

Chapter 6: Lists



Chapter Goals

- To collect elements using lists
- To use the for loop for traversing lists
- To learn common algorithms for processing lists
- To use lists with functions
- To work with tables of data

Contents

- Basic Properties of Lists
- List Operations
- Common List Algorithms
- Using Lists with Functions
- Problem Solving: Adapting Algorithms
- Problem Solving: Discovering Algorithms by Manipulating Physical Objects
- Tables

Basic Properties of Lists

SECTION 6.1

Creating a List

- Specify a list variable with the subscript operator []

Syntax	To create a list:	<code>[value₁, value₂, . . .]</code>
	To access an element:	<code>listReference[index]</code>

Name of list variable $\left\{ \begin{array}{l} \text{moreValues} = [] \\ \text{values} = [32, 54, 67, 29, 35, 80, 115] \end{array} \right.$

Creates an empty list (points to `[]`)

Creates a list with initial values (points to `[32, 54, 67, 29, 35, 80, 115]`)

Initial values (points to the list of numbers)

Use brackets to access an element.

\wedge
`values[i] = 0`
`element = values[i]`

Accessing List Elements

- A list is a sequence of *elements*, each of which has an integer position or *index*
- To access a list element, you specify which index you want to use. That is done with the subscript operator in the same way that you access individual characters in a string

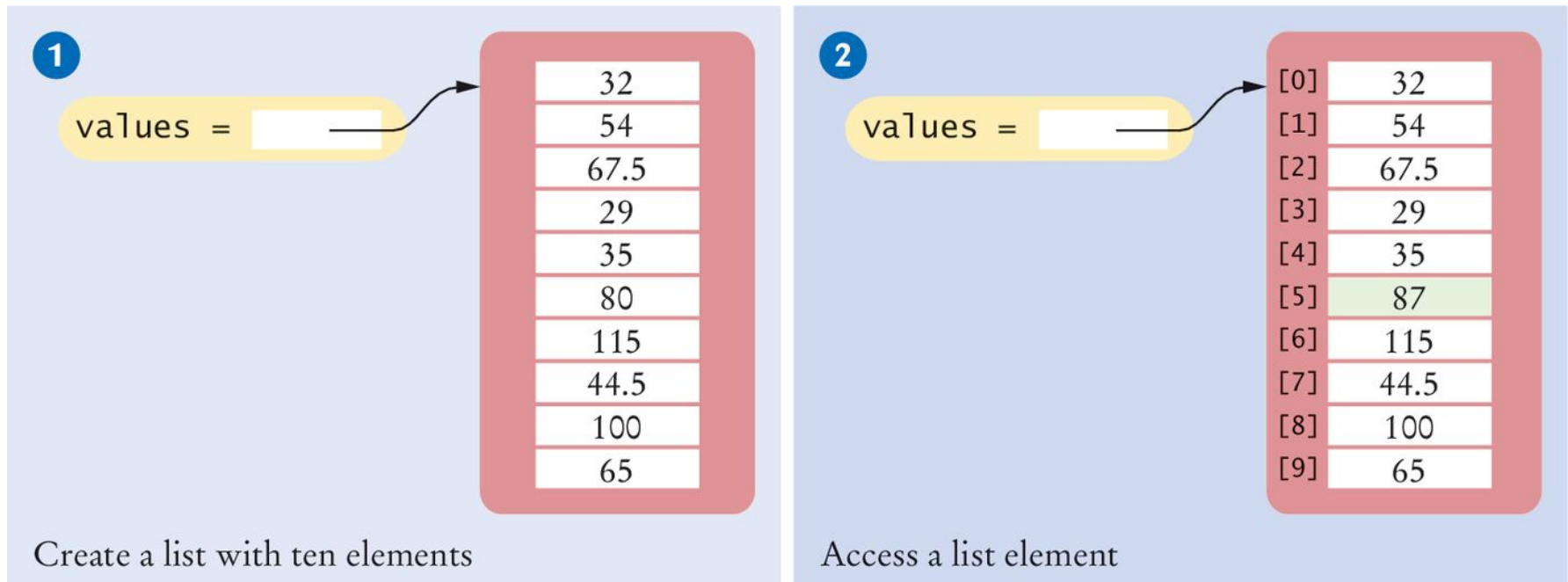
Accessing list
elements

```
print(values[5])
```

Replacing list
elements

```
values[5] = 87
```

Creating Lists/Accessing Elements



1: Creating a list

```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
```

2: Accessing a list element

```
values[5] = 87
```

Lists vs. Strings

- Both lists and strings are **sequences**, and the `[]` operator is used to access an element in any sequence
- There are two differences between lists and strings:
 - Lists can hold values of any type, whereas strings are sequences of characters
 - Moreover:
 - strings are immutable— you cannot change the characters in the sequence
 - Lists are *mutable*

Out of Range Errors

- Out-of-Range Errors:
- Perhaps the most common error in using lists is accessing a nonexistent element

```
values = [2.3, 4.5, 7.2, 1.0, 12.2, 9.0, 15.2, 0.5]
values[8] = 5.4
# Error--values has 8 elements,
# and the index can range from 0 to 7
```

- If your program accesses a list through an out-of-range index, the program will generate an exception at run time

Determining List Length

- You can use the `len()` function to obtain the length of the list; that is, the number of elements:

```
numElements = len(values)
```

Using The Square Brackets

- Note that there are two distinct uses of the square brackets. When the square brackets immediately follow a variable name, they are treated as the subscript operator:

```
values[4]
```

- When the square brackets follow an “=” they create a list:

```
values = [4]
```

Loop Over the Index Values

- Given the values list that contains 10 elements, we will want to set a variable, say *i*, to 0, 1, 2, and so on, up to 9

```
# First version (list index used)
for i in range(10) :
    print(i, values[i])
```

```
# Better version (list index used)
for i in range(len(values)) :
    print(i, values[i])
```

```
# Third version: index values not needed (traverse
# list elements)
for element in values :
    print(element)
```

List References

- Make sure you see the difference between the:
 - List variable: The named 'alias' or pointer to the list
 - List contents: Memory where the values are stored

```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
```

List variable



Reference

List contents

[0]	10
[1]	9
[2]	7
[3]	4
[4]	5

Values

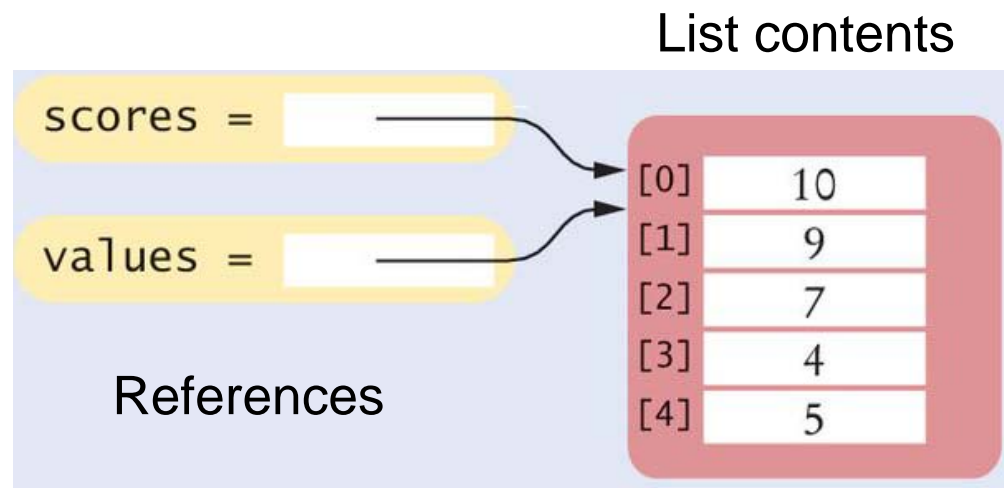
A list variable contains a *reference* to the list contents. The *reference* is the location of the list contents (in memory).

List Aliases

- When you **copy** a list variable into another, both variables refer to the same list
 - The second variable is an *alias* for the first because both variables reference the same list

```
scores = [10, 9, 7, 4, 5]
values = scores      # Copying list reference
```

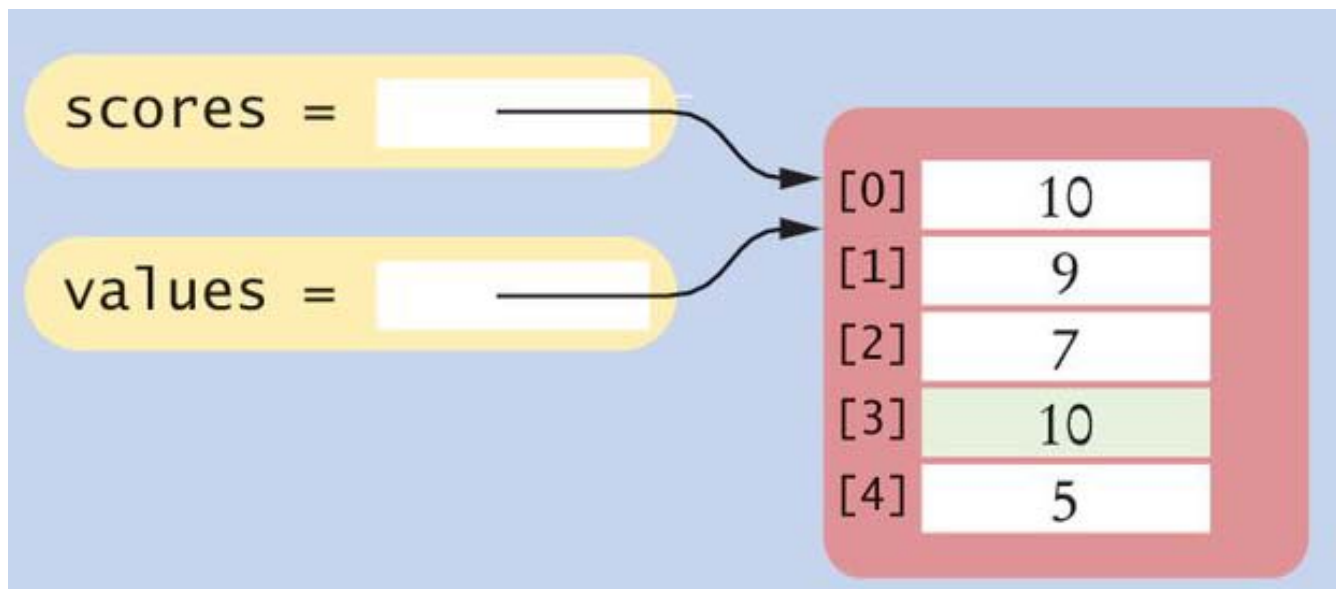
A list variable specifies the location of a list. Copying the reference yields a second reference to the same list.



Modifying Aliased Lists

- You can **modify** the list through either of the variables:

```
scores[3] = 10  
print(values[3])    # Prints 10
```



Reverse Subscripts

- Python, unlike other languages, uses negative subscripts to provide access to the list elements in reverse order.
 - For example, a subscript of -1 provides access to the last element in the list:
 - Similarly, `values[-2]` is the second-to-last element.

Just because you can do this, does not mean you should...

```
last = values[-1]
print("The last element in the list is", last)
```

values =



[0]	32	[-10]
[1]	54	[-9]
[2]	67	[-8]
[3]	29	[-7]
[4]	35	[-6]
[5]	60	[-5]
[6]	115	[-4]
[7]	44	[-3]
[8]	100	[-2]
[9]	65	[-1]

List Operations

SECTION 6.2

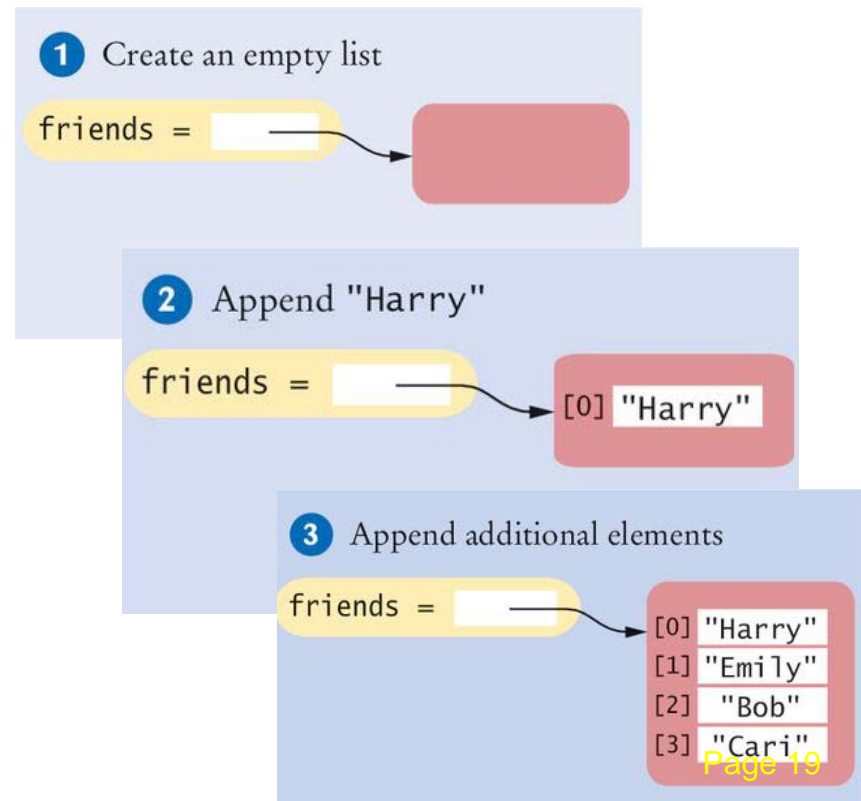
List Operations

- Appending Elements
- Inserting an Element
- Finding an Element
- Removing an Element
- Concatenation
- Equality / Inequality Testing
- Sum, Maximum, Minimum, and Sorting
- Copying Lists

Appending Elements

- Sometimes we may not know the values that will be contained in the list when it's created
- In this case, we can create an empty list and **add elements** to the end as needed

```
#1  
friends = []  
  
#2  
friends.append("Harry")  
  
#3  
friends.append("Emily")  
friends.append("Bob")  
friends.append("Cari")
```

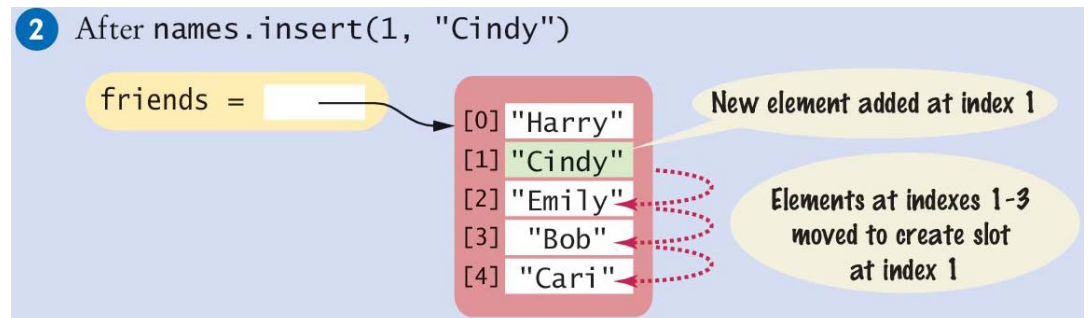
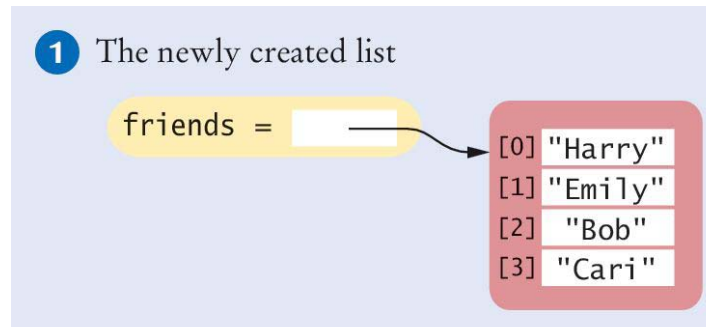


Inserting an Element

- Sometimes the order in which elements are added to a list is important
 - A new element has to be **inserted at a specific position** in the list

```
#1
friends = ["Harry",
           "Emily", "Bob", "Cari"]

#2
friends.insert(1,
              "Cindy")
```



Finding an Element

- If you simply want to know **whether an element is present in a list**, use the **in** operator:

```
if "Cindy" in friends :  
    print("She's a friend")
```

- Often, you want to know the **position at which an element occurs**
 - The `index()` method yields the index of the first match

```
friends = ["Harry", "Emily", "Bob", "Cari", "Emily"]  
n = friends.index("Emily") # Sets n to 1
```

Removing an Element

- The `pop()` method removes the element at a given position

```
friends = ["Harry", "Cindy", "Emily", "Bob", "Cari", "Bill"]  
friends.pop(1)
```

- All of the elements following the removed element are moved up one position to close the gap
- The length of the list is reduced by 1

1 The item at index 1 is removed

friends =

[0] "
[1] "
[2] "
[3] "
[4] "
[5] "

2 The items following the removed element are moved up one position

friends =

[0] "Harry"
[1] "Emily"
[2] "Bob"
[3] "Cari"
[4] "Bill"

Elements at indexes 2-5
moved up one position

Concatenation

- The **concatenation** of two lists is a new list that contains the elements of the first list, followed by the elements of the second

```
myFriends = ["Fritz", "Cindy"]  
yourFriends = ["Lee", "Pat", "Phuong"]
```

- Two lists can be concatenated by using the plus (+) operator:

```
ourFriends = myFriends + yourFriends  
# Sets ourFriends to ["Fritz", "Cindy", "Lee", "Pat", "Phuong"]
```

Replication

- As with string [replication](#) of two lists is a new list that contains the elements of the first list, followed by the elements of the second

```
monthInQuarter = [ 1, 2, 3 ] * 4
```

- Results in the list [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
- You can place the integer on either side of the “*” operator
- The integer specifies how many copies of the list should be concatenated
- One common use of replication is to initialize a list with a fixed value

```
monthlyScores = [0] * 12
```


Equality / Inequality Testing

- You can use the `==` operator to compare whether two lists have the same elements, in the same order.

```
[1, 4, 9] == [1, 4, 9]      # True  
[1, 4, 9] == [4, 1, 9]     # False.
```

- The opposite of `==` is `!=`.

```
[1, 4, 9] != [4, 9]        # True.
```

Sum, Maximum, Minimum

- If you have a list of numbers, the `sum()` function yields the sum of all values in the list.

```
sum([1, 4, 9, 16]) # Yields 30
```

- For a list of numbers or strings, the `max()` and `min()` functions return the largest and smallest value:

```
max([1, 16, 9, 4])           # Yields 16  
min("Fred", "Ann", "Sue")    # Yields "Ann"
```

Sorting

- The `sort()` method sorts a list of numbers or strings.

```
values = [1, 16, 9, 4]  
values.sort() # Now values is [1, 4 , 9, 16]
```

Copying Lists

- As discussed, list variables do not themselves hold list elements
- They hold a reference to the actual list
- If you copy the reference, you get another reference to the same list:

```
prices = values
```

1 After the assignment `prices = values`

values =

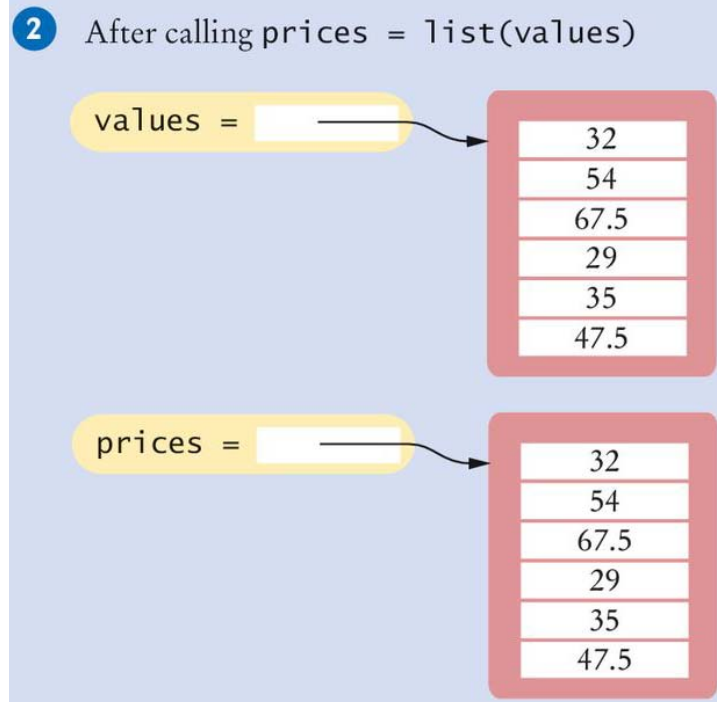
prices =

32
54
67.5
29
35
47.5

Copying Lists (2)

- Sometimes, you want to make a copy of a list; that is, a new list that has the same elements in the same order as a given list
- Use the `list()` function:

```
prices = list(values)
```



Slices of a List

- Sometimes you want to look at a part of a list. Suppose you are given a list of temperatures, one per month:

```
temperatures = [18, 21, 24, 33, 39, 40, 39, 36, 30, 22, 18]
```

- You are only interested in the temperatures for the third quarter, with index values 6, 7, and 8
- You can use the slice operator to obtain them:

```
thirdQuarter = temperatures[6 : 9]
```

- The arguments are the first element to include, and the first to exclude
 - So in our example we get elements 6, 7, and 8

Slices (2)

- Both indexes used with the slice operator are optional
- If the first index is omitted, all elements from the first are included
- The slice

`temperatures[: 6]`

- Includes all elements up to, but not including, position 6
- The slice

`temperatures[6 :]`

- Includes all elements starting at position 6 to the end of the list
- You can assign values to a slice:

`temperatures[6 : 9] = [45, 44, 40]`

- Replaces the values in elements 6, 7, and 8

Common List Functions And Operators

Table 1 Common List Functions and Operators

Operation	Description
<code>[]</code> <code>[<i>elem</i>₁, <i>elem</i>₂, ..., <i>elem</i>_{<i>n</i>}]</code>	Creates a new empty list or a list that contains the initial elements provided.
<code>len(<i>l</i>)</code>	Returns the number of elements in list <i>l</i> .
<code>list(<i>sequence</i>)</code>	Creates a new list containing all elements of the sequence.
<code>values * num</code>	Creates a new list by replicating the elements in the values list <i>num</i> times.
<code>values + moreValues</code>	Creates a new list by concatenating elements in both lists.

Common List Functions And Operators (2)

Table 1 Common List Functions and Operators

Operation	Description
$l[\text{from} : \text{to}]$	Creates a sublist from a subsequence of elements in list l starting at position from and going through but not including the element at position to . Both from and to are optional. (See Special Topic 6.2.)
$\text{sum}(l)$	Computes the sum of the values in list l .
$\text{min}(l)$ $\text{max}(l)$	Returns the minimum or maximum value in list l .
$l_1 == l_2$	Tests whether two lists have the same elements, in the same order.

Common List Methods

Table 2 Common List Methods

Method	Description
<i>l.pop()</i> <i>l.pop(position)</i>	Removes the last element from the list or from the given position. All elements following the given position are moved up one place.
<i>l.insert(position, element)</i>	Inserts the element at the given position in the list. All elements at and following the given position are moved down.
<i>l.append(element)</i>	Appends the element to the end of the list.
<i>l.index(element)</i>	Returns the position of the given element in the list. The element must be in the list.
<i>l.remove(element)</i>	Removes the given element from the list and moves all elements following it up one position.
<i>l.sort()</i>	Sorts the elements in the list from smallest to largest.

Common List Algorithms

SECTION 6.3

Common List Algorithms: Examples and Self-Study

- Filling a List
- Combining List Elements
- Element Separators
- Maximum and Minimum
- Linear Search
- Collecting and Counting Matches
- Removing Matches
- Swapping Elements
- Reading Input

Filling a List

- This loop creates and fills a list with squares (0, 1, 4, 9, 16, ...)

```
values = []  
for i in range(n) :  
    values.append(i * i)
```

Combining List Elements

- Here is how to **compute a sum of numbers**:

```
result = 0.0
for element in values :
    result = result + element
```

- To **concatenate strings**, you only need to change the initial value:

```
result = ""
for element in names :
    result = result + element
```

Element Separators

- When you display the elements of a list, you usually want to separate them, often with commas or vertical lines, like this:

Harry, Emily, Bob

Element Separators (2)

- Add the separator before each element (there's one fewer separator than there are numbers) in the sequence except the initial one (with index 0), like this:

```
for i in range(len(names)) :  
    if i > 0 :  
        result = result + ", "  
    result = result + names[i]
```


Element Separators (3)

- If you want to print values without adding them to a string:

```
for i in range(len(values)) :  
    if i > 0 :  
        print(" | ", end="")  
    print(values[i], end="")  
print()
```

Maximum and Minimum

- Here is the implementation of the max algorithm (already covered in Chapter 4, this one is just specific to a list):

```
largest = values[0]
for i in range(1, len(values)) :
    if values[i] > largest :
        largest = values[i]
```

```
smallest = values[0]
for i in range(1, len(values)) :
    if values[i] < smallest :
        smallest = values[i]
```

Linear Search

- Finding the first value that is > 100 . You need to visit all elements until you have found a match or you have come to the end of the list:

```
limit = 100
pos = 0
found = False
while pos < len(values) and not found :
    if values[pos] > limit :
        found = True
    else :
        pos = pos + 1
if found :
    print("Found at position:", pos)
else :
    print("Not found")
```

A linear search
inspects
elements
in sequence
until a
match is found.

Collecting and Counting Matches

- Collecting all matches

```
limit = 100
result = []
for element in values :
    if (element > limit) :
        result.append(element)
```

- Counting matches

```
limit = 100
counter = 0
for element in values :
    if (element > limit) :
        counter = counter + 1
```

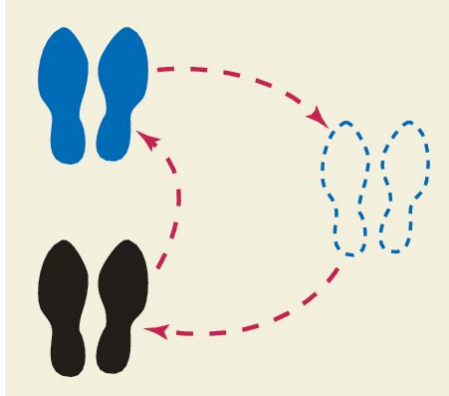
Removing Matches

- Remove all elements that match a particular condition
 - Example: remove all strings of length < 4 from a list

```
i = 0
while i < len(words) :
    word = words[i]
    if len(word) < 4 :
        words.pop(i)
    else :
        i = i + 1
```

Swapping Elements

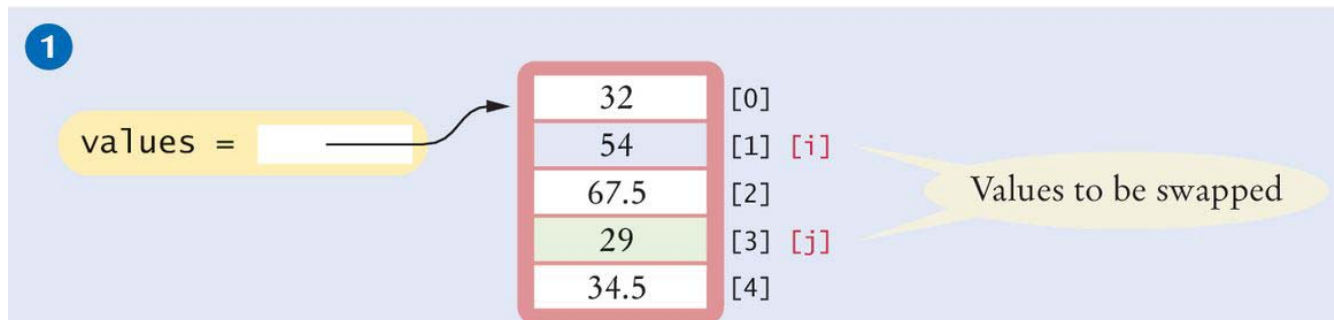
- For example, you can sort a list by repeatedly swapping elements that are not in order
- Swap the elements at positions i and j of a list values
- We'd like to set `values[i]` to `values[j]`. But that overwrites the value that is currently stored in `values[i]`, so we want to save that first:



Before moving a new value into a location (say blue) copy blue's value elsewhere and then move black's value into blue. Then move the temporary value (originally in blue) into black.

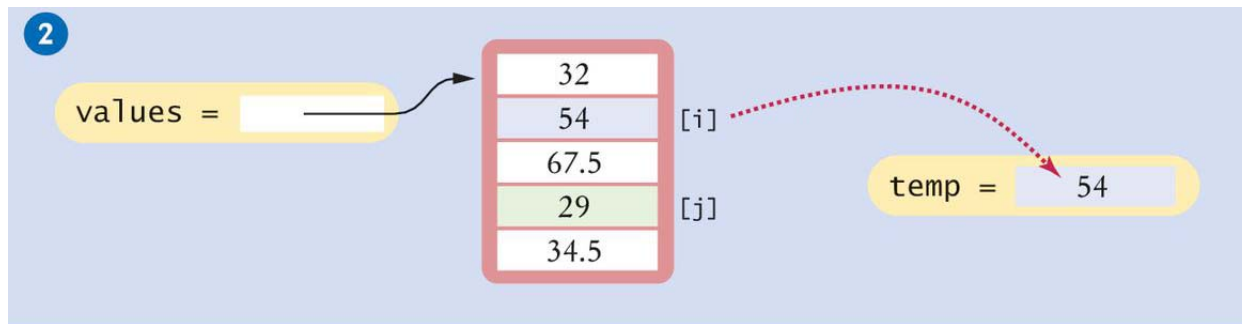
Swapping Elements (2)

- Swapping elements [1] and [3]
 - This sets up the scenario for the actual code that will follow



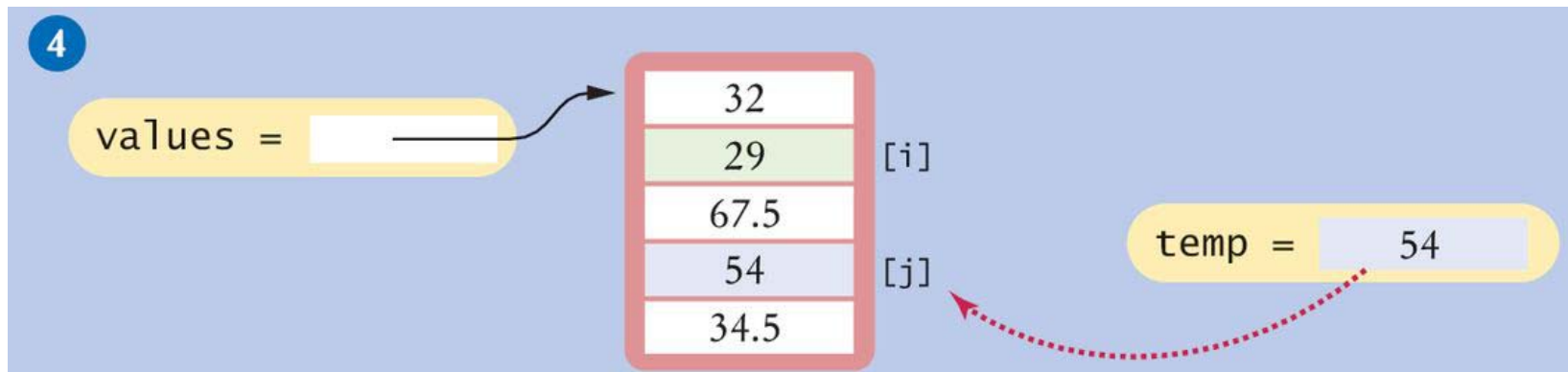
Swapping Elements (3)

```
# Step 2  
temp = values[i]  
  
# Step 3  
values[i] = values[j]
```



Swapping Elements (4)

```
# Step 4  
# temp contains values[i]  
values[j] = temp
```



Reading Input

- It is very common to read input from a user and store it in a list for later processing.

```
values = []  
print("Please enter values, Q to quit:")  
userInput = input("")  
while userInput.upper() != "Q" :  
    values.append(float(userInput))  
    userInput = input("")
```

```
Please enter values, Q to quit:  
32  
29  
67.5  
Q
```

Program execution

Built-In Operations For Lists

- Use the `insert()` method to insert a new element at any position in a list
- The `in` operator tests whether an element is contained in a list
- Use the `pop()` method to remove an element from any position in a list
- Use the `remove()` method to remove an element from a list by value
- Two lists can be concatenated using the plus (+) operator
- Use the `list()` function to copy lists
- Use the slice operator (`:`) to extract a sublist or substrings

Using Lists With Functions

SECTION 6.4

Using Lists With Functions

- A function can accept a [list as an argument](#)
- The following function visits the list elements, but it does not modify them

```
def sum(values) :  
    total = 0  
    for element in values :  
        total = total + element  
    return total
```

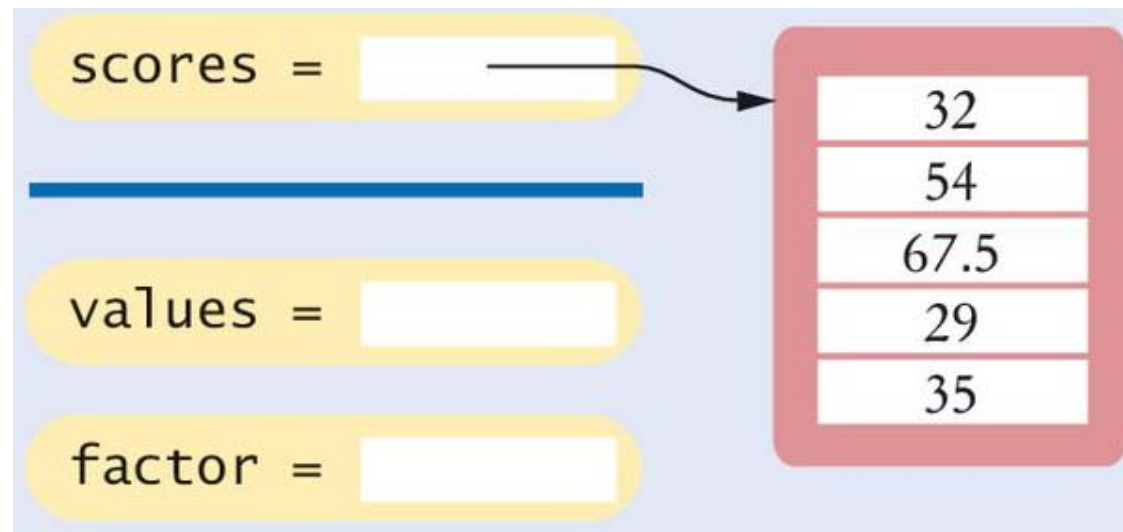
Modifying List Elements

- The following function **multiplies all elements of a list** by a given factor:

```
def multiply(values, factor) :  
    for i in range(len(values)) :  
        values[i] = values[i] * factor
```

Example: Step 1

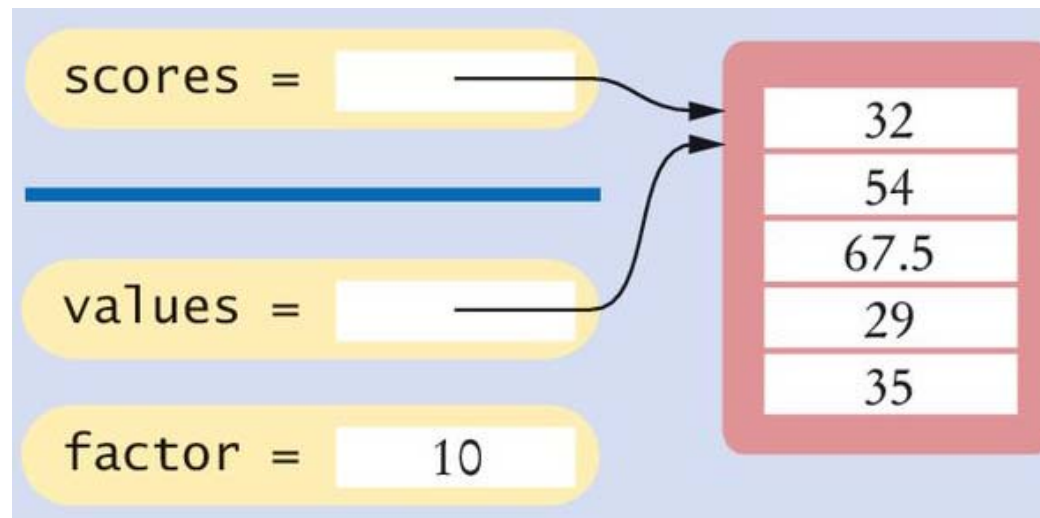
- The parameter variables `values` and `factor` are created



Example: Step 2

- The parameter variables are initialized with the arguments that are passed in the call
- In our case, `values` is set to `scores` and `factor` is set to 10
 - Note that `values` and `scores` are references to the *same* list

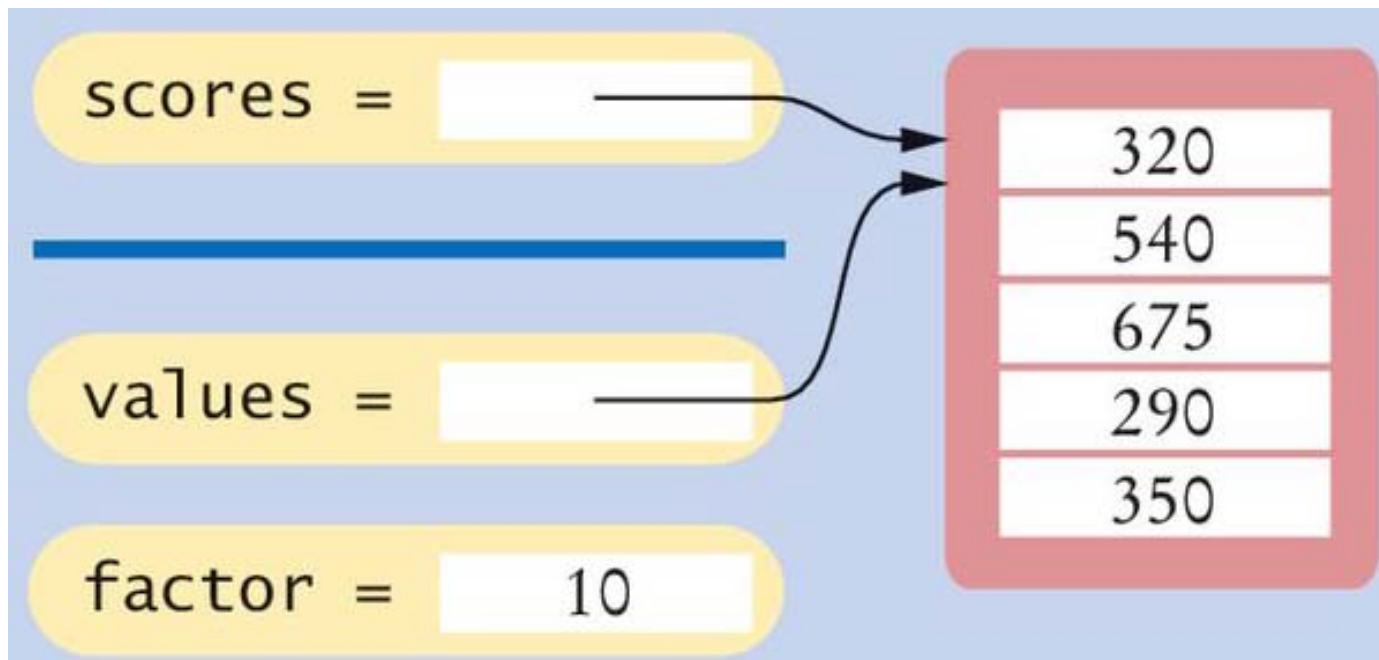
```
# Function call  
multiply(scores, 10)
```



Example: Step 3

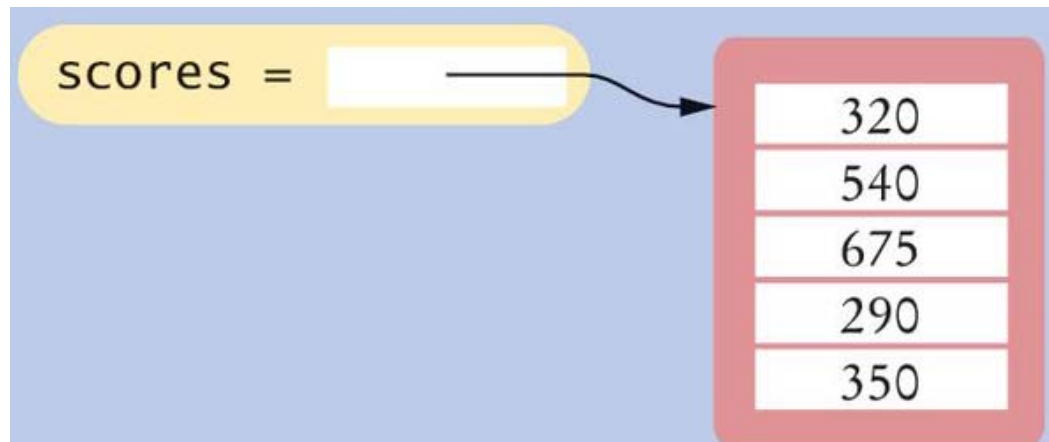
- The function multiplies all list elements by 10

```
def multiply(values, factor) :  
    for i in range(len(values)) :  
        values[i] = values[i] * factor
```



Example: Step 4

- The function returns. Its parameter variables are removed
- However, scores still refers to the list with the modified elements



Returning Lists From Functions

- Simply build up the result in the function and return it
- In this example, the `squares()` function returns a list of squares from 0^2 up to $(n - 1)^2$:

```
def squares(n) :  
    result = []  
    for i in range(n) :  
        result.append(i * i)  
    return result
```

Call By: Value vs. Reference

- Call by value:
 - When the contents of a variable that was passed to a function can never be changed by that function
- Call by reference:
 - Function can change the arguments of a method call
 - A Python method can mutate the contents of a list when it receives an reference to

Tuples

- A tuple is similar to a list, but once created, its contents cannot be modified (a tuple is an immutable version of a list).
- A tuple is created by specifying its contents as a comma-separated sequence. You can enclose the sequence in parentheses:

```
triple = (5, 10, 15)
```

- If you prefer, you can omit the parentheses:

```
triple = 5, 10, 15
```

Returning Multiple Values

- It is common practice in Python, however, to use tuples to return multiple values.

```
# Function definition
def readDate() :
    print("Enter a date:")
    month = int(input(" month: "))
    day = int(input(" day: "))
    year = int(input(" year: "))
    return (month, day, year) # Returns a tuple.

# Function call: assign entire value to a tuple
date = readDate()

# Function call: use tuple assignment:
(month, day, year) = readDate()
```

Tables

SECTION 6.7

Tables

- Lists can be used to store data in two dimensions (2D) like a spreadsheet
 - Rows and Columns
 - Also known as a 'matrix'

	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Figure 10 Figure Skating Medal Counts

Creating Tables

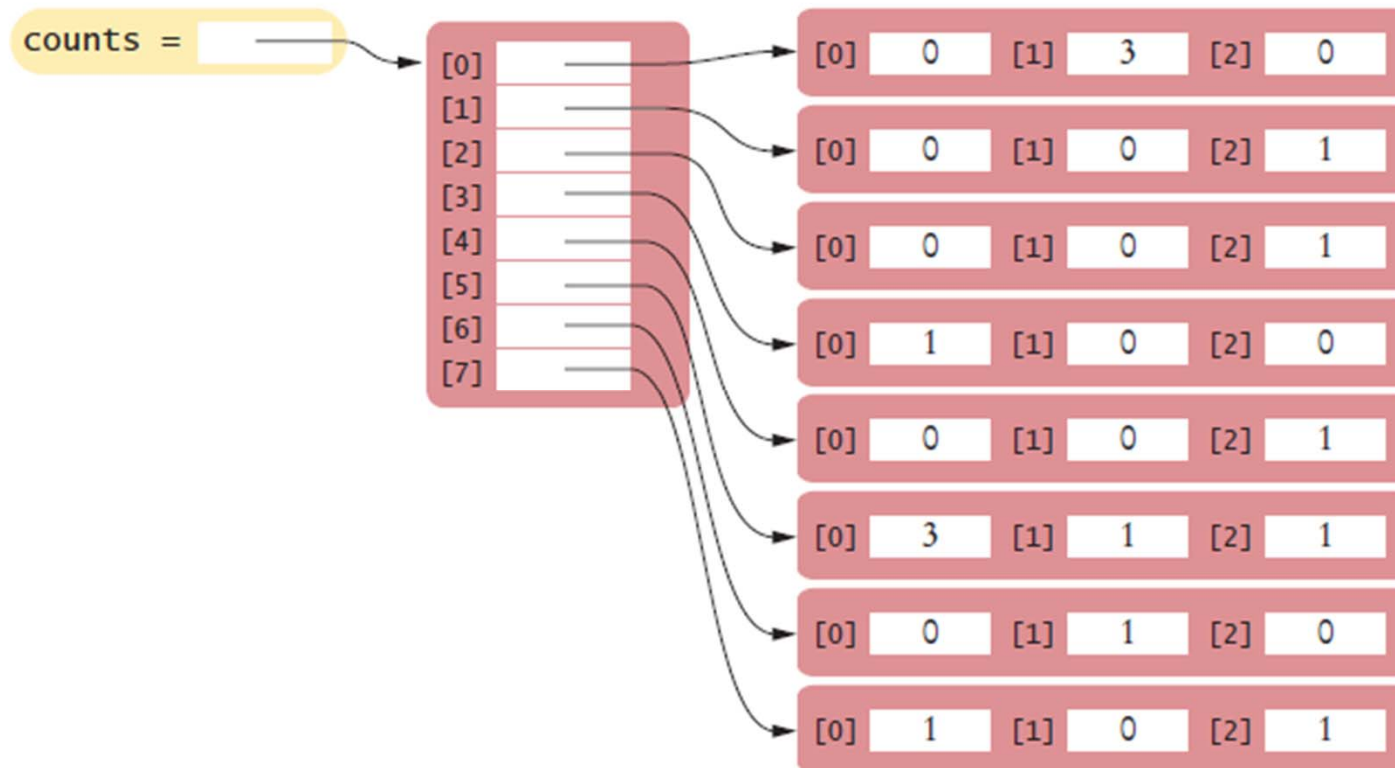
- Here is the code for creating a table that contains 8 rows and 3 columns, which is suitable for holding our medal count data:

```
COUNTRIES = 8
MEDALS = 3

counts = [
    [ 0, 3, 0 ],
    [ 0, 0, 1 ],
    [ 0, 0, 1 ],
    [ 1, 0, 0 ],
    [ 0, 0, 1 ],
    [ 3, 1, 1 ],
    [ 0, 1, 0 ],
    [ 1, 0, 1 ]
]
```

Creating Tables (2)

- This creates a list in which each element is itself another list:



Creating Tables (3)

- Sometimes, you may need to create a table with a size that is too large to initialize with literal values
- First, create a list that will be used to store the individual rows

```
table = []
```

Creating Tables (4)

- Then create a new list using replication (with the number of columns as the size) for each row in the table and append it to the list of rows:

```
ROWS = 5
COLUMNS = 20
for i in range(ROWS) :
    row = [0] * COLUMNS
    table.append(row)
```

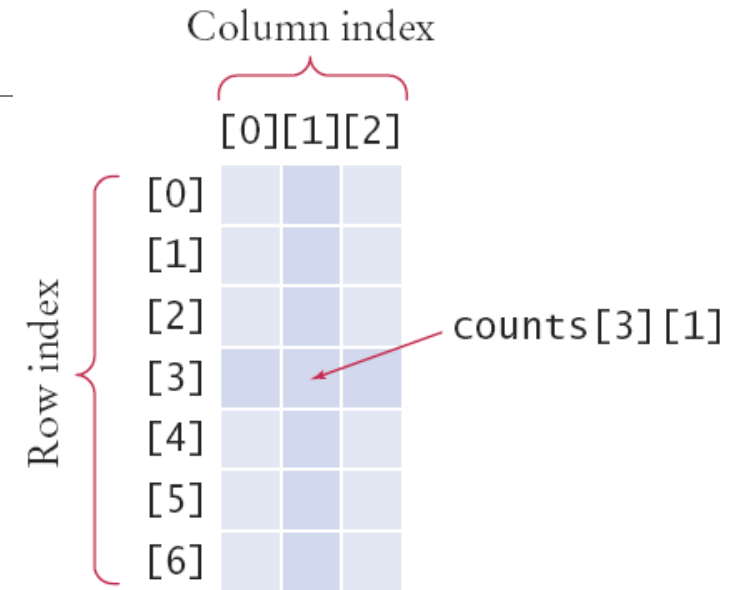
- The result is a table that consists of 5 rows and 20 columns

Accessing Elements

- Use two index values:
 - Row then column

```
medalCount = counts[3][1]
```

- To print
 - Use nested for loops
 - Outer row(i) , inner column(j) :



```
for i in range(COUNTRIES):  
    # Process the ith row  
    for j in range(MEDALS) :  
        # Process the jth column in the ith row  
        print("%8d" % counts[i][j], end="")  
    print() # Start a new line at the end of the row
```

Locating Neighboring Elements

- Some programs that work with two-dimensional lists need to locate the elements that are adjacent to an element
- This task is particularly common in games
- You are at loc i, j
- Watch out for edges!
 - No negative indexes!
 - Not off the 'board'

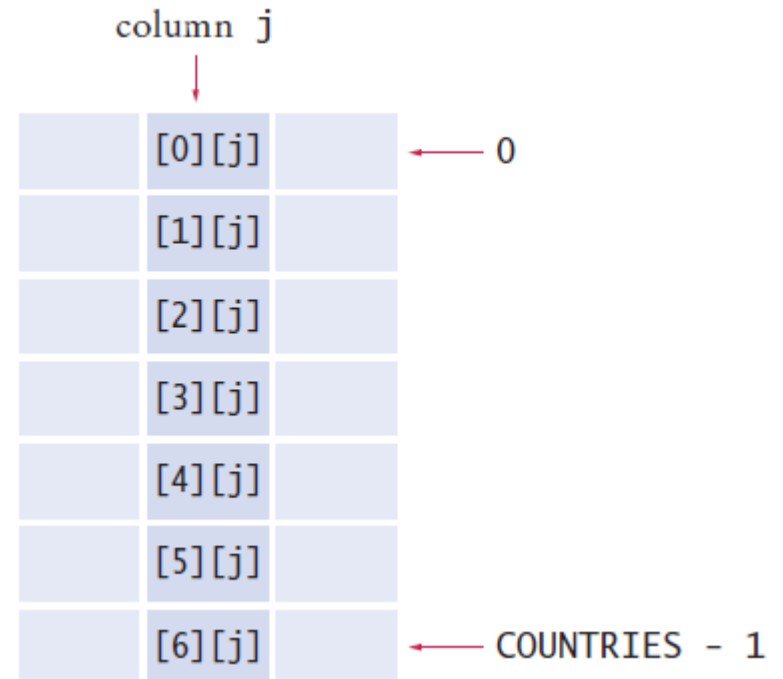
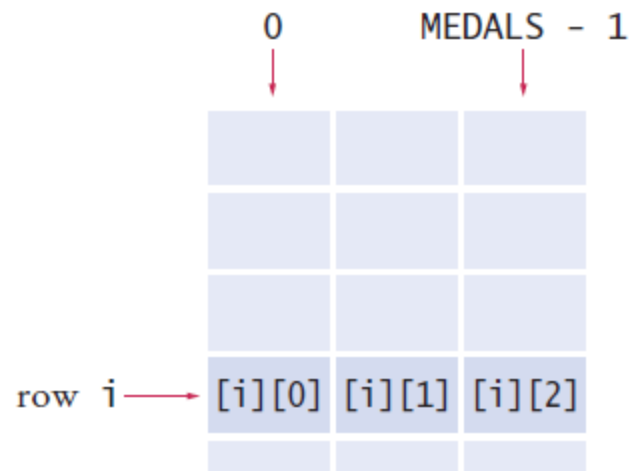
$[i - 1][j - 1]$	$[i - 1][j]$	$[i - 1][j + 1]$
$[i][j - 1]$	$[i][j]$	$[i][j + 1]$
$[i + 1][j - 1]$	$[i + 1][j]$	$[i + 1][j + 1]$

Adding Rows and Columns

- Rows (x)

Columns (y)

```
total = 0
for j in range(MEDALS):
    total = total + counts[i][j]
```



```
total = 0
for i in range(MEDALS):
    total = total + counts[i][j]
```

Using Tables With Functions

- When you pass a table to a function, you will want to recover the dimensions of the table. If `values` is a table, then:

`len(values)` is the number of rows

`len(values[0])` is the number of columns

- For example, the following function computes the sum of all elements in a table:

```
def sum(values) :  
    total = 0  
    for i in range(len(values)) :  
        for j in range(len(values[0])) :  
            total = total + values[i][j]  
    return total
```


Summary

Summary: Lists

- A list is a container that stores a sequence of values
- Each individual element in a list is accessed by an integer index i , using the notation `list[i]`
- A list index must be at least zero and less than the number of elements in the list
- An out-of-range error, which occurs if you supply an invalid list index, can cause your program to terminate
- You can iterate over the index values or the elements of a list

Summary: Lists

- A list reference specifies the location of a list. Copying the reference yields a second reference to the same list
- A linear search inspects elements in sequence until a match is found
- Use a temporary variable when swapping elements
- Lists can occur as function parameters and return values

Summary: Lists

- When calling a function with a list argument, the function receives a list reference, not a copy of the list
- A tuple is created as a comma-separated sequence enclosed in parentheses
- By combining fundamental algorithms, you can solve complex programming tasks
- You should be familiar with the implementation of fundamental algorithms so that you can adapt them

Summary: Lists

- Use a two-dimensional list to store tabular data
- Individual elements in a two-dimensional list are accessed by using two index values, `table[i][j]`

Built-In Operations For Lists

- Use the `insert()` method to insert a new element at any position in a list
- The `in` operator tests whether an element is contained in a list
- Use the `pop()` method to remove an element from any position in a list
- Use the `remove()` method to remove an element from a list by value
- Two lists can be concatenated using the plus (+) operator
- Use the `list()` function to copy lists

Built-In Operations For Lists

- Use the slice operator (:) to extract a sublist or substrings