

07 Prepare: Lists and Repetition

During this lesson, you will learn how to store many items 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

Within a Python program, we 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 words. 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.

elements	"yellow"	"red"	"green"	"yellow"	"blue"
indexes	[0]	[1]	[2]	[3]	[4]

← list length is 5 →

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 words.
    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 `del`, `pop`, and `remove`. 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 fabrics.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 programmer to store other data types in a list, including other lists.

Compound Lists

A **compound list** is a list that contains other lists. Compound lists are used to store lots of related data. Example 3 shows how to create a compound list, retrieve an inner list from the compound list, and retrieve an individual number from the inner list.

```
# Example 3

def main():
    # Create a compound list that stores smaller 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]
    ]

    # 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

    # 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_3.py
height: 2.3
```

Repetition

We can cause a computer to repeat a group of statements by writing **for** and **while** loops.

for

A **for** loop iterates over a range of numbers, such as `range(3, 10)` or a sequence, such as a list. Example 4 shows three **for** loops that iterate over a range of numbers. Notice that just like **if** statements in Python, the body of a loop starts and ends with indentation.

```
# Example 4

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

0
2
4
6
8

100
97
94
:
73
70
```

break

A **break** statement causes a loop to end early. In example 5 at [line 8](#), 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 **break** statement at [line 11](#). The **for** loop in example 5 uses underscore (`_`) as the counting variable, which is a valid variable name in Python. Many Python programmers use underscore as the name of the counting variable to indicate that the counting variable will not be used within the body of the loop.

```
1 # Example 5
2
3 def main():
4     sum = 0
5
6     # Get ten or fewer numbers from the user and
7     # add them together.
8     for _ 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 add_numbers.py
Please enter a number: 6
Please enter a number: 4
Please enter a number: -2
Please enter a number: 0
sum: 8.0
```

for each

In example 6 at [line 10](#), there is a **for** loop that processes all the elements in a list. The code in the body of the **for** loop is very small and simply prints one element from the list each time through the loop. However, we can write as much code as we need in the body of a loop to repeatedly perform all sorts of computations for each element in the list.

```
1 # Example 6
```

```

2
3 def main():
4     # Create a list.
5     colors = ["red", "orange", "yellow", "green", "blue"]
6
7     # Use a for loop to print each element in the list.
8     # Of course, the code in the body of a loop can do
9     # much more with each element than simply print it.
10    for color in colors:
11        print(color)
12
13
14 # Call main to start this program.
15 if __name__ == "__main__":
16     main()

```

```

> python print_colors.py
red
orange
yellow
green
blue

```

while

A **while** loop is more flexible than a **for** loop and repeats while some condition is true. The program in example 7 is a number guessing game that repeatedly asks the user to guess a number between 1 and 100. At [line 21](#), the program includes a **while** loop, which will cause the computer to repeatedly execute the statements at [lines 22–35](#). The while loop will end when the user enters the correct number.

```

1  # Example 7
2
3  import random
4
5
6  def main():
7      print("I'm thinking of a number between 1 and 100.")
8      print("Try to guess it!")
9      print()
10
11     # Prepare a string that the input function will show to
12     # the user during the first iteration of the while loop.
13     prompt = "Please enter an integer between 1 and 100: "
14
15     # Get a random integer between 1 and 100.
16     answer = random.randint(1, 100)
17
18     guess = -1
19     attempts = 0
20
21     while guess != answer:
22         # Get a guess from the user.
23         guess = int(input(prompt))
24         attempts += 1
25         guide = ""
26
27         # Compare the user's guess to the answer.
28         if guess < answer:
29             guide = "too low"
30         elif guess > answer:
31             guide = "too high"
32
33         # Prepare a string that the input function will show to
34         # the user during the next iteration of the while loop.
35         prompt = f"{guess} is {guide}. Please enter another integer: "
36
37     print()
38     print(f"{guess} is correct!")
39     print(f"You used {attempts} guesses to guess the number.")
40
41
42 # Call main to start this program.

```

```

43 if __name__ == "__main__":
44     main()

```

```

> python number_guess.py
I'm thinking of a number between 1 and 100.
Try to guess it!

Please enter an integer between 1 and 100: 50
50 is too low. Please enter another integer: 75
75 is too high. Please enter another integer: 62
62 is too high. Please enter another integer: 56

56 is correct!
You used 4 guesses to guess the number.

```

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 8 and the output of that program:

```

1 # Example 8
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_8.py
Before changing x: x 17 y 17
After changing x: x 18 y 17

```

The program in example 8 contains two integer variables named *x* and *y*. The program in example 8 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*. The value of *x* was changed to 18. The value of *y* remained unchanged.

Why did [line 7](#) (*x* += 1) change the value of *x* but not change the value of *y*? Because [line 5](#) copied *the value* that was in *x* into *y*.

Example 9 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.

```

1 # Example 9
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_9.py
Before changing lx: lx [7, -2] ly [7, -2]
After changing lx: lx [7, -2, 5] ly [7, -2, 5]
```

From the output of example 9, 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 9 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 appended 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`.

From examples 8 and 9, 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 value in memory.

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 others) has important implications for passing arguments into functions. Consider the Python program in example 10 with two functions named `main` and `modify_args`.

```
1 # Example 10
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()
```

The program in example 10 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 `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.

```
> python example_10.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 10, 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. 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 value in memory.

Rationale for Pass by Reference

Why are lists passed into 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 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 into a function, the computer would have to make a copy of a list each time it is passed into 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 into a function by reference.

Tutorials

If the concepts in this prepare 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 program 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. A compound list is a list that stores smaller lists. You can write a `for` each loop that processes each element in a list.

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.