COMP2152 LAB MANUAL

OPEN SOURCE DEVELOPMENT

This booklet will help the reader understand the concepts, principles, and implementation of the Python programming language. By the end of the booklet, the reader will be able to code comfortably in Python.

Prepared by Ben Blanc

TABLE OF CONTENTS

CHAPTER 8

LIST & TUPLES

List and tuples are ordered collections of data that most resemble arrays in other programming languages. List and tuples can contain any data types as elements (string, integer, float, or even other list and tuples)

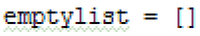
CREATING LIST

You create a list with the following syntax in either one of two ways in python

EMPTY LIST DECLARATION

Takes the following syntax:

VARIABLE_*NAME* = [ ]

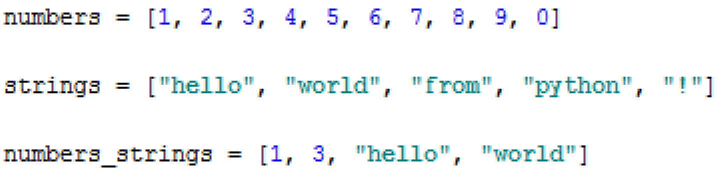


```
emptylist = []
```

LIST WITH VALUES DECLARATION

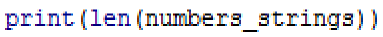
Takes the following syntax:

VARIABLE_*NAME* = [each, declared, element, separated, by, comma]



```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

strings = ["hello", "world", "from", "python", "!"]

numbers_strings = [1, 3, "hello", "world"]
```

The last list declared above has an list size of 4. To output the list size, use the **len()** method.



```
print(len(numbers_strings))
```

CHAPTER 8

AT A GLANCE

In this chapter you will learn how about create and manipulate two types of ordered collections of data in Python:

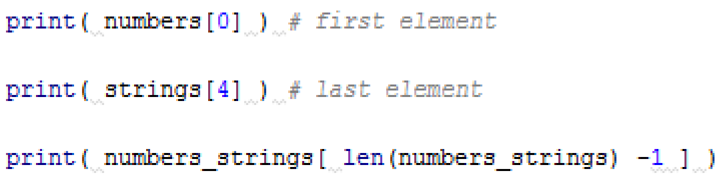
- List
- Tuple

ACCESSING LIST ELEMENTS

The syntax for accessing an list element is:

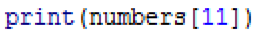
List_Name[index_number]

The first element of an list starts at index 0. The last element of an list is one less than its list length.

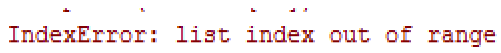


```
print( numbers[0] ) # first element

print( strings[4] ) # last element

print( numbers_strings[ len(numbers_strings) -1 ] )
```

If you select an index that is out of range, you will not get a syntax error but when you run the program, you will get a runtime error



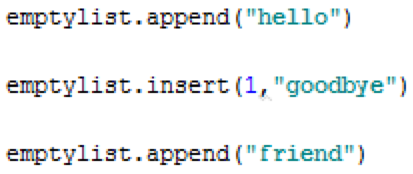
```
print(numbers[11])
```



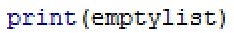
```
IndexError: list index out of range
```

ADDING LIST ELEMENTS

Lists are sequences. Therefore, you must add elements in order. If there is a new element being added (at a new index), use the append() or insert() method

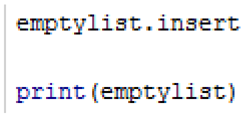


```
emptylist.append("hello")

emptylist.insert(1,"goodbye")

emptylist.append("friend")
```



```
print(emptylist)
```

If you use the insert() method and add at an end out of the range of the current indices of the list, the method will word as append() method

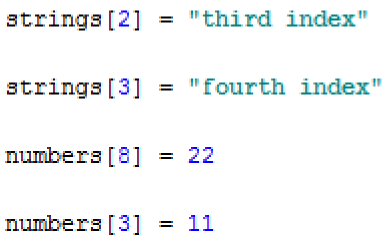


```
emptylist.insert(10,True)

print(emptylist)
```

UPDATING LIST ELEMENTS

The syntax for updating an list element is very similar to accessing an list element, but with the presence of assigning a value to the specified index

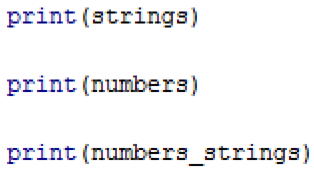
List_Name[index_number] = NEW_VALUE



```
strings[2] = "third index"

strings[3] = "fourth index"

numbers[8] = 22

numbers[3] = 11
```

DISPLAYING ALL LIST ELEMENTS

To display all of your list elements, you can use the print() method



```
print(strings)

print(numbers)

print(numbers_strings)
```

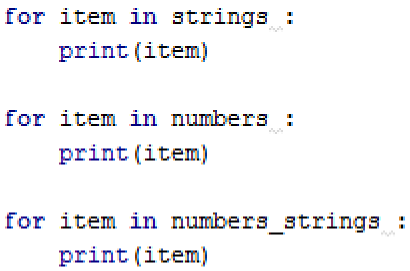
Or you can use a for loop.

As mentioned in Chapter 4, the syntax for the foreach loop is:

for variable_identifier in collection_of_data :

statement(s)

The variable_indentifier is a variable that will represent each individual value in your collection_of_data. The collection_of_data is your list.

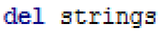


```
for item in strings :
    print(item)

for item in numbers :
    print(item)

for item in numbers_strings :
    print(item)
```

DELETING A LIST

To delete a list, use the del keyword and pass the list variable

del List_Name

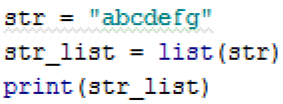


```
del strings
```

TYPECASTING TO A LIST

You can typecast a string or another collection of items by using the list() method

List_Name = list(collection_of_items)



```
str = "abcdefg"
str_list = list(str)
print(str_list)
```

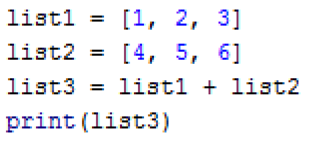
LIST OPERATIONS & METHODS

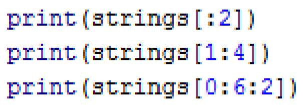
You can execute list operations and use built-in list methods that help with manipulating lists.

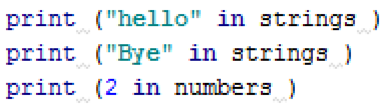
LIST OPERATIONS

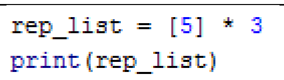
Below is a table of some of the List operations

Operation	Description	Example

Concatenation	Merges two or more lists together
	Return: list
		
		`list1 = [1, 2, 3]`
		`list2 = [4, 5, 6]`
		`list3 = list1 + list2`
		`print(list3)`

Slicing	Returns a subsection of the list.
	Return: list
		
		`print(strings[:2])`
		`print(strings[1:4])`
		`print(strings[0:6:2])`

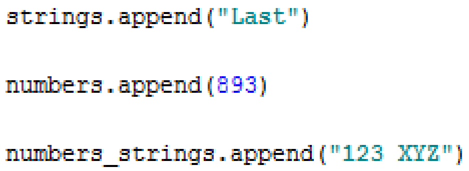
Search	Use the IN operator to determine if item is in list
	Return: Boolean
		
		`print ("hello" in strings )`
		`print ("Bye" in strings )`
		`print (2 in numbers )`

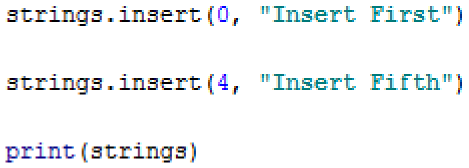
Repetition	Use the * operator to repeat a value in a list	
		`rep_list = [5] * 3`
		`print(rep_list)`

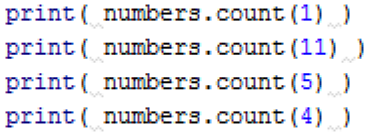
LIST METHODS

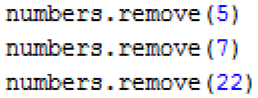
Below is a table of some of the List methods

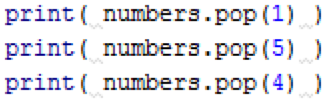
Method	Description	Arguments	Return	Example

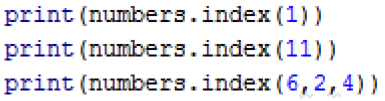
append()	Adds object to the end of list	Any single object	None	 strings.append("Last")
				numbers.append(893)
				numbers_strings.append("123 XYZ")

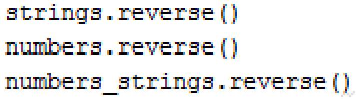
insert()	Adds an object at the specified index location	1)Int Index location
		2)obj element
			None	 strings.insert(0, "Insert First")
				strings.insert(4, "Insert Fifth")
				print(strings)

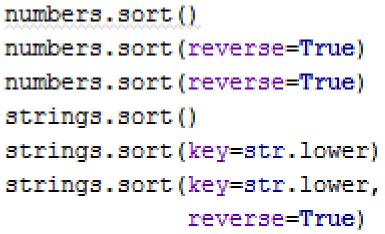
count()	Determines the number of times an object occurs in a list	Object to search	Integer count of number of occurrences	 print(numbers.count(1))
				print(numbers.count(11))
				print(numbers.count(5))
				print(numbers.count(4))

remove()	Removes an object	Any single object	None	 numbers.remove(5)
				numbers.remove(7)
				numbers.remove(22)

pop()	Removes an from the end of the list OR at a specified indexobject	Optional: Index to remove	Object removed	 print(numbers.pop(1))
				print(numbers.pop(5))
				print(numbers.pop(4))

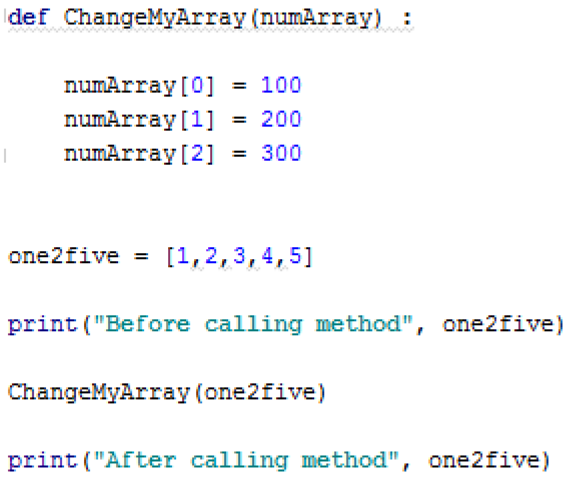
index()	Determines the lowest index of an object.	1)Object to search
		2)Int starting index
		3)Int Ending index
			Integer index of object. Raises exception if not found	 print(numbers.index(1))
				print(numbers.index(11))
				print(numbers.index(6,2,4))

reverse()	Reserves the order of the elements	None	None	 strings.reverse()
				numbers.reverse()
				numbers_strings.reverse()

sort()	Sorts the values by numerical values or by ASCII values. Mixed data types results in error. Can pass function to method	1)key=function to pass when sorted values
		2)reverse=Boolean value to sort in descending order
			None	 numbers.sort()
				numbers.sort(reverse=True)
				numbers.sort(reverse=True)
				strings.sort()
				strings.sort(key=str.lower)
				strings.sort(key=str.lower,
				reverse=True)

LISTS AS METHOD PARAMETERS

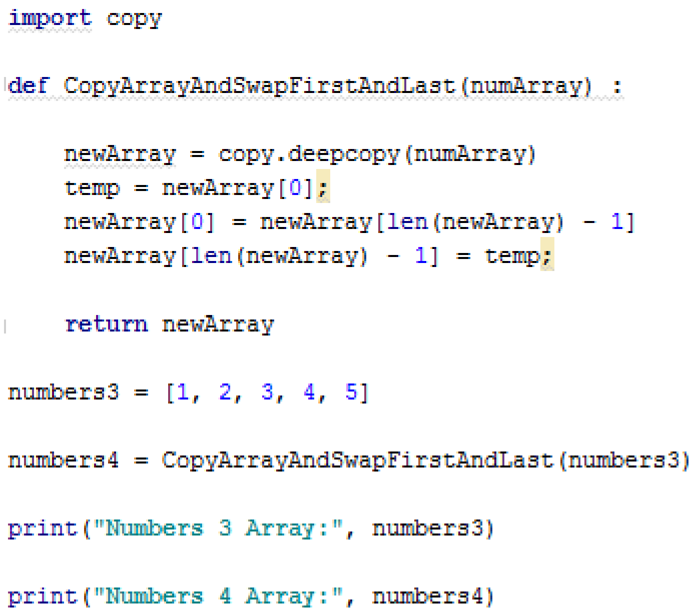
The list data type is passed by reference by default. That means that whatever change your method makes to the list will be reflected in the original list variable. Take below as an example.



```
def ChangeMyArray(numArray) :

    numArray[0] = 100
    numArray[1] = 200
    numArray[2] = 300


one2five = [1,2,3,4,5]

print("Before calling method", one2five)

ChangeMyArray(one2five)

print("After calling method", one2five)
```

From the example above, there is no need to return anything in our method since we will be altering the original list.

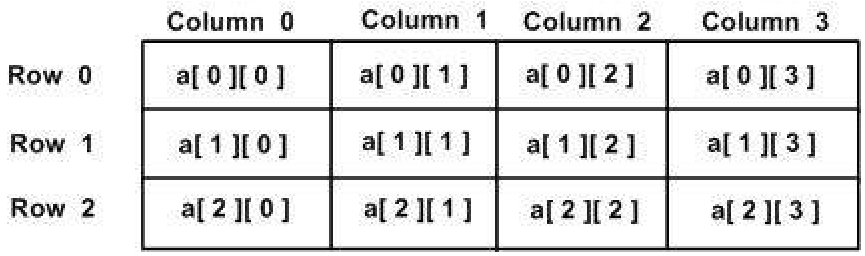
If we would like to make a copy of the list and change the first and last values, we would create a method like this:



```
import copy

def CopyArrayAndSwapFirstAndLast(numArray) :

    newArray = copy.deepcopy(numArray)
    temp = newArray[0];
    newArray[0] = newArray[len(newArray) - 1]
    newArray[len(newArray) - 1] = temp;

    return newArray

numbers3 = [1, 2, 3, 4, 5]

numbers4 = CopyArrayAndSwapFirstAndLast(numbers3)

print("Numbers 3 Array:", numbers3)

print("Numbers 4 Array:", numbers4)
```

TWO-DIMENSIONAL LISTS

A two-dimensional list can be thought of like a table. Below is a table of 3 rows and 4 columns



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

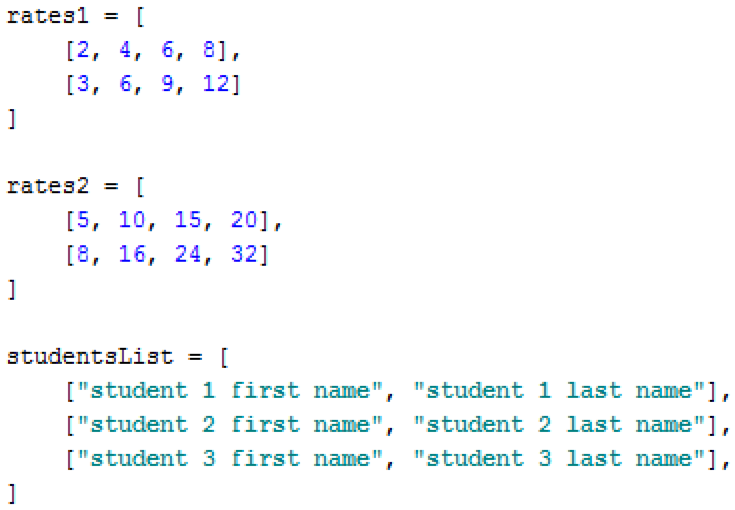
Notice that the first index is always 0.

LIST WITH VALUES DECLARATION

Takes the either of the following syntax:

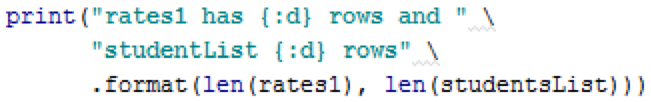
VARIABLE_*NAME* = [
[row1_column1, row1_column2, row1_columnN],
[row2_column1, row2_column2, row2_columnN]
]

Note that the last row of data does not have a comma, however, including one will NOT result in an error.



```
rates1 = [
    [2, 4, 6, 8],
    [3, 6, 9, 12]
]

rates2 = [
    [5, 10, 15, 20],
    [8, 16, 24, 32]
]

studentsList = [
    ["student 1 first name", "student 1 last name"],
    ["student 2 first name", "student 2 last name"],
    ["student 3 first name", "student 3 last name"],
]
```

The **len()** method gives us the number or rows of a list

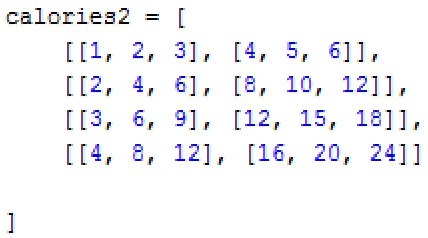


```
print("rates1 has {:d} rows and " \
      "studentList {:d} rows" \
      .format(len(rates1), len(studentsList)))
```

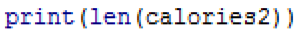
MULTI-DIMENSIONAL LISTS

To create more than two-dimensions in a list, you would add more commas to indicate more dimensions to the list:

LIST WITH VALUES DECLARED



```
calories2 = [
    [[1, 2, 3], [4, 5, 6]],
    [[2, 4, 6], [8, 10, 12]],
    [[3, 6, 9], [12, 15, 18]],
    [[4, 8, 12], [16, 20, 24]]

]
```



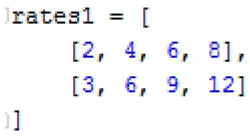
```
print(len(calories2))
```

ACCESSING MULTI-DIMENSIONAL ELEMENTS

Whether accessing two-dimensional or multi-dimensional list elements, the syntax is:

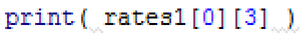
List_Name[first_dimension][second_dimension][Nth_dimension]

Take the following list



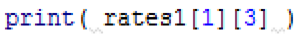
```
rates1 = [
    [2, 4, 6, 8],
    [3, 6, 9, 12]
]
```

You can access the value 8 with the following code:



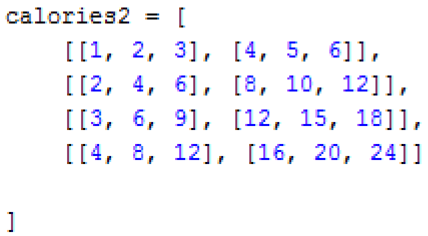
```
print( rates1[0][3] )
```

You can access the value 12 with the following code:



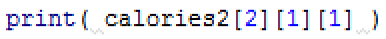
```
print( rates1[1][3] )
```

Take the following list



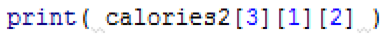
```
calories2 = [
    [[1, 2, 3], [4, 5, 6]],
    [[2, 4, 6], [8, 10, 12]],
    [[3, 6, 9], [12, 15, 18]],
    [[4, 8, 12], [16, 20, 24]]

]
```

You can access the value 15 with the following code



```
print( calories2[2][1][1] )
```

You can access the value 24 with the following code



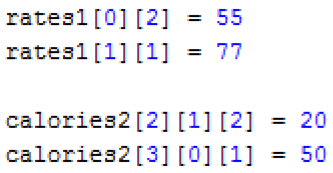
```
print( calories2[3][1][2] )
```

If you select an index that is out of range, you will not get a syntax error but when you run the program, you will get a runtime error

Updating List Elements

The syntax for updating an list element is very similar to accessing an list element, but with the presence of assigning a value to the specified index

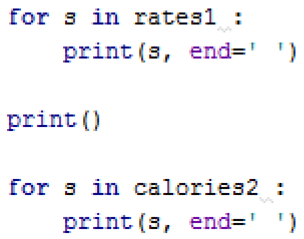
List_Name[first_dimension][second_dimension][Nth_dimension] = NEW_VALUE



```
rates1[0][2] = 55
rates1[1][1] = 77

calories2[2][1][2] = 20
calories2[3][0][1] = 50
```

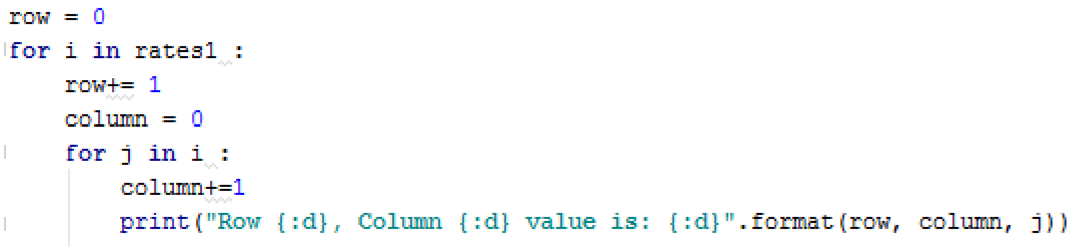
Displaying All List Elements

To display all of your list elements, you use the for loop



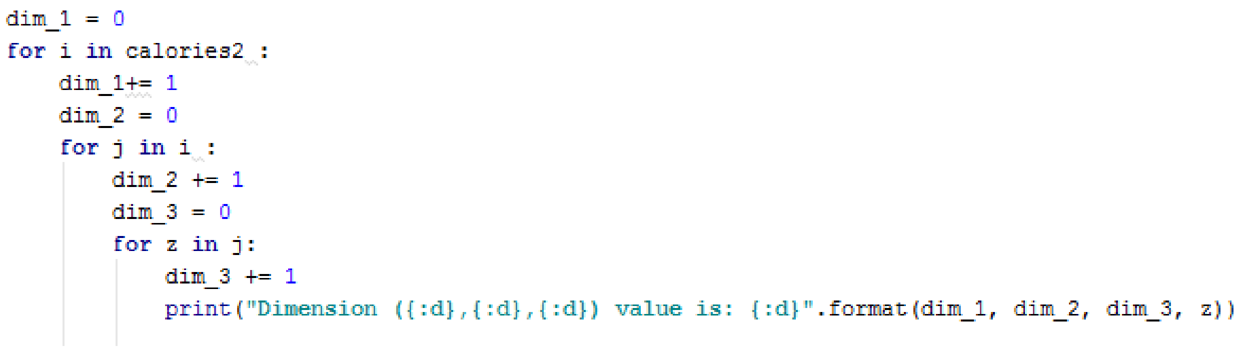
```
for s in rates1 :
    print(s, end=' ')

print()

for s in calories2 :
    print(s, end=' ')
```

And you can use the for loop, but be sure to code the loop within the loop



```
row = 0
for i in rates1 :
    row+= 1
    column = 0
    for j in i :
        column+=1
        print("Row {:d}, Column {:d} value is: {:d}".format(row, column, j))
```

The more dimensions, the more inner loops



```
dim_1 = 0
for i in calories2 :
    dim_1+= 1
    dim_2 = 0
    for j in i :
        dim_2 += 1
        dim_3 = 0
        for z in j:
            dim_3 += 1
            print("Dimension ({:d},{:d},{:d}) value is: {:d}".format(dim_1, dim_2, dim_3, z))
```

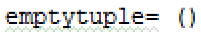
CREATING TUPLE

You create a tuple with the following syntax in either one of two ways in python

EMPTY LIST DECLARATION

Takes the following syntax:

VARIABLE_*NAME* = ()

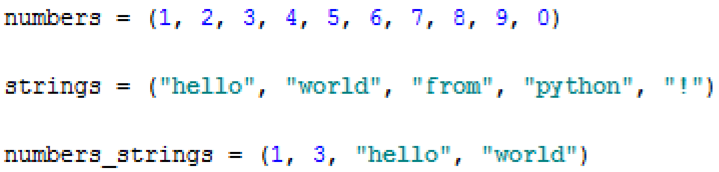


```
emptytuple= ()
```

LIST WITH VALUES DECLARATION

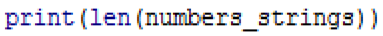
Takes the following syntax:

VARIABLE_*NAME* = (each, declared, element, separated, by, comma)



```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 0)

strings = ("hello", "world", "from", "python", "!")

numbers_strings = (1, 3, "hello", "world")
```

The last list declared above has a tuple size of 4. To output the list size, use the **len()** method.



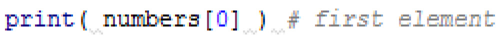
```
print(len(numbers_strings))
```

ACCESSING TUPLE ELEMENTS

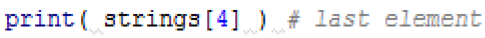
The syntax for accessing an list element is:

Tuple_Name[index_number]

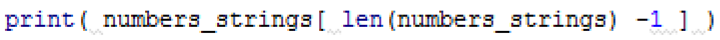
The first element of an list starts at index 0. The last element of an list is one less than its list length.



```
print( numbers[0] ) # first element
```

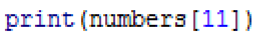


```
print( strings[4] ) # last element
```

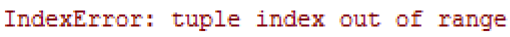


```
print( numbers_strings[ len(numbers_strings) -1 ] )
```

If you select an index that is out of range, you will not get a syntax error but when you run the program, you will get a runtime error



```
print(numbers[11])
```



```
IndexError: tuple index out of range
```

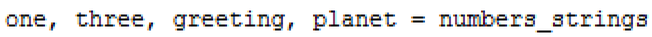
UPDATING TUPLE ELEMENTS

Tuples are immutable. They cannot be updated.

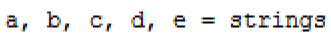
UPACKING TUPLE ELEMENTS

To extract the values of a tuple, the tuple must be unpacked by the following syntax:

Variable, indentifiers, separated, by, comma = Tuple_Variable



```
one, three, greeting, planet = numbers_strings
```

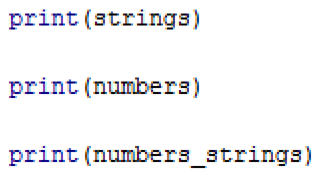


```
a, b, c, d, e = strings
```

Notice that the number of variables separated by commas matches the number of tuple elements.

DISPLAYING ALL LIST ELEMENTS

To display all of your list elements, you can use the print() method



```
print(strings)

print(numbers)

print(numbers_strings)
```

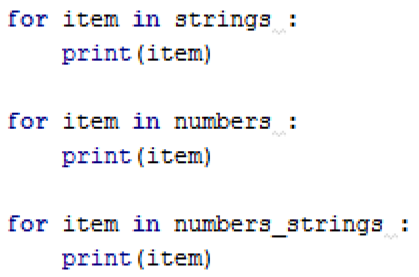
Or you can use a for loop.

As mentioned in Chapter 4, the syntax for the foreach loop is:

for variable_identifier in collection_of_data :

    statement(s)

The variable_indentifier is a variable that will represent each individual value in your collection_of_data. The collection_of_data is your list.

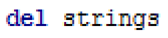


```
for item in strings :
    print(item)

for item in numbers :
    print(item)

for item in numbers_strings :
    print(item)
```

DELETING A TUPLE

To delete a tuple, use the del keyword and pass the tuple variable

del Tuple_Name

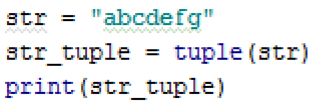


```
del strings
```

TYPECASTING TO A TUPLE

You can typecast a string or another collection of items by using the tuple() method

Tuple_Name = tuple(collection_of_items)



```
str = "abcdefg"
str_tuple = tuple(str)
print(str_tuple)
```

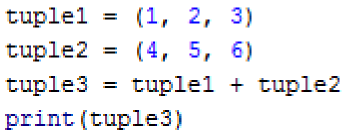
TUPLE OPERATIONS

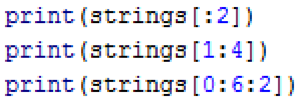
You can execute tuple operations that help with manipulating tuples.

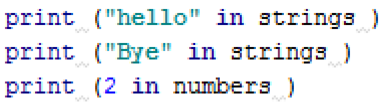
LIST OPERATIONS

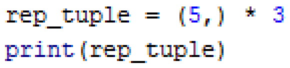
Below is a table of some of the List operations

Operation	Description	Example

Concatenation	Merges two or more tuples together
	Return: tuple
		
		`tuple1 = (1, 2, 3)`
		`tuple2 = (4, 5, 6)`
		`tuple3 = tuple1 + tuple2`
		`print(tuple3)`

Slicing	Returns a subsection of the tuple.
	Return: tuple
		
		`print(strings[:2])`
		`print(strings[1:4])`
		`print(strings[0:6:2])`

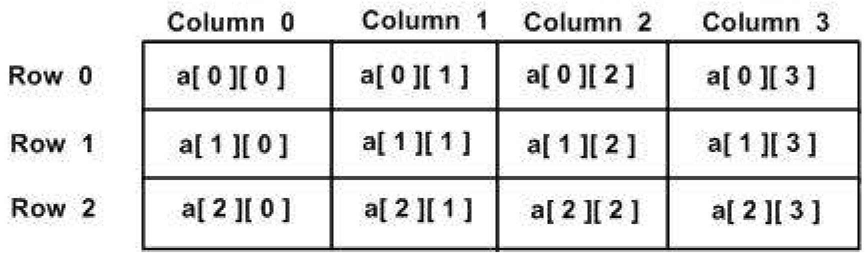
Search	Use the IN operator to determine if item is in tuple
	Return: Boolean
		
		`print ("hello" in strings )`
		`print ("Bye" in strings )`
		`print (2 in numbers )`

Repetition	Use the * operator to repeat a value in a list	
		`rep_tuple = (5,) * 3`
		`print(rep_tuple)`

NOTE: a single tuple value must have trailing comma.

TWO-DIMENSIONAL TUPLES

A two-dimensional tuple can be thought of like a table. Below is a table of 3 rows and 4 columns



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

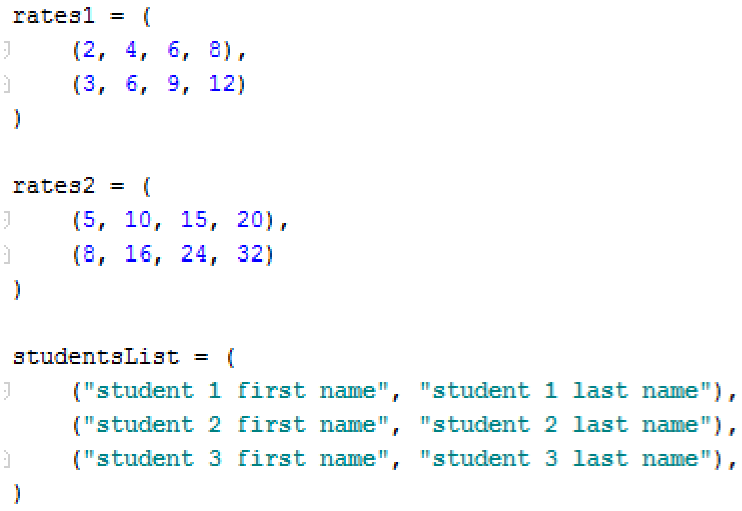
Notice that the first index is always 0.

TUPLE WITH VALUES DECLARATION

Takes the either of the following syntax:

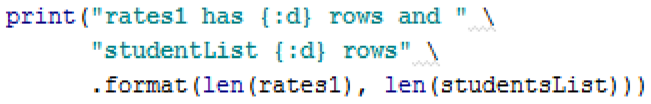
VARIABLE_*NAME* = (
(row1_column1, row1_column2, row1_columnN),
(row2_column1, row2_column2, row2_columnN)
)

Note that the last row of data does not have a comma, however, including one will NOT result in an error.



```
rates1 = (
    (2, 4, 6, 8),
    (3, 6, 9, 12)
)

rates2 = (
    (5, 10, 15, 20),
    (8, 16, 24, 32)
)

studentsList = (
    ("student 1 first name", "student 1 last name"),
    ("student 2 first name", "student 2 last name"),
    ("student 3 first name", "student 3 last name"),
)
```

The **len()** method gives us the number or rows of a list

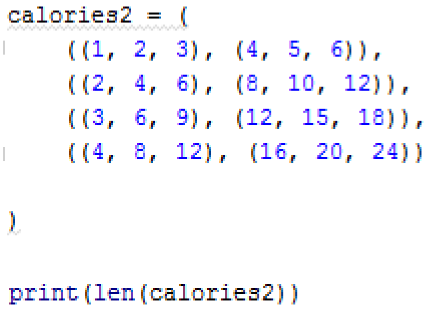


```
print("rates1 has {:d} rows and " \
      "studentList {:d} rows" \
      .format(len(rates1), len(studentsList)))
```

MULTI-DIMENSIONAL TUPLES

To create more than two-dimensions in a tuple, you would add more commas to indicate more dimensions to the tuple:

LIST WITH VALUES DECLARED



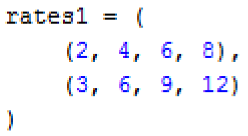
```
calories2 = (
    ((1, 2, 3), (4, 5, 6)),
    ((2, 4, 6), (8, 10, 12)),
    ((3, 6, 9), (12, 15, 18)),
    ((4, 8, 12), (16, 20, 24))

)

print(len(calories2))
```

ACCESSING MULTI-DIMENSIONAL ELEMENTS

Whether accessing two-dimensional or multi-dimensional tuple elements, the syntax is:

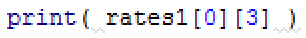
Tuple_Name[first_dimension][second_dimension][Nth_dimension]

Take the following list



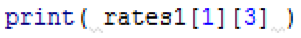
```
rates1 = (
    (2, 4, 6, 8),
    (3, 6, 9, 12)
)
```

You can access the value 8 with the following code:



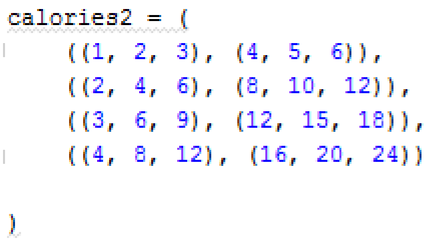
```
print( rates1[0][3] )
```

You can access the value 12 with the following code:



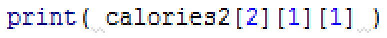
```
print( rates1[1][3] )
```

Take the following list



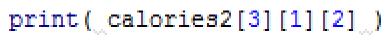
```
calories2 = (
    ((1, 2, 3), (4, 5, 6)),
    ((2, 4, 6), (8, 10, 12)),
    ((3, 6, 9), (12, 15, 18)),
    ((4, 8, 12), (16, 20, 24))

)
```

You can access the value 15 with the following code



```
print( calories2[2][1][1] )
```

You can access the value 24 with the following code

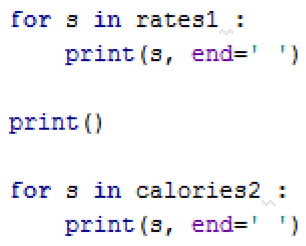


```
print( calories2[3][1][2] )
```

If you select an index that is out of range, you will not get a syntax error but when you run the program, you will get a runtime error

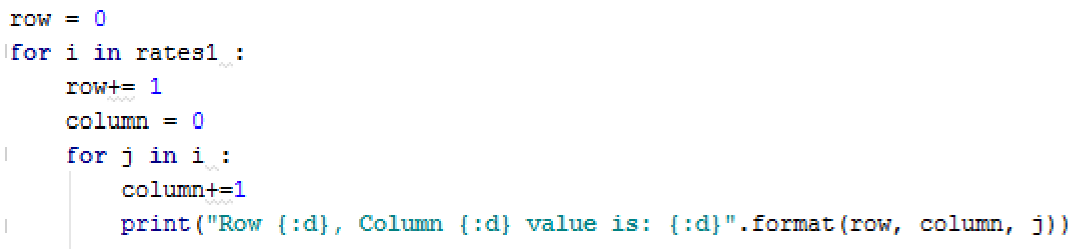
Displaying All List Elements

To display all of your list elements, you use the for loop



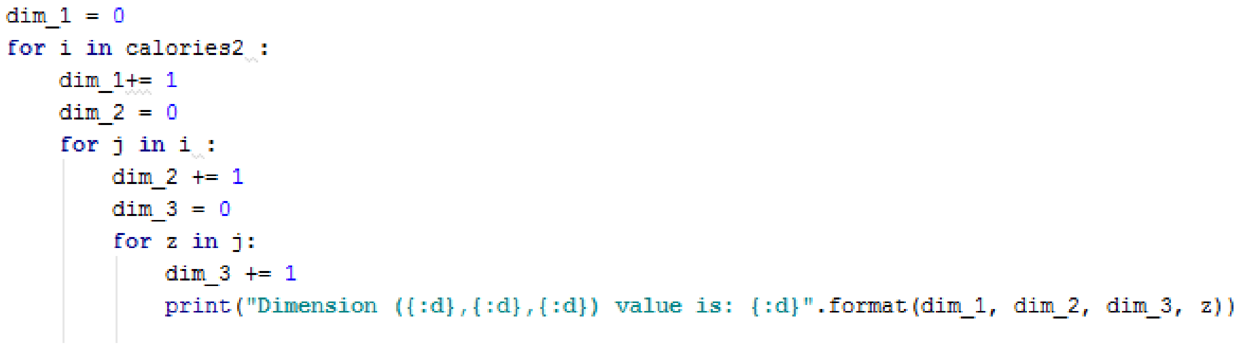
```
for s in rates1 :
    print(s, end=' ')

print()

for s in calories2 :
    print(s, end=' ')
```

And you can use the for loop, but be sure to code the loop within the loop



```
row = 0
for i in rates1 :
    row+= 1
    column = 0
    for j in i :
        column+=1
        print("Row {:d}, Column {:d} value is: {:d}".format(row, column, j))
```

The more dimensions, the more inner loops



```
dim_1 = 0
for i in calories2 :
    dim_1+= 1
    dim_2 = 0
    for j in i :
        dim_2 += 1
        dim_3 = 0
        for z in j:
            dim_3 += 1
            print("Dimension ({:d},{:d},{:d}) value is: {:d}".format(dim_1, dim_2, dim_3, z))
```