COSC 1306 - Prog for Non-Majors

Lists and Tuples

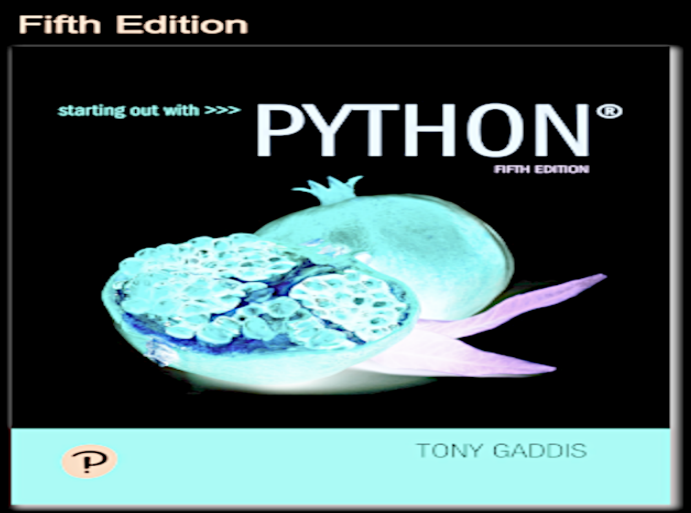
Dr. Mohan

McMurry University

November 7, 2023

Lesson Topics Overview (1 of 2)

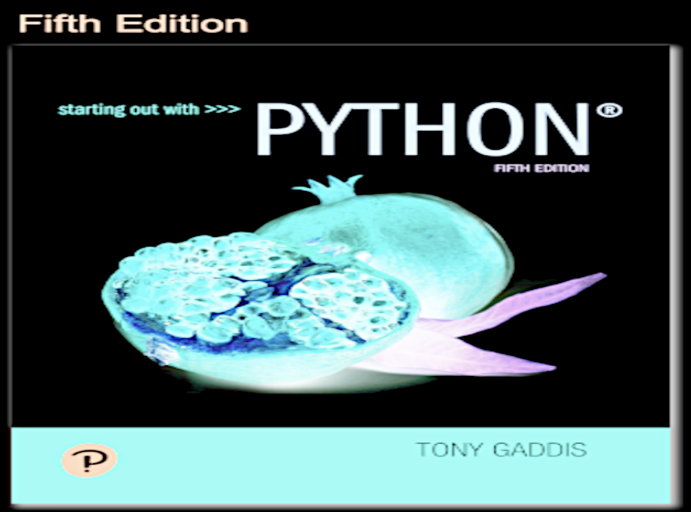
- Sequences.
- Introduction to Lists.
- List Slicing.
- Finding Items in Lists with the in Operator.
- List Methods and Useful Built-in Functions.



Chapter 7
Lists and Tuples

Lesson Topics Overview (2 of 2)

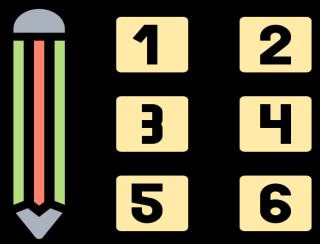
- Copying Lists.
- Processing Lists.
- List Comprehensions.
- Two-Dimensional Lists.
- Tuples.



Chapter 7
Lists and 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 (1 of 2)

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

Introduction to Lists (2 of 2)

Figure 7-1 A list of integers

Figure 7-2 A list of strings

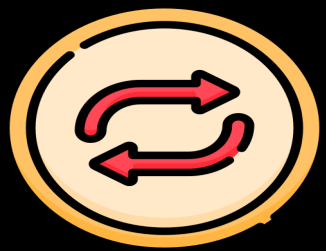
Figure 7-3 A list holding different types

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
- You can **iterate** over a list using a for loop.
  - Format:

    for x in list:



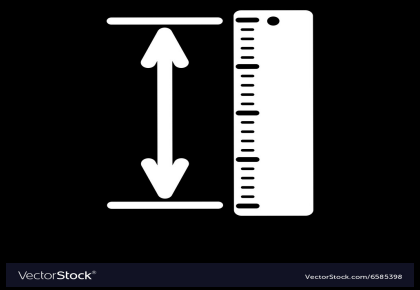
Indexing

- **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.

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:

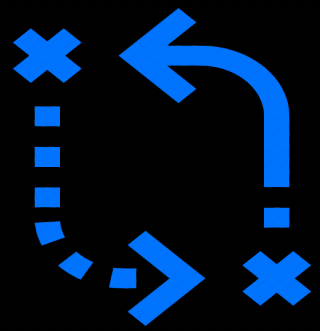
    size = len(my_list)
  - Returns the number of elements in the list, so the index of last element is **len(list)-1**.
  - Can be used to prevent an **IndexError exception** when iterating over a list with a loop.



Lists Are Mutable

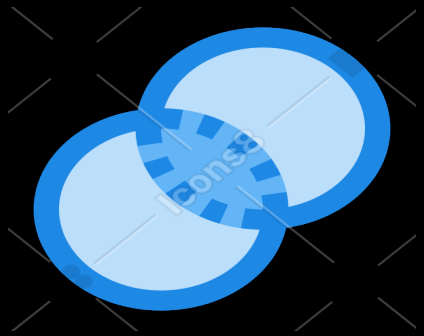
- **Mutable sequence:** the items in the sequence can be changed.
  - Lists are mutable, and so their elements can be changed.
- An expression such as:

  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**.



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.



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.

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.

List Methods and Useful Built-in Functions (1 of 4)

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

List Methods and Useful Built-in Functions (2 of 4)

- **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.
- **remove(item):** removes the first occurrence of item in the list.
- **reverse():** reverses the order of the elements in the list.

List Methods and Useful Built-in Functions (3 of 4)

Table 7-1 A few of the list methods

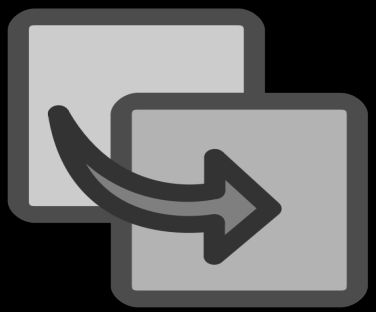
Method	Description
`append(item)`	Adds *item* to the end of the list.
`index(item)`	Returns the index of the first element whose value is equal to item. A `ValueError` exception is raised if item is not found in the list.
`insert(index, item)`	Inserts *item* into the list at the specified *index*. When 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.

List Methods and Useful Built-in Functions (4 of 4)

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

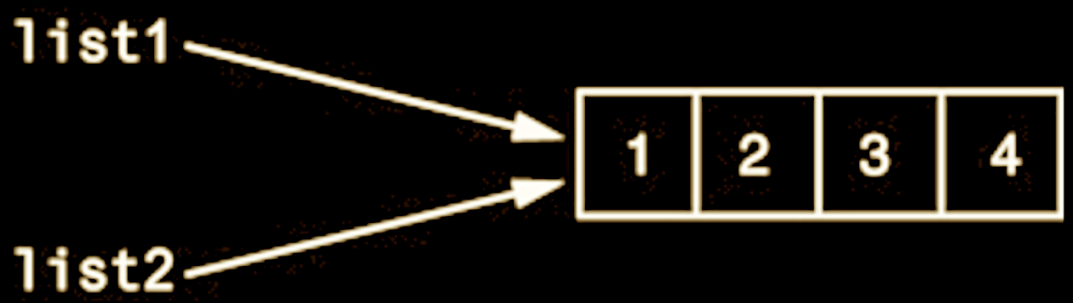
List Copying Lists (1 of 2)

- 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.



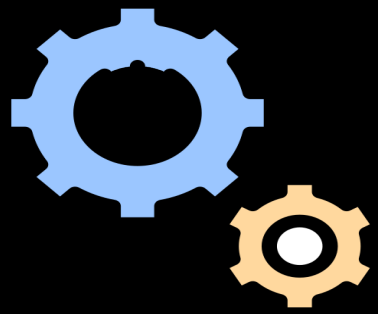
List Copying Lists (2 of 2)

Figure 7-5 list1 and list2 reference the same list



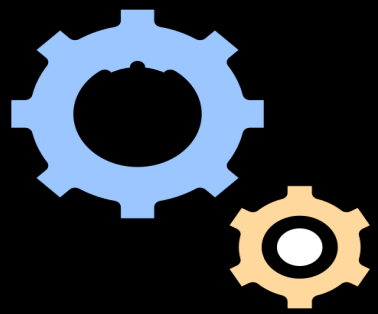
Processing Lists (1 of 2)

- List elements can be used in calculations.
- To calculate total of numeric values in a list use loop with accumulator variable.
- To average numeric values in a list:
  - Calculate total of the values.
  - Divide total of the values by len(list).
- List can be passed as an argument to a function.



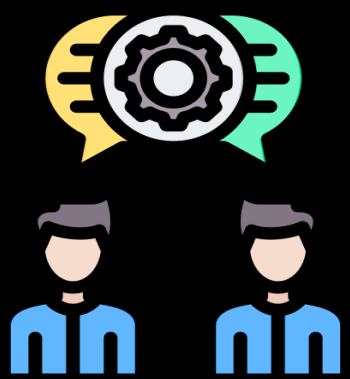
Processing Lists (2 of 2)

- A function can return a reference to a list.
- To save the contents of a list to a file:
  - Use the file object's writelines method.
    - Does not automatically write \n at then end of each item.
  - Use a for loop to write each element and \n.
- To read data from a file use the file object's readlines method.



List Comprehensions (1 of 7)

- **List comprehension:** a concise expression that creates a new list by iterating over the elements of an existing list.



List Comprehensions (2 of 7)

- The **following code** uses a for loop to make a copy of a list:

```
list1 = [1, 2, 3, 4]
list2 = []
for item in list1:
    list2.append(item)
```

- The **following code** uses a list comprehension to make a copy of a list:

```
list1 = [1, 2, 3, 4]
list2 = [item for item in list1]
```

List Comprehensions (3 of 7)

```
list2 = [item for item in list1]
```

Result Expression — Iteration Expression

- The **iteration expression** works like a for loop.
- In this example, it iterates over the elements of list1.
- Each time it iterates, the **target variable** item is assigned the value of an element.
- At the end of each iteration, the value of the result expression is appended to the new list.

List Comprehensions (4 of 7)

```
list1 = [1, 2, 3, 4]
list2 = [item**2 for item in list1]
```

- After this **code executes**, list2 will contain the values [1, 4, 9, 16].

List Comprehensions (5 of 7)

```
str_list = ['Winken', 'Blinken', 'Nod']
len_list = [len(s) for s in str_list]
```

- After this **code executes**, len_list will contain the values [6, 7, 3].

List Comprehensions (6 of 7)

- You can use an **if clause** in a list comprehension to select only certain elements when processing a list.

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = []

for n in list1:
    if n < 10:
        list2.append(n)
```

Works the same as...

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = [item for item in list1 if item < 10]
```

List Comprehensions (7 of 7)

- After this **code executes**, list2 will contain [1, 2, 3, 4].

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = [item for item in list1 if item < 10]
```

Two-Dimensional Lists (1 of 3)

- **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 (2 of 3)

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

Two-Dimensional Lists (3 of 3)

	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]

Tuples (1 of 3)

- **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.
    - Slicing expressions.
    - The in, +, and * operators.

Tuples (2 of 3)

- Tuples do not support the methods:
  - append
  - remove
  - insert
  - reverse
  - sort

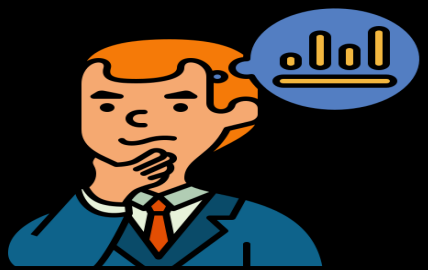
Tuples (3 of 3)

- 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.

Lesson Summary

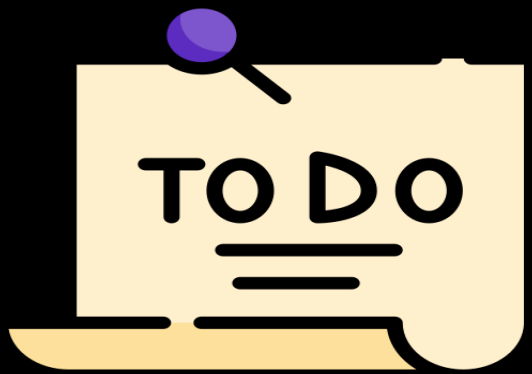
This chapter covered:

- 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.



Things to do

- Get started with Assignment-3!
- Read Textbook Chapter-7.
- Practice the exercise problems at the end of Chapter-7.



Questions?

Please ask your Questions to clarify!