Chapter 6: Python Lists and Tuples Oh My!

Contents

Chapter 6: Python Lists and Tuples Oh My! 1
DRY 2
Lists and Tuples 3
A List is a Sequence 3
Creating a List 4
Accessing a List 4
Lists are Mutable 5
Changing List Element Values 6
Traversing a List with a Loop 7
While Loop and Lists 8
Tutorial 6.1: A List of Numbers 9
List Operations 10
List Functions 11
Examples 11
If in 13
If not in 13
List Methods 13
append() 14
insert() 14
sort() 14
reverse() 15
index() 15
remove() 15
del 15
pop() 16
extend() 16
Tutorial 6.2: Working with Lists 16
Slice 18

Tutorial 6.3: Input a Contact List 19
Tutorial 6.4: Irene's Interactive Shopping List 21
Tutorial 6.5: Parallel Lists 26
Lists and the Random Module 27
Choice 27
Tutorial 6.7: Random Events 28
Choices 28
Sample 28
Shuffle 29
List Arguments 29
Tuples 31
Working with Tuples 31
List Examples 32
Example 1: Random List 32
Example 2: Squared List 32
Example 3: Count Items 33
Example 5: Largest and Smallest 33
Example 6: Quiz Game 33
Glossary 34
Assignment Submission 35

Red light: No AI

Time required: 120 minutes

DRY

Don't **R**epeat **Y**ourself

Lists and Tuples

Lists and tuples are ordered collections or sequences of data items. Many operations will work with either. The major difference is that lists are mutable, tuples are immutable. This means that once created, a list can be changed, a tuple cannot.

A List is a Sequence

We need to get thirty test scores from a user and do something with them. We want to put them in order. We could create thirty variables, score1, score2, . . . , score30, but that would be very tedious. To put the scores in order would be extremely difficult. The solution is to use lists.

A list is a sequence of any type of values. The values in a list are called elements or sometimes items.

The following is a list named **a_list** which contains string values.

```
a_list = ["p", "y", "t", "h", "o" "n"]
```

This diagram shows how this list is organized and accessed. List index positions start at 0.

	← length = 6 →					
	"p"	"y"	"t"	"h"	"o"	"n"
Index	0	1	2	3	4	5
Negative index	-6	-5	-4	-3	-2	-1

The program below shows how to create a list and access items in a list.

```
# simple_list.py
# Create a simple list with three integers
my_list = [1, 2, 3]
# Print the list
print(my_list)
# Access the first element of the list
print(my_list[0])
# Access the second element of the list
print(my_list[1])
# Access the last element of the list
print(my_list[-1])
```

Creating a List

There are several ways to create a new list; the simplest is to enclose the elements in square brackets. An empty list is `[]`. It is the list equivalent of 0 or `''`.

```
empty_list = []
integer_list = [1, 2, 3, 10]
string_list = ["crunchy frog", "ram bladder", "lark vomit"]
```

The first example above is an empty list. The second is a list of four integers. The third is a list of three strings.

The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list:

```
monty_list = ["spam", 2.0, 5, [10, 20]]
```

Long Lists: If you have a long list to enter, you can split it across several lines, like below:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40]
```

Printing Lists: You can use the print function to print the entire contents of a list.

```
user_list = [1, 2, 3]
print(user_list)
[1, 2, 3]
```

Accessing a List

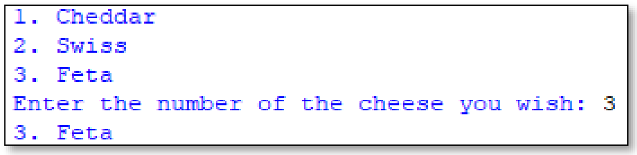
You can access individual items in a list by the index number. Remember, the index of a list starts at 0.

```
# simple_list.py
# Create a simple list with three integers
my_list = [1, 2, 3]
# Print the list
print(my_list)
# Access the first element of the list
print(my_list[0])
# Access the second element of the list
print(my_list[1])
# Access the last element of the list
print(my_list[-1])
```

The for loop prints out the list in a nice format. The user can choose which element of the list to access.

```
# access_a_list.py
# Create a list of strings
cheeses = ["1. Cheddar", "2. Swiss", "3. Feta"]

# Print the list using a for loop
# This iterates through the list one element at a time
for cheese in cheeses:
    print(cheese)

# Prompt the user for a choice
choice = int(input("Enter the number of the cheese you wish: "))

# Print the user choice
# Subtract one to align the user choice with the list index
print(cheeses[choice - 1])
```

Example run:



```
1. Cheddar
2. Swiss
3. Feta
Enter the number of the cheese you wish: 3
3. Feta
```

Lists are Mutable

The syntax for accessing the elements of a list is the bracket operator. The expression inside the brackets specifies the index. Indices start at 0:

```
cheeses = ["Cheddar", "Swiss", "Feta"]
print(cheeses[0])
Cheddar
```

Lists are mutable, they can be changed. You can change the order of items in a list or reassign an item in a list.

When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

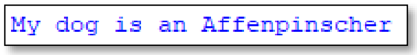
```
numbers = [17, 123]
numbers[1] = 5
print(numbers)
[17, 5]
```

The number 1 element of the numbers list, which used to be 123, is now 5.

Values from a list can be used just like a variable.

```
# print_message.py
dogs = ["Affenpinscher", "Afghan Hound", "Akita"]
message = f"My dog is an {dogs[0]}"
print(message)
```

Example run:



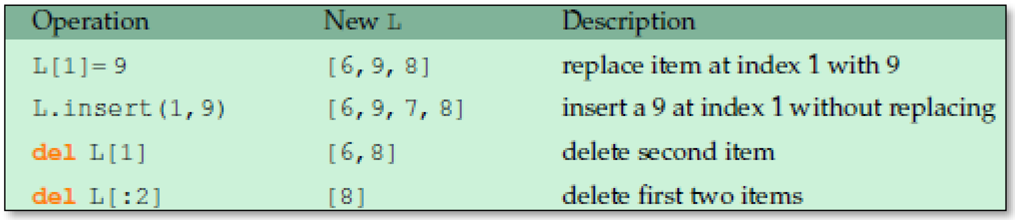
```
My dog is an Affenpinscher
```

You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index "maps to" one of the elements.

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list.

Changing List Element Values

To change the value in location 2 of L to 100, we simply say L[2]=100. If we want to insert the value 100 into location 2 without overwriting what is currently there, we can use the insert method. To delete an entry from a list, we can use the del operator. Some examples are shown below. Assume L=[6, 7, 8] for each operation.



Operation	New L	Description
`L[1]=9`	`[6, 9, 8]`	replace item at index 1 with 9
`L.insert(1,9)`	`[6, 9, 7, 8]`	insert a 9 at index 1 without replacing
`del L[1]`	`[6,8]`	delete second item
`del L[:2]`	`[8]`	delete first two items

Traversing a List with a Loop

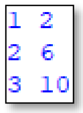
The most common way to iterate or traverse the elements of a list is with a for loop. The syntax is the same as for strings.

```
for cheese in cheeses:
    print(cheese)
cheddar
colby
swiss
```

If you want to write or update the elements, you need the indices. A common way to do that is to combine the functions **range** and **len**:

```
# range_len.py
numbers = [1, 3, 5]
for i in range(len(numbers)):
    # Update element in list
    numbers[i] = numbers[i] * 2
    print(f"{i + 1} {numbers[i]}")
```

Example run:



```
1 2
2 6
3 10
```

This loop traverses the list and updates each element. **len** returns the number of elements in the list. **range** returns a list of indices from 0 to n − 1, where **n** is the length of the list. Each time through the loop, **i** gets the index of the next element. The assignment statement in the body uses **i** to read the old value of the element and to assign the new value.

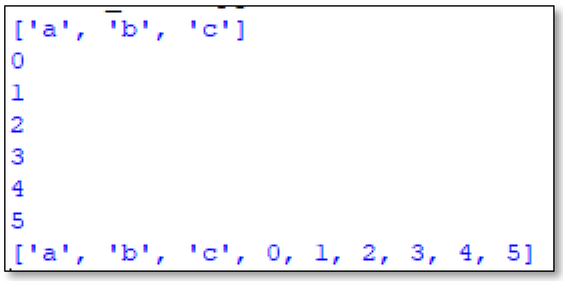
Nested List: Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
["spam", 1, ["Brie", "Roquefort", "Pol le Veq"], [1, 2, 3]]
```

While Loop and Lists

```
# list_while.py
a_list = ["a", "b", "c"]
print(a_list)

count = 0
while count <= 5:
    print(count)
    a_list.append(count)
    count = count + 1

print(a_list)
```

Example run:



```
['a', 'b', 'c']
0
1
2
3
4
5
['a', 'b', 'c', 0, 1, 2, 3, 4, 5]
```

The code above begins by creating a list named **a_list**. The list is initially populated with the strings "a", "b", and "c". The list is printed producing the first line of output text shown in the example above.

Create and Initialize a Counter Variable

Following that, a variable named count is created and initialized with the value 0. This variable will be used to control the number of iterations in the while loop.

Execute a While Loop

The next four lines of code constitute a while loop. The first line in the while loop contains a decision structure terminated by a colon.

Recall that the initial value of count is 0. The behavior of the first line in the while loop can be paraphrased as follows:

> While the value of count is less than or equal (note the relational operator that consists of a left angle bracket and an equal character) to the literal

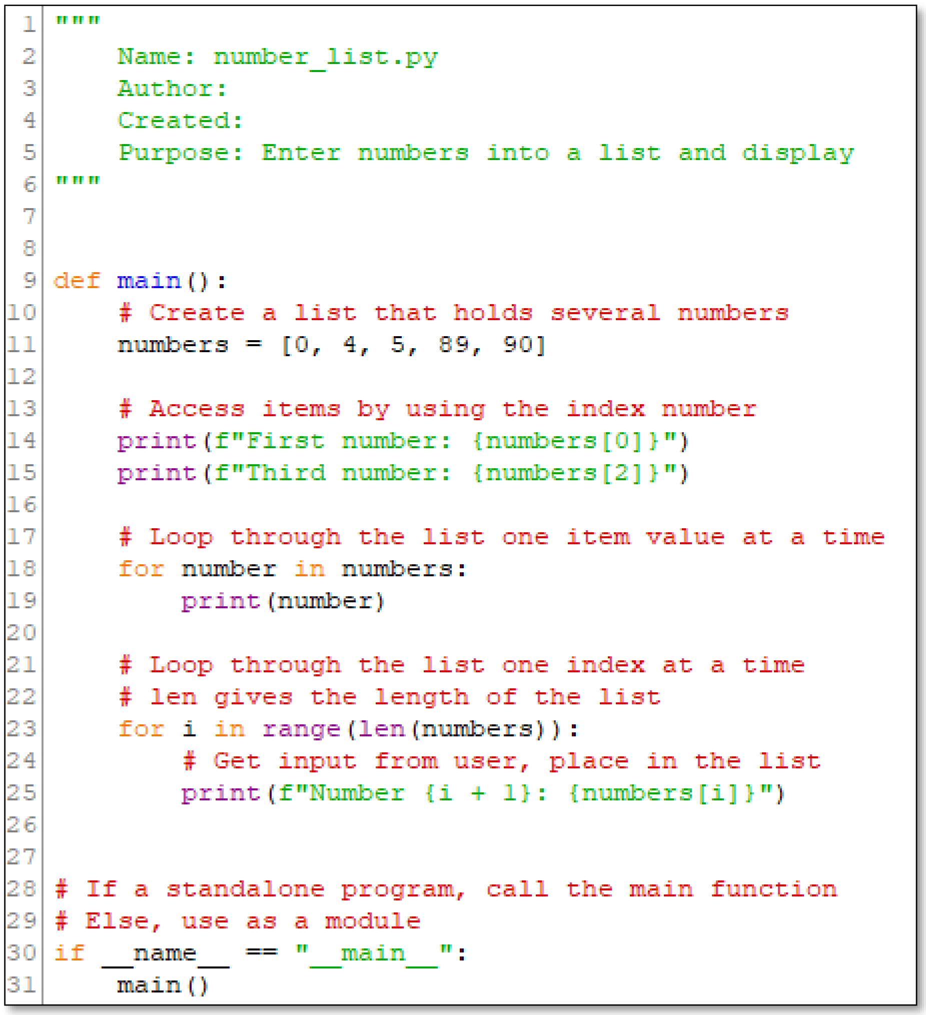
value 5, execute all the statements that follow the colon at the same indentation level.

Stated differently, for as long as the conditional clause (count less than or equal to 5) continues to be true, execute all the statements that follow the colon at the same indentation level. When the conditional clause is no longer true, skip the indented statements and transfer control to whatever follows the indented statements.

In this case, there are three statements (followed by a comment) at the same indentation level following the colon. The first of the three statements prints the current value stored in the variable named **count** as shown by the second line of text in the visualization.

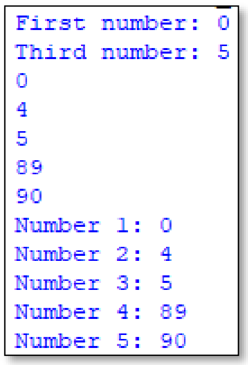
Tutorial 6.1: A List of Numbers

Add numbers to a list from user input and display the list. This tutorial demonstrates two ways to use a for loop to go through a list one item at a time.



```python
"""
    Name: number_list.py
    Author:
    Created:
    Purpose: Enter numbers into a list and display
"""


def main():
    # Create a list that holds several numbers
    numbers = [0, 4, 5, 89, 90]

    # Access items by using the index number
    print(f"First number: {numbers[0]}")
    print(f"Third number: {numbers[2]}")

    # Loop through the list one item value at a time
    for number in numbers:
        print(number)

    # Loop through the list one index at a time
    # len gives the length of the list
    for i in range(len(numbers)):
        # Get input from user, place in the list
        print(f"Number {i + 1}: {numbers[i]}")


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

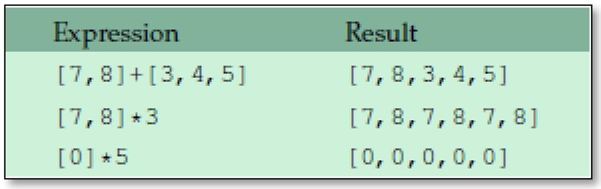
Example run:



```
First number: 0
Third number: 5
0
4
5
89
90
Number 1: 0
Number 2: 4
Number 3: 5
Number 4: 89
Number 5: 90
```

List Operations

Many list operations work the same as string operations.



Expression	Result
`[7,8]+[3,4,5]`	`[7,8,3,4,5]`
`[7,8]*3`	`[7,8,7,8,7,8]`
`[0]*5`	`[0,0,0,0,0]`

The + operator concatenates lists:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
[1, 2, 3, 4, 5, 6]
```

The ***** operator repeats a list a given number of times:

```
alist = [0]
alist = alist * 4
print(alist)
[0, 0, 0, 0]
blist = [1, 2, 3]
blist = blist * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats four times. The second example repeats the list three times.

List Functions

There are many built-in functions that operate on lists without writing your own loops.

Function	Description
`len()`	Returns the number of items in the list
`sum()`	Returns the sum of the items in the list
`min()`	Returns the minimum of the items in the list
`max()`	Returns the maximum of the items in the list
`sorted()`	Returns a sorted list

Examples

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
>>> print(sorted(nums))
[3, 9, 12, 15, 41, 74]
```

The **sum()** function only works when the list elements are numbers. The other functions (**max()**, **len()**, etc.) work with lists of strings and other types that can be comparable.

This is a program that computes the average of a list of numbers entered by the user using a list.

A program to compute an average without a list:

```
total = 0
count = 0
while (True):
    inp = input("Enter a number: ")
    if inp == "done":
        break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print(f"Average: {average}”)
```

In this program, we have count and total variables to keep the number and running total of the user’s numbers as we repeatedly prompt the user for a number.

We could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

```
# Filename: average_numbers_with_list.py
# Create empty list for accumulation
num_list = list()
while (True):
    num = input("Enter a number (Enter to quit): ")
    if num == "":
        break
    # Cast input value to float
    value = float(num)
    # Append value to list
    num_list.append(value)
average = sum(num_list) / len(num_list)
print(f"Average: {average}")
```

Replit [average_numbers_with_list.py]()

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

If in

in is a Boolean operator that checks for membership in a sequence. A sequence is a string, list, tuple, anything that is indexed.

This example uses a string.

```
# Filename: if_in.py
words = "Norwegian Wood"
letter = input("Enter a character: ")
if letter in words:
    print(f"{letter} is in {words}")
else:
    print("I don't need that letter")
```

If not in

if not in is the opposite of if in.

```
# Filename: if_not_in.py
activity = input("What game would you like to play? ")
# .casefold() is a way to convert a string to lower case
# for comparison
if "golf" or "baseball" not in activity.casefold().casefold():
    print("But I want to play golf or baseball")
```

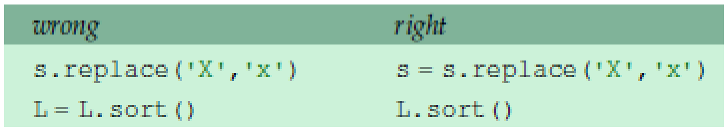
List Methods

Python provides methods that operate on lists. Methods use the **.** (dot) operator.

Method	Description
append(x)	add **x** to the end of the list
sort(x)	sort the list
count(x)	return the number of times **x** occurs in the list
index(x)	return the location of the first occurrence of **x**
reverse()	reverse the list
clear()	remove all the elements from the list

`remove(x)`	remove the first occurrence of **x** from the list
`pop(p)`	remove the item at index **p** and return its value
`insert(p, x)`	insert **x** at index **p** of the list

Important note: There is a big difference between list methods and string methods: String methods do not change the original string, but list methods do change the original list. To sort a list L, use `L.sort()` and not `L = L.sort()`.



wrong	*right*
`s.replace('X','x')`	`s = s.replace('X','x')`
`L = L.sort()`	`L.sort()`

append()

Append adds a new element to the end of a list:

```
t = ["a", "b", "c"]
t.append('d')
print(t)

['a', 'b', 'c', 'd']
```

insert()

Insert places a new element at the designate index and moves the others aside.

```
t = ["a", "b", "c"]
t.insert(1, "d")
print(t)
['a', 'd', 'b', 'c']
```

sort()

Arranges the elements of the list from low to high:

```
t = ["d", "c", "e", "b", "a"]
t.sort()
print(t)
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

reverse()

The reverse() reverses the original order of the list. It doesn't sort backward alphabetically, it reverses the current order of the list,

```
t = ["d", "c", "e", "b", "a"]
t.reverse()
print(t)
t.sort()
print(t)
['d', 'c', 'e', 'b', 'a']
['a', 'b', 'c', 'd', 'e']
```

index()

Returns the index of the first occurrence on the list of the element that is given to index() as argument. A runtime error is generated if the element is not found on the list.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
print(fruitlist.index("banana"))
1
```

count()

Returns an integer that indicates how often the element that is passed to it as an argument occurs in the list.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
print(fruitlist.count("banana"))
2
```

remove()

Removes an item with a certain value.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
fruitlist.remove("banana")
print(fruitlist)
['apple', 'cherry', 'banana', 'durian']
```

del

del is a function, not a method. **del** removes an item at an index.

```
t = ["a", "b", "c"]
del t[1]
print(t)
['a', 'c']
```

pop()

pop removes an item from a list at a designated index and returns a value. You can assign that value to a variable. If you don’t provide an index, it deletes and returns the last element.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
popfruit = fruitlist.pop(1)
print(fruitlist)
print(popfruit)
['apple', 'cherry', 'banana', 'durian']
Banana
```

extend()

Takes a list as an argument and appends all the elements. This example leaves t2 unmodified.

```
t1 = ["a", "b", "c"]
t2 = ["d", "e"]
t1.extend(t2)
print(t1)
['a', 'b', 'c', 'd', 'e']
```

join()

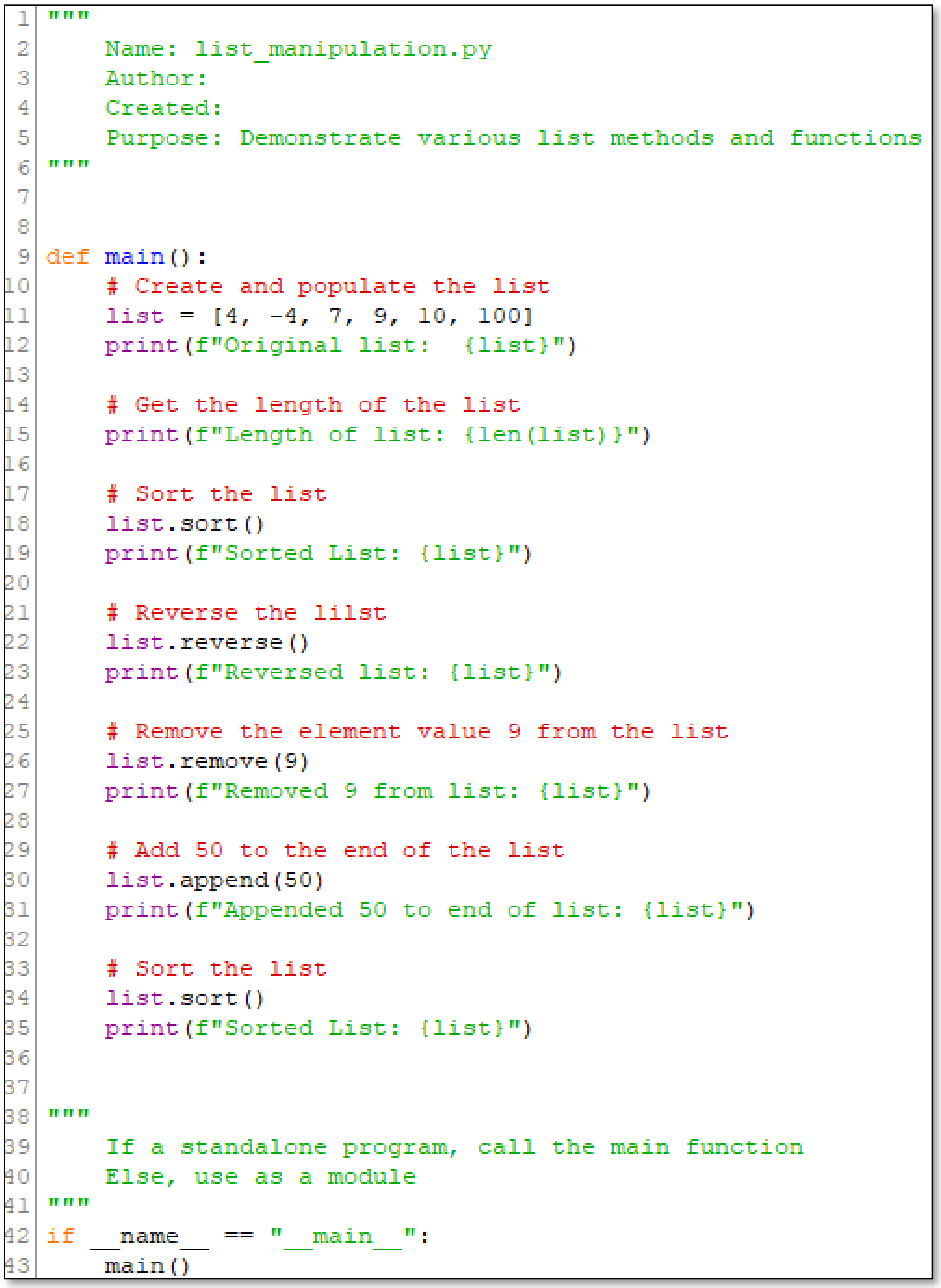
.join() is not a list method, it is a string method. It does work very well with lists. .join() iterates through the list joining each string with the separator.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
separator = " | "
output = separator.join(fruitlist)
print(output)
apple | banana | cherry | banana | durian
```

Tutorial 6.2: Working with Lists

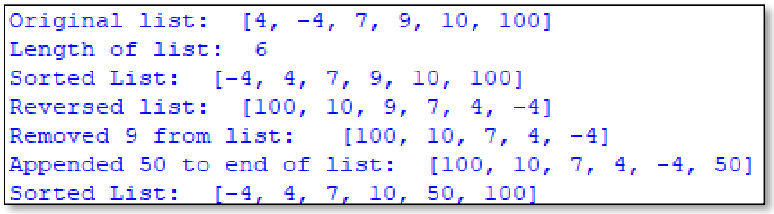
The following program manipulates a list using various methods and functions.

Name the file: **list_manipulation.py**



```python
"""
    Name: list_manipulation.py
    Author:
    Created:
    Purpose: Demonstrate various list methods and functions
"""


def main():
    # Create and populate the list
    list = [4, -4, 7, 9, 10, 100]
    print(f"Original list:  {list}")

    # Get the length of the list
    print(f"Length of list: {len(list)}")

    # Sort the list
    list.sort()
    print(f"Sorted List: {list}")

    # Reverse the lilst
    list.reverse()
    print(f"Reversed list: {list}")

    # Remove the element value 9 from the list
    list.remove(9)
    print(f"Removed 9 from list: {list}")

    # Add 50 to the end of the list
    list.append(50)
    print(f"Appended 50 to end of list: {list}")

    # Sort the list
    list.sort()
    print(f"Sorted List: {list}")


"""
    If a standalone program, call the main function
    Else, use as a module
"""
if __name__ == "__main__":
    main()
```

Example run:



```
Original list:  [4, -4, 7, 9, 10, 100]
Length of list:  6
Sorted List:  [-4, 4, 7, 9, 10, 100]
Reversed list:  [100, 10, 9, 7, 4, -4]
Removed 9 from list:   [100, 10, 7, 4, -4]
Appended 50 to end of list:  [100, 10, 7, 4, -4, 50]
Sorted List:  [-4, 4, 7, 10, 50, 100]
```

Slice

The **slice** operator works on lists. The start is the index of the first element of the slice. The end is the index denoting the end of the slice. The end is not inclusive.

`list_name[start : end]`

```
t = ["a", "b", "c", "d", "e", "f"]
print(t[1:3])
['b', 'c']
print(t[:4])
['a', 'b', 'c', 'd']
print(t[3:])
['d', 'e', 'f']
```

`t[1:3]` slices the index 1 and 2 items.

`t[:4]` slices everything from the beginning of the list to index 3.

`t[3:]` slices everything after and including index 3.

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. If you omit both, the slice is a copy of the whole list.

```
print(t[:])
['a', 'b', 'c', 'd', 'e', 'f']
```

As lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
t = ["a", "b", "c", "d", "e", "f"]
t[1:3] = ['x', 'y']
print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

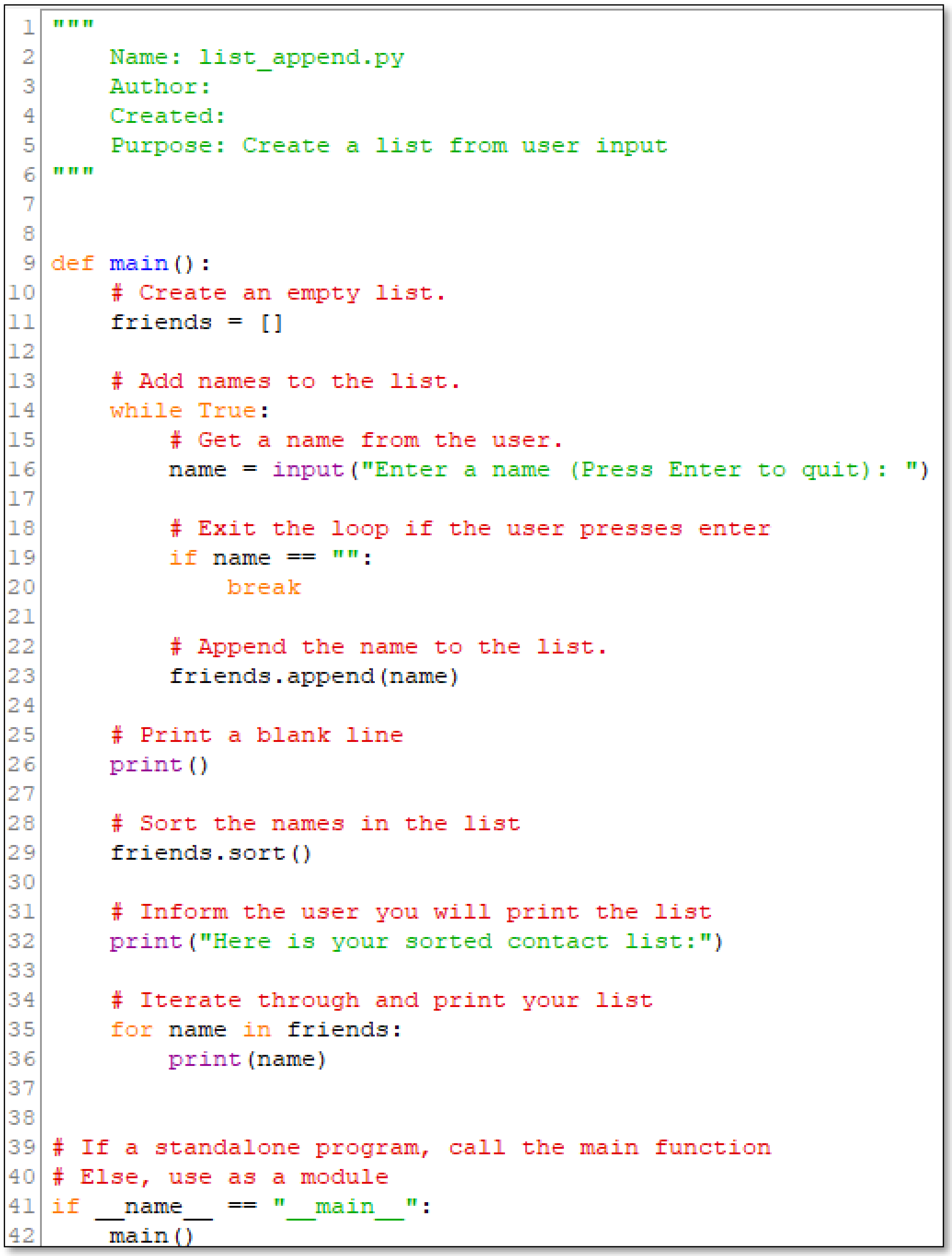
To remove more than one element, you can use **del** with a **slice** index:

```
t = ["a", "b", "c", "d", "e", "f"]
del t[1:5]
print(t)
['a', 'f']
```

slice selects all the elements up to, but not including the second index.

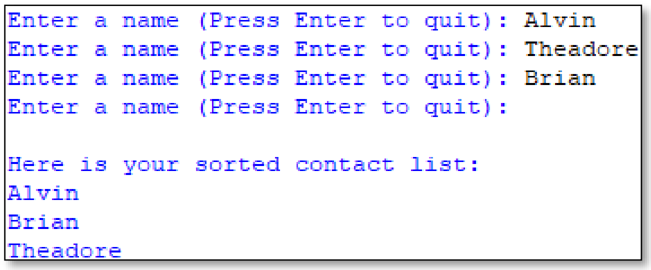
Tutorial 6.3: Input a Contact List

Let's use the append and sort functions to add items to a list. Create a program named **list_append.py**



```python
"""
    Name: list_append.py
    Author:
    Created:
    Purpose: Create a list from user input
"""


def main():
    # Create an empty list.
    friends = []

    # Add names to the list.
    while True:
        # Get a name from the user.
        name = input("Enter a name (Press Enter to quit): ")

        # Exit the loop if the user presses enter
        if name == "":
            break

        # Append the name to the list.
        friends.append(name)

    # Print a blank line
    print()

    # Sort the names in the list
    friends.sort()

    # Inform the user you will print the list
    print("Here is your sorted contact list:")

    # Iterate through and print your list
    for name in friends:
        print(name)


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:

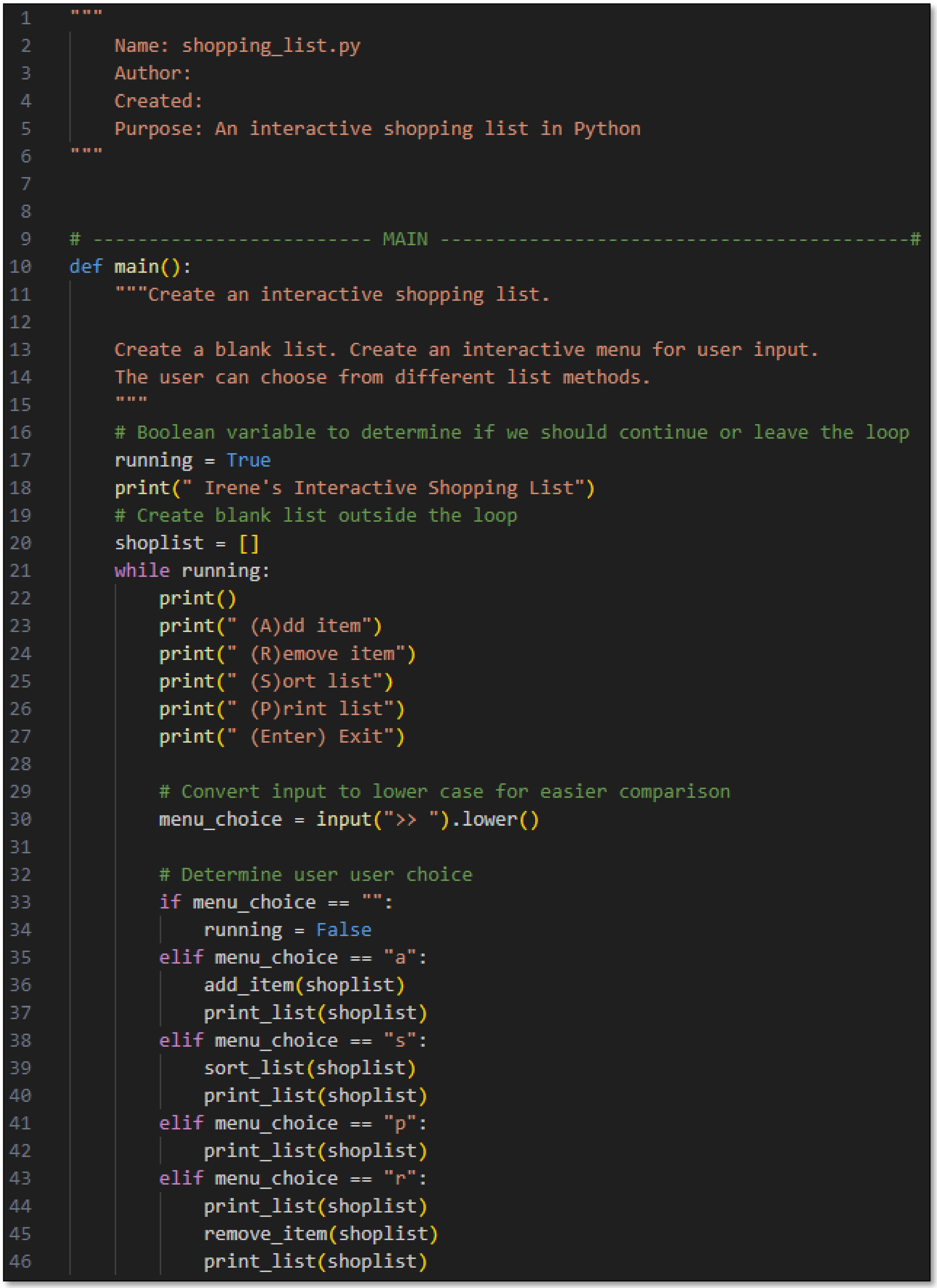


```
Enter a name (Press Enter to quit): Alvin
Enter a name (Press Enter to quit): Theadore
Enter a name (Press Enter to quit): Brian
Enter a name (Press Enter to quit):

Here is your sorted contact list:
Alvin
Brian
Theadore
```

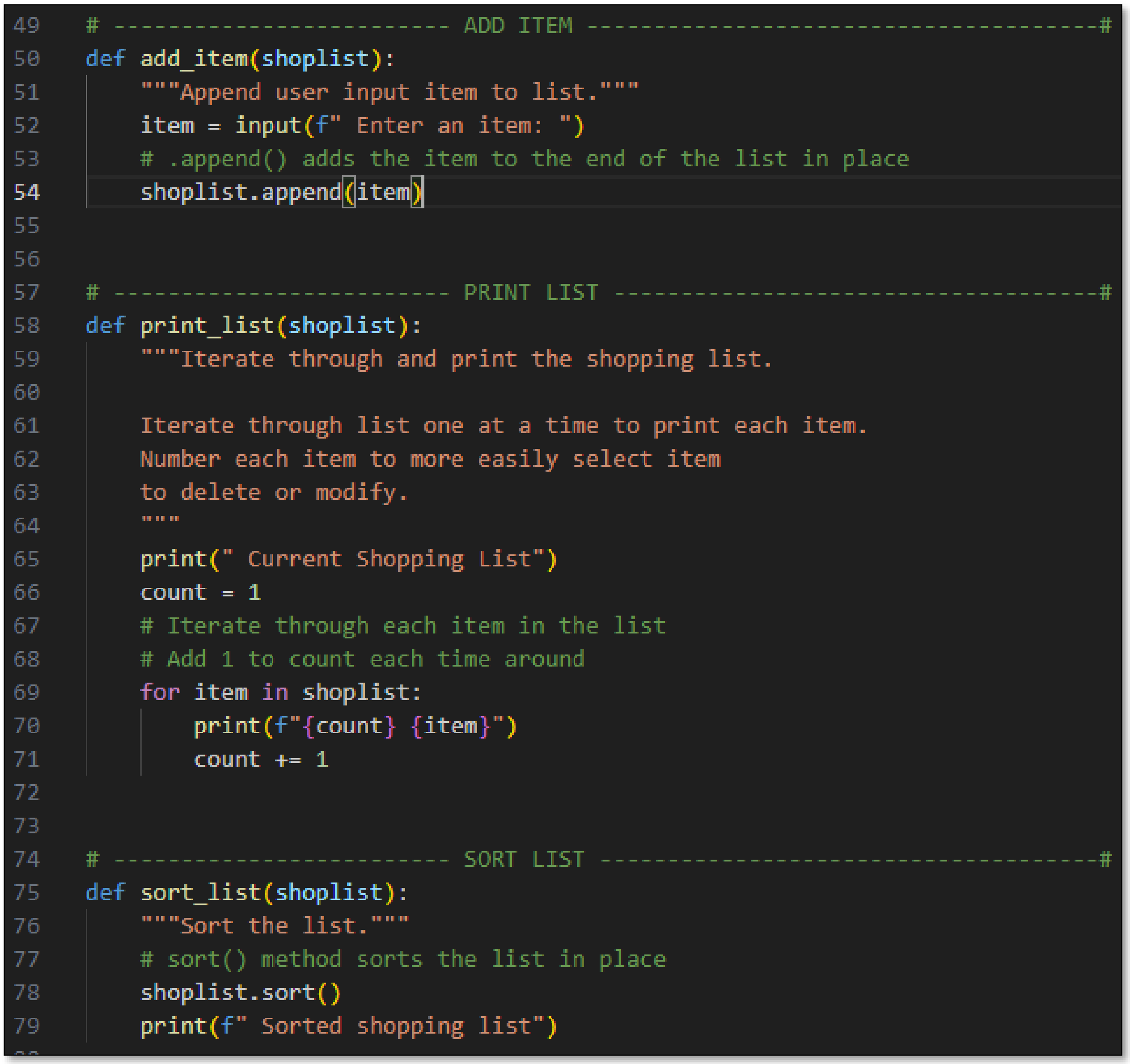
Tutorial 6.4: Irene’s Interactive Shopping List

The following program demonstrates how to create and manipulate a list with user input. This list also demonstrates how to Write Python Beautiful Code with PEP8.

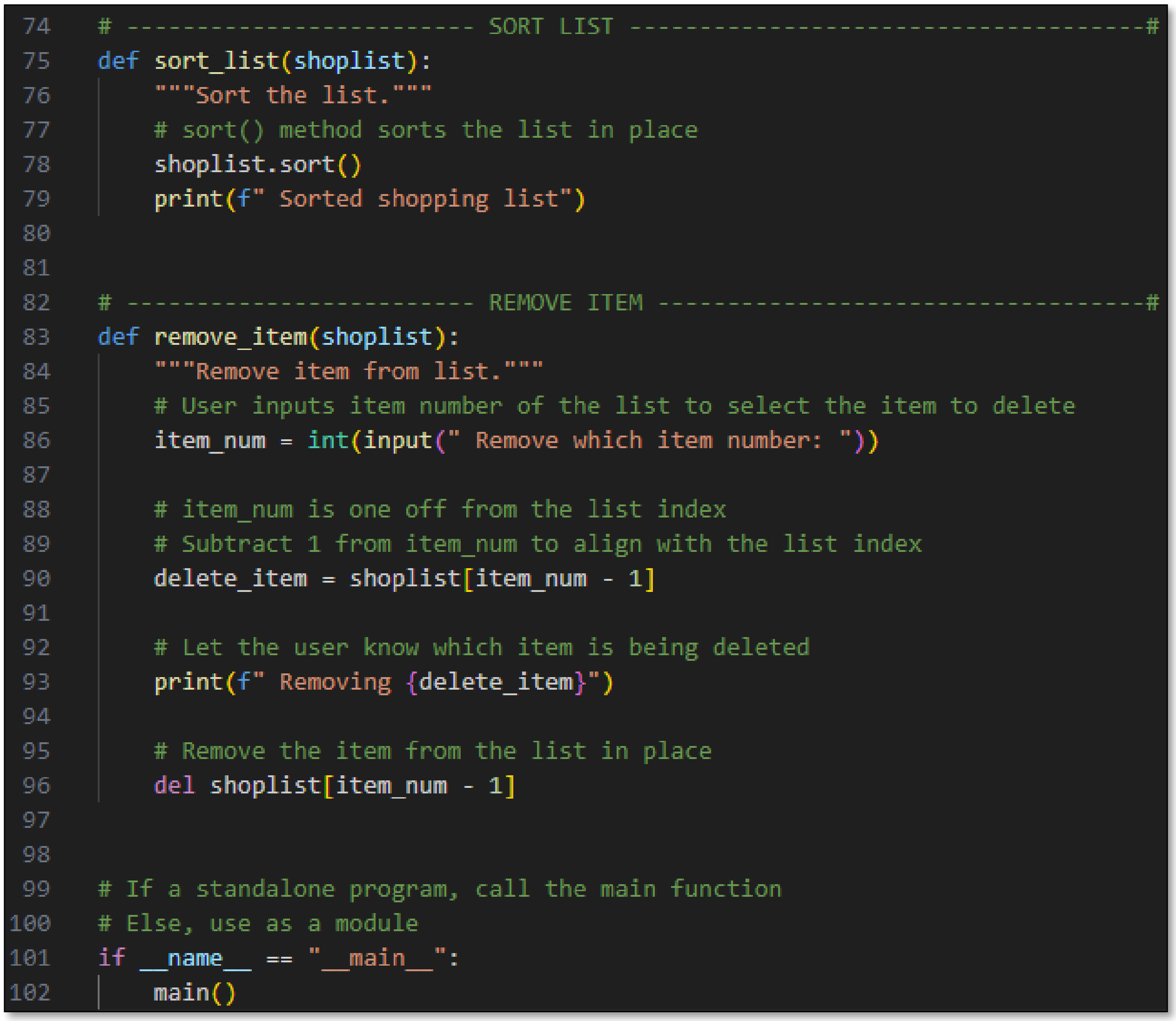
Create a Python program called **shopping_list.py**.



```python
"""
    Name: shopping_list.py
    Author:
    Created:
    Purpose: An interactive shopping list in Python
"""


# ------------------------- MAIN -------------------------------------------#
def main():
    """Create an interactive shopping list.

    Create a blank list. Create an interactive menu for user input.
    The user can choose from different list methods.
    """
    # Boolean variable to determine if we should continue or leave the loop
    running = True
    print(" Irene's Interactive Shopping List")
    # Create blank list outside the loop
    shoplist = []
    while running:
        print()
        print(" (A)dd item")
        print(" (R)emove item")
        print(" (S)ort list")
        print(" (P)rint list")
        print(" (Enter) Exit")

        # Convert input to lower case for easier comparison
        menu_choice = input(">> ").lower()

        # Determine user user choice
        if menu_choice == "":
            running = False
        elif menu_choice == "a":
            add_item(shoplist)
            print_list(shoplist)
        elif menu_choice == "s":
            sort_list(shoplist)
            print_list(shoplist)
        elif menu_choice == "p":
            print_list(shoplist)
        elif menu_choice == "r":
            print_list(shoplist)
            remove_item(shoplist)
            print_list(shoplist)
```

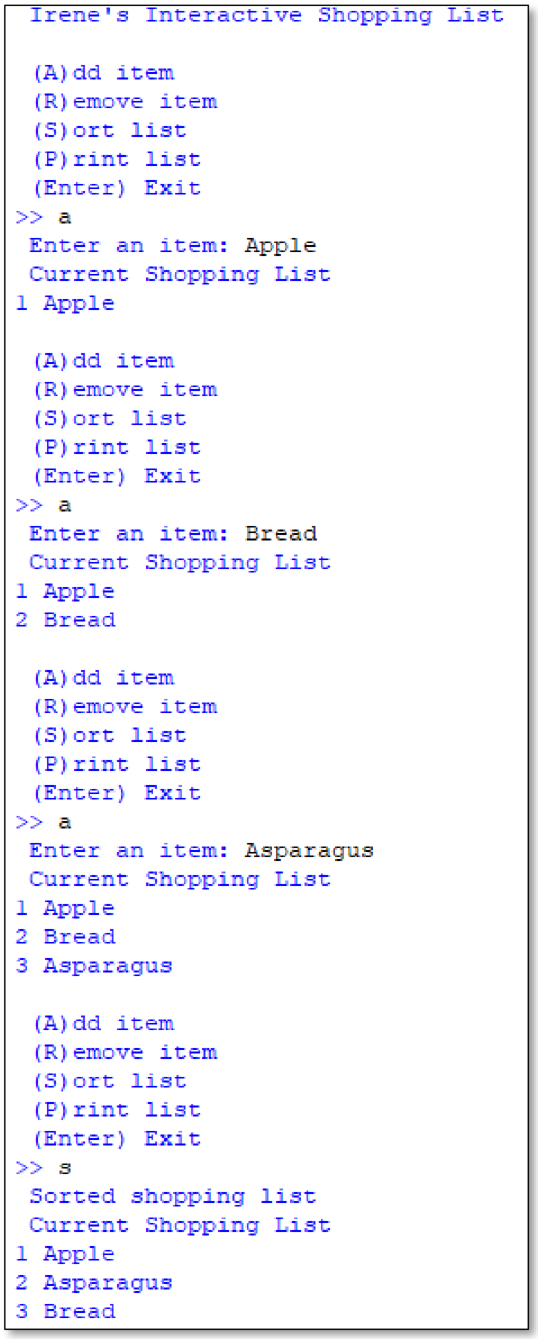


```python
# ------------------------ ADD ITEM -------------------------------------#
def add_item(shoplist):
    """Append user input item to list."""
    item = input(f" Enter an item: ")
    # .append() adds the item to the end of the list in place
    shoplist.append(item)


# ------------------------ PRINT LIST -----------------------------------#
def print_list(shoplist):
    """Iterate through and print the shopping list.

    Iterate through list one at a time to print each item.
    Number each item to more easily select item
    to delete or modify.
    """
    print(" Current Shopping List")
    count = 1
    # Iterate through each item in the list
    # Add 1 to count each time around
    for item in shoplist:
        print(f"{count} {item}")
        count += 1


# ------------------------ SORT LIST ------------------------------------#
def sort_list(shoplist):
    """Sort the list."""
    # sort() method sorts the list in place
    shoplist.sort()
    print(f" Sorted shopping list")
```

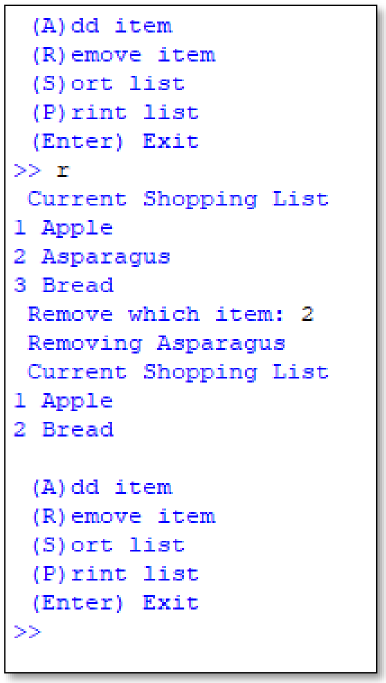


```python
# ------------------------- SORT LIST ------------------------------------#
def sort_list(shoplist):
    """Sort the list."""
    # sort() method sorts the list in place
    shoplist.sort()
    print(f" Sorted shopping list")


# ------------------------- REMOVE ITEM ----------------------------------#
def remove_item(shoplist):
    """Remove item from list."""
    # User inputs item number of the list to select the item to delete
    item_num = int(input(" Remove which item number: "))

    # item_num is one off from the list index
    # Subtract 1 from item_num to align with the list index
    delete_item = shoplist[item_num - 1]

    # Let the user know which item is being deleted
    print(f" Removing {delete_item}")

    # Remove the item from the list in place
    del shoplist[item_num - 1]


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



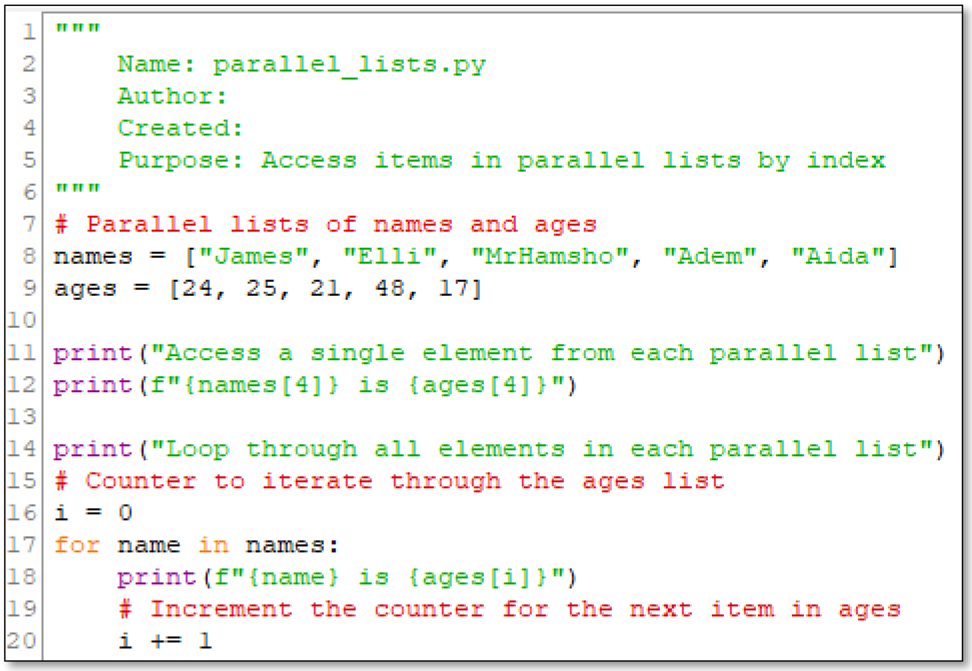
```
 Irene's Interactive Shopping List

  (A)dd item
  (R)emove item
  (S)ort list
  (P)rint list
  (Enter) Exit
>> a
 Enter an item: Apple
 Current Shopping List
1 Apple

  (A)dd item
  (R)emove item
  (S)ort list
  (P)rint list
  (Enter) Exit
>> a
 Enter an item: Bread
 Current Shopping List
1 Apple
2 Bread

  (A)dd item
  (R)emove item
  (S)ort list
  (P)rint list
  (Enter) Exit
>> a
 Enter an item: Asparagus
 Current Shopping List
1 Apple
2 Bread
3 Asparagus

  (A)dd item
  (R)emove item
  (S)ort list
  (P)rint list
  (Enter) Exit
>> s
 Sorted shopping list
 Current Shopping List
1 Apple
2 Asparagus
3 Bread
```



```
  (A)dd item
  (R)emove item
  (S)ort list
  (P)rint list
  (Enter) Exit
>> r
 Current Shopping List
1 Apple
2 Asparagus
3 Bread
 Remove which item: 2
 Removing Asparagus
 Current Shopping List
1 Apple
2 Bread

  (A)dd item
  (R)emove item
  (S)ort list
  (P)rint list
  (Enter) Exit
>>
```

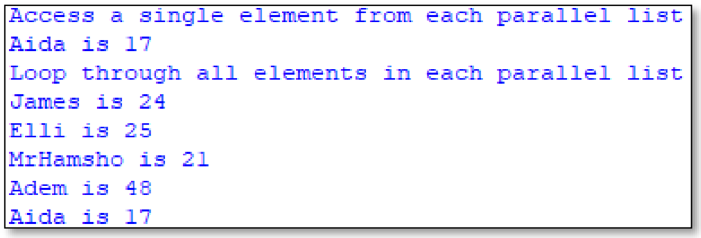
Tutorial 6.5: Parallel Lists

A way to reduce the number of related variables is to use parallel lists. You can have multiple parallel lists. This example uses two.



```python
"""
    Name: parallel_lists.py
    Author:
    Created:
    Purpose: Access items in parallel lists by index
"""
# Parallel lists of names and ages
names = ["James", "Elli", "MrHamsho", "Adem", "Aida"]
ages = [24, 25, 21, 48, 17]

print("Access a single element from each parallel list")
print(f"{names[4]} is {ages[4]}")

print("Loop through all elements in each parallel list")
# Counter to iterate through the ages list
i = 0
for name in names:
    print(f"{name} is {ages[i]}")
    # Increment the counter for the next item in ages
    i += 1
```

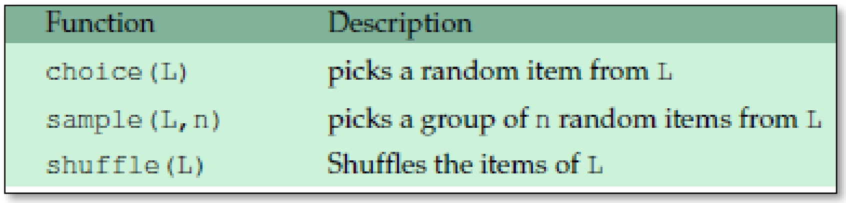
Example run:



```
Access a single element from each parallel list
Aida is 17
Loop through all elements in each parallel list
James is 24
Elli is 25
MrHamsho is 21
Adem is 48
Aida is 17
```

Lists and the Random Module

There are some nice functions in the random module that work on lists.



Function	Description
choice(L)	picks a random item from L
sample(L,n)	picks a group of n random items from L
shuffle(L)	Shuffles the items of L

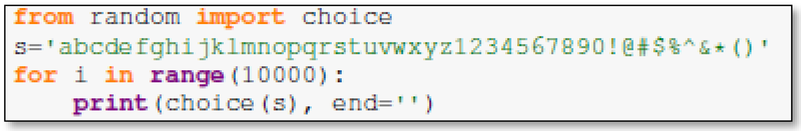
Note: The shuffle function modifies the original list. If you don't want your list changed, you'll need to make a copy of it.

Choice

This can be used to pick a name from a list of names.

```
from random import choice
names = ["Joe", "Bob", "Sue", "Sally"]
current_player = choice(names)
print(current_player)
```

This function also works with strings, picking a random character from a string. Here is an example that uses choice to fill the screen with a bunch of random characters.



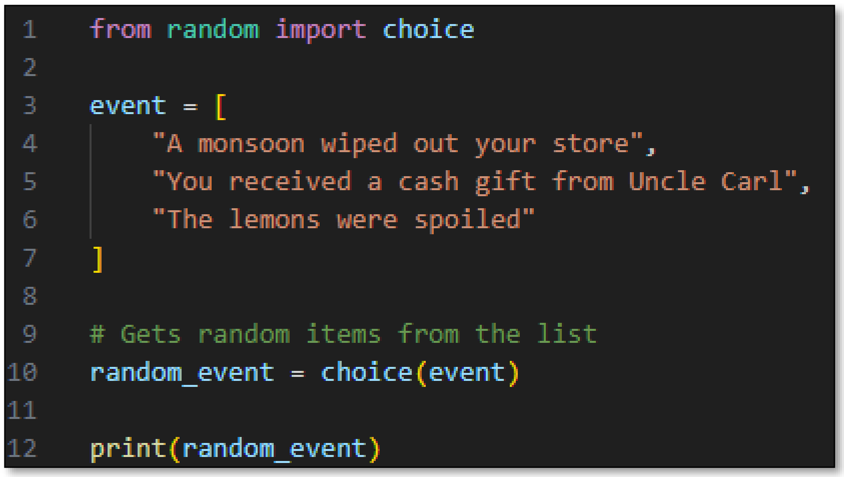
```
from random import choice
s='abcdefghijklmnopqrstuvwxyz1234567890!@#$%^&*()'
for i in range(10000):
    print(choice(s), end='')
```

Choice can also pick random numbers out of a list.

```
# Import the choice function from random
from random import choice
numbers = [1, 2, 3, 4]
random_choice = choice(numbers)
print(random_choice)
```

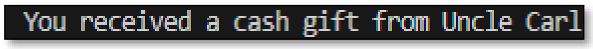
Tutorial 6.7: Random Events

This tutorial chooses a random event from a list. The advantage of using a list is that it is easy to add, remove, or modify an event without changing the code.



```python
from random import choice

event = [
    "A monsoon wiped out your store",
    "You received a cash gift from Uncle Carl",
    "The lemons were spoiled"
]

# Gets random items from the list
random_event = choice(event)

print(random_event)
```

Example run:



```
You received a cash gift from Uncle Carl
```

Choices

Choices allows picking more than one element from a list. The second argument k = 2, sets how many random elements to return.

```
# Import the choice function from random
from random import choices
numbers = [1, 2, 3, 4]
random_choices = choices(numbers, k = 2)
print(random_choices)
```

Sample

This function is similar to choice. Choice picks one item from a list, sample can be used to pick several.

```
from random import sample
names = ["Joe", "Bob", "Sue", "Sally"]
team = sample(names, 2)
```

Shuffle

Shuffle picks a random ordering of players in a game. The shuffle function modifies the original list. If you don’t want your list changed, you’ll need to make a copy of it.

Here is a nice use of shuffle to pick a random ordering of players in a game.

```
from random import shuffle
players = ["Joe", "Bob", "Sue", "Sally"]
shuffle(players)
for p in players:
    print(p, 'it is your turn.')
    # code to play the game goes here
```

Here we use shuffle divide a group of people into teams of two.

```
from random import shuffle
names = ["Joe", "Bob", "Sue", "Sally"]
shuffle(names)
teams = []
for i in range(0, len(names),2):
   teams.append([names[i], names[i+1]])
```

Each element in the teams list is a list of two names.

1. Shuffle the names list into a random order.
2. The first two names in the shuffled list become the first team, the next two names become the second team, etc.

Notice that we use the optional third argument to range to skip ahead by two through the list of names.

List Arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change.

For example, **delete_head** removes the first element from a list:

```
def delete_head(t):
   del t[0]
```

Here’s how it is used:

```
def delete_head(t):
   del t[0]
letters = ["a", "b", "c"]
delete_head(letters)
print(letters)
['b', 'c']
```

The parameter **t** and the variable **letters** are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists.

For example, the append method modifies a list, but the + operator creates a new list:

```
t1 = [1, 2]
t2 = t1.append(3)

print(t1)
[1, 2, 3]

print(t2)
None
t3 = t1 + [3]

print(t3)
[1, 2, 3]

print(t2 is t3)
False
```

This difference is important when you write functions that are supposed to modify lists.

For example, tail returns all but the first element of a list:

```
def tail(t):
  return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
def tail(t):
  return t[1:]
letters = ["a", "b", "c"]
rest = tail(letters)

print(rest)
['b', 'c']
```

Tuples

A tuple is like a list whose values cannot be modified. It is an ordered list of objects, and it can contain references to any type of object. A Python tuple is called an array in other programming languages.

- Tuples are normally written as a sequence of items contained in (optional) matching parentheses.
- A tuple is an immutable sequence.
- Items in a tuple are accessed using a numeric index.
- Tuples can be nested.

A tuple is like a list whose values cannot be modified. In other words, a tuple is immutable.

Tuples provide some degree of integrity to the data stored in them. You can pass a tuple around through a program and be confident that its value can't be accidentally changed.

Working with Tuples

A tuple is a comma-separated list of values:

```
t = ("a", "b", "c", "d", "e")
```

Most list operators also work on tuples. The ones that modify the list itself will not work.

You can’t modify the elements of a tuple, but you can replace one tuple with another:

```
t = ("a", "b", "c", "d", "e")
t = ("A",) + t[1:]
print(t)
('A', 'b', 'c', 'd', 'e')
```

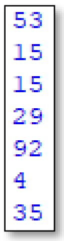
List Examples

Example 1: Random List

Write a program that generates a list named **random_list** of 50 random numbers between 1 and 100.

```
from random import randint
# Create an empty list
random_list = []
# Generate and add random integers to the list
for i in range(50):
    randum_number = randint(1, 100)
    random_list.append(random_number)
    print(random_list[i])
```

Example run (Only a partial list is shown):



```
53
15
15
29
92
4
35
```

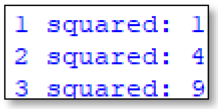
We used the append method to build up the list one item at a time starting with the empty list, [].

Example 2: Squared List

Replace each element in a list **square_list** with its square.

```
square_list = [1, 2, 3]
for i in range(len(square_list)):
    # Use end="" to keep everything on the same line
    print(square_list[i], end="")
    square_list[i] = square_list[i]**2
    print(f" squared: {square_list[i]})
```

Example run:



```
1 squared: 1
2 squared: 4
3 squared: 9
```

Example 3: Count Items

Count how many items in a list L are greater than 50.

```
count = 0
for item in L:
   if item > 50:
      count = count + 1
```

Example 4: Count Each Item

Given a list L that contains numbers between 1 and 10, create a new list whose first element is how many ones are in L, whose second element is how many twos are in L, etc.

```
L = [1, 2, 3, 1, 5, 5, 9, 9]
frequencies = []
for i in range(1, 10):
    frequencies.append(L.count(i))
    print(frequencies[i - 1])
```

The key is the list method count that tells how many times a something occurs in a list.

Example 5: Largest and Smallest

Write a program that prints out the two largest and two smallest elements of a list called scores.

```
scores = [24, 45, 1, 6]
scores.sort()
print(f"Two smallest: {scores[0]} {scores[1]}")
print(f"Two largest: {scores[-1]} {scores[-2]} ")
```

Once we sort the list, the smallest values are at the beginning and the largest are at the end.

Example 6: Quiz Game

Here is a program to play a simple quiz game.

```
questions = ["What is the capital of France? ",
"Which state has only one neighbor? "]
answers = ["Paris", "Maine"]
num_right = 0
for i in range(len(questions)):
    guess = input(questions[i])
    if guess.lower()==answers[i].lower():
        print("Correct")
        num_right=num_right+1
    else:
        print(f"Wrong. The answer is {answers[i]} ")
    print(f"You have {num_right} out of {I + 1} right. ")
```

This illustrates the general technique: If you find yourself repeating the same code over and over, try lists and a for loop. The few parts of your repetitious code that are varying are where the list code will go.

The benefits of this are that to change a question, add a question, or change the order, only the questions and answers lists need to be changed. If you want to make a change to the program, like not telling the user the correct answer, you can modify a single line, instead of twenty copies of that line spread throughout the program.

Glossary

aliasing A circumstance where two or more variables refer to the same object.

delimiter A character or string used to indicate where a string should be split.

element One of the values in a list (or other sequence); also called items.

equivalent Having the same value.

index An integer value that indicates an element in a list.

identical Being the same object (which implies equivalence).

list A sequence of values.

list traversal Sequential accessing of each element in a list.

nested list A list that is an element of another list.

object Something a variable can refer to. An object has a type and a value.

reference The association between a variable and its value.

Assignment Submission

1. Attach the pseudocode or create a TODO.
2. Attach all tutorials and assignments.
3. Attach screenshots showing the successful operation of each tutorial program.
4. Submit in Blackboard.