\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC121 Python Programming**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lesson 06 Lists [Part 2]

# **Objectives**

In this lesson, students will learn:

- How to use for statements to iterate over lists

- How to copy and concatenate lists

- How to create and use nested lists

- How to create and use tuples

# **6.1 List Concatenation**

We can combine two or even more lists into a longer one. The following is an example:

list1 = [3, 7, 2]  
list2 = [4, 1, 9, 5]  
list3 = list1 + list2  
print(**"List1:"**, list1)  
print(**"List2:"**, list2)  
print(**"List3:"**, list3)

The third statement in the program above creates a 7-element new list named list3. The first 3 elements are copied from list1, while the last 4 elements are copied from list2. The following is the output of the program:

List1: [3, 7, 2]

List2: [4, 1, 9, 5]

List3: [3, 7, 2, 4, 1, 9, 5]

Here is another example:

list1 = [**"Physics"**, **"Math"**, **"Biology"**]  
list2 = [**"History"**, **"Music"**]  
list3 = [**"Psychology"**]  
list4 = list1 + list2 + list3  
print(**"List1:"**, list1)  
print(**"List2:"**, list2)  
print(**"List3:"**, list3)  
print(**"List4:"**, list4)

The fourth statement in the program above creates a new list named list4, which is equal to the combination of list1, list2 and list3. The following is the output of the program:

List1: ['Physics', 'Math', 'Biology']

List2: ['History', 'Music']

List3: ['Psychology']

List4: ['Physics', 'Math', 'Biology', 'History', 'Music', 'Psychology']

# **6.2 Copying List Elements**

You can copy elements from one list to another list. The following is an example:

listx = [**"Amy"**, **"Bill"**, **"Carol"**, **"Dan"**, **"Eve"**]  
listy = listx[1:4]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

The syntax listx[1:4] returns part of listx. The number before the colon is the starting index, while the number after the colon is the ending index. listx[1:4] includes three elements: listx[1], listx[2] and listx[3]. It does not include listx[4] because the element with the ending index is always excluded. The following is the output of the program:

listx: ['Amy', 'Bill', 'Carol', 'Dan', 'Eve']

listy: ['Bill', 'Carol', 'Dan']

Here is another example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[2:6]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

The syntax listx[2:6] returns listx[2], listx[3], listx[4] and listx[5]. The element listx[6] is excluded. The following is the output of the program:

listx: [10, 11, 12, 13, 14, 15, 16]

listy: [12, 13, 14, 15]

The following program copies listx[0], listx[1], listx[2] and listx[3]to the new list listy:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[0:4]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

Here is the output of the program:

listx: [10, 11, 12, 13, 14, 15, 16]

listy: [10, 11, 12, 13]

The following program copies listx[4], listx[5] and listx[6]to the new list listy:

listx = [10, 11, 12, 13, 14, 15, 16]  
ending\_index = len(listx)  
listy = listx[4:ending\_index]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

The length of listx is 7. Since we use it as the ending index, the program copies elements up to and including listx[6]. The following is the output of the program:

listx: [10, 11, 12, 13, 14, 15, 16]

listy: [14, 15, 16]

We can shorten the program above by combining the second and third statements:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[4:len(listx)]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

This shortened version will generate the same output as before.

Python syntax allows us to omit the starting index and the ending index when we copy list elements. If the starting element is omitted, it means starting from the first element of the list. The following is an example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[:4]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

The syntax listx[:4] is the same as listx[0:4]. Do not forget the colon. Here is the output of the program:

listx: [10, 11, 12, 13, 14, 15, 16]

listy: [10, 11, 12, 13]

If the ending element is omitted, it means ending at the end of the list and including the last element. The following is an example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[4:]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

The syntax listx[4:] is the same as listx[4:len(listx)]. Do not forget the colon. Here is the output of the program:

listx: [10, 11, 12, 13, 14, 15, 16]

listy: [14, 15, 16]

You can omit both the starting and ending indices at the same time. It just means copying every element in the list. The following is an example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[:]  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

The syntax listx[:] is the same as listx[0:len(listx)]. Do not forget the colon. Here is the output of the program:

listx: [10, 11, 12, 13, 14, 15, 16]

listy: [10, 11, 12, 13, 14, 15, 16]

The elements of the two lists are exactly the same.

Before we end this section, let’s talk a little bit more about copy the elements of the whole list. When we copy a list, our goal is to create a clone, i.e. a new list whose elements are exactly the same as the old list. This can be done in a few different ways. Using syntax such as listy = listx[:] is one way to copy all list elements.

Another way is using list concatenation. The following is an example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = [] + listx  
print(**"listx:"**, listx)  
print(**"listy:"**, listy)

The syntax [] + listx forces the computer to create a new list to store the combination of the elements of an empty list and listx. Since the empty list has no elements, the new list will have exactly the same elements as listx. The following is the output:

listx: [10, 11, 12, 13, 14, 15, 16]

listy: [10, 11, 12, 13, 14, 15, 16]

The second statement in the program above can be written as the following if you want to:

listy = listx + []

Either version will work.

You may wonder why we bother to concatenate an empty list when we do list copying. Why don’t we simply write listy = listx? The reason is that listy = listx does not work. It does not create a new list of elements in computer memory. Instead, it only associates two variables to the same list. In other words, it just gives an additional name to the old list.

When listy = listx[:] or listy = listx + [] is used, a new list is created in computer memory and the new variable listy is associated to the new list:

listx

[10, 11, 12, 13, 14, 15, 16]

listy

[10, 11, 12, 13, 14, 15, 16]

When listy = listx is used, no new list is created in computer memory. The new variable listy is associated to the old list:

listx

[10, 11, 12, 13, 14, 15, 16]

listy

Remember, do not write listy = listx if you want a new list with elements copied from the old list.

# **6.3 Other Linear Data Structures**

A list is a sequence of data. There are other data structures in Python that also store data in sequences.

## Tuples

Tuples are similar to lists, except that once a tuple is created, it cannot be altered. You cannot add elements to a tuple or remove elements from a tuple. You cannot change any element of a tuple. The following example shows how to create a tuple and access its elements:

tuple1 = (8, 2, 6, 4)  
print(**"Element 0 of tuple1:"**, tuple1[0])  
total = tuple1[0] + tuple1[1] + tuple1[2] + tuple1[3]  
print(**"Total of the elements:"**, total)

The following is the output of the program:

Element 0 of tuple1: 8

Total of the elements: 20

You can concatenate tuples or copy elements of a tuple:

tuple1 = (8, 2, 6, 4)  
tuple2 = (7, 3)  
tuple3 = tuple1 + tuple2  
tuple4 = tuple1[0:3]  
print(**"tuple3:"**, tuple3)  
print(**"tuple4:"**, tuple4)

The following is the output of the program:

tuple3: (8, 2, 6, 4, 7, 3)

tuple4: (8, 2, 6)

Tuples are best used for storing data that should remain unchanged throughout the whole program. If necessary, you can convert a tuple to a list and vice versa.

tuple1 = (8, 2, 6, 4)  
print(**"Original tuple:"**, tuple1)  
list1 = list(tuple1)  
list1[0] = 1  
print(**"List after element change:"**, list1)  
tuple1 = tuple(list1)  
print(**"Tuple after element change:"**, tuple1)

The program above first creates a 4-element tuple. It then creates a list and copies the elements from the tuple to it. The syntax list(tuple1) returns a list that has elements exactly the same as tuple1. The program then changes the first element of the list from 8 to 1 and copies the changed list back to a tuple. The syntax tuple(list1) returns a tuple that has elements exactly the same as list1. The following is the output of the program:

Original tuple: (8, 2, 6, 4)

List after element change: [1, 2, 6, 4]

Tuple after element change: (1, 2, 6, 4)

## Strings

Strings are another type of linear data structures in Python. You can consider a string as a tuple of characters. You can access individual string elements. You can concatenate strings or copy part of a string.

first\_name = **"John"**print(**"Print first name:"**, first\_name)  
print(**"Print one element:"**, first\_name[0])  
last\_name = **"Doe"**full\_name = first\_name + last\_name  
print(**"Concatenation:"**, full\_name)  
substring = full\_name[1:4]  
print(**"Part of a string:"**, substring)

Output of the program:

Print first name: John

Print one element: J

Concatenation: JohnDoe

Part of a string: ohn

Similar to tuples, string elements are immutable. You cannot use an assignment statement to change an element of a string.

first\_name = **"John"**first\_name[0] = **"X"** *# not allowed*

You will get the following error message if you run the program above:

TypeError: 'str' object does not support item assignment

If you want to change the letter J to X, you can use the replace function:

first\_name = **"John"**first\_name = first\_name.replace(**"J"**, **"X"**)  
print(**"Print first name:"**, first\_name)

The replace function returns a new string with J replaced by X. We use an assignment statement to save the new string in the same variable first\_name to overwrite the old string. The following is the output:

Print first name: Xohn

There is another way to change an element in a string:

first\_name = **"John"**first\_name = **"X"** + first\_name[1:]  
print(**"Print first name:"**, first\_name)

The second statement concatenates the letter X with the original letters except the J. The result of this operation is stored in the same variable first\_name to overwrite the original string. The following is the output:

Print first name: Xohn

# **6.4 Nested Lists And Nested For Loops**

A list can contain elements of any type, including other lists. Therefore, you can create a list of lists if you want to. The list that holds the other lists as elements is called the outer list and the lists that are elements of the outer lists are called inner lists. The following is an example.

Suppose there are 3 students in a class and each student has two lab scores. The lists of lab scores are the inner lists and the list of students that have these lab scores is the outer list. In the following program we first create three lists, with each one storing two scores of a student. Then we create the outer list to store the three student scores lists.

The **student0, student1,** and **student2** lists are the 3 inner lists of student scores.

student0 = [90, 91]  
student1 = [80, 81]  
student2 = [70, 71]

These 3 lists are added as elements in the **outer list** called **all\_students,** as follows:

all\_students = [student0, student1, student2]

Each element in the list **all\_students** is a list itself. We can access the **inner lists** using bracket indexing. This is precisely the same as accessing elements of a list which are integers, floats, booleans or strings. Let’s display an element of **all\_students**:

print ("all\_students[0]:", all\_students[0])

print ("all\_students[1]:", all\_students[1])

print ("all\_students[2]:", all\_students[2])

The following is the output:

all\_students[0]: [90, 91]

all\_students[1]: [80, 81]

all\_students[2]: [70, 71]

The **print** statements display the two scores of the student. The square brackets in the output indicate that the elements **all\_student[0], all\_student[1],**and **all\_student[2],** are themselves a list.

We can also access the elements of the inner lists using bracket indexing by adding a second bracket. Use the following **print** statement to display the first test of the second student:

print ("all\_students[1][0]:", all\_students[1][0])

The syntax **all\_students[1][0]** accesses element **0** of **all\_students[1]**, which is a list itself. The following is the output:

all\_students[1][0]: 80

Similarly, use the following **print** statement to display the second test of the third student or in other words, element **1** of **all\_student[2]:**

print ("all\_students[2][1]:", all\_students[2][1])

The following is the output:

all\_students[2][1]: 71

student0 = [90, 91]  
student1 = [80, 81]  
student2 = [70, 71]  
all\_students = [student0, student1, student2]

The program code above can be shortened to one statement like this:

all\_students = [[90, 91], [80, 81], [70, 71]]

The inner lists can be thought of as rows and the position of their elements within the list as columns, like in a matrix:

all\_students = [[90, 91],

[80, 81],

[70, 71]]

A nested list in which the number of elements for each inner list are the same can be traversed using a **nested set of two for loops**. The order of traversal is row-by-row and column-by-column within each row. The **outer** **for** loop uses an index for the rows. The **inner** **for** loop uses an index for the columns.

student0 = [90, 91]

student1 = [80, 81]

student2 = [70, 71]

all\_students = [student0, student1, student2]

for row in range(3):

print (‘Scores for Student’, row + 1)

for col in range(2):

print (all\_students[row][col])

**Scores for Student 1**

**90**

**91**

**Scores for Student 2**

**80**

**81**

**Scores for Student 3**

**70**

**71**

# **6.5 Iteration, Iterators, and Iterables**

**Iteration** is the act of repeating a process until a condition is met. It is also a term for performing a task on the elements of a sequence, one after another. Any time you use a loop, to go over or traverse a group of items, that is **iteration**. To iterate over a sequence means to visit each element of the sequence, and do some operation for each element. In Python, we say that a value is an iterable when your program can iterate over it. In short, an **iterable** is a value that represents a sequence of one more values.

All instances of Python's sequence types are iterables. These types may be referred to as container types: a string is a container for 32-bit characters, and lists and tuples are general-purpose containers that can contain any value type, including other sequences, such as with nested lists.

Two of the most common uses for an **iterable** are in a **for** loop or a **list comprehension**, where you want to perform some operation on a sequence of values. In Python iterables create **iterators**. The **iterators** are the entities that actually iterate over the elements of the iterable sequence.

# **6.6 List Comprehensions**

**List comprehensions** provide a concise way to create lists. This is a feature which is widely used in Python. It can be found in many examples and in the source code of many Python libraries. The syntax is a little cryptic, but it is an important construct of the Python language. Common applications of list comprehensions are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares of only the even numbers, like:

*# List of even squares using a for loop*

squares = []

for x in range(13):

if x % 2 == 0:

squares.append (x\*\*2)

print (squares)

[0, 4, 16, 36, 64, 100, 144]

Note that this uses a loop variable named **x** that still exists after the loop completes. We can create the list of squares without using the variable x, which is more concise and readable, once you are comfortable with the syntax, as follows:

*# List of even squares using a list comprehension*

squares = [x\*\*2 for x in range(13) if x % 2 == 0]

print (squares)

[0, 4, 16, 36, 64, 100, 144]

The parts of a **list comprehension** include:

* an input sequence.
* a variable representing elements of the input sequence.
* an optional conditional expression.
* an output expression producing elements of the output list from elements of the input sequence that satisfy the condition.

A **list comprehension** consists of brackets containing an expression followed by a **for** clause, then **zero or more** **for** or **if** clauses. There are no colons in a list comprehension. The expressions can be anything, meaning you can put in all kinds of objects in lists. The result will be a new list resulting from evaluating the expression in the context of the **for** and **if** clauses which follow it. The list comprehension always returns a list.

The list comprehension in the example above is decomposed as:

[x\*\*2 for x in range(13) if x % 2 == 0]

* input sequence: **range(13)**
* variable representing elements of the input sequence: **x**
* conditional expression: **if x % 2 == 0**
* output expression – what to append to new list: **x\*\*2**
* required **for** clause: **for x in range(13)**

The following list comprehension combines the elements of two lists into a list of tuples, if the corresponding list elements are not equal:

*# Combine two lists into tuples, when elements are not equal*

comb\_list =

[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]

print (comb\_list)

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

and it is equivalent to. Remember how the nested **for** loop works:

*# Combine two lists into tuples, when elements are not equal*

comb\_list = []

for x in [1,2,3]:

for y in [3,1,4]:

if x != y:

comb\_list.append ((x, y))

print (comb\_list)

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

Note how the order of the **for** and **if** clauses is the same in both the list comprehension and the for loop equivalent. The list comprehension always starts with what is being appended to the list – the output expression. If the output expression is a tuple (e.g. the (x, y) in the previous example), it must be parenthesized.

Here are a few more simple examples of list comprehensions:

Given **vector = [-4, -2, 0, 2, 4]**

1. Create a new list with the values doubled:

print ( [x \* 2 for x in vector] )

Output:

[-8, -4, 0, 4, 8]

1. Filter the list to exclude negative numbers:

print ( [x for x in vector if x >= 0] )

Output:

[0, 2, 4]

1. Apply a function to all the elements:

print ( [abs (x) for x in vector] )

Output:

[4, 2, 0, 2, 4]

The initial output expression in a list comprehension can be any arbitrary expression, including another list comprehension. Consider the following example of a 3x4 matrix – a nested list implemented as a list of 3 lists of length 4.

matrix = [ [1,2,3,4],

[5,6,7,8],

[9,10,11,12] ]

This matrix can be transposed into a 4x3 matrix, where the rows become columns and columns become rows:

**matrix\_T = [ [1,5,9],**

**[2,6,10],**

**[3,7,11],**

**[4,8,12] ]**

The following list comprehension can be used to do this transposition:

matrix\_T = [[row[i] for row in matrix] for i in range(4)]

print (matrix\_T)

Output:

**[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]**

Note how the **output expression** is itself a list comprehension. The following steps can be used to create the equivalent **for** loop code to do this same transposition, removing the list comprehensions.

Step 1:

matrix\_T = []

for i in range(4):

matrix\_T.append ([row[i] for row in matrix])

Step 2:

matrix\_T = []

for i in range(4):

row\_T = []

for row in matrix:

row\_T.append (row[i])

matrix\_T.append (row\_T)

print (matrix\_T)

Output:

**[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]**

The middle of the above Step 2 code is the conversion of the output expression list comprehension – code lines 3-5