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

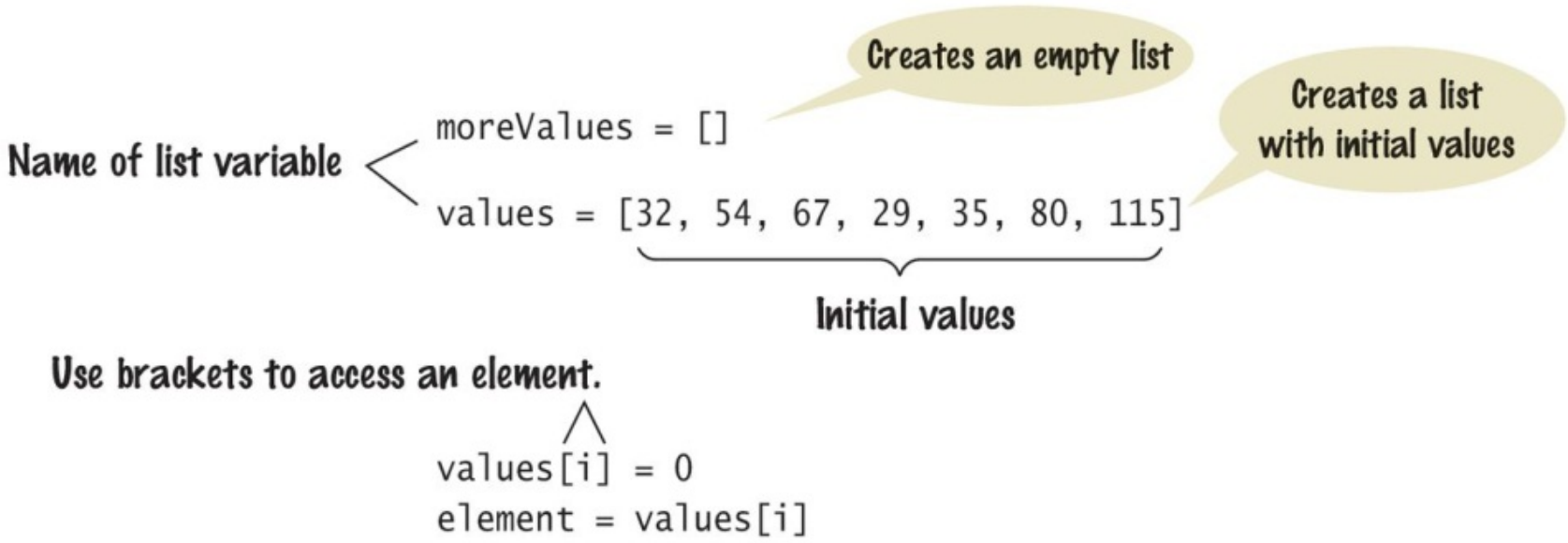
Creating a List

- Specify a list variable with the subscript operator []

Syntax

To create a list: $[value_1, value_2, \ldots]$

To access an element: *listReference*[*index*]



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

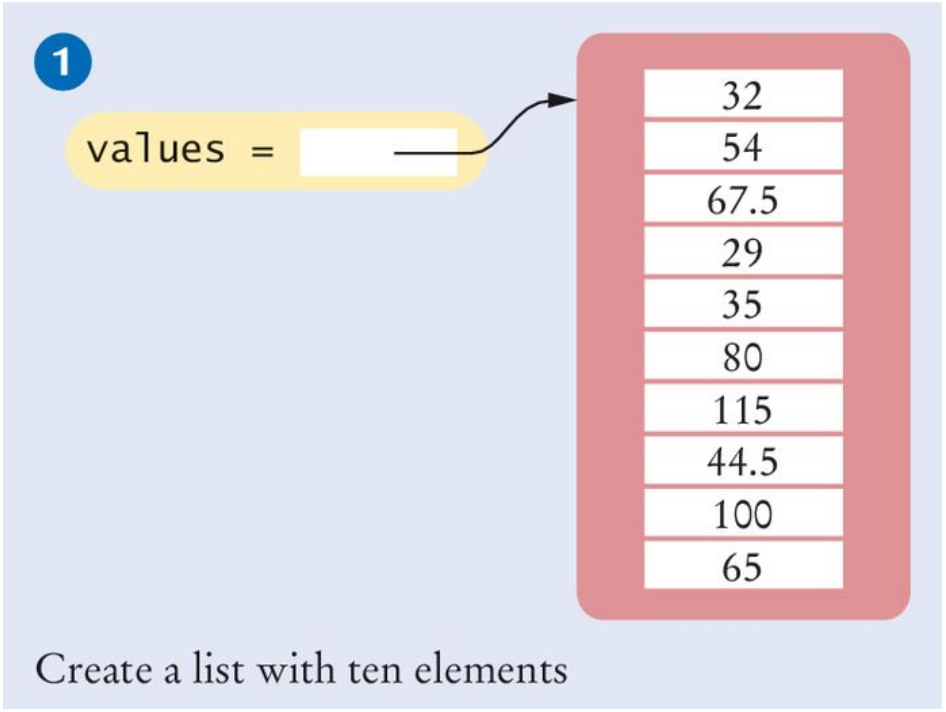
Accessing list elements

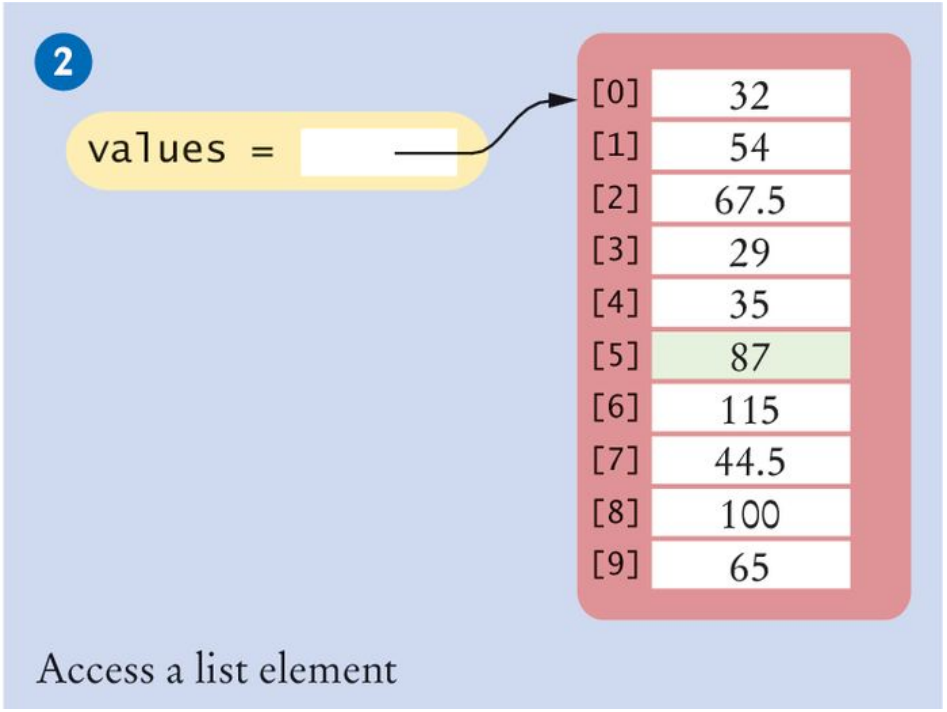
```
print(values[5])
```

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])
```

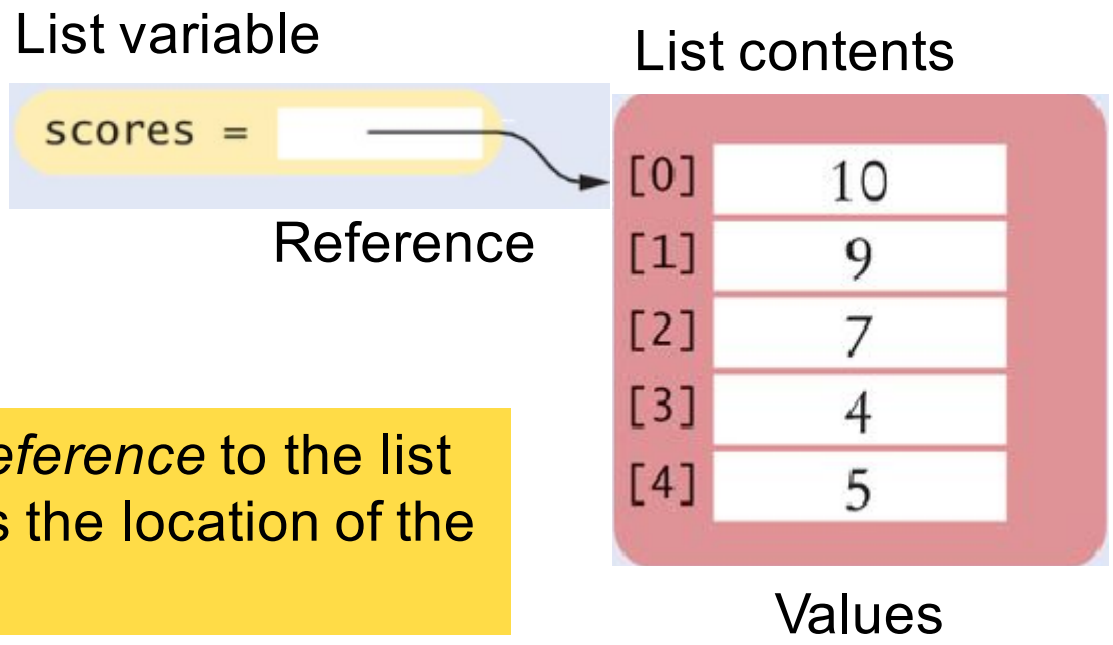
```
# Better version (list index used)
for i in range(len(values)) :
    print(i, values[i])
```

```
# 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]
```



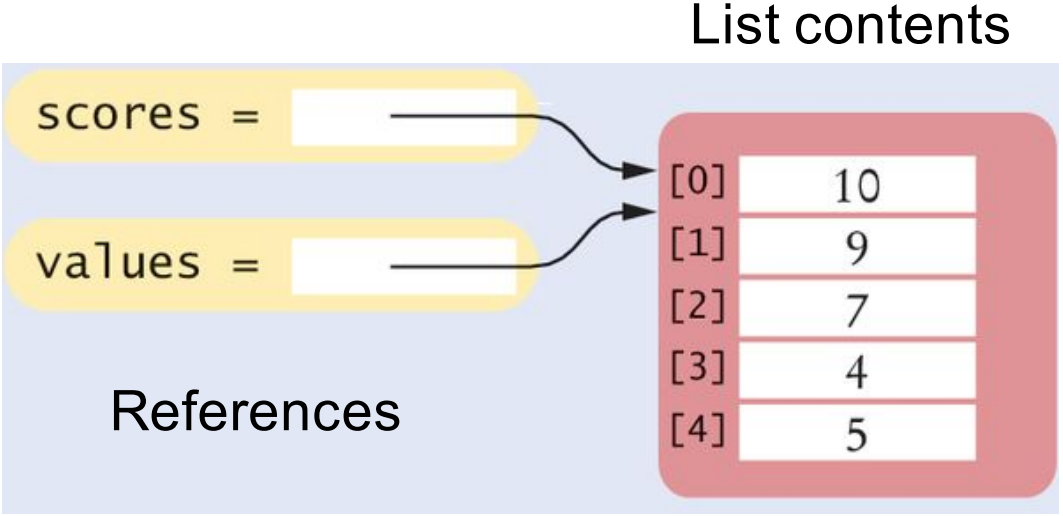
A list variable contains a *reference* to the list contents. The *reference* is the location of the list contents (in memory).

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
```

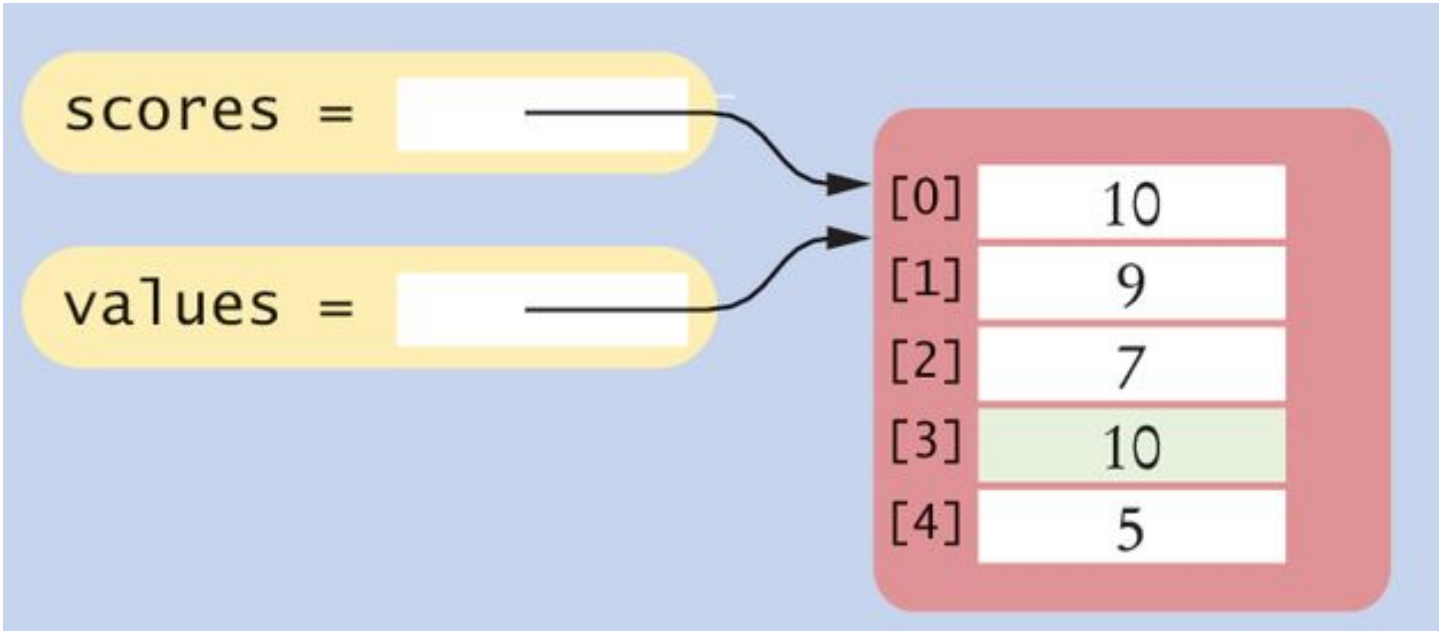
A list variable specifies the location of a list. Copying the reference yields a second reference to the same list.



Modifying Aliased Lists

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

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

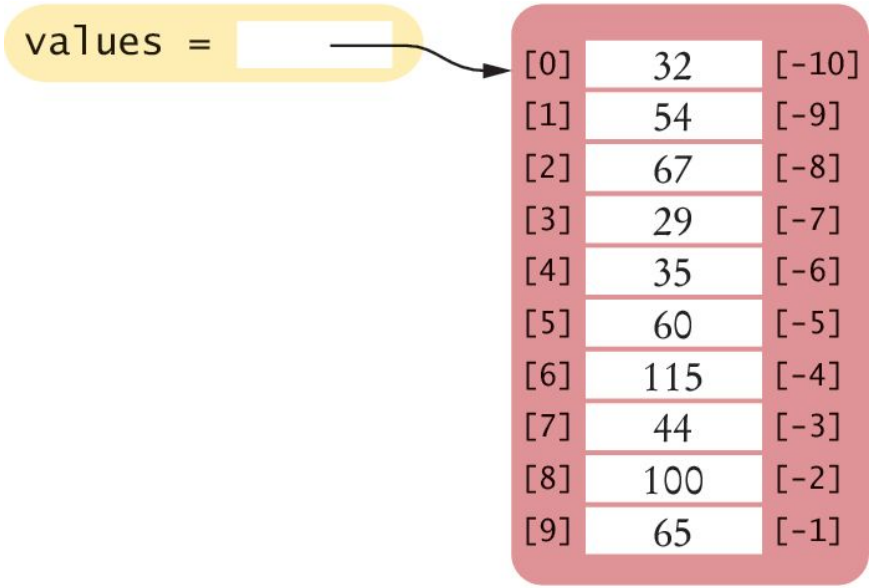


Reverse Subscripts

- 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)
```



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

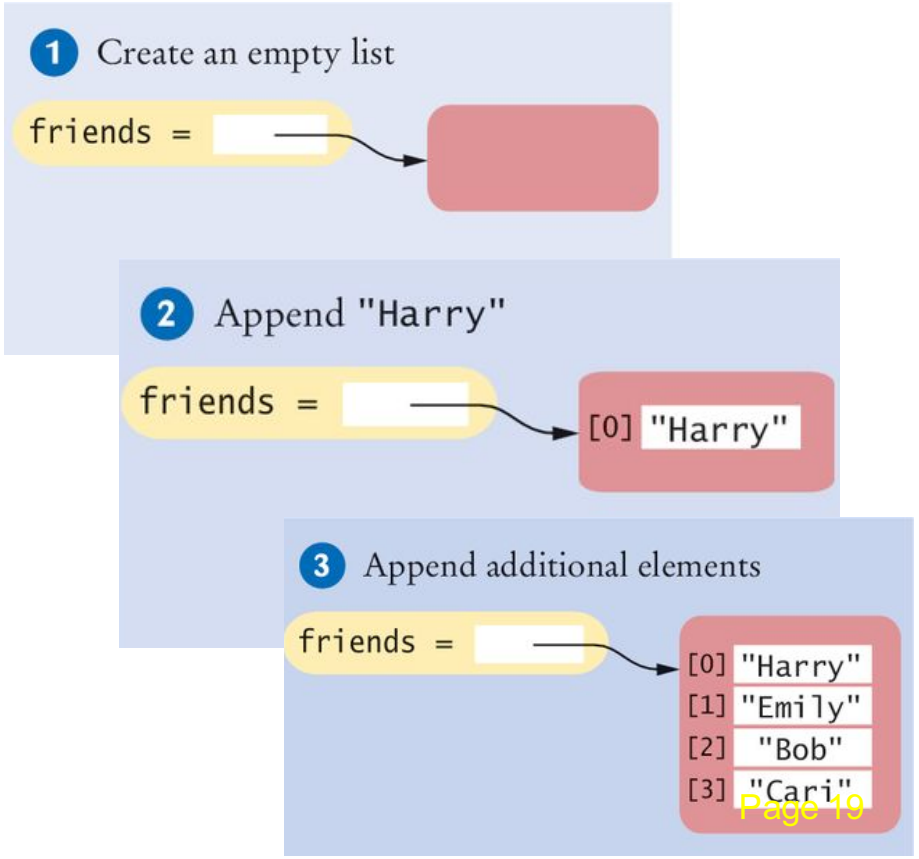
Appending Elements

- 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")
```

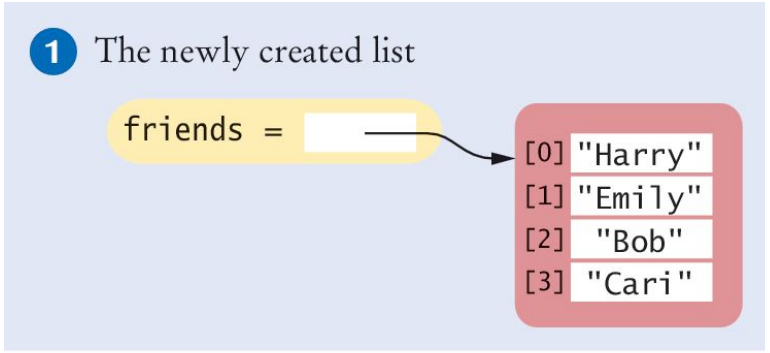


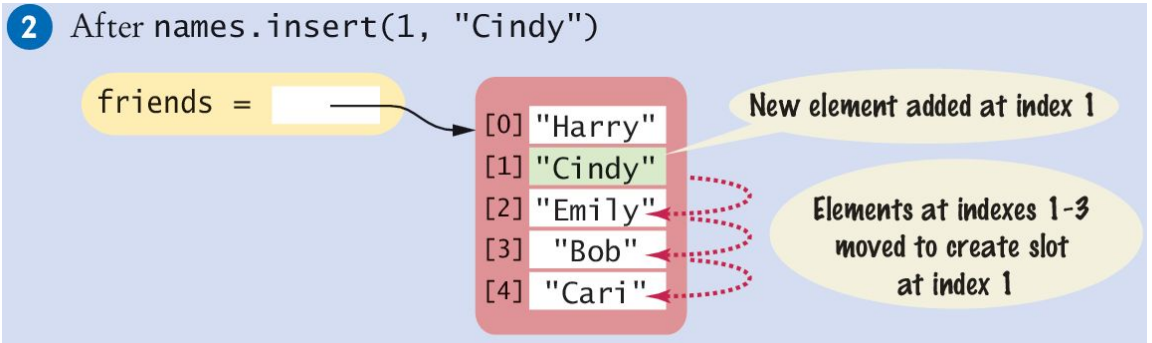
Inserting an Element

- 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

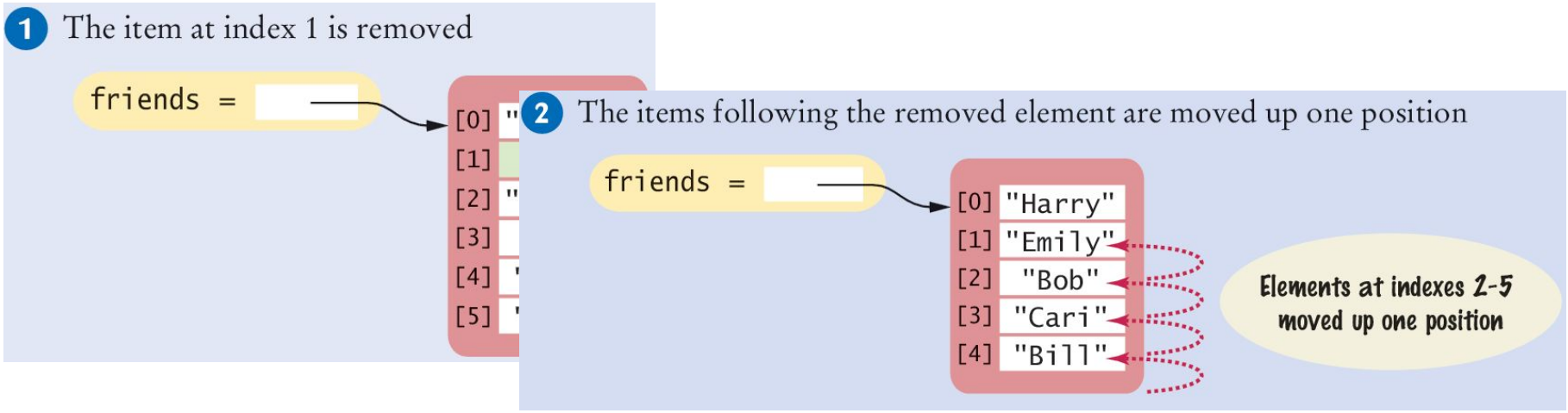
```
friends = ["Harry", "Emily", "Bob", "Cari", "Emily"]
n = friends.index("Emily") # Sets n to 1
```

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



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

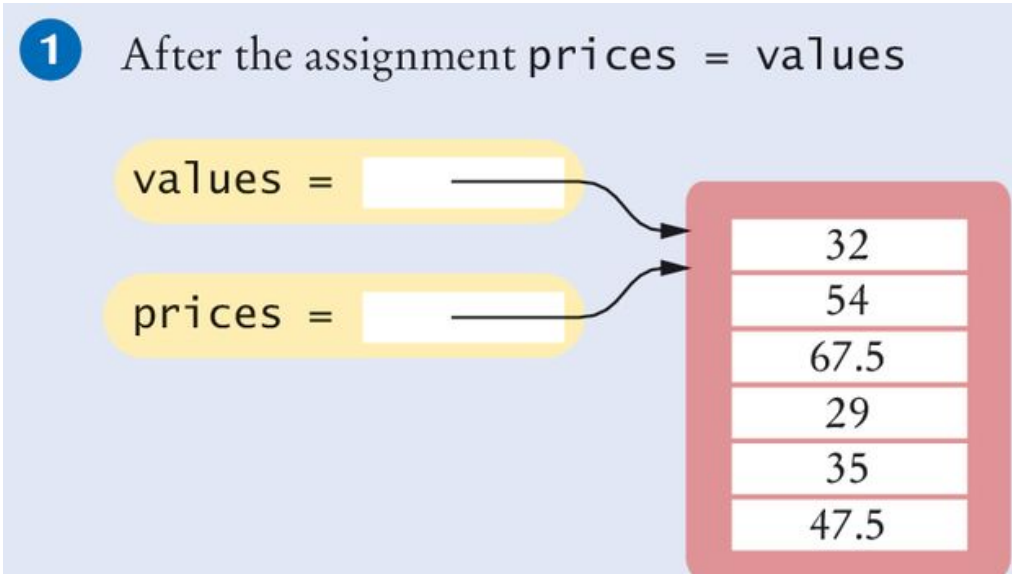
- The sort() method sorts a list of numbers or strings.

```
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:

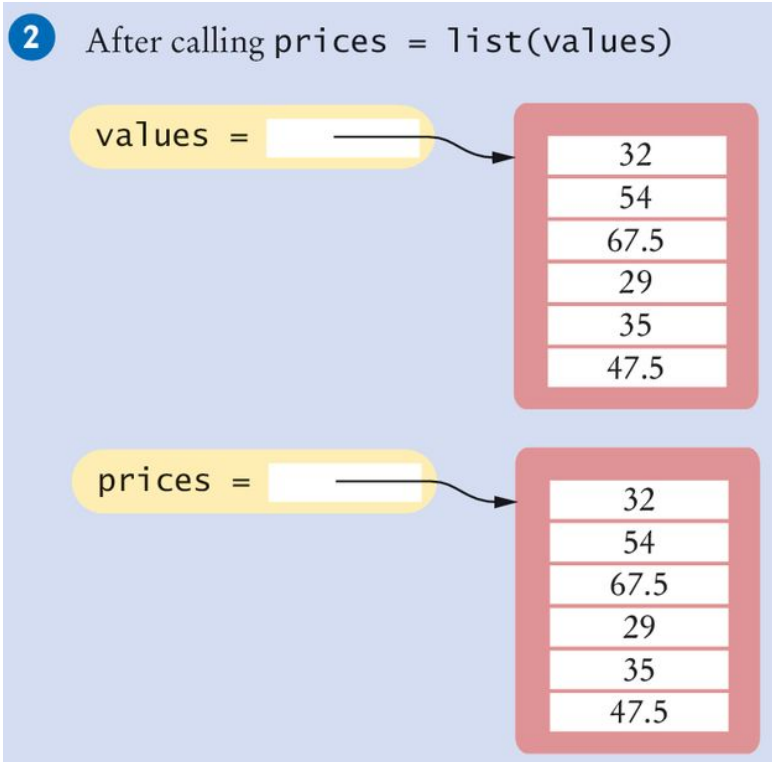
```
prices = values
```



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
`[]` `[`$elem_1$, $elem_2$, ..., $elem_n$`]`	Creates a new empty list or a list that contains the initial elements provided.
`len(`*l*`)`	Returns the number of elements in list *l*.
`list(`*sequence*`)`	Creates a new list containing all elements of the sequence.
`values * num`	Creates a new list by replicating the elements in the `values` list `num` times.
`values + moreValues`	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[from : 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.)
`sum(l)`	Computes the sum of the values in list *l*.
`min(l)` `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
l.pop() *l*.pop(*position*)	Removes the last element from the list or from the given position. All elements following the given position are moved up one place.
l.insert(*position*, *element*)	Inserts the element at the given position in the list. All elements at and following the given position are moved down.
l.append(*element*)	Appends the element to the end of the list.
l.index(*element*)	Returns the position of the given element in the list. The element must be in the list.
l.remove(*element*)	Removes the given element from the list and moves all elements following it up one position.
l.sort()	Sorts the elements in the list from smallest to largest.

Common List Algorithms

SECTION 6.3

Common List Algorithms

- 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
```

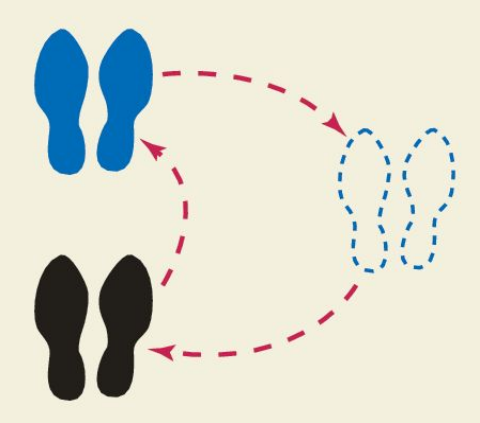
Removing Matches

- 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

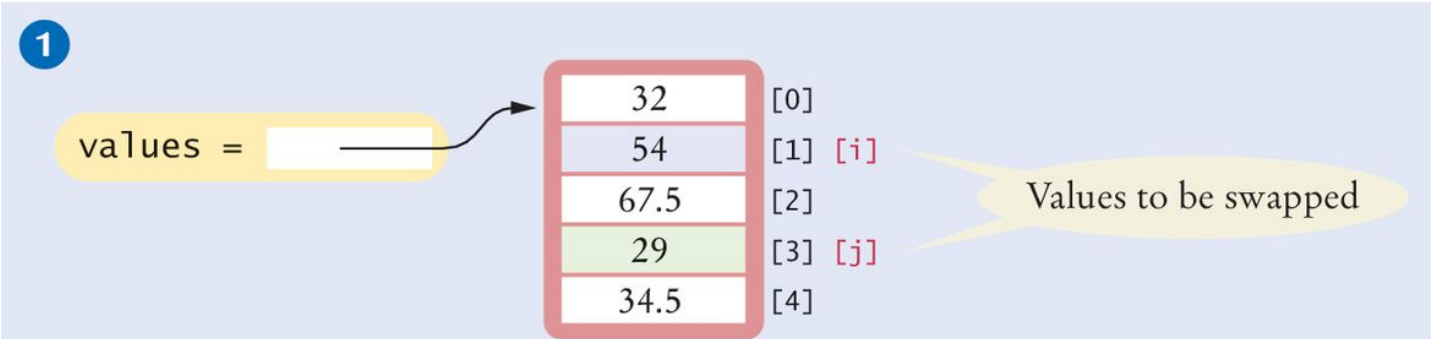
- For example, you can sort a list by repeatedly swapping elements that are not in order
- Swap the elements at positions i and j of a list values
- We'd like to set values[i] to values[j]. But that overwrites the value that is currently stored in values[i], so we want to save that first:



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)

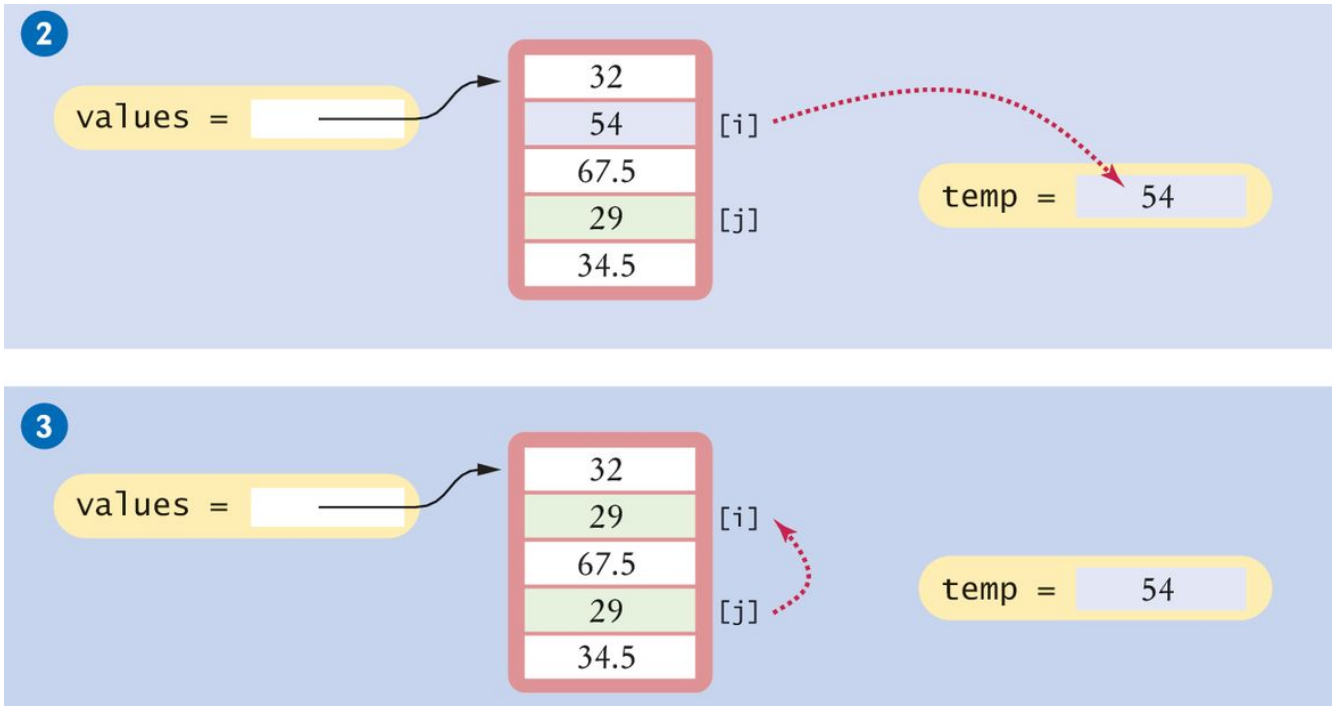
- Swapping elements [1] and [3]
  - This sets up the scenario for the actual code that will follows



Swapping Elements (3)

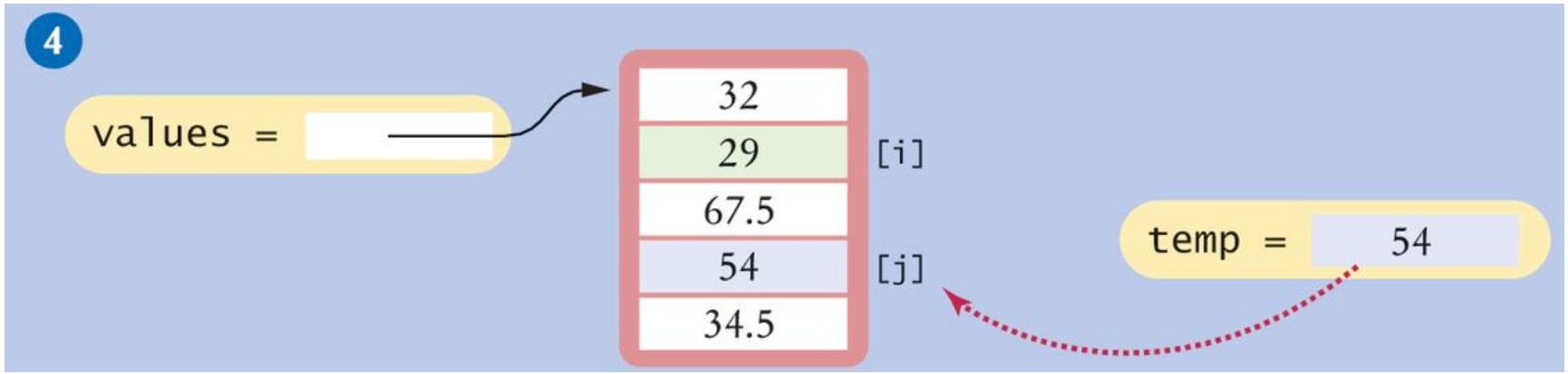
```
# Step 2
temp = values[i]

# 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

Example One

- Open the file largest.py in Wing

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

Example Problems

- Open the file largest.py in Wing
- Modify the program to find and print both the largest and smallest number
  - Find the largest number
  - Print the list
    - Print the string " <== largest value“ next to the largest number
  - Find the smallest number
  - Print the list
    - Print the string " <== smallest value“ next to the smallest number
- Modify the program again
  - Find the largest number
  - Find the smallest number
  - Print the list
    - Print the string " <== largest value“ next to the largest number
    - Print the string " <== smallest value“ next to the smallest number

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
```

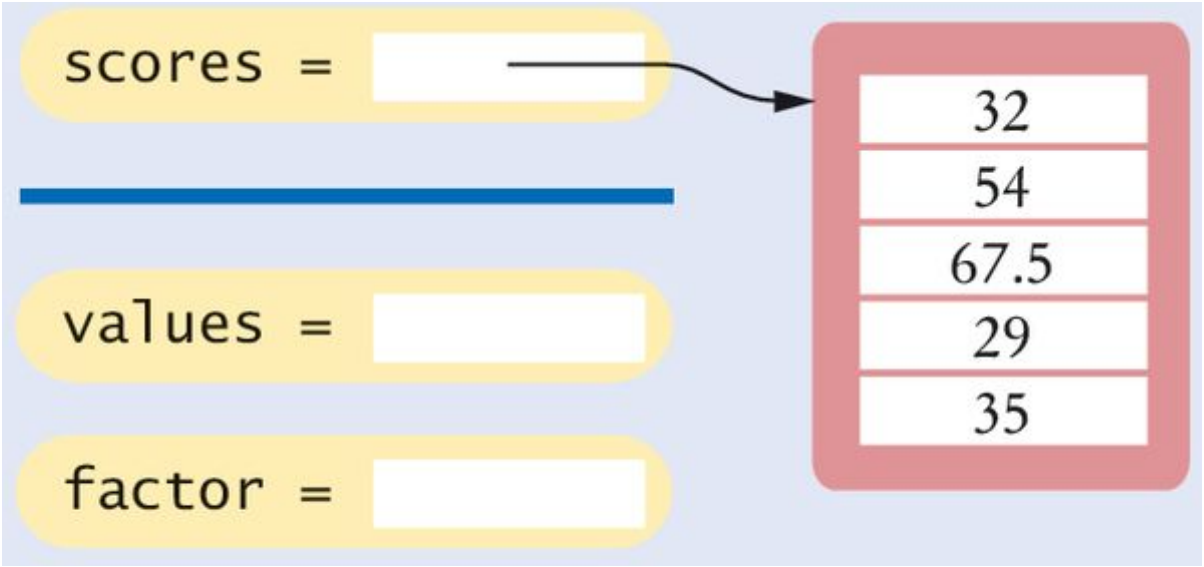
Modifying List Elements

- 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

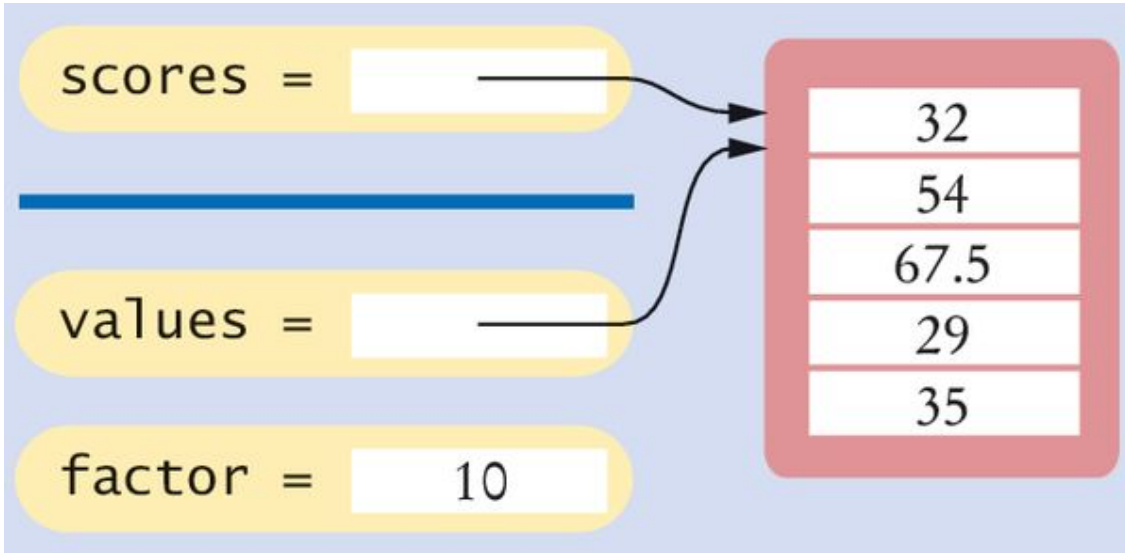
- The parameter variables `values` and `factor` are created



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

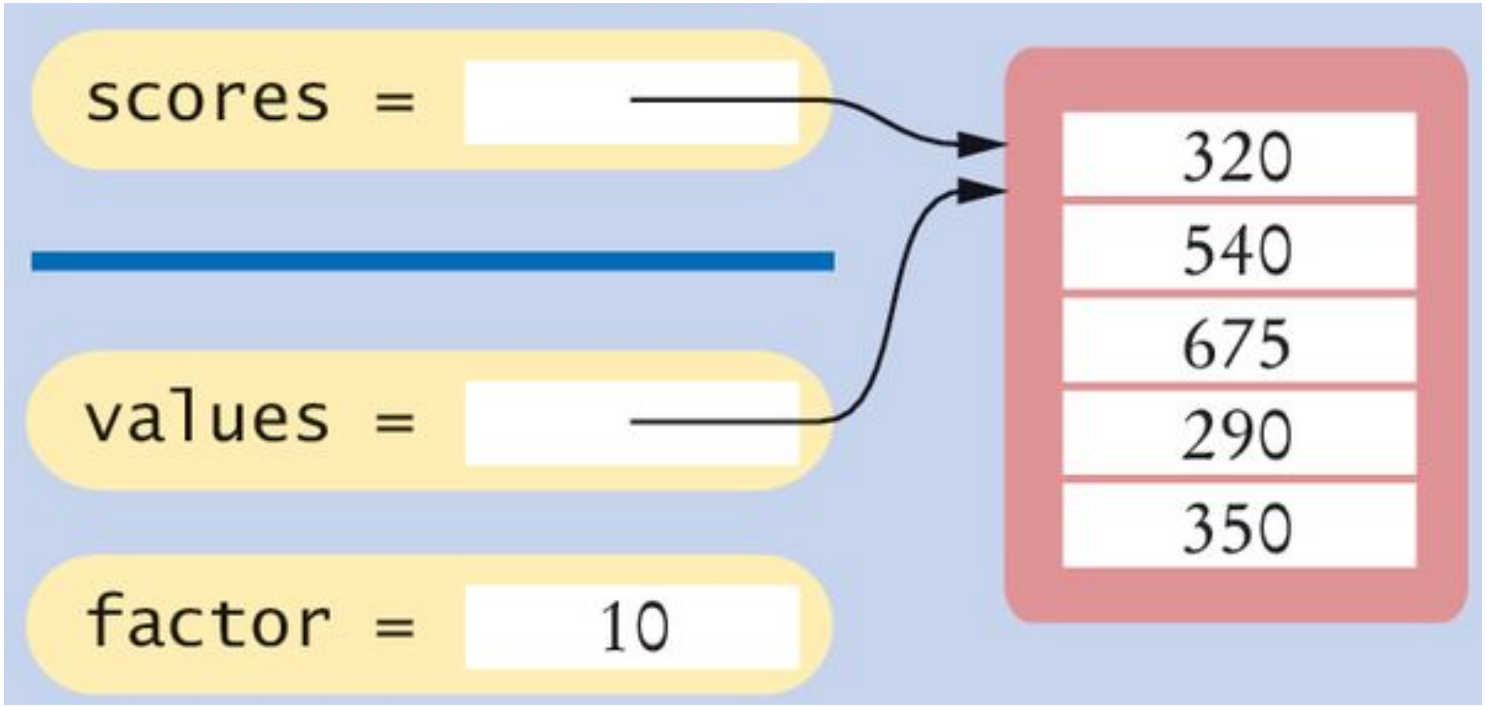
```
# Function call
multiply(scores, 10)
```



Example: Step 3

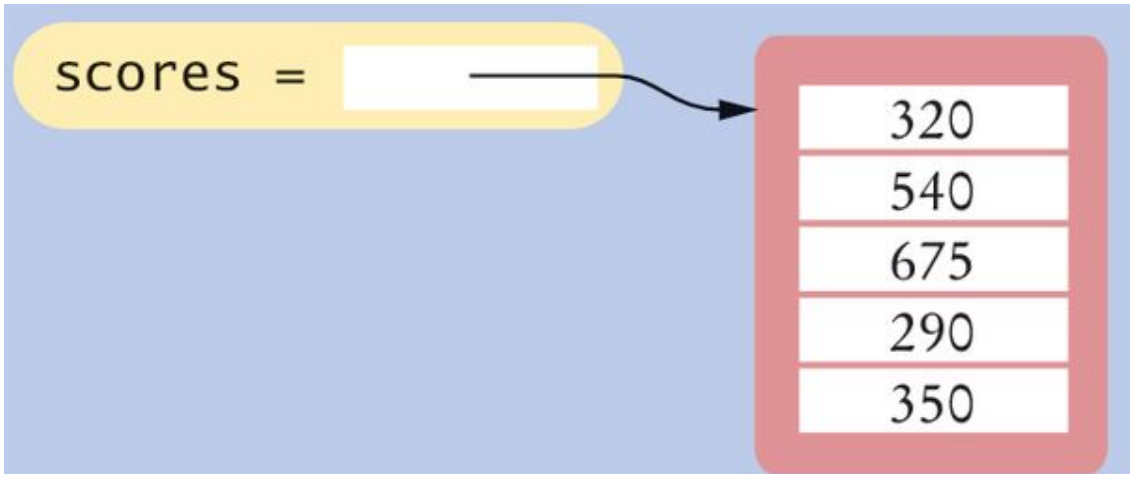
- The function multiplies all list elements by 10

```
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
```

Example One

- Open the file reverse.py
- This program reads values from the user, multiplies them by 10, and prints them in reverse order
- The readFloats function returns a list
- The multiply function has a list argument, it modifies the list elements
- The printReversed function has a list argument, but it does not modify the list elements

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()
```

Problem Solving

SECTION 6.5: ADAPTING ALGORITHMS

Adapting Algorithms

- Consider this example problem: You are given the quiz scores of a student. You are to compute the final quiz score, which is the sum of all scores after dropping the lowest one
  - For example, if the scores are

    8 7 8.5 9.5 7 5 10
  - then the final score is 50

Adapting a Solution

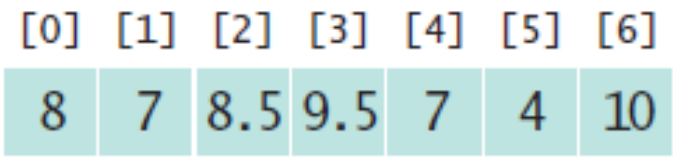
- What steps will we need?
  - Find the minimum
  - Remove it from the list
  - Calculate the sum

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	7	8.5	9.5	7	4	10

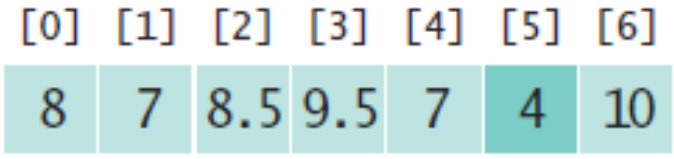
- What tools do we know?
  - Finding the minimum value (Section 6.3.4)
  - Removing matches (Section 6.3.7)
  - Calculating the sum (Section 6.4)
- But wait… We need to find the POSITION of the minimum value, not the value itself
  - Hmmm. Time to adapt

Planning a Solution

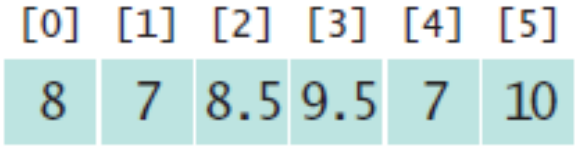
- Refined Steps:
  - Find the minimum value
  - Find its position
  - Remove it from the list
  - Calculate the sum



- Let's try it
  - Find the position of the minimum:
    - At position 5



  - Remove it from the list
  - Calculate the sum



Adapting the code

- Adapt smallest value to smallest position:

Original algorithm

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

Adapted algorithm

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

Working Out an Example

Problem Statement: The final quiz score for a student is computed by adding up all of the scores except the lowest two

For example, if the scores are: 8, 4, 7, 9, 9, 7, 5, 10

The final score is 50

We are going develop the algorithm and write a program to compute the final score

Step One

- We want to start with a high level decomposition of the problem:
  - Read the data into a list
  - Process the data
  - Display the results
- We will refer back to the algorithms and list operations to help guide our design. Most of the tasks associated with this problem can be solved by using or adapting one or more of the algorithms
- Our next step in the stepwise refine is to identify the step we need to process the data:
  1. Read inputs
  2. Remove the minimum
  3. Remove the minimum again
  4. Calculate the sum

Step Two

- Now we start to determine the algorithms we need
- We have working algorithms for reading the inputs, and calculating the sum
- To remove the minimum value we can find the minimum (we have an algorithm for that) and remove it.
  - It is a bit more efficient to find the position of the minimum value and “pop” that position

Step Three

- Plan the functions we need
  - We can compute the sum with the existing sum function
  - We need a function to read the floating point numbers; readFloats()
  - We need a function to remove the minimum; removeMinimum() (we will call this twice)
- Our main function can be structured as:

```
scores = readFloats()
removeMinimum(scores)
removeMinimum(scores)
total = sum(scores)
print("Final Score : ", total)
```

Step Four

- Assemble and test your code
- Review your code and make sure you handle the “normal” and “exceptional” cases.
  - How do you handle an empty list?
  - A list with a single element?
  - What if you don’t find a smallest number?
- Remember in our problem statement we are dropping two grades
- It is not possible to compute a minimum if the list is empty or has a single element
  - In that case we should terminate the program with an error message ***before*** attempting to call the remove minimum function
- Develop your test cases, and the expected outputs

Testing

- Develop your test cases, and the expected outputs

Test Case	Expected Output	Comment
8 4 7 8.5 9.5 7 5 10	50	Example case
8 7 7 7 9	24	Make sure only two instances of the low score are removed
8 7	0	After removing the two low scores, none remain
(no inputs)	Error	That is not a legal input

scores.py

- Open the file scores.py in Wing

A Second Example

Problem Statement: Our task is to analyze whether a die is fair by counting how often each value (1, 2, 3, 4, 5, 6) appears

Our input will be a series of die toss values

For example, if the scores are: 1, 2, 1, 3, 4, 6, 5, 6

The result is 1: 2; 2: 1; 3: 1; 4: 1; 5: 1; 6: 2

We are going develop the algorithm and write a program to compute and print the frequency of each die value

Step One

- We want to start with a high level decomposition of the problem:
  - Read the die values
  - Count how often the values (1, 2, …, 6) appear
  - Print the counts
- If we think about this we can simplify; do we need to store the values?
  - We are only counting the number of times each die toss occurs. If we create a list of counter we can read and then discard the inputs
- Our next step in the stepwise refine is to identify the steps we need to process the data:
  1. Read input
  2. For each input value:
     1. Increment the corresponding counter
  3. Print the counters

Step Two

- Determine the algorithms we need:
- We don't have an algorithm for reading inputs and incrementing a counter (yet) but it is easy to build one
  - If we have a list of length 6 we can simply

```
counters[value – 1] = counters[value – 1] + 1
```

- To make it easier was can can not use the [0] position and have

```
counters[value] = counters[value] + 1
```

- So, if we define counters = [0] * (sides + 1)
- Now we can focus on printing the counters
- We can use a count controlled loop and a format string to print the results

Step Three

- Plan the Functions we need:
  - countInputs(sides) # will count the inputs
  - printCounters(counters) # will print the counters
- The main function calls these functions:

```
counters = countInputs(6)
printCounters(counters)
```

Step Four

- Assemble and test your program:
- When updating a counter we have to make sure we do not generate an boundary error; we have to reject inputs < 1 and > 6

Test Case	Expected Output	Comment
1 2 3 4 5 6	1 1 1 1 1 1	Each number occurs once
1 2 3	1 1 1 0 0 0	Numbers that do not appear have a count of “0”
1 2 3 1 2 3 4	2 2 2 1 0 0	The counters must be correct
No input	0 0 0 0 0 0	All counters are “0”
0 1 2 3 4 5 6 7	ERROR	Inputs out of bounds

dice.py

- Open the file dice.py

Discovering Algorithms by Manipulating Physical Objects

SECTION 6.6

Discovering Algorithms

- Consider this example problem:
  - You are given a list whose size is an even number, and you are to switch the first and the second half
- For example, if the list contains the eight numbers:

- Rearrange it to:

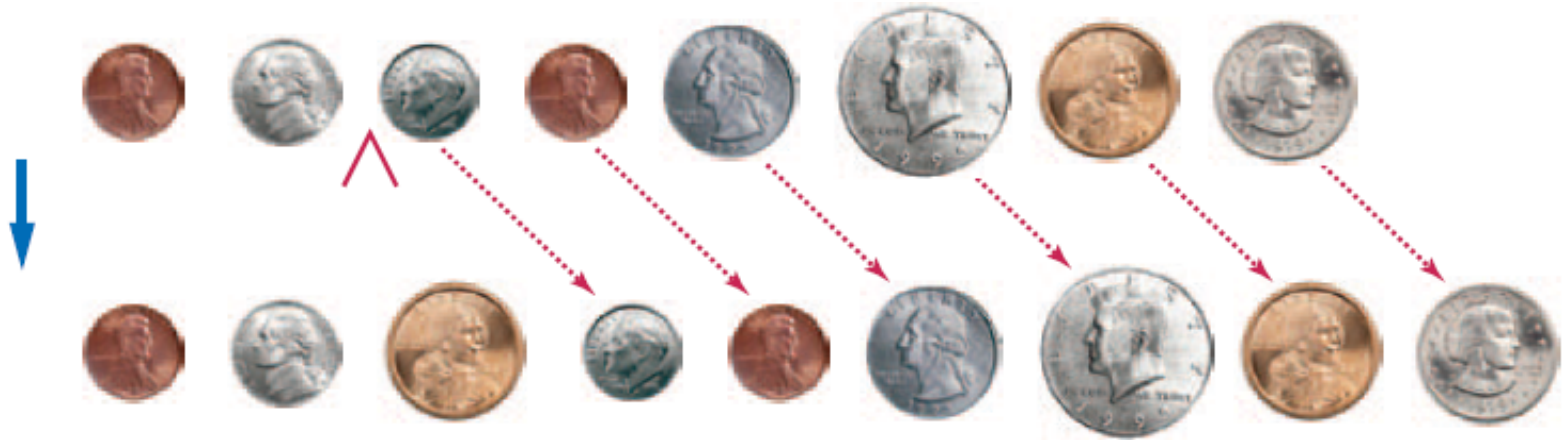
Manipulating Objects

- One useful technique for discovering an algorithm is to manipulate physical objects
- Start by lining up some objects to denote an array
  - Coins, playing cards, or small toys are good choices

- Visualize removing one object

Manipulating Objects

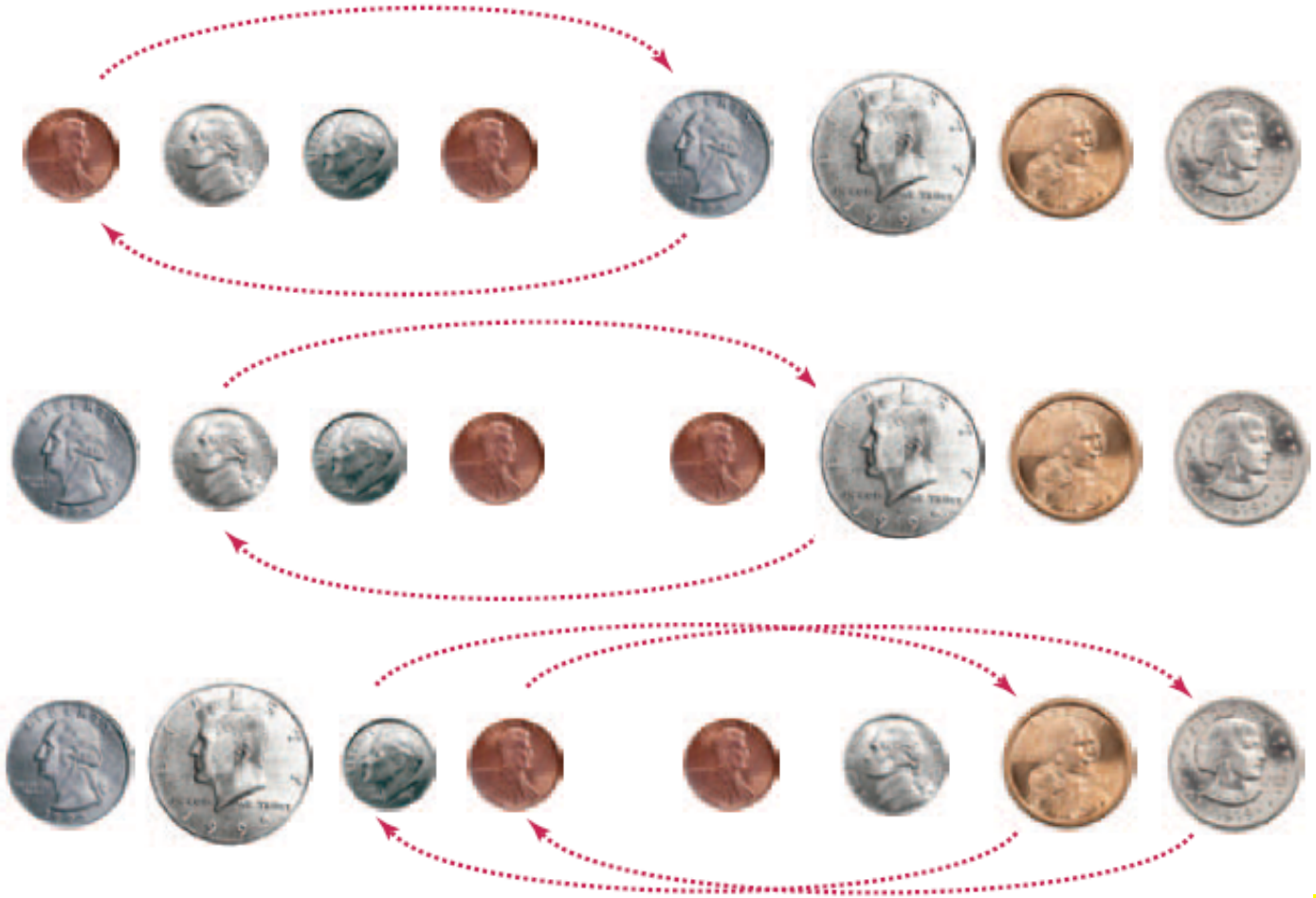
- Visualize inserting one object



- How about swapping two coins?

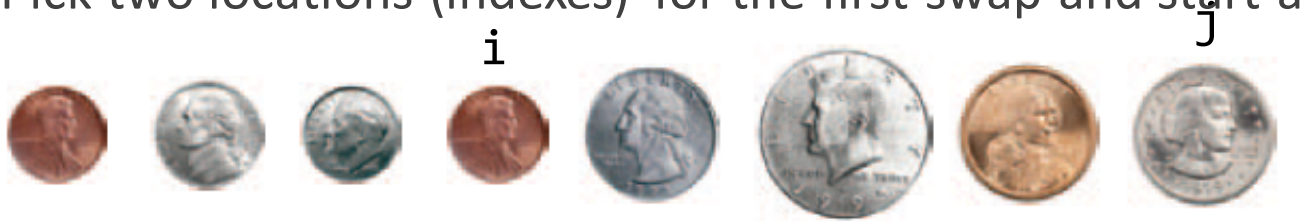
Manipulating Objects

- Back to our original problem. Which tool(s) to use?
  - How about swapping two coins? Four times?



Develop an Algorithm

- Pick two locations (indexes) for the first swap and start a loop



- How can j be set to handle any number of items?
  - … if size is 8, j is index 4… (size / 2)
- And when do we stop our loop?… Also (size / 2)

```
i = 0
j = ... (we'll think about that in a minute)
While (don't know yet)
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

```
i = 0
j = length / 2
While (i < length / 2)
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

swaphalves.py

- Open the file swaphalves.py

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

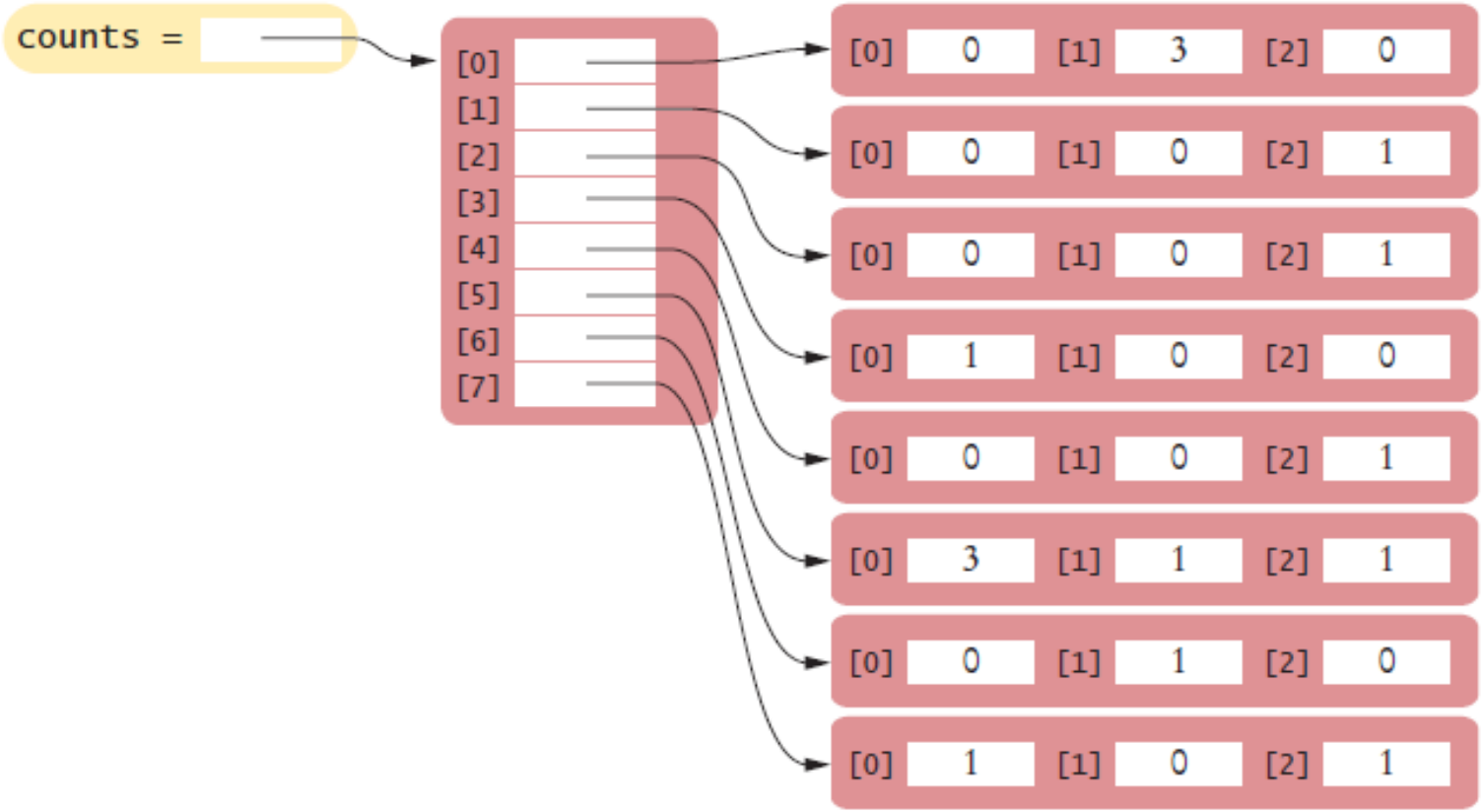
- Here is the code for creating a table that contains 8 rows and 3 columns, which is suitable for holding our medal count data:

```
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:

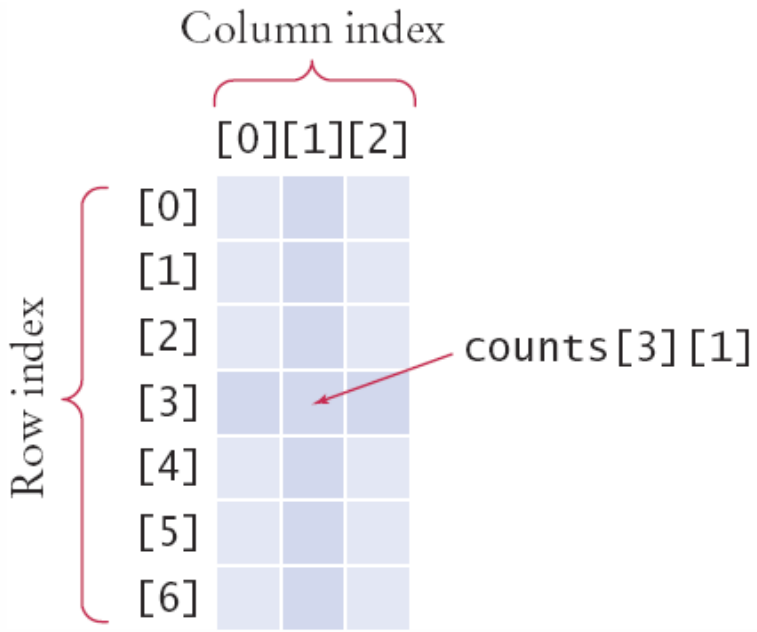
```
ROWS = 5
COLUMNS = 20
for i in range(ROWS) :
    row = [0] * COLUMNS
    table.append(row)
```

- 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]
```

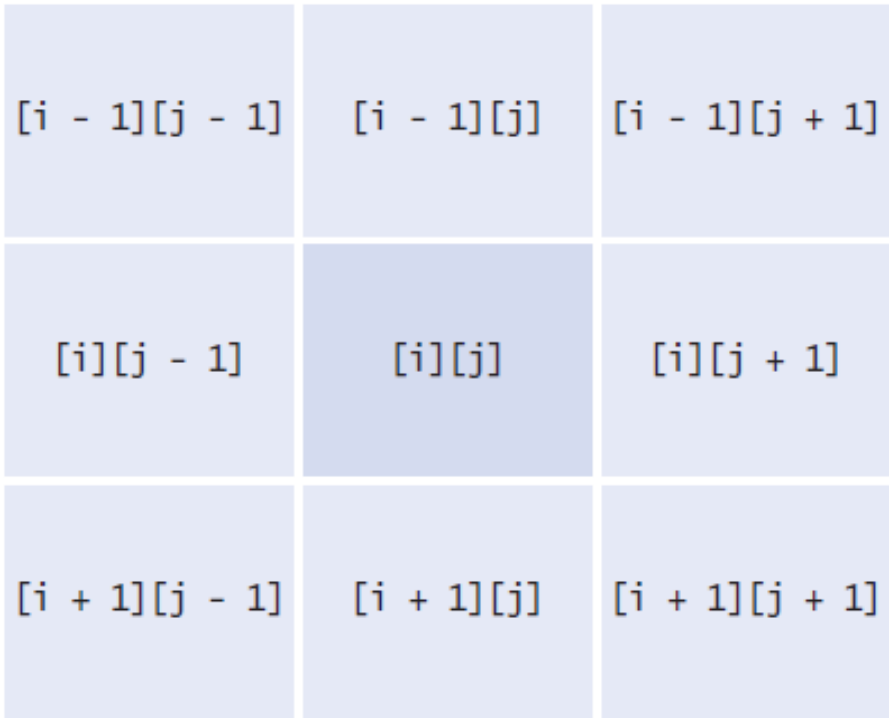


- To print
  - Use nested for loops
  - Outer row(i) , inner column(j) :

```
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

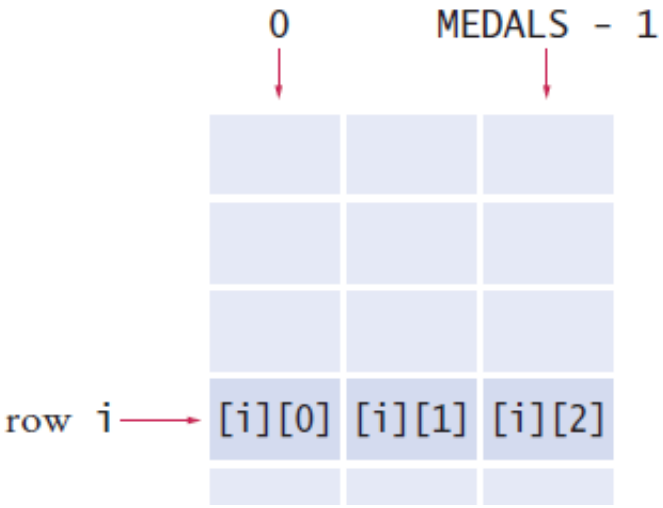
- Some programs that work with two-dimensional lists need to locate the elements that are adjacent to an element
- This task is particularly common in games
- You are at loc i, j
- Watch out for edges!
  - No negative indexes!
  - Not off the 'board'

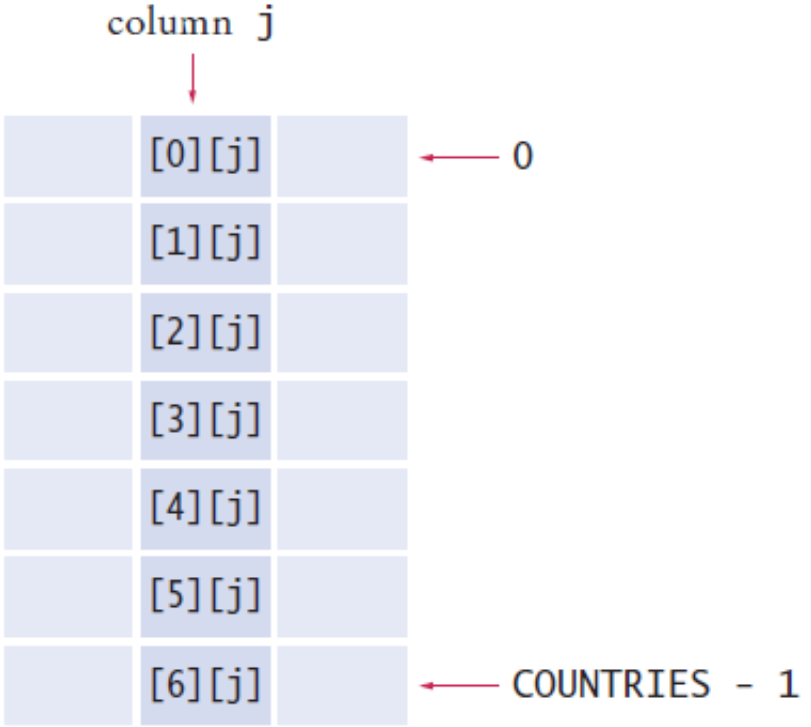


Adding Rows and Columns

- 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
```

Example

- Open the file medals.py

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
- Discover algorithms by manipulating physical objects

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