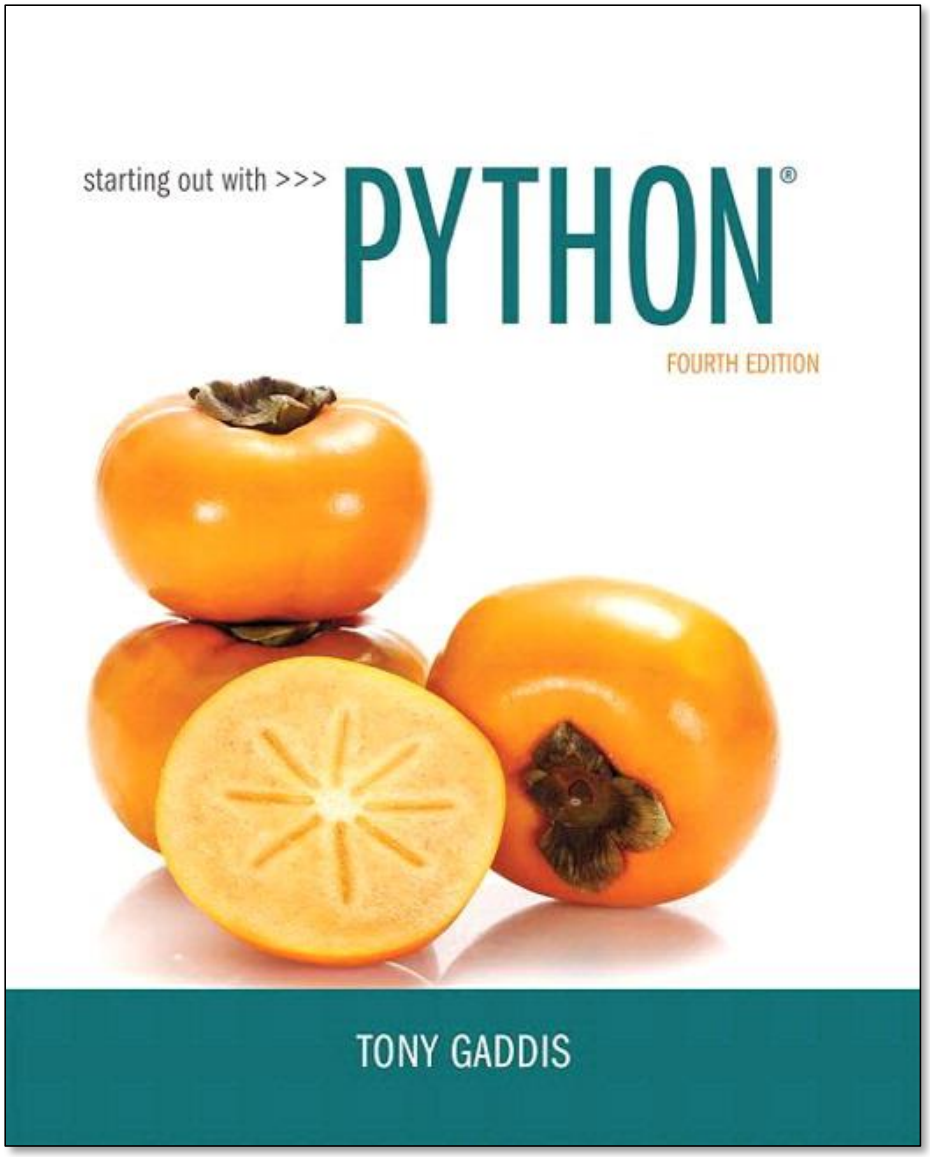
CHAPTER 7

Lists and Tuples



Dr. Bita Shadgar

Topics

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

Learning Outcomes

- At the end of this week the students must be able to:
  - Define sequences, mutable and immutable objects
  - Differentiate lists from tuples
  - Declare, initialize, process lists and tuples
  - Apply operation on sequences like slicing and in
  - Iterate on sequences and use index for naming
  - Recognize list methods and useful built-in functions
  - Copy and process a list
  - Use 2D lists and naming with index

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
  - A list is mutable and a tuple is immutable
  - **Mutable objects** have fields that can be changed, **immutable objects** have no fields that can be changed after the **object** is created.

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 (strong future in Python)
- `print` function can be used to display an entire list

```
>>>ls = ["Hello, world!", 12, 12.5, True]
>>>print (ls)
```

Examples : Lists

Figure 7-1 A list of integers

Figure 7-2 A list of strings

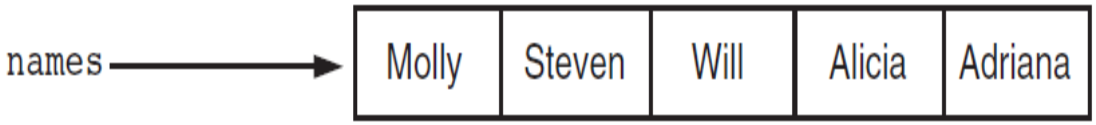


Figure 7-3 A list holding different types

Repetition Operator and Iterating a List

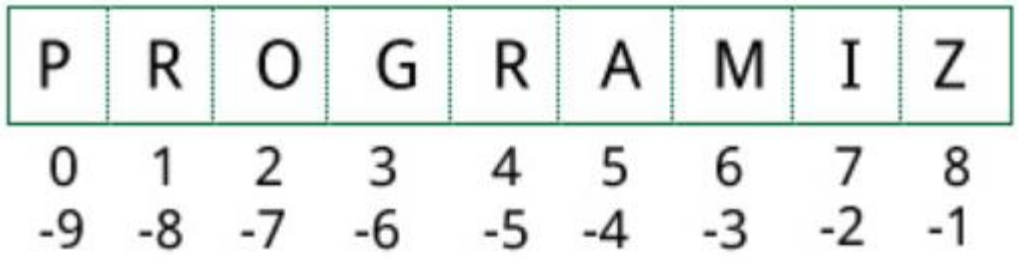
- **Repetition operator**: makes multiple copies of a list and joins them together
  - The * symbol is a repetition operator when applied to a sequence and an integer
  - General format: *list * n*

```
>>>x = [0, 1] * 5
>>>print (x)
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

- 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

```
>>> print([1,2,3] + ['A', 5])
[1, 2, 3, 'A', 5]
```

- The += augmented assignment operator can also be used to concatenate lists

```
>>> x = [0, 1]
>>> x += [2,3]
>>> print (x)
[0, 1, 2, 3]
```

List Slicing

- 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

```
>>> y = [0, 1, 2, 3, 4, 5,6]
>>> print(y[2:5])          #[2,3,4]
```

  - Slicing expressions can include a step value and negative indexes relative to end of list

```
>>> y = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
>>> print(y[0: :2])       #[0, 0, 0, 0, 0]
>>> print(y[-1: : -2])    # [1, 1, 1, 1, 1]
```

Finding Items in Lists using `in` Operator

- Use `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

- **`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
- **`insert(index, item)`:** used to insert *`item`* at position *`index`* in the list
- Hands-on: insert_list.py, index_list.py

List Methods (cont.)

- **`reverse()`**: reverses the order of the elements 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
  - Raises `ValueError` exception if *`item`* is not found in the list
- Hands-on : open and run remove_item.py

Useful Built-in Functions

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

```
>>> y = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del y[5]
>>> print(y)
[0, 1, 2, 3, 4, 6, 7, 8, 9]
```

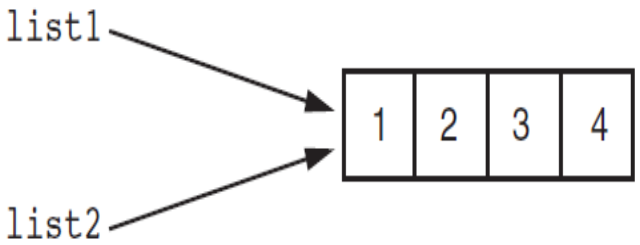
- **`min` and `max` functions**: built-in functions that returns the item that has the lowest or highest value in a sequence
  - The sequence is passed as an argument

Copying Lists

- Assigning a list to a new variable doesn't make a new list

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = list1       #a wrong way of copying
>>> del list1[2]
>>> print(list1)    # [1, 2, 4]
>>> print(list2)    # [1, 2, 4]
```

Figure 7-4 `list1` and `list2` reference the same list



Copying Lists (cont.)

- To make a copy of a list you must copy each element of the list
  1. 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
  2. Creating a new empty list and concatenating the old list to the new empty list
- Hands-on: Modify the code in previous slide to copy list1 to list2 and print them.

Processing Lists

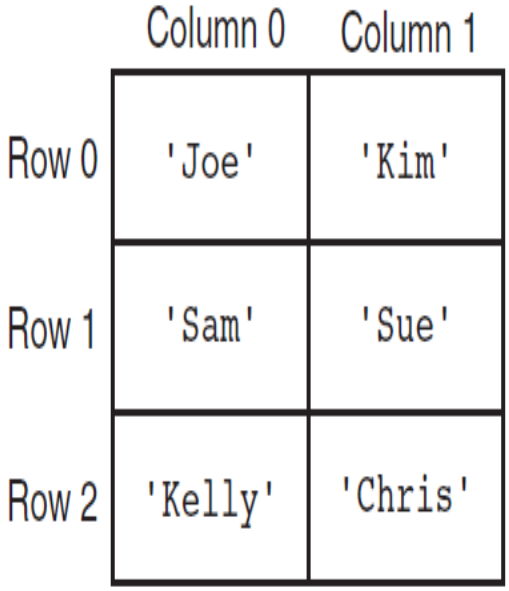
- List elements can be used in calculations
- Hands-on:
  - To calculate total of numeric values in a list use loop with accumulator variable (total_list.py)
  - To average numeric values in a list: (average_list.py)
    - Calculate total of the values
    - Divide total of the values by `len(list)`
  - List can be passed as an argument to a function
    - write a function that given a list of integers, calculates and returns the average. (very similar to total_function.py)
  - A function can return a reference to a list (return_list.py)

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

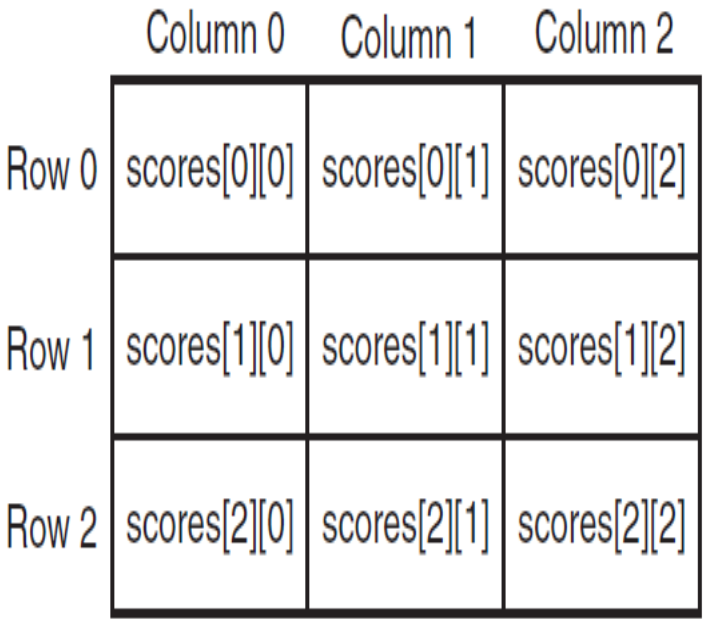
Two-Dimensional Lists (cont'd.)

Figure 7-5 A two-dimensional list



Two-Dimensional Lists (cont'd.)

Figure 7-7 Subscripts for each element of the `scores` list



Passing list as Parameter of a Function

- When you pass list as an argument to a function, you are passing the address of list (Pass by reference), therefore changes on the list in the function is visible by caller function.

```
def test(ls, value):
    ls.append(12)
    value = value + 1
def main():
    myls = [1, 2, 3]
    x = 5
    test(myls, x)
    print(myls, x)
main()
```

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

Tuples (cont'd.)

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

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

Summary

- 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

More Practice

- Check out review questions in chapter 6 of the textbook including :
  - Multiple Choices,
  - True or False
  - Short Answer
  - Algorithm WorkBench
  - Programming Exercises (1, 2, 4, 6, 11)