# Python Programming
# Module 4: Working with Lists

## Learning objectives

1. Working efficiently with the elements in a list.
2. Working through a list using a for loop, how Python uses indentation to structure a program.
3. Making simple numerical lists, as well as a few operations you can perform on numerical lists.
4. Slicing a list to work with a subset of items and how to copy lists properly using a slice.
5. Working with tuples, which provide a degree of protection to a set of values that shouldn't change.

# Working with Lists

## Looping Through an Entire List

Let's say we have a list of magicians' names, and we want to print out each name in the list. We could do this by retrieving each name from the list individually, but this approach could cause several problems. For one, it would be repetitive to do this with a long list of names. Also, we'd have to change our code each time the list's length changed. A for loop avoids both of these issues by letting Python manage these issues internally.

Let's use a for loop to print out each name in a list of magicians:

```
1 magicians=['alice','david','carolina']
2 for magician in magicians:
3     print(magician)
```

We begin by defining a list at `1`. At `2`, we define a for loop. This line tells Python to pull a name from the list magicians, and store it in the variable magician. At `3`, we tell Python to print the name that was just stored in magician. Python then repeats lines `2` and `3`, once for each name in the list. It might help to read this code as "For every magician in the list of magicians, print the magician's name." The output is a simple printout of each name in the list:

```
alice
david
carolina
```

### *A Closer Look at Looping*

In a simple loop like we used in *magicians.py*, Python initially reads the first line of the loop:

```
for magician in magicians:
```

This line tells Python to retrieve the first value from the list magicians and store it in the variable magician. This first value is `'alice'`. Python then reads the next line:

```
print(magician)
```

Python prints the current value of magician, which is still `'alice'`. Because the list contains more values, Python returns to the first line of the loop:

```
for magician in magicians:
```

Python retrieves the next name in the list, `'david'`, and stores that value in magician. Python then executes the line:

```
print(magician)
```

Python prints the current value of magician again, which is now `'david'`. Python repeats the entire loop once more with the last value in the list,`'carolina'`. Because no more values are in the list, Python moves on to the next line in the program. In this case nothing comes after the for loop, so the program simply ends.

***Doing More Work Within a for Loop***

You can do just about anything with each item in a `for` loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
1     print(magician.title()+", that was a great trick!")
```

The only difference in this code is at `1` where we compose a message to each magician, starting with that magician's name. The first time through the loop the value of magician is `'alice'`, so Python starts the first message with the name `'Alice'`. The second time through the message will begin with `'David'`, and the third time through the message will begin with `'Carolina'`.

The output shows a personalized message for each magician in the list:

```
  Alice, that was a great trick!
  David, that was a great trick!
  Carolina, that was a great trick!
```

You can also write as many lines of code as you like in the `for` loop. Every indented line following the line for `magician in magicians` is considered `inside the loop`, and each indented line is executed once for each value in the list. Therefore, you can do as much work as you like with each value in the list.

Let's add a second line to our message, telling each magician that we're looking forward to their next trick:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
     print(magician.title() + ", that was a great trick!")
1   print("I can't wait to see your next trick,"+magician.title()+
          ".\n")
```

Because we have indented both print statements, each line will be executed once for every magician in the list. The newline `("\n")` in the second print statement `1` inserts a blank line after each pass through the loop. This creates a set of messages that are neatly grouped for each person in the list:

```
  Alice, that was a great trick!
  I can't wait to see your next trick, Alice.

  David, that was a great trick!
```

```
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

You can use as many lines as you like in your `for` loops. In practice, you'll often find it useful to do a number of different operations with each item in a list when you use a for loop.

***Doing Something After a for Loop***

Any lines of code after the `for` loop that are not indented are executed once without repetition. Let's write a thank you to the group of magicians as a whole, thanking them for putting on an excellent show. To display this group message after all of the individual messages have been printed, we place the thank you message after the for loop without indentation:

```
magicians=['alice','david','carolina']
for magician in magicians:
    print(magician.title()+", that was a great trick!")
    print("I can't wait to see your next trick,"+magician.title()+
    ".\n")

1   print("Thank you, everyone. That was a great magic show!")
```

The first two print statements are repeated once for each magician in the list, as you saw earlier. However, because the line at `1` is not indented, it's printed only once:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

When you're processing data using a `for` loop, you'll find that this is a good way to summarize an operation that was performed on an entire data set. Python uses indentation to determine when one line of code is connected to the line above it. In the previous examples, the lines that printed messages to individual magicians were part of the for loop because they were indented. Python's use of indentation makes code very easy to read. Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure.

## Forgetting to Indent

Always indent the line after the for statement in a loop. If you forget, Python will remind you:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
1    print(magician)
```

The print statement at `1` should be indented, but it's not. When Python expects an indented block and doesn't find one, it lets you know which line it had a problem with.

```
  File "magicians.py", line 3
       print(magician)
             ^
  Indentation Error: expected an indented block
```

You can usually resolve this kind of indentation error by indenting the line or lines immediately after the for statement.

***Forgetting to Indent Additional Lines***

This is what happens when we forget to indent the second line in the loop that tells each magician we're looking forward to their next trick:

```
  magicians=['alice','david','carolina']
  for magician in magicians:
     print(magician.title()+", that was a great trick!")
1    print("I can't wait to see your next trick, "+magician.title()+
           ".\n")
```

The print statement at `1` is supposed to be indented, but because Python finds at least one indented line after the for statement, it doesn't report an error. As a result, the first print statement is executed once for each name in the list because it is indented. The second print statement is not indented, so it is executed only once after the loop has finished running. Because the final value of magician is `'carolina'`, she is the only one who receives the "`looking forward to the next trick`" message:

```
  Alice, that was a great trick!
  David, that was a great trick!
  Carolina, that was a great trick!
  I can't wait to see your next trick, Carolina.
```

This is a `logical error`. The syntax is valid Python code, but the code does not produce the desired result because a problem occurs in its logic. If you expect to see a certain action repeated once for each item in a list and it's executed only once, determine whether you need to simply indent a line or a group of lines.

***Indenting Unnecessarily***

If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

```
  message="Hello Python world!"
1     print(message)
```

We don't need to indent the print statement at `1`, because it doesn't belong to the line above it; hence, Python reports that error:

```
  File "hello_world.py", line 2
       print(message)
            ^
  Indentation Error: unexpected indent
```

You can avoid unexpected indentation errors by indenting only when you have a specific reason to do so. In the programs, you're writing at this point, the only lines you should indent are the actions you want to repeat for each item in a for loop.

***Indenting Unnecessarily After the Loop***

Let's see what happens when we accidentally indent the line that thanked the magicians as a group for putting on a good show:

```
  magicians=['alice', 'david', 'carolina']
  for magician in magicians:
     print(magician.title()+", that was a great trick!")
     print("I can't wait to see your next trick,"+magician.title()+
     ".\n")

1 print("Thank you everyone, that was a great magic show!")
```

Because the line at `1` is indented, it's printed once for each person in the list, as you can see at `2`:

```
  Alice, that was a great trick!
  I can't wait to see your next trick, Alice.

2 Thank you everyone, that was a great magic show!
  David, that was a great trick!
  I can't wait to see your next trick, David.

2 Thank you everyone, that was a great magic show!
  Carolina, that was a great trick!
  I can't wait to see your next trick, Carolina.

2 Thank you everyone, that was a great magic show!
```

This is another logical error, similar to the one in "`Forgetting to Indent Additional Lines`". Because Python doesn't know what you're trying to accomplish with your code, it will run all code that is written in valid syntax. If an action is repeated many times when it should be executed only once, determine whether you just need to unintent the code for that action.

***Forgetting the Colon***

The colon at the end of a for statement tells Python to interpret the next line as the start of a loop.

```
  magicians=['alice','david','carolina']
1 for magician in magicians
    print(magician)
```

If you accidentally forget the colon, as shown at `1`, you'll get a syntax error because Python doesn't know what you're trying to do.

## Using the `range()` Function

Python's `range()` function makes it easy to generate a series of numbers. For example, you can use the `range()` function to print a series of numbers like this:

```
for value in range(1,5):
    print(value)
```

Although this code looks like it should print the numbers from `1` to `5`, it doesn't print the number `5`:

```
1
2
3
4
```

In this example, `range()` prints only the numbers `1` through `4`. This is another result of the off-by-one behaviour you'll see often in programming languages. The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide. Because it stops at that second value, the output never contains the end value, which would have been `5` in this case.

To print the numbers from `1` to `5`, you would use `range(1,6)`:

```
for value in range(1,6):
print(value)
```

This time the output starts at 1 and ends at 5:

```
1
2
3
4
```

```
5
```

If your output is different than what you expect when you're using `range()`, try adjusting your end value by `1`.

### *Using `range()` to Make a List of Numbers*

If you want to make a list of numbers, you can convert the results of `range()` directly into a list using the `list()` function. When you wrap `list()` around a call to the `range()` function, the output will be a list of numbers.

In the example in the previous section, we simply printed out a series of numbers. We can use `list()` to convert that same set of numbers into a list:

```
numbers=list(range(1,6))
print(numbers)
```

And this is the result:

```
[1, 2, 3, 4, 5]
```

We can also use the `range()` function to tell Python to skip numbers in a given range. For example, here's how we would list the even numbers between 1 and 10:

```
even_numbers=list(range(2,11,2))
print(even_numbers)
```

In this example, the `range()` function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

```
[2, 4, 6, 8, 10]
```

You can create almost any set of numbers you want to using the `range()` function. For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10). In Python, two asterisks (`**`) represent exponents. Here's how you might put the first 10 square numbers into a list:

```
squares=[]
for value in range(1,11):
    square=value**2
    squares.append(square)

print(squares)
```

We start with an empty list called squares at `1`. At `2`, we tell Python to loop through each value from 1 to 10 using the `range()` function. Inside the loop, the current value is raised to the second power and

stored in the variable square at 3. At 4, each new value of square is appended to the list squares. Finally, when the loop has finished running, the list of squares is printed at 5:

```
  [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To write this code more concisely, omit the temporary variable square and append each new value directly to the list:

```
  squares=[]
  for value in range(1,11):
1     squares.append(value**2)

  print(squares)
```

The code at 1 does the same work as the lines at 3 and 4 in *squares.py*. Each value in the loop is raised to the second power and then immediately appended to the list of squares.

## Simple Statistics with a List of Numbers

A few Python functions are specific to lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers:

```
>>> digits=[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

*The examples in this section use short lists of numbers in order to fit easily on the page. They would work just as well if your list contained a million or more numbers.*

### *List Comprehensions*

The approach described earlier for generating the list squares consisted of using three or four lines of code. A list comprehension allows you to generate this same list in just one line of code. A list comprehension combines the for loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but we have included them here because you'll most likely see them as soon as you start looking at other people's code.

The following example builds the same list of square numbers you saw earlier but uses a list comprehension:

```
  squares=[value**2 for value in range(1,11)]
```

```
print(squares)
```

To use this syntax, begin with a descriptive name for the list, such as squares. Next, open a set of square brackets and define the expression for the values you want to store in the new list. In this example, the expression is `value**2`, which raises the value to the second power. Then, write a `for` loop to generate the numbers you want to feed into the expression, and close the square brackets. The `for` loop in this example is for value in `range(1,11)`, which feeds the values 1 through 10 into the expression `value**2`. Notice that no colon is used at the end of the for statement.
The result is the same list of square numbers you saw earlier:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

## Slicing a List

To make a slice, you specify the index of the first and last elements you want to work with. As with the `range()` function, Python stops one item before the second index you specify. To output the first three elements in a list, you would request indices 0 through 3, which would return elements 0, 1, and 2.
The following example involves a list of players on a team:

```
  players=['charles','martina','michael','florence','eli']
1 print(players[0:3])
```

The code at `1` prints a slice of this list, which includes just the first three players. The output retains the structure of the list and includes the first three players in the list:

```
['charles','martina','michael']
```

You can generate any subset of a list. For example, if you want the second, third, and fourth items in a list, you would start the slice at index-1 and end at index-4:

```
players=['charles','martina','michael','florence','eli']
print(players[1:4])
```

This time the slice starts with `'martina'` and ends with `'florence'`:

```
['martina','michael','florence']
```

If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

```
players=['charles','martina','michael','florence','eli']
print(players[:4])
```

Without a starting index, Python starts at the beginning of the list:

```
['charles','martina','michael','florence']
```

A similar syntax works if you want a slice that includes the end of a list. For example, if you want all items from the third item through the last item, you can start with index-2 and omit the second index:

```
players=['charles','martina','michael','florence','eli']
print(players[2:])
```

Python returns all items from the third item through the end of the list:

```
['michael','florence','eli']
```

This syntax allows you to output all of the elements from any point in your list to the end regardless of the length of the list. Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice `players[-3:]`:

```
players=['charles','martina','michael','florence','eli']
print(players[-3:])
```

This prints the names of the last three players and would continue to work as the list of players changes in size.

***Looping Through a Slice***

You can use a slice in a for loop if you want to loop through a subset of the elements in a list. In the next example we loop through the first three players and print their names as part of a simple roster:

```
players=['charles','martina','michael','florence','eli']
print("Here are the first three players on my team:")
for player in players[:3]:
    print(player.title())
```

Instead of looping through the entire list of players at `1`, Python loops through only the first three names:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Slices are very useful in a number of situations. For instance, when you're working with data, you can use slices to process your data in chunks of a specific size. Or, when you're building a web application, you could use slices to display information in a series of pages with an appropriate amount of information on each page.

## Copying a List

To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index (`[:]`). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list. For example,

```
1 my_foods=['pizza','falafel','carrot cake']
2 friend_foods=my_foods[:]

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

At `1` we make a list of the foods we like called `my_foods`. At `2` we make a new list called `friend_foods`. We make a copy of `my_foods` by asking for a slice of `my_foods` without specifying any indices and store the copy in `friend_foods`. When we print each list, we see that they both contain the same foods:

```
  My favourite foods are:
  ['pizza','falafel','carrot cake']

  My friend's favourite foods are:
  ['pizza','falafel','carrot cake']
```

To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favourite foods:

```
  my_foods=['pizza','falafel','carrot cake']
1 friend_foods=my_foods[:]

2 my_foods.append('cannoli')
3 friend_foods.append('ice cream')

  print("My favourite foods are:")
  print(my_foods)
  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

At `1`, we copy the original items in `my_foods` to the new list `friend_foods`, as we did in the previous example. Next, we add a new food to each list: at `2`, we add `'cannoli'` to `my_foods`, and at `3`, we add `'ice cream'` to `friend_foods`. We then print the two lists to see whether each of these foods is in the appropriate list.

```
  My favourite foods are:
4 ['pizza','falafel','carrot cake','cannoli']

  My friend's favourite foods are:
5 ['pizza','falafel','carrot cake','ice cream']
```

The output at 4 shows that 'cannoli' now appears in our list of favourite foods but 'ice cream' doesn't. At 5, we can see that 'ice cream' now appears in our friend's list but 'cannoli' doesn't. If we had simply set friend_foods equal to my_foods, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice:

```
  my_foods=['pizza','falafel','carrot cake']

  #This doesn't work:
1 friend_foods=my_foods

  my_foods.append('cannoli')
  friend_foods.append('ice cream')

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

Instead of storing a copy of my_foods in friend_foods at 1, we set friend_foods equal to my_foods. This syntax actually tells Python to connect the new variable friend_foods to the list that is already contained in my_foods, so now both variables point to the same list. As a result, when we add 'cannoli' to my_foods, it will also appear in friend_foods. Likewise, 'ice cream' will appear in both lists, even though it appears to be added only to friend_foods.
The output shows that both lists are the same now, which is not what we wanted:

```
  My favourite foods are:
  ['pizza','falafel','carrot cake','cannoli','ice cream']
  My friend's favourite foods are:
  ['pizza','falafel','carrot cake','cannoli','ice cream']
```

*Don't worry about the details in this example for now. Basically, if you're trying to work with a copy of a list and you see unexpected behaviour, make sure you are copying the list using a slice, as we did in the first example.*

## Defining a Tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.
For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

```
1 dimensions=(200,50)
2 print(dimensions[0])
  print(dimensions[1])
```

We define the tuple dimensions at 1, using parentheses instead of square brackets. At 2, we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
  200 50
```

Let's see what happens if we try to change one of the items in the tuple dimensions:

```
  dimensions=(200,50)
1 dimensions[0]=250
```

The code at 1 tries to change the value of the first dimension, but Python returns a type error. Basically, because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
  Traceback (most recent call last):
     File "dimensions.py", line 3, in <module>
          dimensions[0] = 250
  TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

***Looping Through All Values in a Tuple***

You can loop over all the values in a tuple using a for loop, just as you did with a list:

```
  dimensions=(200,50)
  for dimension in dimensions:
     print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
  200
  50
```

***Writing over a Tuple***

Although you can't modify a tuple, you can assign a new value to a variable that holds a tuple. So if we wanted to change our dimensions, we could redefine the entire tuple:

```
1 dimensions=(200,50)
  print("Original dimensions:")
  for dimension in dimensions:
     print(dimension)

2 dimensions=(400,100)
3 print("\nModified dimensions:")
  for dimension in dimensions:
```

```
    print(dimension)
```

The block at 1 defines the original tuple and prints the initial dimensions. At 2, we store a new tuple in the variable dimensions. We then print the new dimensions at 3. Python doesn't raise any errors this time, because overwriting a variable is valid:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.

**Exercises**

1. Write a python program to sum and multiply all the items in a list
2. Write a python program to get the largest nad smallest number from a list.
3. Write a python program to count the number of strings where the string length is 2 or more and the first and last character are same from a given list of strings. Go to the editor

   **Sample List** : `['abc', 'xyz', 'aba', '1221']`

   **Expected Result** : `2`
4. Write a python program to remove duplicates from a list.
5. Write a python program to clone or copy a list.
6. Write a python program to print a specified list after removing the 0th, 4th and 5th elements.

   **Sample List** : `['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow']`

   **Expected Output** : `['Green', 'White', 'Black']`
7. Write a python program to generate all permutations of a list in Python.
8. Write a python program to find the index of an item in a specified list.
9. Write a python program to append a list to the second list.
10. Write a python program to get the frequency of the elements in a list.
11. Write a python program to find the list in a list of lists whose sum of elements is the highest.

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

    **Expected Output**: [10, 11, 12]
12. Write a python program to convert a tuple to a string.
13. Write a python program to find the repeated items of a tuple.
14. Write a python program to check whether an element exists within a tuple.
15. Write a python program to convert a list to a tuple.
16. Write a python program to remove an item from a tuple.

17. Write a python program to find the index of an item of a tuple.

18. Write a python program to reverse a tuple.