedure is as follows.

Let's begin this with a list of integer numbers.

number = [2, 7, 9, 5, 4, 1]

Let's follow through this iteration below to sort this numbers:

First iteration

[2, 7, 9, 5, 4, 1]

Here we first compare the first two elements 2 > 7, since 2 is not greater than 7, there will be no swap, our list remain the same.

Bubble Sort

In the second comparison: $7 < 9$, there will be no swap, our list remain the same $[2, 7, 9, 5, 4, 1]$.

In the third comparison: $9 > 5$, there is a swap, the new list is $[2, 7, 5, 9, 4, 1]$

In the fourth comparison: $9 > 4$, then swap, the new list is $[2, 7, 5, 4, 9, 1]$

In the fifth comparison: $9 > 1$, then swap, the new list is $[2, 7, 5, 4, 1, 9]$

Now we have completed the first set of iteration, and we succeeded in getting the largest element to the end of the list. Now we need to follow the process again from the first element to the ***len(number) – 1*** to complete the rest of the iteration.

At the end of the completed iterations, we should have bubble sorted our list to be $[1, 2, 4, 5, 7, 9]$.

Bubble Sort: Example

Let's look at an example to help us understand how this is done in Python.

```
#!/usr/bin/env python
"""
A program to perform bubble sort on list
"""
def bubble_sort(number):
        for i in range(0, len(number)-1): # from the first element to the last
                for j in range(len(number) - 1):
                        if(number[j] > number[j + 1]):
                                temp = number[j]
                                number[j] = number[j + 1]
                                number[j + 1] = temp
        return number_list
number_list = [2, 7, 9, 5, 4, 1]
print("The unsorted list is: ", number_list)
print("The sorted list is: ", bubble_sort(number_list)) # function call
```

Bubble Sort: Output

```
The unsorted list is: [2, 7, 9, 5, 4, 1]
The sorted list is: [1, 2, 4, 5, 7, 9]
```

Selection Sort

Selection sort is used to sort a list of items in ascending and descending order. In selection sort, the first element in the list is chosen and compared to the rest of the list's elements repeatedly. Both elements are **swapped** if one is smaller than another (in ascending order). The element in the second place in the list is then selected and compared to the remaining elements in the list. Both elements are swapped if one of them is smaller than the other. This process is repeated until all the items in the list have been sorted. The selection sorting procedure is as follows:

Suppose we have a list that contains the following elements: [3, 5, 1, 2, 4]

We will firstly start with the list that has not been sorted: 35124

All the list elements can be found in the unsorted portion. We examine each item and decide that the smallest element is **1**. As a result, we swap 1 with 3, this becomes 15324. The remaining unsorted element becomes [5, 3, 2, 4]. Now 2 becomes the smallest number. We now swap 2 with 5.This will result to: 12354 and so on.

Selection Sort: Example

This selection sort procedure is repeated until the list has been sorted fully as: 12345.

Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python
"""
A program to perform selection sort on a list
"""
def selection_sort(number):
        for i in range(len(number) - 1):
                num = i
                for j in range(i + 1, len(number) - 1):
                        if number[j] < number[num]:
                                num = j
                number[i], number[num] = number[num], number[i]
number_list = [18, 90, 56, 2, 30, 88, 97]
print("The Original list: ", number_list)
selection_sort(number_list)
print("The selection sorted list: ",number_list)
```

Selection Sort: Output

Output:
The Original list: [18, 90, 56, 2, 30, 88, 97]
The selection sorted list: [2, 18, 30, 56, 88, 90, 97]

Insertion Sort

The insertion sort algorithm places elements in a specific order in a list. In the insertion sort algorithm, each iteration transfers an element from an unsorted portion to a sorted piece until all the elements in the list are sorted. To sort the list using insertion sort. Let's follow this steps below to help us understand the procedure:

- We split a list into two parts, i.e. sorted and unsorted lists.
- Iterate from list[1] to list[n] over the given list
- Compare the current element to the next element.
- If the current element is smaller that the next element, compared it to the element before it. Move to the greater elements one position up to make space for the swapped element.

Insertion Sort

Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python
"""
A program to perform insertion sort on a list
"""
# creating a function for insertion sort
def insertion_sort(number):
        for i in range(1, len(number)):
                item = number[i]
                j = i – 1
                while j >= 0 and item < number[j]:
                                number[j + 1] = number[j]
                                j -= 1
                number[j + 1] = item
        return number
```

Insertion Sort

Let's look at an example to help us understand this concept better.

```
number_list = [100, 18, 90, 56, 2, 30, 88, 97, 1]
print("The unsorted list is: ", number_list)
print("The sorted list is: ", insertion_sort(number_list))
```

Output:

The unsorted list is: [100, 18, 90, 56, 2, 30, 88, 97, 1]
The sorted list is: [1, 2, 18, 30, 56, 88, 90, 97, 100]

Quicksort

Quicksort is an efficient sorting algorithm because it is based on a partition of a list. It uses a **divide and conquer** approach and divides a list into smaller parts. First, picks the pivot element from a list and applies a partition techniques. The result of the partition approach is to locate the pivot element into its actual position. In this technique there are two types of partitions one at the *left-side* of the **key** and the other at the *right-side*.

Let's look at an example of a quicksort to understand this concept better.

Quicksort: Example

```
#!/usr/bin/env python
def partitionMethod(number, low, high): # divide function
        i = (low - 1 )
        pivot = number[high] # pivot element
        for j in range(low, high):
                # if current element is smaller
                if number[j] <= pivot:
                        # increment
                i = i + 1
                number[i], number[j] = number[j], number[i]
        number[i + 1], number[high] = number[high], number[i + 1]
        return (i + 1)
```

Quicksort: Example

```
# quick sort
def quickSortMethod(number, low, high):
        if low < high:
                # index
                num = partitionMethod(number, low, high) # call the partitionMethod()
                # sort the partitions
                quickSortMethod(number, low, num - 1)
                quickSortMethod(number, num + 1, high)
number_list = [100, 18, 90, 56, 2, 30, 88, 97, 1, 33, 12]
length = len(number_list)
quickSortMethod(number_list, 0, length - 1)
print("The sorted list is:")
for i in range(length):
        print(number_list[i], end= " ")
print("\n")
```

Output:

The sorted list is:

1 2 12 18 30 33 56 88 90 97 100

Summary

Summary

- We discussed about list and their creation, accessing, and updating values in lists.
- We looked at nesting, aliasing, and cloning lists.
- We discussed some list paraments and passing list as arguments. We discussed about list operators and methods.
- We looked at passing and returning list from functions and explore more of list functions such as filter(), range(), len(), copy(), append(), slice() and so on.
- We also discussed searching and sorting methods in the list such as linear search, binary search, bubble sort, selection, insertion and quick sort.

Important: Further Reading

- Concept of a collection
- Lists and definite loops
- Indexing and lookup
- List mutability
- Functions: len, min, max, sum
- List methods: append, remove, pop etc
- Sorting lists
- Splitting strings into lists of words
- Using split to parse strings

Points to Remember

- In a list, values are enclosed with square brackets.
- The first item on the list is indexed as [0]
- A slice is represented by a colon(:)
- A nested list means a list within a list.
- The list() construct takes a single argument
- A list is used in an operator for searching in a list
- The sort() method is used for sorting elements of a given list.

Further Reading

You can read further this week's lecture from the following chapters:

- Python for Everyone (3/e) : By **Cay Horstmann** & **Rance Necaise**
  - Chapter 6 Lists
- Learning Python (5th Edition): By **Mark Lutz**
  - Chapter 8 Lists and Dictionaries
- Basic Core Python Programming: By **Meenu Kohli**
  - Chapter 5 Lists and Arrays

Next Lecture 7

Lecture 7 - Tuples