Object Oriented Programming II

Dr. Burak Kaleci

April 14, 2019

Content

Introduction

- A sequence is an object that contains multiple items of data.
- The items that are in a sequence are stored one after the other.
- Python provides various ways to perform operations on the items that are stored in a sequence.
- There are several different types of sequence objects in Python.
- We will look at two of the fundamental sequence types: lists and tuples.
- Both lists and tuples are sequences that can hold various types of data. The difference between lists and tuples is simple: a list is mutable, which means that a program can change its contents, but a tuple is immutable, which means that once it is created, its contents cannot be changed.

Introduction to Lists

- A list is an object that contains multiple data items.
- Each item that is stored in a list is called an element.
- Here is a statement that creates a list of integers:

  even_numbers = [2, 4, 6, 8, 10]
- The items that are enclosed in brackets and separated by commas are the list elements.
- After this statement executes, the variable even_numbers will reference the list, as shown in Figure

Introduction to Lists

- The following is another example:

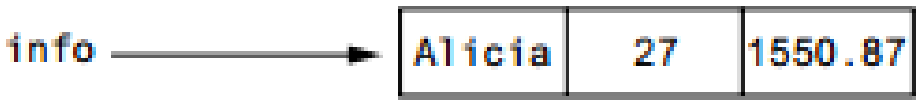
  names = ['Molly', 'Steven', 'Will', 'Alicia', 'Adriana']

- This statement creates a list of five strings. After the statement executes, the names variable will reference the list as shown in Figure.

- A list can hold items of different types, as shown in the following example:

  info = ['Alicia', 27, 1550.87]

- This statement creates a list containing a string, an integer, and a floating-point number. After the statement executes, the info variable will reference the list as shown in Figure



Introduction to Lists

- You can use the print function to display an entire list, as shown here:

```
numbers = [5, 10, 15, 20]
print(numbers)
```

- Python also has a built-in list()function that can convert certain types of objects to lists.
- You can use a statement such as the following to convert the range function's iterable object to a list:

```
numbers = list(range(5))
```

Introduction to Lists

- When this statement executes, the following things happen:
  - The range function is called with 5 passed as an argument. The function returns an iterable containing the values 0, 1, 2, 3, 4.
  - The iterable is passed as an argument to the list()function. The list()function returns the list [0, 1, 2, 3, 4].
  - The list [0, 1, 2, 3, 4]is assigned to the numbers variable.
- Here is another example:

  numbers = list(range(1, 10, 2))
- When you pass three arguments to the range function, the first argument is the starting value, the second argument is the ending limit, and the third argument is the step value.
- This statement will assign the list [1, 3, 5, 7, 9] to the numbers variable.

The Repetition Operator

- The * symbol multiplies two numbers. However, when the operand on the left side of the * symbol is a sequence (such as a list) and the operand on the right side is an integer, it becomes the repetition operator.
- The repetition operator makes multiple copies of a list and joins them all together.
- Here is the general format:

  list * n
- In the general format, listis a list, and nis the number of copies to make.

```
1  >>> numbers = [0] * 5 Enter
2  >>> print(numbers) Enter
3  [0, 0, 0, 0, 0]
4  >>>
```

Introduction to Lists

- Let's take a closer look at each statement:
  - In line 1, the expression [0] * 5 makes five copies of the list [0] and joins them all together in a single list.
  - The resulting list is assigned to the numbers variable.
  - In line 2, the numbers variable is passed to the print function. The function's output is shown in line 3.
- Here is another interactive mode demonstration:

```
1  >>> numbers = [1, 2, 3] * 3 Enter
2  >>> print(numbers) Enter
3  [1, 2, 3, 1, 2, 3, 1, 2, 3]
4  >>>
```

Loops and Indexing

- You can iterate over a list with the for loop, as shown here:

```
numbers = [99, 100, 101, 102]
for n in numbers:
    print(n)
```

- Another way that you can access the individual elements in a list is with an index.
- Each element in a list has an index that specifies its position in the list.
- Indexing starts at 0, so the index of the first element is 0, the index of the second element is 1, and so forth. The index of the last element in a list is 1 less than the number of elements in the list.
- For example, the following statement creates a list with 4 elements:

```
my_list = [10, 20, 30, 40]
```

- The indexes of the elements in this list are 0, 1, 2, and 3. We can print the elements of the list with the following statement:

```
print(my_list[0], my_list[1], my_list[2], my_list[3])
```

Loops and Indexing

- The following loop also prints the elements of the list:

```
index = 0
while index < 4:
    print(my_list[index])
    index += 1
```

- You can also use negative indexes with lists to identify element positions relative to the end of the list.
- The Python interpreter adds negative indexes to the length of the list to determine the element position.
- The index -1 identifies the last element in a list, -2 identifies the next to last element, and so forth. The following code shows an example:

```
my_list = [10, 20, 30, 40]
print(my_list[-1], my_list[-2], my_list[-3], my_list[-4])
```

The len Function

- Python has a built-in function named len that returns the length of a sequence, such as a list.
- The following code demonstrates:

```
my_list = [10, 20, 30, 40]
size = len(my_list)
```

- The first statement assigns the list [10, 20, 30, 40] to the my_list variable. The second statement calls the len function, passing the my_list variable as an argument.
- The function returns the value 4, which is the number of elements in the list. This value is assigned to the size variable.
- The len function can be used to prevent an IndexError exception when iterating over a list with a loop. Here is an example:

```
my_list = [10, 20, 30, 40]
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1
```

Lists Are Mutable

- Lists in Python are mutable, which means their elements can be changed.
- Consequently, an expression in the form list[index] can appear on the left side of an assignment operator.
- The following code shows an example:

```
numbers = [1, 2, 3, 4, 5]
print(numbers)
numbers[0] = 99
print(numbers)
```

- The statement in line 3 assigns 99 to numbers[0]. This changes the first value in the list to 99.
- When you use an indexing expression to assign a value to a list element, you must use a valid index for an existing element or an IndexError exception will occur.

Lists Are Mutable

- If you want to use indexing expressions to fill a list with values, you have to create the list first, as shown here:

```
1  # Create a list with 5 elements.
2  numbers = [0] * 5
3
4  # Fill the list with the value 99.
5  index = 0
6  while index < len(numbers):
7      numbers[index] = 99
8      index += 1
```

- The statement in line 2 creates a list with five elements, each element assigned the value 0.
- The loop in lines 6 through 8 then steps through the list elements, assigning 99 to each one.

Lists Are Mutable

```
1  # The NUM_DAYS constant holds the number of
2  # days that we will gather sales data for.
3  NUM_DAYS = 5
4
5  def main():
6  # Create a list to hold the sales
7  # for each day.
8      sales = [0] * NUM_DAYS
9
10     # Create a variable to hold an index.
11     index=0
12
13     print('Enter the sales for each day.')
14
15     # Get the sales for each day.
16     while index < NUM_DAYS:
17         print('Day #', index + 1, ': ', sep='', end='')
18         sales[index] = float(input())
19         index += 1
20
21     # Display the values entered.
22     print('Here are the values you entered:')
23     for value in sales:
24         print(value)
25
26 # Call the main function.
27 main()
```

- The statement in line 3 creates the variable NUM_DAYS, which is used as a constant for the number of days.
- The statement in line 8 creates a list with five elements, with each element assigned the value 0.
- Line 11 creates a variable named index and assigns the value 0 to it.
- The loop in lines 16 through 19 iterates 5 times. The first time it iterates, index references the value 0, so the statement in line 18 assigns the user's input to sales[0].
- The second time the loop iterates, index references the value 1, so the statement in line 18 assigns the user's input to sales[1]. This continues until input values have been assigned to all the elements in the list.

Concatenating Lists

- To concatenate means to join two things together. You can use the + operator to concatenate two lists.
- Here is an example:

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
list3 = list1 + list2
```

- After this code executes, list1 and list2 remain unchanged, and list3 references the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

- The following interactive mode session also demonstrates list concatenation:

```
girl_names = ['Joanne', 'Karen', 'Lori']
boy_names = ['Chris', 'Jerry', 'Will']
all_names = girl_names + boy_names
print(all_names)
['Joanne', 'Karen', 'Lori', 'Chris', 'Jerry', 'Will']
```

Concatenating Lists

- You can also use the += augmented assignment operator to concatenate one list to another.
- Here is an example:

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
list1 += list2
```

- The last statement appends list2 to list1. After this code executes, list2 remains unchanged, but list1 references the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

- The following interactive mode session also demonstrates the += operator used for list concatenation:

```
girl_names = ['Joanne', 'Karen', 'Lori']
girl_names += ['Jenny', 'Kelly']
print(girl_names)
['Joanne', 'Karen', 'Lori', 'Jenny', 'Kelly']
```

List Slicing

- You have seen how indexing allows you to select a specific element in a sequence.
- Sometimes you want to select more than one element from a sequence.
- In Python, you can write expressions that select subsections of a sequence, known as slices.
- A slice is a span of items that are taken from a sequence.
- When you take a slice from a list, you get a span of elements from within the list.
- To get a slice of a list, you write an expression in the following general format:

  list_name[start: end]

- In the general format, start is the index of the first element in the slice, and end is the index marking the end of the slice.
- The expression returns a list containing a copy of the elements from start up to (but not including) end.

List Slicing

- For example, suppose we create the following list:
  days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
- The following statement uses a slicing expression to get the elements from indexes 2 up to, but not including, 5:
  mid_days = days[2:5]
- After this statement executes, the mid_days variable references the following list:
  ['Tuesday', 'Wednesday', 'Thursday']

List Slicing

- Here is a summary of each line:

  In line 1, we created the list and [1, 2, 3, 4, 5]and assigned it to the numbers variable.

  In line 2, we passed numbers as an argument to the print function. The print function displayed the list in line 3.

  In line 4, we sent the slice numbers[1:3] as an argument to the print function. The print function displayed the slice in line 5.

  ```
  numbers = [1, 2, 3, 4, 5]
  print(numbers)
  [1, 2, 3, 4, 5]
  print(numbers[1:3])
  [2, 3]
  ```

List Slicing

- If you leave out the start index in a slicing expression, Python uses 0 as the starting index.

```
numbers = [1, 2, 3, 4, 5]
print(numbers)
[1, 2, 3, 4, 5]
print(numbers[:3])
[1, 2, 3]
```

- Notice line 4 sends the slice numbers[:3]as an argument to the print function. Because the starting index was omitted, the slice contains the elements from index 0 up to 3.
- If you leave out the end index in a slicing expression, Python uses the length of the list as the end index.

```
numbers = [1, 2, 3, 4, 5]
print(numbers)
[1, 2, 3, 4, 5]
print(numbers[2:])
[3, 4, 5]
```

- Notice line 4 sends the slice numbers[2:]as an argument to the print function. Because the ending index was omitted, the slice contains the elements from index 2 through the end of the list.

List Slicing

- If you leave out both the start and end index in a slicing expression, you get a copy of the entire list.

```
numbers = [1, 2, 3, 4, 5]
print(numbers)
[1, 2, 3, 4, 5]
print(numbers[:])
[1, 2, 3, 4, 5]
```

- The slicing examples we have seen so far get slices of consecutive elements from lists. Slicing expressions can also have step value, which can cause elements to be skipped in the list.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[1:8:2])
[2, 4, 6, 8]
```

- In the slicing expression in line 4, the third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every second element from the specified range in the list.
- You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the list. Python adds a negative index to the length of a list to get the position referenced by that index.

Finding Items in Lists with the in Operator

```
1  # This program demonstrates the in operator
2  # used with a list.
3
4  def main():
5      # Create a list of product numbers.
6      prod_nums = ['V475', 'F987', 'Q143', 'R688']
7
8      # Get a product number to search for.
9      search = input('Enter a product number: ')
10
11     # Determine whether the product number is in the list.
12     if search in prod_nums:
13         print(search, 'was found in the list.')
14     else:
15         print(search, 'was not found in the list.')
16
17 # Call the main function.
18 main()
```

Program Output (with input shown in bold)

```
Enter a product number: Q143 Enter
Q143 was found in the list.
```

Program Output (with input shown in bold)

```
Enter a product number: B000 Enter
B000 was not found in the list.
```

- In Python, you can use the in operator to determine whether an item is contained in a list.
- Here is the general format of an expression written with the in operator to search for an item in a list:

  item in list

- In the general format, item is the item for which you are searching, and list is a list.
- The expression returns true if item is found in the list, or false otherwise.

List Methods and Useful Built-in Functions

- Lists have numerous methods that allow you to add elements, remove elements, change the ordering of elements, and so forth.

Method	Description
`append(item)`	Adds *item* to the end of the list.
`index(item)`	Returns the index of the first element whose value is equal to item. A `ValueError` exception is raised if item is not found in the list.
`insert(index, item)`	Inserts *item* into the list at the specified *index*. When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
`sort()`	Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).
`remove(item)`	Removes the first occurrence of *item* from the list. A `ValueError` exception is raised if item is not found in the list.
`reverse()`	Reverses the order of the items in the list.

The append Method

```
1  # This program demonstrates how the append
2  # method can be used to add items to a list.
3
4  def main():
5      # First, create an empty list.
6      name_list = []
7
8      # Create a variable to control the loop
9      again='y'
10
11     # Add some names to the list.
12     while again == 'y':
13         # Get a name from the user.
14         name = input('Enter a name: ')
15
16         # Append the name to the list.
17         name_list.append(name)
18
19         # Add another one?
20         print('Do you want to add another name?')
21         again = input('y = yes, anything else = no: ')
22         print()
23
24     # Display the names that were entered.
25     print('Here are the names you entered.')
26
27     for name in name_list:
28         print(name)
29
30 # Call the main function.
31 main()
```

- The append method is commonly used to add items to a list.
- The item that is passed as an argument is appended to the end of the list's existing elements.
- Line 6 creates an empty list (a list with no elements) and assigns it to the name_list variable.
- Inside the loop, the append method is called to build the list. The first time the method is called, the argument passed to it will become element 0. The second time the method is called, the argument passed to it will become element 1. This continues until the user exits the loop.

The index Method

```
1  # This program demonstrates how to get the
2  # index of an item in a list and then replace
3  # that item with a new item.
4
5  def main():
6      # Create a list with some items.
7      food = ['Pizza', 'Burgers', 'Chips']
8
9      # Display the list.
10     print('Here are the items in the food list:')
11     print(food)
12
13     # Get the item to change.
14     item = input('Which item should I change? ')
15
16     try:
17         # Get the item's index in the list.
18         item_index = food.index(item)
19
20         # Get the value to replace it with.
21         new_item = input('Enter the new value: ')
22
23         # Replace the old item with the new item.
24         food[item_index] = new_item
25
26         # Display the list.
27         print('Here is the revised list:')
28         print(food)
29     except ValueError:
30         print('That item was not found in the list.')
31
32 # Call the main function.
33 main()
```

- Earlier, you saw how the in operator can be used to determine whether an item is in a list.
- Sometimes you need to know not only whether an item is in a list, but where it is located.
- The index method is useful in these cases. You pass an argument to the inde xmethod, and it returns the index of the first element in the list containing that item.
- If the item is not found in the list, the method raises a ValueError exception.

The index Method

```
1  # This program demonstrates how to get the
2  # index of an item in a list and then replace
3  # that item with a new item.
4
5  def main():
6      # Create a list with some items.
7      food = ['Pizza', 'Burgers', 'Chips']
8
9      # Display the list.
10     print('Here are the items in the food list:')
11     print(food)
12
13     # Get the item to change.
14     item = input('Which item should I change? ')
15
16     try:
17         # Get the item's index in the list.
18         item_index = food.index(item)
19
20         # Get the value to replace it with.
21         new_item = input('Enter the new value: ')
22
23         # Replace the old item with the new item.
24         food[item_index] = new_item
25
26         # Display the list.
27         print('Here is the revised list:')
28         print(food)
29     except ValueError:
30         print('That item was not found in the list.')
31
32 # Call the main function.
33 main()
```

- The elements of the foodlist are displayed in line 11, and in line 14, the user is asked which item he or she wants to change.
- Line 18 calls the index method to get the index of the item.
- Line 21 gets the new value from the user, and line 24 assigns the new value to the element holding the old value.

The insert Method

```
1  # This program demonstrates the insert method.
2
3  def main():
4      # Create a list with some names.
5      names = ['James', 'Kathryn', 'Bill']
6
7      # Display the list.
8      print('The list before the insert:')
9      print(names)
10
11     # Insert a new name at element 0.
12     names.insert(0, 'Joe')
13
14     # Display the list again.
15     print('The list after the insert:')
16     print(names)
17
18 # Call the main function.
19 main()
```

- The insert method allows you to insert an item into a list at a specific position.
- You pass two arguments to the insert method: an index specifying where the item should be inserted and the item that you want to insert.

The sort Method

- The sort method rearranges the elements of a list so they appear in ascending order (from the lowest value to the highest value). Here is an example:

```
my_list = [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
print('Original order:', my_list)
my_list.sort()
print('Sorted order:', my_list)
```

- When this code runs, it will display the following:

```
Original order: [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
Sorted order: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The sort Method

- Here is another example:

```
my_list = ['beta', 'alpha', 'delta', 'gamma']
print('Original order:', my_list)
my_list.sort()
print('Sorted order:', my_list)
```

- When this code runs, it will display the following:

```
Original order: ['beta', 'alpha', 'delta', 'gamma']
Sorted order: ['alpha', 'beta', 'delta', 'gamma']
```

The remove Method

```
1  # This program demonstrates how to use the remove
2  # method to remove an item from a list.
3
4  def main():
5      # Create a list with some items.
6      food = ['Pizza', 'Burgers', 'Chips']
7
8      # Display the list.
9      print('Here are the items in the food list:')
10     print(food)
11
12     # Get the item to change.
13     item = input('Which item should I remove? ')
14
15     try:
16         # Remove the item.
17         food.remove(item)
18
19         # Display the list.
20         print('Here is the revised list:')
21         print(food)
22
23     except ValueError:
24         print('That item was not found in the list.')
25
26 # Call the main function.
27 main()
```

- The remove method removes an item from the list.
- You pass an item to the method as an argument, and the first element containing that item is removed.
- This reduces the size of the list by one element.
- All of the elements after the removed element are shifted one position toward the beginning of the list.
- A ValueError exception is raised if the item is not found in the list.

The reverse Method

- The reverse method simply reverses the order of the items in the list. Here is an example:

```
my_list = [1, 2, 3, 4, 5]
print('Original order:', my_list)
my_list.reverse()
print('Reversed:', my_list)
```

- This code will display the following:

```
Original order: [1, 2, 3, 4, 5]
Reversed: [5, 4, 3, 2, 1]
```

The del Statement

- The remove method you saw earlier removes a specific item from a list, if that item is in the list.
- Some situations might require you remove an element from a specific index, regardless of the item that is stored at that index.
- This can be accomplished with the del statement.
- Here is an example of how to use the del statement:

```
my_list = [1, 2, 3, 4, 5]
print('Before deletion:', my_list)
del my_list[2]
print('After deletion:', my_list)
```

- This code will display the following:

```
Before deletion: [1, 2, 3, 4, 5]
After deletion: [1, 2, 4, 5]
```

The min and max Functions

- Python has two built-in functions named min and max that work with sequences.
- The min function accepts a sequence, such as a list, as an argument and returns the item that has the lowest value in the sequence. Here is an example:

```
my_list = [5, 4, 3, 2, 50, 40, 30]
print('The lowest value is', min(my_list))
```

- This code will display the following:

```
The lowest value is 2
```

- The max function accepts a sequence, such as a list, as an argument and returns the item that has the highest value in the sequence. Here is an example:

```
my_list = [5, 4, 3, 2, 50, 40, 30]
print('The highest value is', max(my_list))
```

- This code will display the following:

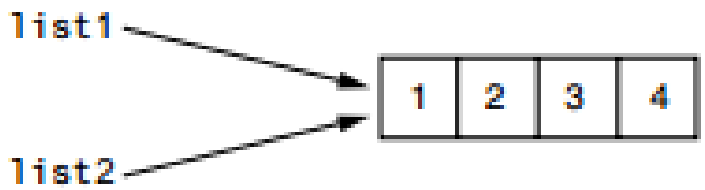
```
The highest value is 50
```

Copying Lists

- Recall that in Python, assigning one variable to another variable simply makes both variables reference the same object in memory.
- For example, look at the following code:

  list1 = [1, 2, 3, 4]

  list2 = list1
- After this code executes, both variables list1 and list2 will reference the same list in memory.
- This is shown in Figure.



Copying Lists

```
1  >>> list1 = [1, 2, 3, 4] Enter
2  >>> list2 = list1 Enter
3  >>> print(list1) Enter
4  [1, 2, 3, 4]
5  >>> print(list2) Enter
6  [1, 2, 3, 4]
7  >>> list1[0] = 99 Enter
8  >>> print(list1) Enter
9  [99, 2, 3, 4]
10 >>> print(list2) Enter
11 [99, 2, 3, 4]
12 >>>
```

- In line 1, we create a list of integers and assign the list to the list1 variable.
- In line 2, we assign list1 to list2. After this, both list1 and list2 reference the same list in memory.
- In line 3, we print the list referenced by list1. The output of the print function is shown in line 4.
- In line 5, we print the list referenced by list2. The output of the print function is shown in line 6. Notice it is the same as the output shown in line 4.

Copying Lists

```
1  >>> list1 = [1, 2, 3, 4] Enter
2  >>> list2 = list1 Enter
3  >>> print(list1) Enter
4  [1, 2, 3, 4]
5  >>> print(list2) Enter
6  [1, 2, 3, 4]
7  >>> list1[0] = 99 Enter
8  >>> print(list1) Enter
9  [99, 2, 3, 4]
10 >>> print(list2) Enter
11 [99, 2, 3, 4]
12 >>>
```

- In line 7, we change the value of list[0]to 99.
- In line 8, we print the list referenced by list1. The output of the print function is shown in line 9. Notice the first element is now 99.
- In line 10, we print the list referenced by list2. The output of the print function is shown in line 11. Notice the first element is 99.
- In this interactive session, the list1 and list2 variables reference the same list in memory.

Copying Lists

- Suppose you wish to make a copy of the list, so list1 and list2 reference two separate but identical lists. One way to do this is with a loop that copies each element of the list. Here is an example:

```
# Create a list with values.
list1 = [1, 2, 3, 4]
# Create an empty list.
list2 = []
# Copy the elements of list1 to list2.
for item in list1:
    list2.append(item)
```

- After this code executes, list1 and list2 will reference two separate but identical lists.
- A simpler and more elegant way to accomplish the same task is to use the concatenation operator, as shown here:

  list1 = [1, 2, 3, 4]
  list2 = [] + list1

- The last statement in this code concatenates an empty list with list1 and assigns the resulting list to list2.
- As a result, list1 and list2 will reference two separate but identical lists.

Processing Lists

```
1  # This program calculates the gross pay for
2  # each of Megan's baristas.
3
4  # NUM_EMPLOYEES is used as a constant for the
5  # size of the list.
6  NUM_EMPLOYEES = 6
7
8  def main():
9      # Create a list to hold employee hours.
10     hours = [0]* NUM_EMPLOYEES
11
12     # Get each employee's hours worked.
13     for index in range(NUM_EMPLOYEES):
14         print('Enter the hours worked by employee ',
15               index + 1, ': ', sep='', end='')
16         hours[index] = float(input())
17
18     # Get the hourly pay rate.
19     pay_rate = float(input('Enter the hourly pay rate: '))
20
21     # Display each employee's gross pay.
22     for index in range(NUM_EMPLOYEES):
23         gross_pay = hours[index] * pay_rate
24         print('Gross pay for employee ', index + 1,': $',
25               format(gross_pay, ',.2f'), sep='')
26
27 # Call the main function.
28 main()
```

- Megan owns a small neighborhood coffee shop, and she has six employees who work as baristas (coffee bartenders).
- All of the employees have the same hourly pay rate.
- Megan has asked you to design a program that will allow her to enter the number of hours worked by each employee, then display the amounts of all the employees' gross pay.
- You determine the program should perform the following steps:

  For each employee: get the number of hours worked and store it in a list element.

  For each list element: use the value stored in the element to calculate an employee's gross pay.

  Display the amount of the gross pay.

Totaling the Values in a List

```
1  # This program calculates the total of the values
2  # in a list.
3
4  def main():
5      # Create a list.
6      numbers = [2, 4, 6, 8, 10]
7
8      # Create a variable to use as an accumulator.
9      total = 0
10
11     # Calculate the total of the list elements.
12     for value in numbers:
13         total += value
14
15     # Display the total of the list elements.
16     print('The total of the elements is', total)
17
18 # Call the main function.
19 main()
```

- Assuming a list contains numeric values, to calculate the total of those values, you use a loop with an accumulator variable.
- The loop steps through the list, adding the value of each element to the accumulator.

Averaging the Values in a List

```
1  # This program calculates the average of the values
2  # in a list.
3
4  def main():
5      # Create a list.
6      scores = [2.5, 7.3, 6.5, 4.0, 5.2]
7
8      # Create a variable to use as an accumulator.
9      total = 0.0
10
11     # Calculate the total of the list elements.
12     for value in scores:
13         total += value
14
15     # Calculate the average of the elements.
16     average = total / len(scores)
17
18     # Display the total of the list elements.
19     print('The average of the elements is', average)
20
21 # Call the main function.
22 main()
```

- The first step in calculating the average of the values in a list is to get the total of the values.
- You saw how to do that with a loop in the preceding section.
- The second step is to divide the total by the number of elements in the list.

Passing a List as an Argument to a Function

```
1  # This program uses a function to calculate the
2  # total of the values in a list.
3
4  def main():
5      # Create a list.
6      numbers = [2, 4, 6, 8, 10]
7
8      # Display the total of the list elements.
9      print('The total is', get_total(numbers))
10
11 # The get_total function accepts a list as an
12 # argument returns the total of the values in
13 # the list.
14 def get_total(value_list):
15     # Create a variable to use as an accumulator.
16     total = 0
17
18     # Calculate the total of the list elements.
19     for num in value_list:
20         total += num
21
22     # Return the total.
23     return total
24
25 # Call the main function.
26 main()
```

- You can easily pass a list as an argument to a function.
- This gives you the ability to put many of the operations that you perform on a list in their own functions.
- When you need to call these functions, you can pass the list as an argument.

Returning a List from a Function

```
1  # This program uses a function to create a list.
2  # The function returns a reference to the list.
3
4  def main():
5      # Get a list with values stored in it.
6      numbers = get_values()
7
8      # Display the values in the list.
9      print('The numbers in the list are:')
10     print(numbers)
11
12 # The get_values function gets a series of numbers
13 # from the user and stores them in a list. The
14 # function returns a reference to the list.
15 def get_values():
16     # Create an empty list.
17     values = []
18
19     # Create a variable to control the loop.
20     again = 'y'
21
22     # Get values from the user and add them to
23     # the list.
24     while again == 'y':
25         # Get a number and add it to the list.
26         num = int(input('Enter a number: '))
27         values.append(num)
28
29         # Want to do this again?
30         print('Do you want to add another number?')
31         again = input('y = yes, anything else = no: ')
32         print()
33
34     # Return the list
35     return values
36
37 # Call the main function.
38 main()
```

- A function can return a reference to a list. This gives you the ability to write a function that creates a list and adds elements to it, then returns a reference to the list so other parts of the program can work with it.
- The code in Program shows an example. It uses a function named get_values that gets a series of values from the user, stores them in a list, then returns a reference to the list.

Processing a List

- Dr. LaClaire gives a series of exams during the semester in her chemistry class. At the end of the semester, she drops each student's lowest test score before averaging the scores.
- She has asked you to design a program that will read a student's test scores as input and calculate the average with the lowest score dropped.
- Here is the algorithm that you developed:

  Get the student's test scores.

  Calculate the total of the scores.

  Find the lowest score.

  Subtract the lowest score from the total. This gives the adjusted total.

  Divide the adjusted total by 1 less than the number of test scores. This is the average.

  Display the average.

Processing a List

```
1  # This program gets a series of test scores and
2  # calculates the average of the scores with the
3  # lowest score dropped.
4
5  def main():
6      # Get the test scores from the user.
7      scores = get_scores()
8
9      # Get the total of the test scores.
10     total = get_total(scores)
11
12     # Get the lowest test score.
13     lowest = min(scores)
14
15     # Subtract the lowest score from the total.
16     total = total - lowest
17
18     # Calculate the average. Note that we divide
19     # by 1 less than the number of scores because
20     # the lowest score was dropped.
21     average = total / (len(scores) - 1)
22
23     # Display the average.
24     print('The average, with the lowest score dropped',
25               'is:', average)
```

- Line 7 calls the get_scores function. The function gets the test scores from the user and returns a reference to a list containing those scores. The list is assigned to the scores variable.
- Line 10 calls the get_total function, passing the scores list as an argument. The function returns the total of the values in the list. This value is assigned to the total variable.

Processing a List

```
1  # This program gets a series of test scores and
2  # calculates the average of the scores with the
3  # lowest score dropped.
4
5  def main():
6      # Get the test scores from the user.
7      scores = get_scores()
8
9      # Get the total of the test scores.
10     total = get_total(scores)
11
12     # Get the lowest test score.
13     lowest = min(scores)
14
15     # Subtract the lowest score from the total.
16     total = total - lowest
17
18     # Calculate the average. Note that we divide
19     # by 1 less than the number of scores because
20     # the lowest score was dropped.
21     average = total / (len(scores) - 1)
22
23     # Display the average.
24     print('The average, with the lowest score dropped',
25               'is:', average)
```

- Line 13 calls the built-in min function, passing the scores list as an argument. The function returns the lowest value in the list. This value is assigned to the lowest variable.
- Line 16 subtracts the lowest test score from the total variable.
- Then, line 21 calculates the average by dividing total by len(scores)-1. (The program divides by len (scores) - 1 because the lowest test score was dropped.)
- Lines 24 and 25 display the average.

Processing a List

```
27 # The get_scores function gets a series of test
28 # scores from the user and stores them in a list.
29 # A reference to the list is returned.
30 def get_scores():
31     # Create an empty list.
32     test_scores = []
33
34     # Create a variable to control the loop.
35     again = 'y'
36
37     # Get the scores from the user and add them to
38     # the list.
39     while again == 'y':
40         # Get a score and add it to the list.
41         value = float(input('Enter a test score: '))
42         test_scores.append(value)
43
44         # Want to do this again?
45         print('Do you want to add another score?')
46         again = input('y = yes, anything else = no: ')
47         print()
48
49     # Return the list.
50     return test_scores
```

- The get_scores function prompts the user to enter a series of test scores.
- As each score is entered, it is appended to a list.
- The list is returned in line 50.

Processing a List

```
52 # The get_total function accepts a list as an
53 # argument returns the total of the values in
54 # the list.
55 def get_total(value_list):
56     # Create a variable to use as an accumulator.
57     total = 0.0
58
59     # Calculate the total of the list elements.
60     for num in value_list:
61         total += num
62
63     # Return the total.
64     return total
```

- This function accepts a list as an argument.
- It uses an accumulator and a loop to calculate the total of the values in the list.
- Line 64 returns the total.

Two-Dimensional Lists

- The elements of a list can be virtually anything, including other lists.
- A two-dimensional list is a list that has other lists as its elements.
- To demonstrate, look at the following interactive session:

```
students = [['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']]
print(students)
[['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']]
print(students[0])
['Joe', 'Kim']
print(students[1])
['Sam', 'Sue']
print(students[2])
['Kelly', 'Chris']
```

Two-Dimensional Lists

- Let's take a closer look at each line.
- Line 1 creates a list and assigns it to the students variable. The list has three elements, and each element is also a list. The element at students[0] is ['Joe', 'Kim'], the element at students[1] is ['Sam', 'Sue'] and the element at students[2] is ['Kelly', 'Chris'].
- Line 2 prints the entire students list. The output of the print function is shown in line 3.
- Line 4 prints the students[0] element. The output of the print function is shown in line 5.
- Line 6 prints the students[1] element. The output of the print function is shown in line 7.
- Line 8 prints the students[2] element. The output of the print function is shown in line 9.

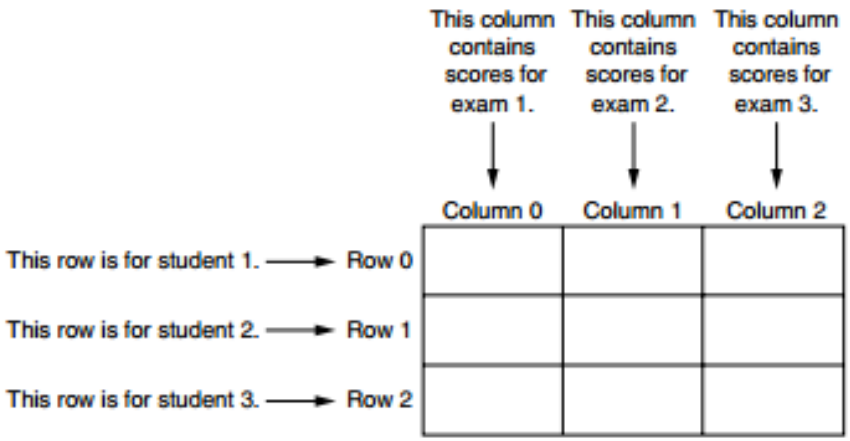
Two-Dimensional Lists

- Lists of lists are also known as nested lists,or two-dimensional lists.
- It is common to think of a two-dimensional list as having rows and columns of elements, as shown in Figure.
- This figure shows the two-dimensional list that was created in the previous interactive session as having three rows and two columns.
- Notice the rows are numbered 0, 1, and 2, and the columns are numbered 0 and 1.
- There is a total of six elements in the list.

	Column 0	Column 1
Row 0	'Joe'	'Kim'
Row 1	'Sam'	'Sue'
Row 2	'Kelly'	'Chris'

Two-Dimensional Lists

- Two-dimensional lists are useful for working with multiple sets of data.
- For example, suppose you are writing a grade-averaging program for a teacher. The teacher has three students, and each student takes three exams during the semester.
- One approach would be to create three separate lists, one for each student. Each of these lists would have three elements, one for each exam score.
- This approach would be cumbersome, however, because you would have to separately process each of the lists.
- A better approach would be to use a two-dimensional list with three rows (one for each student) and three columns (one for each exam score), as shown in Figure



Two-Dimensional List

When processing the data in a two-dimensional list, you need two subscripts: one for the rows, and one for the columns.

```
scores = [[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]]
```

The elements in row 0 are referenced as follows:

```
scores[0][0]
scores[0][1]
scores[0][2]
```

The elements in row 1 are referenced as follows:

```
scores[1][0]
scores[1][1]
scores[1][2]
```

And, the elements in row 2 are referenced as follows:

```
scores[2][0]
scores[2][1]
scores[2][2]
```

	Column 0	Column 1	Column 2
Row 0	`scores[0][0]`	`scores[0][1]`	`scores[0][2]`
Row 1	`scores[1][0]`	`scores[1][1]`	`scores[1][2]`
Row 2	`scores[2][0]`	`scores[2][1]`	`scores[2][2]`

Two-Dimensional List

```
1  # This program assigns random numbers to
2  # a two-dimensional list.
3  import random
4
5  # Constants for rows and columns
6  ROWS = 3
7  COLS = 4
8
9  def main():
10     # Create a two-dimensional list.
11     values = [[0, 0, 0, 0],
12               [0, 0, 0, 0],
13               [0, 0, 0, 0]]
14
15     # Fill the list with random numbers.
16     for r in range(ROWS):
17         for c in range(COLS):
18             values[r][c] = random.randint(1, 100)
19
20     # Display the random numbers.
21     print(values)
22
23 # Call the main function.
24 main()
```

- Let's take a closer look at the program:
- Lines 6 and 7 create global constants for the number of rows and columns.
- Lines 11 through 13 create a two-dimensional list and assign it to the values variable.
- We can think of the list as having three rows and four columns. Each element is assigned the value 0.

Two-Dimensional List

```
1  # This program assigns random numbers to
2  # a two-dimensional list.
3  import random
4
5  # Constants for rows and columns
6  ROWS = 3
7  COLS = 4
8
9  def main():
10     # Create a two-dimensional list.
11     values = [[0, 0, 0, 0],
12               [0, 0, 0, 0],
13               [0, 0, 0, 0]]
14
15     # Fill the list with random numbers.
16     for r in range(ROWS):
17         for c in range(COLS):
18             values[r][c] = random.randint(1, 100)
19
20     # Display the random numbers.
21     print(values)
22
23 # Call the main function.
24 main()
```

- Lines 16 through 18 are a set of nested for loops.
- The outer loop iterates once for each row, and it assigns the variable r the values 0 through 2.
- The inner loop iterates once for each column, and it assigns the variable c the values 0 through 3.
- The statement in line 18 executes once for each element of the list, assigning it a random integer in the range of 1 through 100.
- Line 21 displays the list's contents.

Tuples

- A tuple is a sequence, very much like a list.
- The primary difference between tuples and lists is that tuples are immutable. That means once a tuple is created, it cannot be changed.
- When you create a tuple, you enclose its elements in a set of parentheses, as shown in the following interactive session:

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)
(1, 2, 3, 4, 5)
```

- The first statement creates a tuple containing the elements 1, 2, 3, 4, and 5 and assigns it to the variable my_tuple.
- The second statement sends my_tuple as an argument to the print function, which displays its elements.

Tuples

- The following session shows how a for loop can iterate over the elements in a tuple:

```
names = ('Holly', 'Warren', 'Ashley')
for n in names:
    print(n)
```

- Like lists, tuples support indexing, as shown in the following session:

```
names = ('Holly', 'Warren', 'Ashley')
for i in range(len(names)):
    print(names[i])
```

Tuples

- In fact, tuples support all the same operations as lists, except those that change the contents of the list. Tuples support the following:
  - Subscript indexing (for retrieving element values only)
  - Methods such as index
  - Built-in functions such as len, min,and max
  - Slicing expressions
  - The in operator
  - The + and * operators
- Tuples do not support methods such as append, remove, insert, reverse,and sort.

Tuples

- If the only difference between lists and tuples is immutability, you might wonder why tuples exist.
- One reason that tuples exist is performance. Processing a tuple is faster than processing a list, so tuples are good choices when you are processing lots of data, and that data will not be modified.
- Another reason is that tuples are safe. Because you are not allowed to change the contents of a tuple, you can store data in one and rest assured that it will not be modified (accidentally or otherwise) by any code in your program.
- Additionally, there are certain operations in Python that require the use of a tuple. As you learn more about Python, you will encounter tuples more frequently.

Tuples

```
1  >>> number_tuple = (1, 2, 3) Enter
2  >>> number_list = list(number_tuple) Enter
3  >>> print(number_list) Enter
4  [1, 2, 3]
5  >>> str_list = ['one', 'two', 'three'] Enter
6  >>> str_tuple = tuple(str_list) Enter
7  >>> print(str_tuple) Enter
8  ('one', 'two', 'three')
9  >>>
```

- You can use the built-in list()function to convert a tuple to a list, and the built-in tuple() function to convert a list to a tuple.
- Line 1 creates a tuple and assigns it to the number_tuple variable.
- Line 2 passes number_tuple to the list()function. The function returns a list containing the same values as number_tuple, and it is assigned to the number_list variable.
- Line 3 passes number_list to the print function. The function's output is shown in line 4.

Tuples

```
1  >>> number_tuple = (1, 2, 3) Enter
2  >>> number_list = list(number_tuple) Enter
3  >>> print(number_list) Enter
4  [1, 2, 3]
5  >>> str_list = ['one', 'two', 'three'] Enter
6  >>> str_tuple = tuple(str_list) Enter
7  >>> print(str_tuple) Enter
8  ('one', 'two', 'three')
9  >>>
```

- Line 5 creates a list of strings and assigns it to the str_list variable.
- Line 6 passes str_list to the tuple()function. The function returns a tuple containing the same values as str_list, and it is assigned to str_tuple.
- Line 7 passes str_tuple to the print function. The function's output is shown in line 8.