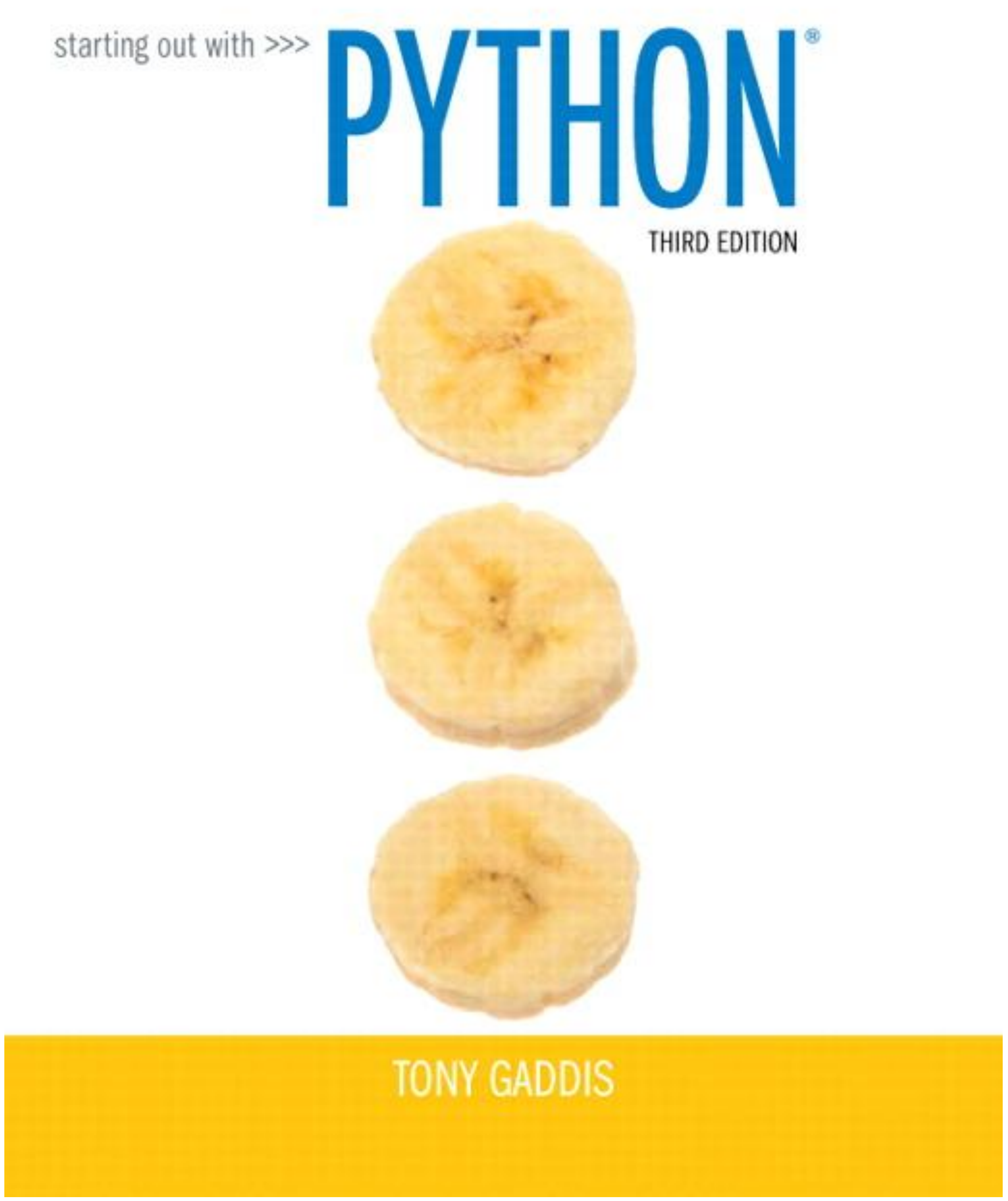
CHAPTER 7

Lists and Tuples



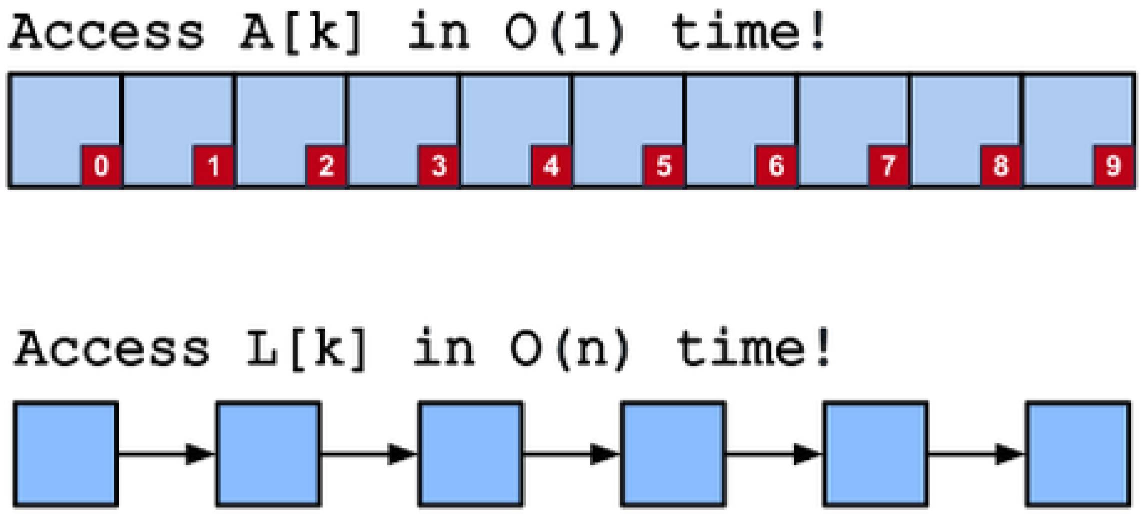
Topics

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

Sequences

- **<u>Sequence</u>: an object that contains multiple items of data**
  - The items are stored in sequence one after another
  - userString = “example”
- **Python provides different types of sequences, including lists and tuples**
  - In Python, lists are used in same manner as arrays
    - Technically lists and arrays are not the same thing
  - The difference between these is that a <u>list is mutable</u> and a <u>tuple is immutable</u>

Sequences

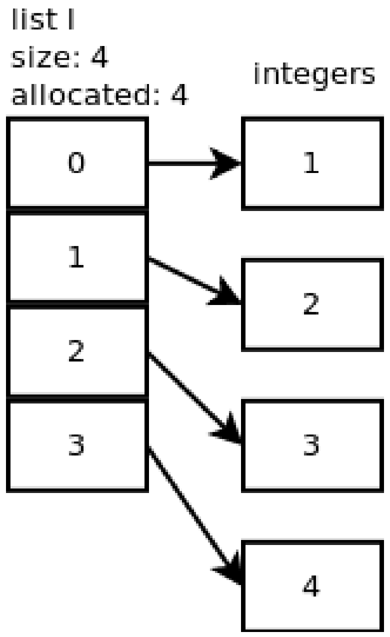


Array & Linked List

- Arrays: faster but need contiguous memory
- Lists: slower but more memory efficient

Python Lists

- Python’s lists are really variable-length arrays. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array’s length in a list head structure.



Introduction to Lists

- **List: an object that contains multiple data items**
  - Element: An item in a list
  - Format: `list = [item1, item2, etc.]`
  - `myList = [1, 2, 3, 4, 5]`
  - Can hold different types: integers, strings, floats, characters
  - Can hold items of different types concurrently
- `print` **function can be used to display an entire list**
- `list()` **function can convert certain types of objects to lists**

Introduction to Lists

Figure 7-1 A list of integers

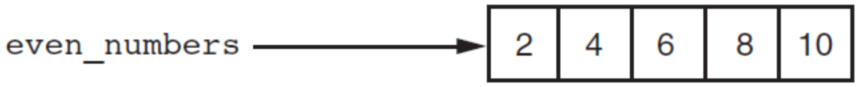


Figure 7-2 A list of strings

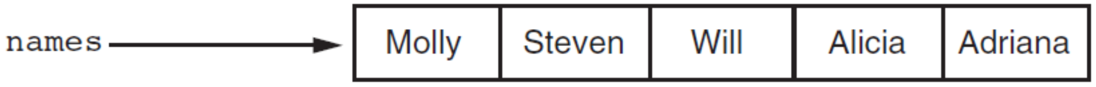
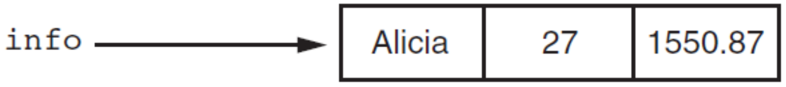


Figure 7-3 A list holding different types



The Repetition Operator and Iterating over a List

- **<u>Repetition operator</u>: 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 element in myList:`

Index/Element

- **<u>Index</u>: a number specifying the position of an element in a list**
  - Enables access to individual element in list
  - Index of first element 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.
- **<u>Element:</u> identifies the item in the list**

usrStr →	**w**	**o**	**r**	**d**	**s**
	0	1	2	3	4

Elements are: w, o, r, d, s

list_Sample1.py

The `len` function

- **An `IndexError` exception is raised if an invalid index is used**
- **<u>`len` function</u>: returns the length of a sequence such as a list**
  - Example: `size = len(myList)`
  - Returns the number of elements in the list, so the index of last element is `len(myList)-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**

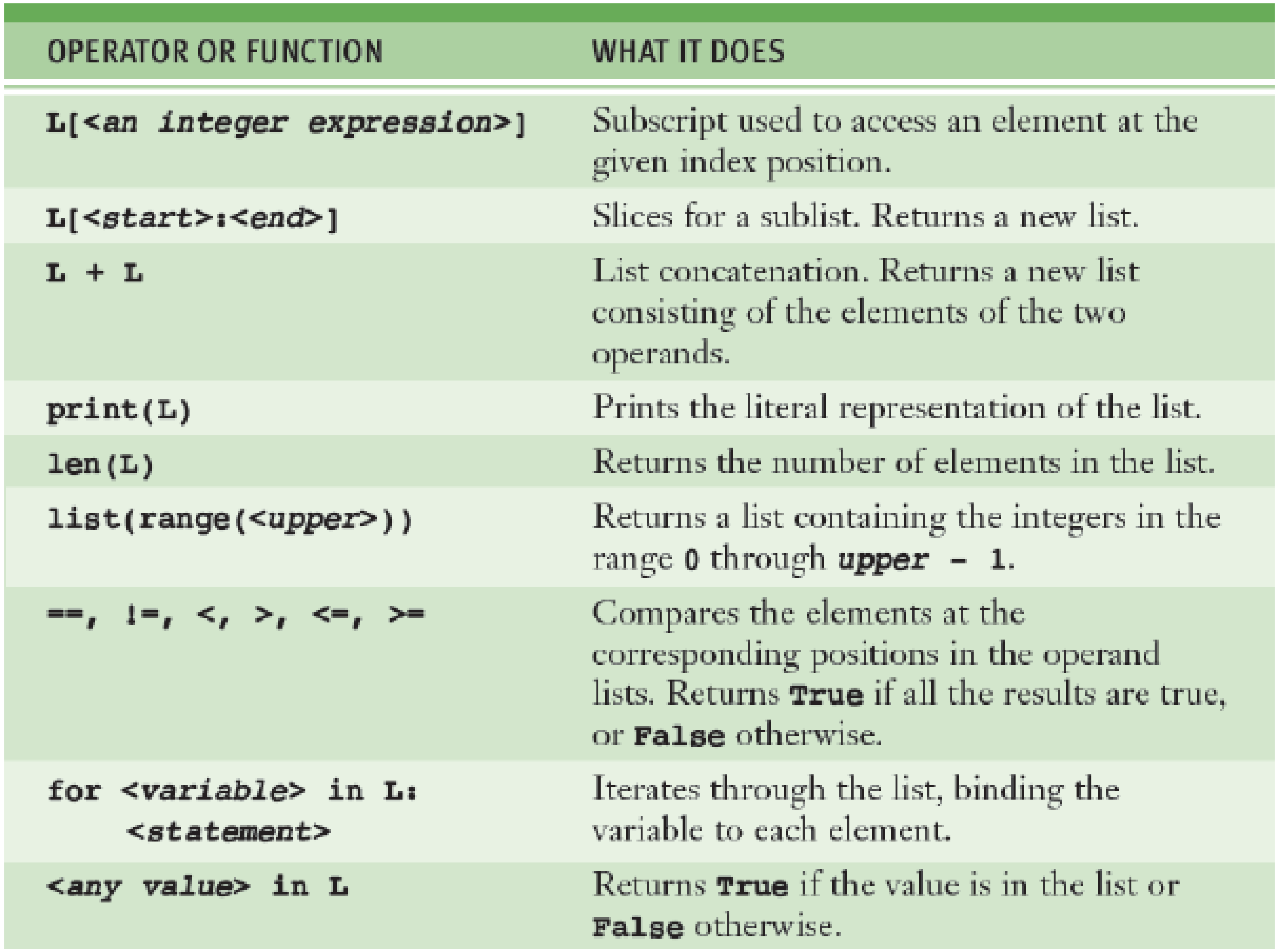
List Slicing

- **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**

Basic List Operators



OPERATOR OR FUNCTION	WHAT IT DOES
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L + L`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range `0` through `upper - 1`.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns `True` if all the results are true, or `False` otherwise.

`for <variable> in L:`
`    <statement>`
	Iterates through the list, binding the variable to each element.

`<any value> in L`	Returns `True` if the value is in the list or `False` otherwise.

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.
`reverse()`	Reverses the order of the items in the list.

List Methods and Useful Built-in Functions

- **`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

- **`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_Sample2.py
list_Sample3.py

List Methods and Useful Built-in Functions

- <u>`insert(`*`index, item`*`)`</u>: used to insert *`item`* at position *`index`* in the list
- <u>`sort()`</u>: used to sort the elements of the list in ascending order
- <u>`remove(`*`item`*`)`</u>: removes the first occurrence of *`item`* in the list
- <u>`reverse()`</u>: reverses the order of the elements in the list

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

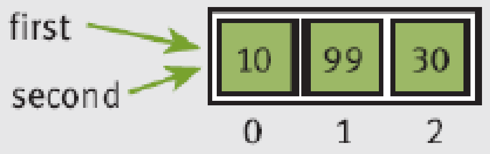
Copying Lists - Side Effects

Aliasing: Mutable property of lists leads to interesting phenomena

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
[10, 99, 30]
>>> second
[10, 99, 30]
```

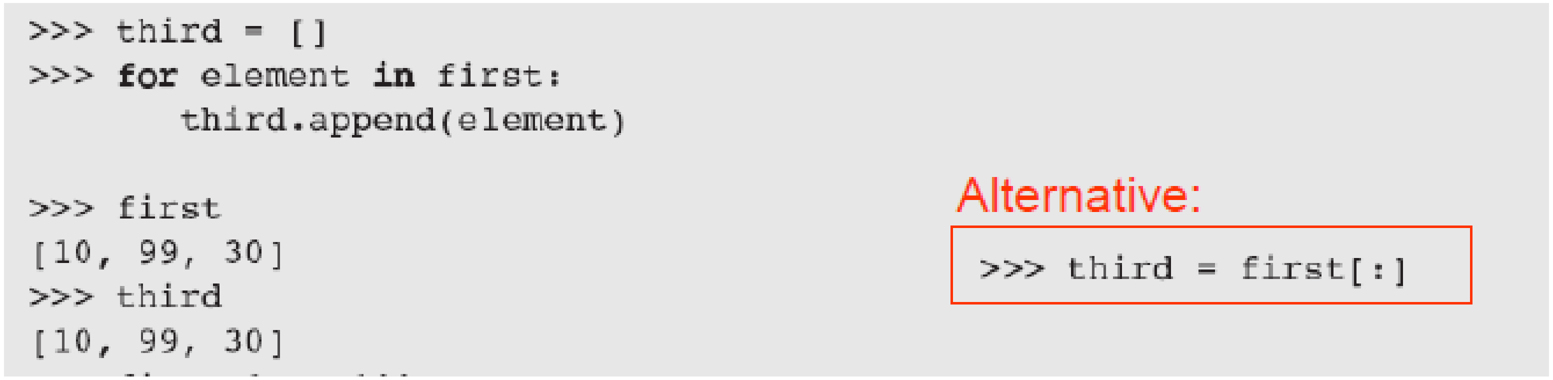
first and second are **aliases**

(refer to the exact same list object)



Copying Lists – Correct Way

To prevent aliasing, copy contents of object

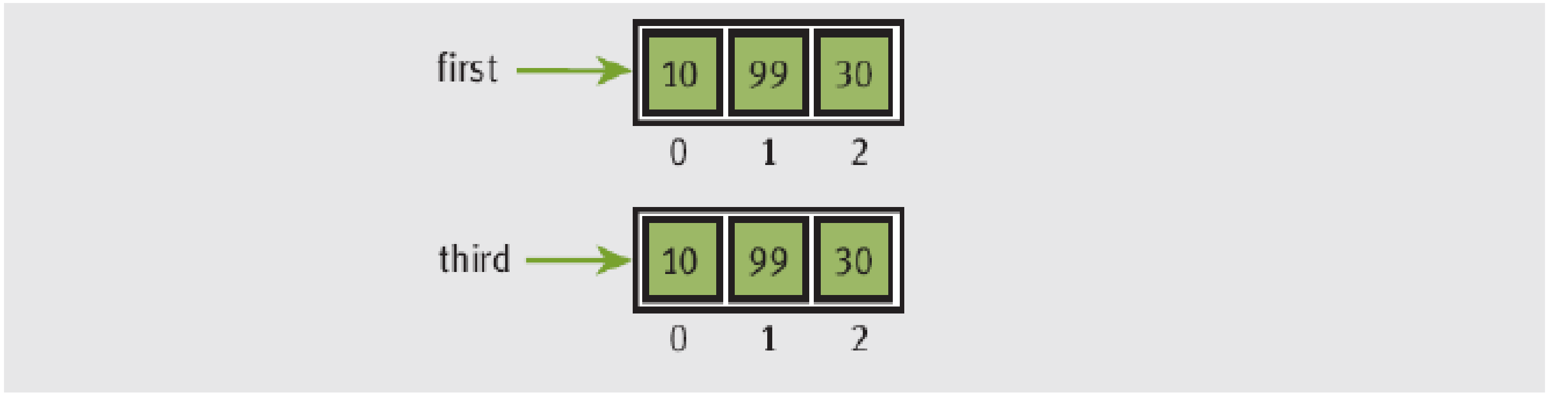


```
>>> third = []
>>> for element in first:
        third.append(element)

>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
```

Alternative:

```
>>> third = first[:]
```



list_Copy.py

Processing Lists

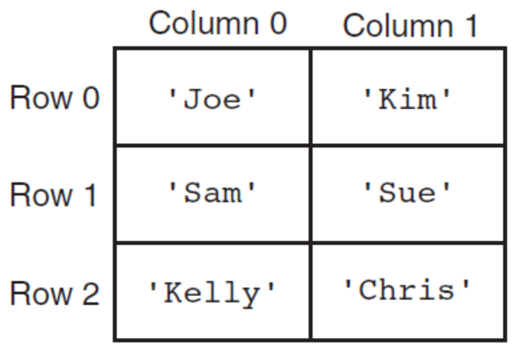
- **List elements can be used in calculations**
- **To calculate total of numeric values in a list use loop with accumulator variable**
- **To average numeric values in a list:**
  - Calculate total of the values
  - Divide total of the values by `len(list)`
- **List can be passed as an argument to a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

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

Figure 7-5 A two-dimensional list



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

Two-Dimensional Lists

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

	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]

list_2D.py

Tuples

- **<u>Tuple</u>: 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

- **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

- **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