


07 Prepare: Lists and Repetition

During this lesson, you will learn how to store many elements in a Python list. You will learn how to write a loop that processes each element in a list. Also, you will learn that lists are passed into a function differently than numbers are passed.

Videos

Watch these videos from Microsoft about lists and repetition in Python.

 [Lists](#) (12 minutes)

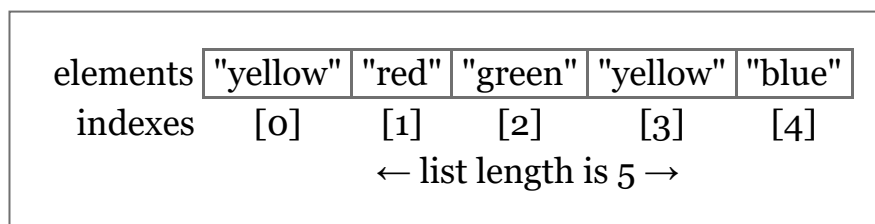
 [Loops](#) (6 minutes)

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson:

Lists

A Python program can store many values in a **list**. Lists are mutable, meaning they can be changed after they are created. Each value in a list is called an **element** and is stored at a unique index. An **index** is always an integer and determines where an element is stored in a list. The first index of a Python list is always zero (0). The following diagram shows a list that contains five strings. The diagram shows both the elements and the indexes of the list. Notice that each index is a unique integer, and that the first index is zero.



In a Python program, we can create a list by using square brackets ([and]). We can determine the number of items in a list by calling the built-in `len` function. We can retrieve an item from a list and replace an item in a list using square brackets ([and]) and an index. Example 1 contains a program that creates a list, prints the length of the list, retrieves and prints one item from the list, changes one item in the list, and then prints the entire list.

```
# Example 1

def main():
    # Create a list that contains five strings.
```

```

colors = ["yellow", "red", "green", "yellow", "blue"]

# Call the built-in len function
# and print the length of the list.
length = len(colors)
print(f"Number of elements: {length}")

# Print the element that is stored
# at index 2 in the colors list.
print(colors[2])

# Change the element that is stored at
# index 3 from "yellow" to "purple".
colors[3] = "purple"

# Print the entire colors list.
print(colors)

# Call main to start this program.
if __name__ == "__main__":
    main()

```

```

> python example_1.py
Number of elements: 5
green
['yellow', 'red', 'green', 'purple', 'blue']

```

We can add an item to a list by using the `insert` and `append` methods. We can determine if an element is in a list by using the Python membership operator, which is the keyword `in`. We can find the index of an item within a list by using the `index` method. We can remove an item from a list by using the `pop` and `remove` methods. Example 2 shows how to create a list and add, find, and remove items from a list.

```

# Example 2

def main():
    # Create an empty list that will hold fabric names.
    fabrics = []

    # Add three elements at the end of the fabrics list.
    fabrics.append("velvet")
    fabrics.append("denim")
    fabrics.append("gingham")

    # Insert an element at the beginning of the fabrics list.
    fabrics.insert(0, "chiffon")
    print(fabrics)

    # Determine if gingham is in the fabrics list.
    if "gingham" in fabrics:
        print("gingham is in the list.")
    else:
        print("gingham is NOT in the list.")

    # Get the index where velvet is stored in the fabrics list.
    i = fabrics.index("velvet")

    # Replace velvet with taffeta.
    fabrics[i] = "taffeta"

    # Remove the last element from the fabrics list.

```

```
fabrics.pop()

# Remove denim from the fabrics list.
fabrics.remove("denim")

# Get the length of the fabrics list and print it.
n = len(fabrics)
print(f"The fabrics list contains {n} elements.")
print(fabrics)

# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_2.py
['chiffon', 'velvet', 'denim', 'gingham']
gingham is in the list.
The fabrics list contains 2 elements.
['chiffon', 'taffeta']
```

The lists in examples 1 and 2 store strings. Of course, it is possible to store numbers in a list, too. In fact, Python allows a program to store any data type in a list, including other lists.

Repetition

A programmer can cause a computer to repeat a group of statements by writing `for` and `while` loops.

for loop

A `for` loop iterates over a sequence, such as a list. This means a `for` loop causes the computer to repeatedly execute the statements in the body of the `for` loop, once for each element in the sequence. In example 3, consider the list of colors at line 5 and the `for` loop at lines 8–9. Notice how the `for` loop causes the computer to repeat line 9 once for each element in the `colors` list. Of course, the code in the body of a loop can do much more with each element than simply print it.

```
1  # Example 3
2
3  def main():
4      # Create a list of color names.
5      colors = ["red", "orange", "yellow", "green", "blue"]
6
7      # Use a for loop to print each element in the list.
8      for color in colors:
9          print(color)
10
11
12 # Call main to start this program.
13 if __name__ == "__main__":
14     main()
```

```
> python example_3.py
red
orange
```

```
yellow
green
blue
```

Notice in example 3 at lines 8–9 that just like `if` statements in Python, the body of a loop starts and ends with indentation.

range function

The Python built-in [range function](#) creates and returns a sequence of numbers. The `range` function accepts one, two, or three parameters as shown in example 4 and its output. Many programmers use the `range` function in a `for` loop to cause the computer to repeat code once for each number in a range of numbers. Example 4 shows four `for` loops that iterate over a range of numbers.

```
# Example 4

def main():
    # Count from zero to nine by one.
    for i in range(10):
        print(i)
    print()

    # Count from five to nine by one.
    for i in range(5, 10):
        print(i)
    print()

    # Count from zero to eight by two.
    for i in range(0, 10, 2):
        print(i)
    print()

    # Count from 100 down to 70 by three.
    for i in range(100, 69, -3):
        print(i)

# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_4.py
0
1
2
:
8
9

5
6
7
8
9

0
2
4
6
8
```

```
100
97
94
:
73
70
```

In example 5 at lines 8–9 and lines 15–17, there are two `for` loops. Both loops print each element from a list named *colors*. The first loop iterates over the elements in the *colors* list. The second loop uses the built-in `len` and `range` functions to iterate over the indexes of the *colors* list. Which style of `for` loop do you prefer to read and write? Most programmers prefer to write a loop like the one at lines 8–9 because it is simpler than the one at lines 15–17.

```
1  # Example 5
2
3  def main():
4      # Create a list of color names.
5      colors = ["red", "orange", "yellow", "green", "blue"]
6
7      # Use a for loop to print each element in the list.
8      for color in colors:
9          print(color)
10
11     print()
12
13     # Use a different for loop to
14     # print each element in the list.
15     for i in range(len(colors)):
16         # Use the index i to retrieve
17         # an element from the list.
18         color = colors[i]
19
20         print(color)
21
22
23 # Call main to start this program.
24 if __name__ == "__main__":
25     main()
```

```
> python example_5.py
red
orange
yellow
green
blue

red
orange
yellow
green
blue
```

In the previous example, the code in the body of both `for` loops is very short and simply prints one element from the list each time through the loop. However, you can write as many lines of code as you need in the body of a loop to repeatedly perform all sorts of computations for each element in a list.

break statement

A `break` statement causes a loop to end early. In example 6 at lines 8–12, there is a `for` loop that asks the user to input ten numbers one at a time. However, the loop will terminate early if the user enters a zero (0) because of the `if` statement and `break` statement at lines 10 and 11.

```

1  # Example 6
2
3  def main():
4      sum = 0
5
6      # Get ten or fewer numbers from the user and
7      # add them together.
8      for i in range(10):
9          number = float(input("Please enter a number: "))
10         if number == 0:
11             break
12         sum += number
13
14     # Print the sum of the numbers for the user to see.
15     print(f"sum: {sum}")
16
17
18 # Call main to start this program.
19 if __name__ == "__main__":
20     main()

```

```

> python example_6.py
Please enter a number: 6
Please enter a number: 4
Please enter a number: -2
Please enter a number: 0
sum: 8.0

```

while loop

A `while` loop is more flexible than a `for` loop and repeats while some condition is true. Imagine that we need a function to compare the contents of two lists? Can we use a loop to compare the contents of two lists? Example 7 contains a `while` loop at lines 35–46 with an `if` statement at line 42 that finds the first index where two lists differ.

```

1  # Example 7
2
3  def main():
4      list1 = ["red", "orange", "yellow", "green", "blue"]
5      list2 = ["red", "orange", "green", "green", "blue"]
6
7      index = compare_lists(list1, list2)
8      if index == -1:
9          print("The contents of list1 and list2 are equal")
10     else:
11         print(f"list1 and list2 differ at index {index}")
12
13
14 def compare_lists(list1, list2):
15     """Compare the contents of two lists. If the contents
16     of the two lists are not equal, return the index of
17     the first difference. If the contents of the two lists
18     are equal, return -1.
19
20     Parameters

```

```

21         list1: a list
22         list2: another list
23     Return: an index or -1
24     """
25     # Get the length of the shortest list.
26     length1 = len(list1)
27     length2 = len(list2)
28     limit = min(length1, length2)
29
30     # Begin at the first index (0) and repeat until the
31     # computer finds two elements that are not equal or
32     # until the computer reaches the end of the shortest
33     # list, whichever comes first.
34     i = 0
35     while i < limit:
36         # Retrieve one element from each list.
37         element1 = list1[i]
38         element2 = list2[i]
39
40         # If the two elements are not
41         # equal, quit the while loop.
42         if element1 != element2:
43             break
44
45         # Add one to the index variable.
46         i += 1
47
48     # If the length of both lists are equal and the
49     # computer verified that all elements are equal,
50     # set i to -1 to indicate that the contents of
51     # the two lists are equal.
52     if length1 == length2 == i:
53         i = -1
54
55     return i
56
57
58 # Call main to start this program.
59 if __name__ == "__main__":
60     main()

```

```

> python example_7.py
list1 and list2 differ at index 2

```

Compound Lists

A **compound list** is a list that contains other lists. Compound lists are used to store lots of related data. Example 8 shows how to create a compound list, retrieve one inner list from the compound list, and retrieve an individual number from the inner list.

```

# Example 8

def main():
    # These are the indexes of each
    # element in the inner lists.
    YEAR_PLANTED_INDEX = 0
    HEIGHT_INDEX = 1
    GIRTH_INDEX = 2
    FRUIT_AMOUNT_INDEX = 3

    # Create a compound list that stores inner lists.

```

```

apple_tree_data = [
    # [year_planted, height, girth, fruit_amount]
    [2012, 2.7, 3.6, 70.5],
    [2012, 2.4, 3.7, 81.3],
    [2015, 2.3, 3.6, 62.7],
    [2016, 2.1, 2.7, 42.1]
]

# Retrieve one inner list from the compound list.
one_tree = apple_tree_data[2]

# Retrieve one value from the inner list.
height = one_tree[HEIGHT_INDEX]

# Print the tree's height.
print(f"height: {height}")

# Call main to start this program.
if __name__ == "__main__":
    main()

```

```

> python example_8.py
height: 2.3

```

Example 9 shows how to process all elements in a compound list. The `for` loop at line 24 causes the computer to repeat lines 24–34 once for each inner list that is inside the compound list named `apple_tree_data`. Line 28 retrieves the fruit amount from one inner list and then line 34 adds one fruit amount to the total fruit amount.

```

1  # Example 9
2
3  def main():
4      # These are the indexes of each
5      # element in the inner lists.
6      YEAR_PLANTED_INDEX = 0
7      HEIGHT_INDEX = 1
8      GIRTH_INDEX = 2
9      FRUIT_AMOUNT_INDEX = 3
10
11     # Create a compound list that stores inner lists.
12     apple_tree_data = [
13         # [year_planted, height, girth, fruit_amount]
14         [2012, 2.7, 3.6, 70.5],
15         [2012, 2.4, 3.7, 81.3],
16         [2015, 2.3, 3.6, 62.7],
17         [2016, 2.1, 2.7, 42.1]
18     ]
19
20     total_fruit_amount = 0
21
22     # This loop will repeat once for each inner list
23     # in the apple_tree_data compound list.
24     for inner_list in apple_tree_data:
25
26         # Retrieve the fruit amount from
27         # the current inner list.
28         fruit_amount = inner_list[FRUIT_AMOUNT_INDEX]
29
30         # Print the fruit amount for the current tree.
31         print(fruit_amount)
32

```



```

33         # Add the current fruit amount to the total.
34         total_fruit_amount += fruit_amount
35
36     # Print the total fruit amount.
37     print(f"Total fruit amount: {total_fruit_amount:.1f}")
38
39
40 # Call main to start this program.
41 if __name__ == "__main__":
42     main()

```

```

> python example_9.py
70.5
81.3
62.7
42.1
Total fruit amount: 256.6

```

Values and References

In a Python program, the computer assigns values to variables differently based on their data type. Consider the small program in example 10 and the output of that program. The program in example 10 contains two integer variables named *x* and *y*. The program in example 10 does the following:

- The statement at line 4 stores the value 17 into the variable *x*.
- Line 5 copies the value that is in the variable *x* into the variable *y*.
- Line 6 prints the values of *x* and *y* which are both 17.
- Line 7 adds one to the value of *x*, making its value 18 instead of 17.
- Line 8 prints the values of *x* and *y* again. The value of *x* was changed to 18. The value of *y* remained unchanged.

Why does line 7 (*x* += 1) change the value of *x* but not change the value of *y*? Because line 5 copies *the value* that was in *x* into *y*. In other words, *x* and *y* are two separate variables, each with its own value.

```

1  # Example 10
2
3  def main():
4      x = 17
5      y = x
6      print(f"Before changing x: x {x} y {y}")
7      x += 1
8      print(f"After changing x: x {x} y {y}")
9
10 # Call main to start this program.
11 if __name__ == "__main__":
12     main()

```

```

> python example_10.py
Before changing x: x 17 y 17
After changing x: x 18 y 17

```

Example 11 shows a small Python program that contains two variables named *lx* and *ly* that each refer to a list. This program is similar to the previous program, but it has two lists instead of two integers. From the output of example 11, we

see there is a big difference between the way a Python program assigns integers and the way it assigns lists. The program in example 11 does the following:

- The statement at line 4 creates a list and stores a reference to that list in the variable *lx*.
- Line 5 copies the reference in the variable *lx* into the variable *ly*. Line 5 does not create a copy of the list but instead causes both the variables *lx* and *ly* to refer to the same list.
- Line 6 prints the values of *lx* and *ly*. Notice that their values are the same as we expect them to be because of line 5.
- Line 7 appends the number 5 onto the list *lx*.
- Line 8 prints the values of *lx* and *ly* again. Notice in the output that when *lx* and *ly* are printed the second time, it appears that the number 5 was appended to both lists.

Why does it appear that appending the number 5 onto *lx* also appends the number 5 onto *ly*? Because *lx* and *ly* refer to the same list. There is really only one list with two references to that list. Because *lx* and *ly* refer to the same list, a change to the list through variable *lx* can be seen through variable *ly*.

```

1  # Example 11
2
3  def main():
4      lx = [7, -2]
5      ly = lx
6      print(f"Before changing lx: lx {lx} ly {ly}")
7      lx.append(5)
8      print(f"After changing lx: lx {lx} ly {ly}")
9
10 # Call main to start this program.
11 if __name__ == "__main__":
12     main()

```

```

> python example_11.py
Before changing lx: lx [7, -2] ly [7, -2]
After changing lx: lx [7, -2, 5] ly [7, -2, 5]

```

From examples 10 and 11, we learn that when a computer executes a Python statement to assign the value of a boolean, integer, or float variable to another variable ($y = x$), the computer copies *the value* of one variable into the other. However, when a computer executes a Python statement to assign the value of a list variable to another variable ($ly = lx$), the computer does not copy *the value* but instead copies *the reference* so that both variables refer to the same list in memory.

Pass by Value and Pass by Reference

The fact that the computer copies the value of some data types (boolean, integer, float) and copies the reference for other data types (list and other large data types) has important implications for passing arguments into functions. Consider the Python program in example 12 with two functions named `main` and `modify_args`. The program in example 12 does the following:

- The statement at line 5 assigns the value 5 to a variable named *x*.

- Line 6 assigns a list to a variable named *lx*.
- Line 7 prints the values of *x* and *lx* before they are passed to the *modify_args* function.
- Line 11 calls the *modify_args* function and passes *x* and *lx* to that function.
- Within the *modify_args* function, line 28 changes the value of the parameter *n* by adding one to it, and line 29 changes the value of the parameter *alist* by appending the number 4 onto it.
- Line 13 prints the values of *x* and *lx* after they were passed to the *modify_args* function. Notice in the output below that the value of *x* was not changed by the *modify_args* function. However, the value of *lx* was changed by the *modify_args* function.

```

1  # Example 12
2
3  def main():
4      print("main()")
5      x = 5
6      lx = [7, -2]
7      print(f"Before calling modify_args(): x {x}  lx {lx}")
8
9      # Pass one integer and one list
10     # to the modify_args function.
11     modify_args(x, lx)
12
13     print(f"After calling modify_args():  x {x}  lx {lx}")
14
15
16 def modify_args(n, alist):
17     """Demonstrate that the computer passes a value
18     for integers and passes a reference for lists.
19     Parameters
20         n: A number
21         alist: A list
22     Return: nothing
23     """
24     print("    modify_args(n, alist)")
25     print(f"    Before changing n and alist: n {n}  alist {alist}")
26
27     # Change the values of both parameters.
28     n += 1
29     alist.append(4)
30
31     print(f"    After changing n and alist:  n {n}  alist {alist}")
32
33
34 # Call main to start this program.
35 if __name__ == "__main__":
36     main()

```

```

> python example_12.py
main()
Before calling modify_args(): x 5  lx [7, -2]
    modify_args(n, alist)
    Before changing n and alist: n 5  alist [7, -2]
    After changing n and alist:  n 6  alist [7, -2, 4]
After calling modify_args():  x 5  lx [7, -2, 4]

```

From the output of example 12, we see that modifying an integer parameter changes the integer within the called function only. However, modifying a list parameter changes the list within the called function and within the calling

function. Why? Because when a computer passes a boolean, integer, or float variable to a function, the computer copies *the value* of that variable into the parameter of the called function. Copying the value of an argument into a parameter is known as **pass by value**. However, when a computer passes a list variable to a function, the computer copies *the reference* so that the original variable and the parameter both refer to the same list in memory. Copying the reference of an argument into a parameter is known as **pass by reference**.

Rationale for Pass by Reference





Why are booleans and numbers passed to a function by value and lists are passed to a function by reference? To understand the answer to this question, consider the work a computer would have to do if lists were passed by value.

When a computer passes a number (or boolean) variable to a function, the number is passed by value which means the computer copies the value of the number variable into the parameter of the called function. This works well for numbers because each number variable occupies a small amount of the computer's memory. Making a copy of a number is fast, and the copy uses a small amount of memory.

However, a list may contain millions of elements and therefore occupy a large amount of the computer's memory. If lists were passed by value to a function, the computer would have to make a copy of a list each time it is passed to a function. If a list is large, copying the list takes a relatively long time and uses a lot of the computer's memory for the copy. Therefore, to make programs fast and use less memory, lists (and other large data types) are passed to a function by reference.

Tutorials

If the concepts in this preparation content seem confusing to you, reading these tutorials may help you better understand the concepts.

-  [Lists in Python](#)
-  [More on Lists](#)
-  [Python "for" Loops](#)
-  [Python "while" Loops](#)

Summary

During this lesson, you are learning how to store many values in a list. Each element in a list is stored at a unique index. Each index is an integer, and the first index in a Python list is always zero (0). The built-in `len` function will return the number of elements stored in a list. The index of the last element in a Python list is always one less than the length of the list. To retrieve or store one element in a list, you can use the square brackets (`[` and `]`) and an index. To

process all the elements in a list, you can write a `for` loop. A compound list is a list that stores smaller lists.

During this lesson, you are also learning the difference between passing arguments into a function by value and by reference. Numbers are passed into a function by value, meaning that the computer copies the number from an argument into a parameter. Lists are passed into a function by reference, meaning the computer does not make a copy of a list argument but instead passes a reference to the list into a called function. Because lists are passed by reference, if a called function changes a list that is a parameter, that function is not changing a copy of the list but instead is changing the original list from the calling function.