CN101

Lecture 11-12

Lists and Tuples

Topics

- Sequences
- Introduction to Lists
- List Slicing
- Finding Items in Lists with the in Operator
- List Methods and Useful Built-in Functions
- Copying Lists
- Two-Dimensional Lists
- Tuples

Sequences

- Sequence: an object that contains multiple items of data
  - The items are stored in sequence one after another
- Python provides different types of sequences, including lists and tuples
  - The difference between these is that a list is mutable and a tuple is immutable

Introduction to Lists

- List: an object that contains multiple data items
  - Element: An item in a list
  - Format: `list = [item1, item2, etc.]`
  - Can hold items of different types
- `print` function can be used to display an entire list

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

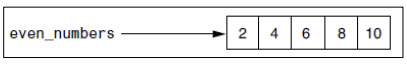
- `list()` function can convert certain types of objects to lists

```
>>> numbers = list(range(1, 10, 2))
>>> print(numbers)
[1, 3, 5, 7, 9]
```


Introduction to Lists (cont'd.)

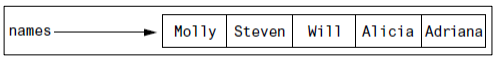
- Here is a statement that creates a list of integers:

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



- The following is another example:

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



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

```
info = ['Alicia', 27, 1550.87]
```

The Repetition Operator and Iterating over a List

- Repetition operator: makes multiple copies of a list and joins them together
  - The * symbol is a repetition operator when applied to a sequence and an integer
    - Sequence is left operand, number is right
  - General format: `list * n`

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

- You can iterate over a list using a `for` loop
  - Format: `for x in list:`

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

Indexing

- Index: a number specifying the position of an element in a list
  - Enables access to individual element in list
  - Index of first element in the list is 0, second element is 1, and n'th element is n-1
  - Negative indexes identify positions relative to the end of the list
    - The index -1 identifies the last element, -2 identifies the next to last element, etc.

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


The `len` function

- An `IndexError` exception is raised if an invalid index is used
- `len` function: returns the length of a sequence such as a list
  - Example: *`size = len(my_list)`*
  - Returns the number of elements in the list, so the index of last element is `len(list)-1`
  - Can be used to prevent an `IndexError` exception when iterating over a list with a loop

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


Lists Are Mutable

- Mutable sequence: the items in the sequence can be changed
  - Lists are mutable, and so their elements can be changed
- An expression such as
- `list[1] = new_value` can be used to assign a new value to a list element
  - Must use a valid index to prevent raising of an `IndexError` exception

```
>>> numbers = [1, 2, 3, 4, 5]
>>> print(numbers)
[1, 2, 3, 4, 5]
>>> numbers[2] = 10
>>> print(numbers)
[1, 2, 10, 4, 5]
```


Program 7-1 (sales_list.py)

```
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()
```

Program Output (with input shown in bold)

```
Enter the sales for each day.
Day #1: 1000 Enter
Day #2: 2000 Enter
Day #3: 3000 Enter
Day #4: 4000 Enter
Day #5: 5000 Enter
Here are the values you entered:
1000.0
2000.0
3000.0
4000.0
5000.0
```


Concatenating Lists

- Concatenate: join two things together
- The + operator can be used to concatenate two lists
  - Cannot concatenate a list with another data type, such as a number
- The += augmented assignment operator can also be used to concatenate lists

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

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


List Slicing

- Slice: a span of items that are taken from a sequence
  - List slicing format: *`list[start : end]`*
  - Span is a list containing copies of elements from *`start`* up to, but not including, *`end`*
    - If *`start`* not specified, `0` is used for start index
    - If *`end`* not specified, `len(list)` is used for end index
  - Slicing expressions can include a step value and negative indexes relative to end of list

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

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

Finding Items in Lists with the `in` Operator

- You can use the `in` operator to determine whether an item is contained in a list
  - General format: *`item in list`*
  - Returns `True` if the item is in the list, or `False` if it is not in the list
- Similarly you can use the `not in` operator to determine whether an item is not in a list

Program 7-2 (in_list.py)

```
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.
```

List Methods

- `append(item)`: used to add items to a list – *`item`* is appended to the end of the existing list

```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers.append(6)
>>> print(numbers)
[1, 2, 3, 4, 5, 6]
```

List Methods (cont'd.)

- `index(item)`: used to determine where an item is located in a list
  - Returns the index of the first element in the list containing `item`
  - Raises `ValueError` exception if *`item`* not in the list

```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers.index(3)
2
>>> numbers.index(7)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    numbers.index(7)
ValueError: 7 is not in list
```

Program 7-3 (list_append.py)


```
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()
```

Program Output (with input shown in bold)

```
Enter a name: Kathryn Enter
Do you want to add another name?
y = yes, anything else = no: y Enter

Enter a name: Chris Enter
Do you want to add another name?
y = yes, anything else = no: y Enter

Enter a name: Kenny Enter
Do you want to add another name?
y = yes, anything else = no: y Enter

Enter a name: Renee Enter
Do you want to add another name?
y = yes, anything else = no: n Enter

Here are the names you entered.
Kathryn
Chris
Kenny
Renee
```

List Methods (cont'd.)

- `insert(index, item)`: used to insert *item* at position *index* in the list
- `sort()`: used to sort the elements of the list in ascending order

```
>>> numbers = [1, 3, 2, 6, 4]
>>> numbers.insert(2, 5)
>>> print(numbers)
[1, 3, 5, 2, 6, 4]
>>> numbers.sort()
>>> print(numbers)
[1, 2, 3, 4, 5, 6]
```


Program 7-5 (insert_list.py)

```
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()
```

Program Output

```
The list before the insert:
['James', 'Kathryn', 'Bill']
The list after the insert:
['Joe', 'James', 'Kathryn', 'Bill']
```


List Methods (cont'd.)

- `remove(item)`: removes the first occurrence of *item* in the list
  - Raises `ValueError` exception if *item* not in the list
- `reverse()`: reverses the order of the elements in the list

```
>>> numbers = [1, 2, 3, 2, 5]
>>> numbers.remove(2)
>>> print(numbers)
[1, 3, 2, 5]
>>> numbers.reverse()
>>> print(numbers)
[5, 2, 3, 1]
```


Useful Built-in Functions

- `del` statement: removes an element from a specific index in a list
  - General format: `del list[i]`

```
>>> numbers = [1, 2, 3, 4, 5]
>>> del numbers[3]
>>> print(numbers)
[1, 2, 3, 5]
>>> del numbers[5]
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    del numbers[5]
IndexError: list assignment index out of range
```


Useful Built-in Functions (cont'd.)

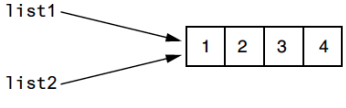
- `min` and `max` functions: built-in functions that returns the item that has the lowest or highest value in a sequence
  - The sequence is passed as an argument
- `Sum` functions: built-in functions that returns the sum of all values in a sequence

```
>>> my_list = [5, 4, 3, 2, 50, 40, 30]
>>> print('The lowest value is', min(my_list))
The lowest value is 2
>>> print('The highest value is', max(my_list))
The highest value is 50
>>> print('The sum is', sum(my_list))
The sum is 134
```


List Referencing

```
# Create a list.
list1 = [1, 2, 3, 4]
# Assign the list to the list2 variable.
list2 = list1
```

- After this code executes, both variables list1 and list2 will reference the same list in memory.




```
>>> list1 = [1, 2, 3, 4] Enter
>>> list2 = list1 Enter
>>> print(list1) Enter
[1, 2, 3, 4]
>>> print(list2) Enter
[1, 2, 3, 4]
>>> list1[0] = 99 Enter
>>> print(list1) Enter
[99, 2, 3, 4]
>>> print(list2) Enter
[99, 2, 3, 4]
>>>
```

Copying Lists

- To make a copy of a list you must copy each element of the list
  - Two methods to do this:
    - Creating a new empty list and using a `for` loop to add a copy of each element from the original list to the new list
    - Creating a new empty list and concatenating the old list to the new empty list

```
# 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)
```

```
# Create a list with values.
list1 = [1, 2, 3, 4]
# Create a copy of list1.
list2 = [] + list1
```

As a result, `list1` and `list2` will reference two separate but identical lists.

Program 7-7 (barista_pay.py)

```
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()
```

Program Output (with input shown in bold)

```
Enter the hours worked by employee 1: 10 Enter
Enter the hours worked by employee 2: 20 Enter
Enter the hours worked by employee 3: 15 Enter
Enter the hours worked by employee 4: 40 Enter
Enter the hours worked by employee 5: 20 Enter
Enter the hours worked by employee 6: 18 Enter
Enter the hourly pay rate: 12.75 Enter
Gross pay for employee 1: $127.50
Gross pay for employee 2: $255.00
Gross pay for employee 3: $191.25
Gross pay for employee 4: $510.00
Gross pay for employee 5: $255.00
Gross pay for employee 6: $229.50
```


Two-Dimensional Lists

- Two-dimensional list: a list that contains other lists as its elements
  - Also known as nested list
  - Common to think of two-dimensional lists as having rows and columns
  - Useful for working with multiple sets of data
- To process data in a two-dimensional list need to use two indexes
- Typically use nested loops to process

Two-Dimensional Lists (cont'd.)

```
>>> 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']
>>> print(students[0][0])
Joe
```

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

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

	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]

Program 7-18 (random_numbers.py)


```
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()
```

Program Output

```
[[4, 17, 34, 24], [46, 21, 54, 10], [54, 92, 20, 100]]
```


Tuples

- Tuple: an immutable sequence
  - Very similar to a list
  - Once it is created it cannot be changed
  - Format: `tuple_name = (item1, item2)`
  - Tuples support operations as lists
    - Subscript indexing for retrieving elements
    - Methods such as `index`
    - Built in functions such as `len, min, max, sum`
    - Slicing expressions
    - The `in`, `+`, and `*` operators

Tuples (cont'd.)

- Tuples do not support the methods:
  - `append`
  - `remove`
  - `insert`
  - `reverse`
  - `sort`
- Tuples do not support **`del`** statement


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

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

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

Holly
Warren
Ashley
```


Tuples (cont'd.)

- Advantages for using tuples over lists:
  - Processing tuples is faster than processing lists
  - Tuples are safe
  - Some operations in Python require use of tuples
- `list()` function: converts tuple to list
- `tuple()` function: converts list to tuple

NOTE: If you want to create a tuple with just one element, you must write a trailing comma after the element's value, as shown here:

```
my_tuple = (1,)     # Creates a tuple with one element.
```

If you omit the comma, you will not create a tuple. For example, the following statement simply assigns the integer value 1 to the `value` variable:

```
value = (1)         # Creates an integer.
```


Summary

- This chapter covered:
  - Lists, including:
    - Repetition and concatenation operators
    - Indexing
    - Techniques for processing lists
    - Slicing and copying lists
    - List methods and built-in functions for lists
    - Two-dimensional lists
  - Tuples, including:
    - Immutability
    - Difference from and advantages over lists