PGR107
Python Programming

Lecture 6 – Lists

Kristiania University College

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

Lists

- In many programs, you need to collect large numbers of values. In Python, you use the **list structure** for this purpose.

- A **list** is a container that stores a collection of elements that are arranged in a linear or sequential order. Lists can automatically grow to any desired size as new items are added and shrink as items are removed.

- In this chapter, you will learn about lists and several common algorithms for processing them.

Basic Properties of Lists

- **Lists** are the fundamental mechanism in Python for collecting multiple values.

- **Creating Lists:**

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

Basic Properties of Lists

- **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. For example,

```
print(values[5])    # Prints the element at index 5
```

- Both **lists** and **strings** are sequences, and the **[]** operator can be used to access an element in any sequence.

Basic Properties of Lists

- 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. But lists are mutable. You can replace one list element with another, like this:

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

Basic Properties of Lists

Syntax

To create a list:	$[value_1, value_2, \ldots]$
To access an element:	$listReference[index]$

Creates an empty list

Creates a list with initial values

Name of list variable

```
moreValues = []
values = [32, 54, 67, 29, 35, 80, 115]
```

Initial values

Use brackets to access an element.

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

Basic Properties of Lists

- **Traversing Lists:** There are two fundamental ways of visiting all elements of a list. You can loop over the index values and look up each element, or you can loop over the elements themselves.

- 1)

```
for i in range(len(values)) :
    print(i, values[i])
```

- 2)

```
for element in values :
    print(element)
```

Basic Properties of Lists

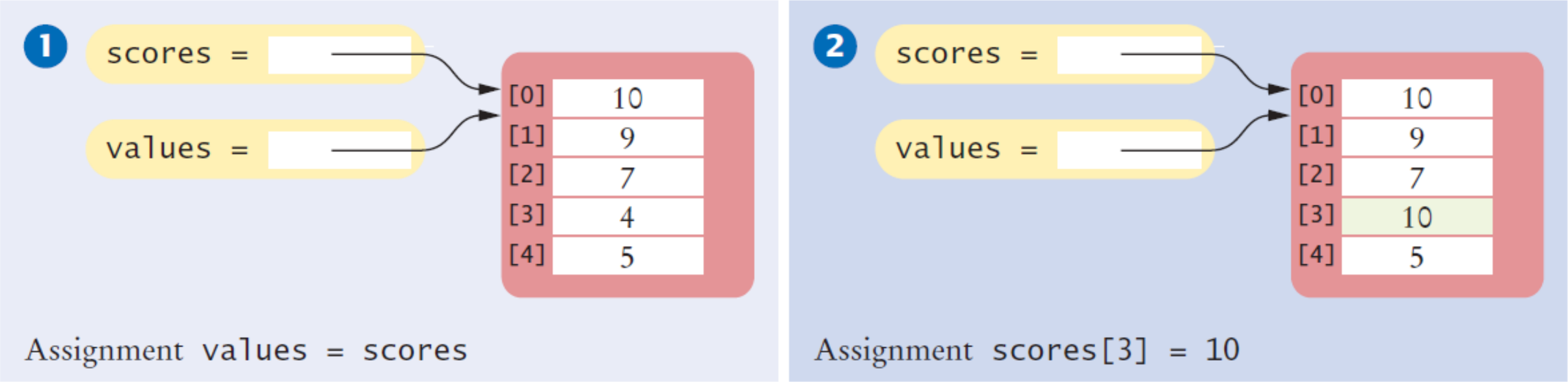
- **List References:** A list reference specifies the location of a list. Copying the reference yields a second reference to the same list.

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

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

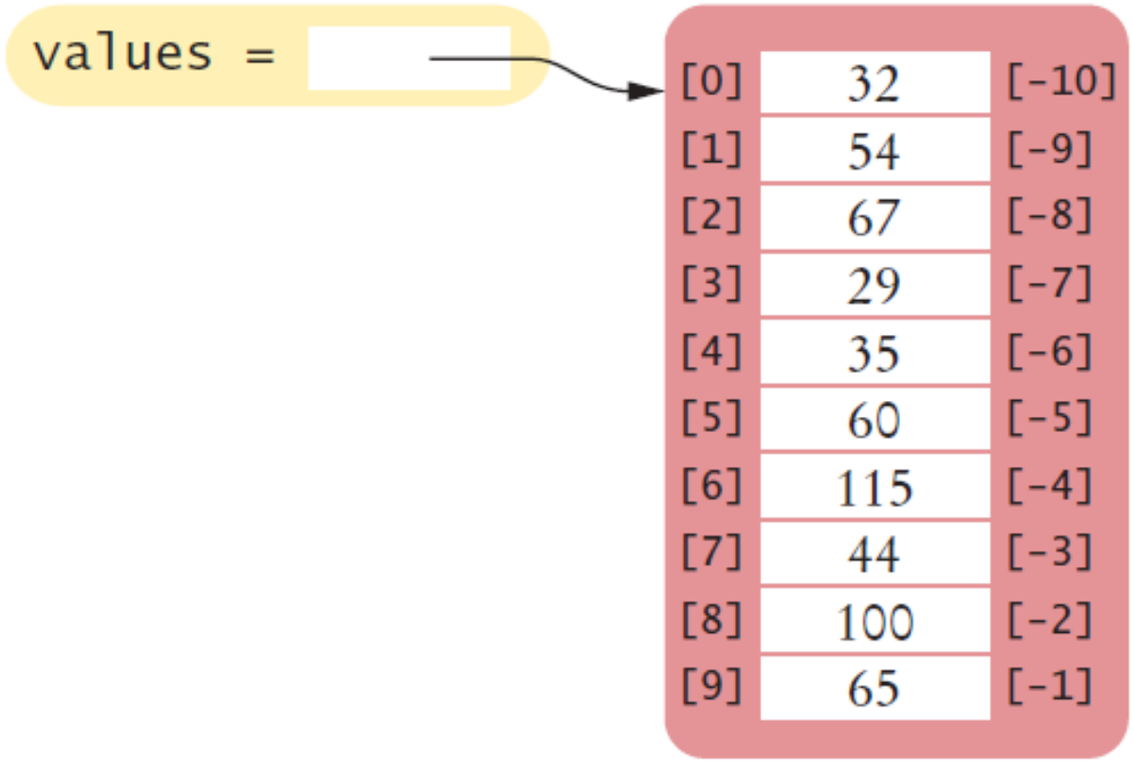
Basic Properties of Lists

- **List References:** A list reference specifies the location of a list. Copying the reference yields a second reference to the same list.



Basic Properties of Lists

- **Reverse Subscripts:** Python allows you to use negative subscripts when accessing an element of a list. The negative subscripts provide access to the list elements in **reverse order**.

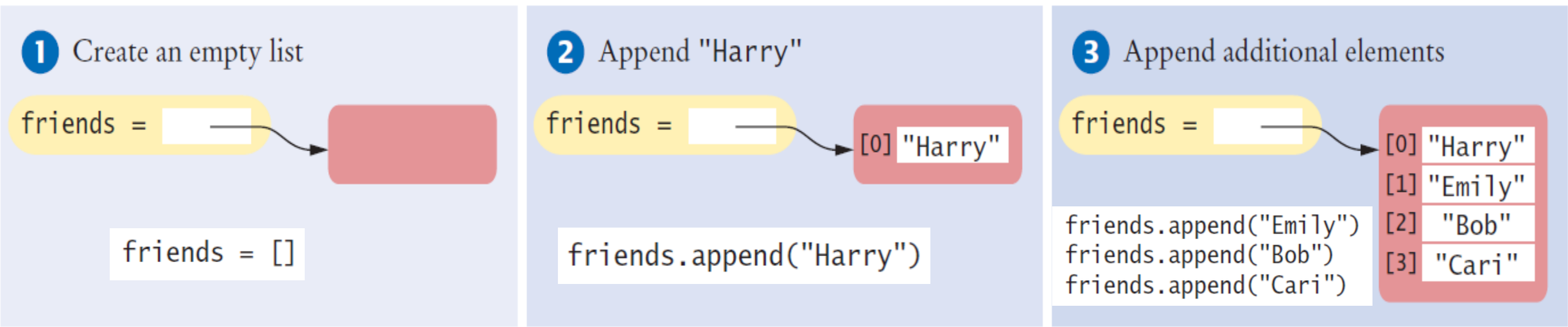


- In general, the valid range of negative subscripts is between **-1** and **-len(values)**.

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

Lists Operations

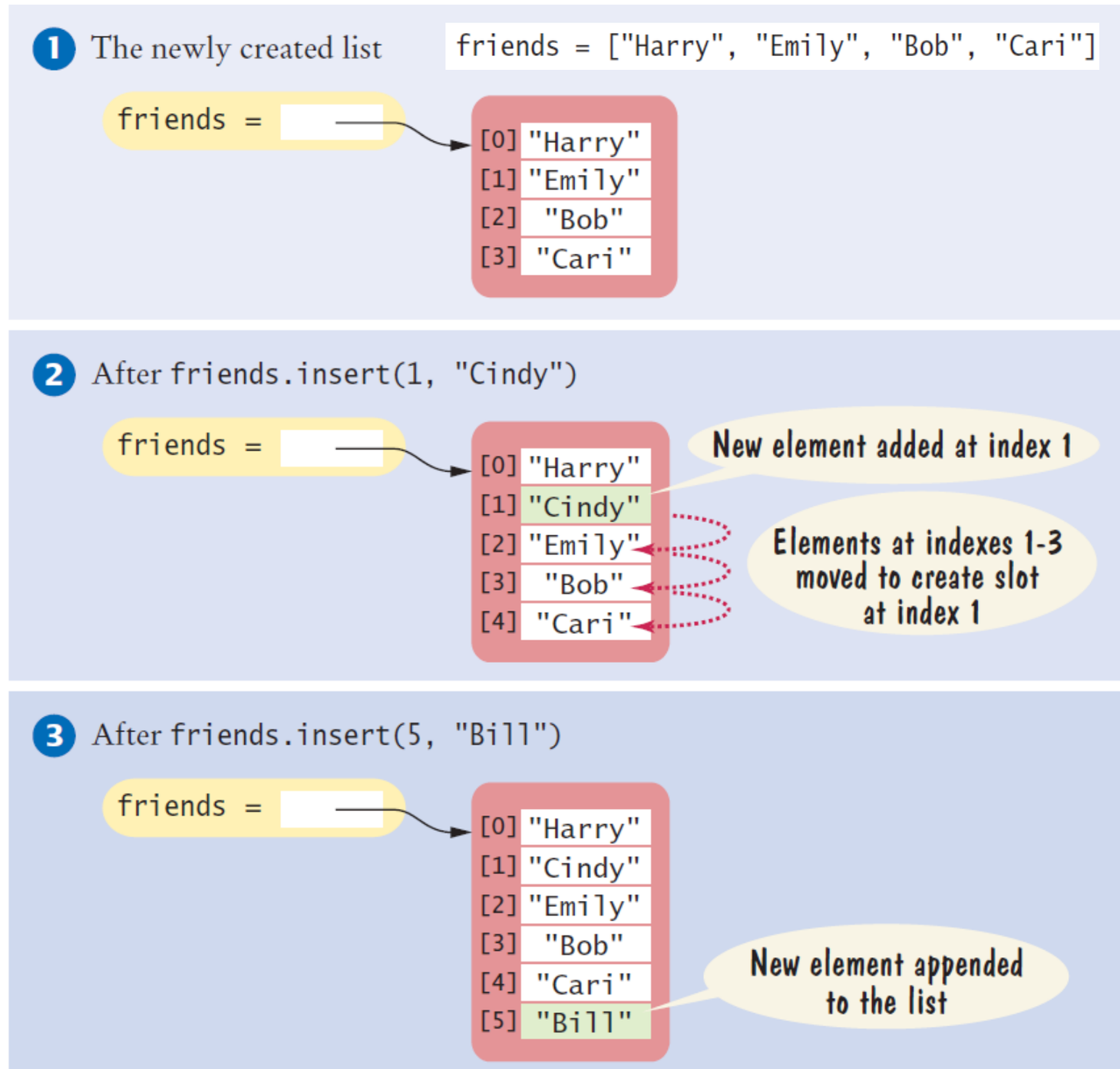
- **Appending Elements:** A new element can be appended to the end of the list with the **append method**. The size, or length, of the list increases after each call to the append method. Any number of elements can be added to a list.



Lists Operations

- **Inserting Elements:**

Use the insert method to insert a new element at any position in a list.



Lists Operations

- **Finding Elements:**
- 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")
```

- Sometimes, you want to know the position at which an element occurs. The **index method** yields the index of the first match. For example,

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

Lists Operations

- **Finding Elements:**
- If a value occurs more than once, you may want to find the position of all occurrences. You can call the index method and specify a starting position for the search. Here, we start the search after the index of the previous match:

```
n2 = friends.index("Emily", n + 1)    # Sets n2 to 4
```

- When you call the index method, the element to be found must be in the list or a runtime exception occurs. It is usually a good idea to test with the in operator before calling the index method:

```
if "Cindy" in friends :
   n = friends.index("Cindy")
else :
   n = -1
```

Lists Operations

- **Removing Elements:**
- The **pop method** removes the element at a given position. For example, suppose we start with the list

```
friends = ["Harry", "Cindy", "Emily", "Bob", "Cari", "Bill"]
```

- To remove the element at index position 1 ("Cindy") in the friends list, you use the command

```
friends.pop(1)
```

- The element removed from the list is **returned** by the pop method.

```
print("The removed item is", friends.pop(1))
```

Lists Operations

- **Removing Elements:**
- If you call the pop method without an argument, it removes and returns the last element of the list. For example, **friends.pop()** removes **"Bill"**.

- The **remove method** removes an element by value instead of by position.

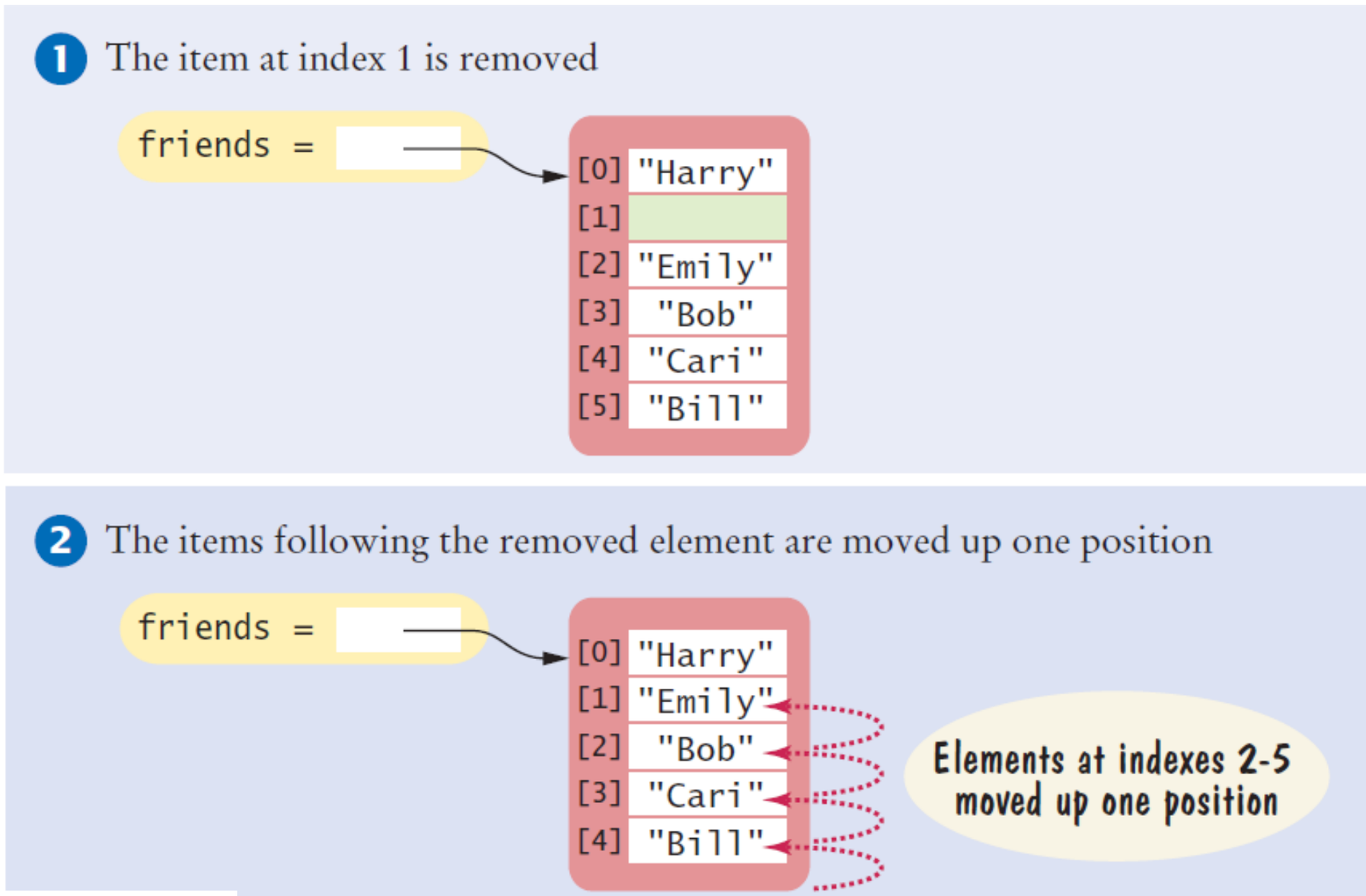
```
friends.remove("Cari")
```

- Note that the value being removed must be in the list or an exception is raised. To avoid a run-time error, you should first verify that the element is in the list before attempting to remove it:

```
element = "Cari"
if element in friends :
   friends.remove(element)
```

Lists Operations

- **Removing Elements:**



Lists Operations

- **Concatenation and Replication:**
- The concatenation of two lists is a new list that contains the elements of the first list, followed by the elements of the second. For example, suppose we have two lists

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

- You want to create a new list that combines the two. Two lists can be concatenated by using the plus (+) operator:

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

Lists Operations

- **Concatenation and Replication:**
- If you want to concatenate the same list multiple times, use the **replication operator** (*****). For example,

```
monthInQuarter = [1, 2, 3] * 4    # The list is [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- One common use of replication is to initialize a list with a fixed value. For example

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

Lists Operations

- **Equality Testing:**
- You can use the == operator to compare whether two lists have the same elements, in the same order. For example,
- [1, 4, 9] == [1, 4, 9] is True, but
- [1, 4, 9] == [4, 1, 9] is False.
- The opposite of **==** is **!=**. The expression [1, 4, 9] != [4, 9] is True.

Lists Operations

- **Sum, Maximum, Minimum, and Sorting:**
- If you have a list of numbers, the **sum function** yields the sum of all values in the list. For example:

```
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:

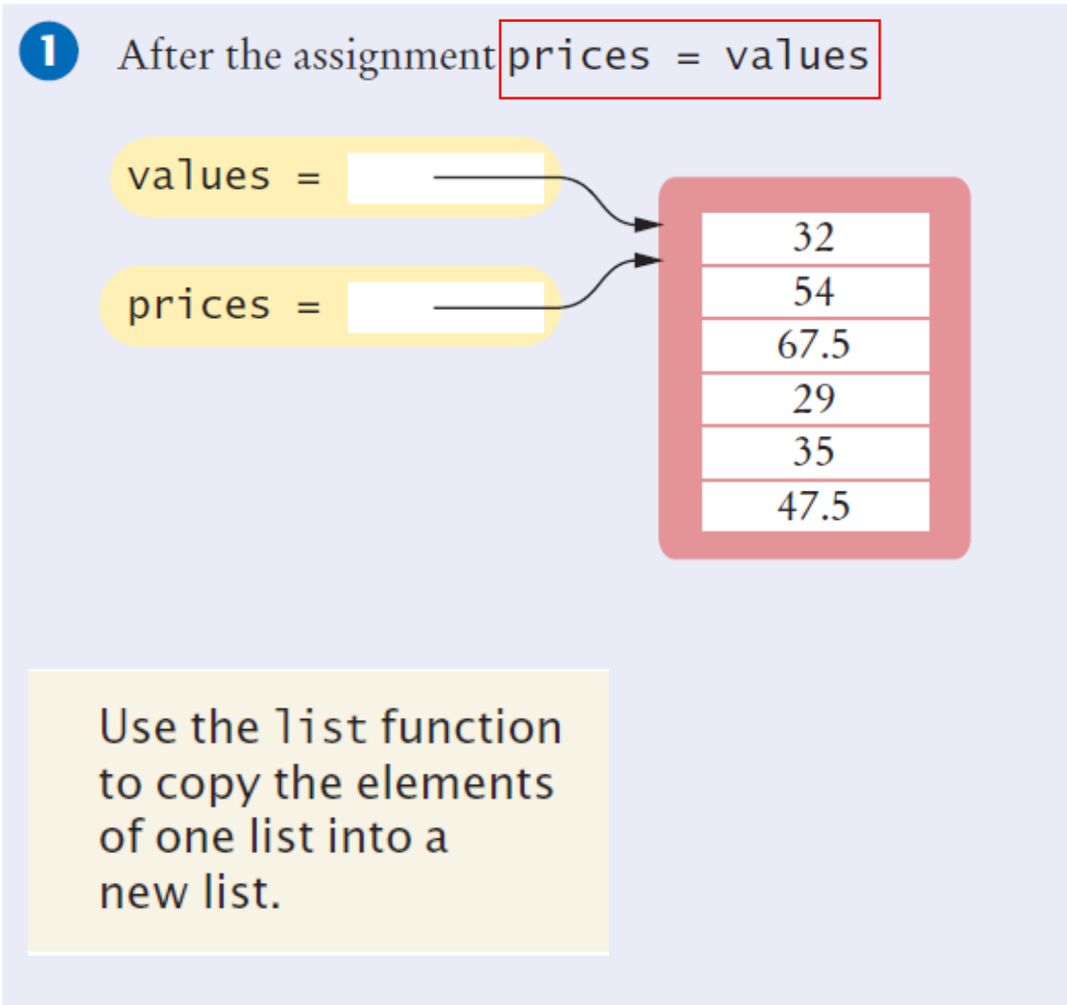
```
max([1, 16, 9, 4])    # Yields 16
min("Fred", "Ann", "Sue")    # Yields "Ann"
```

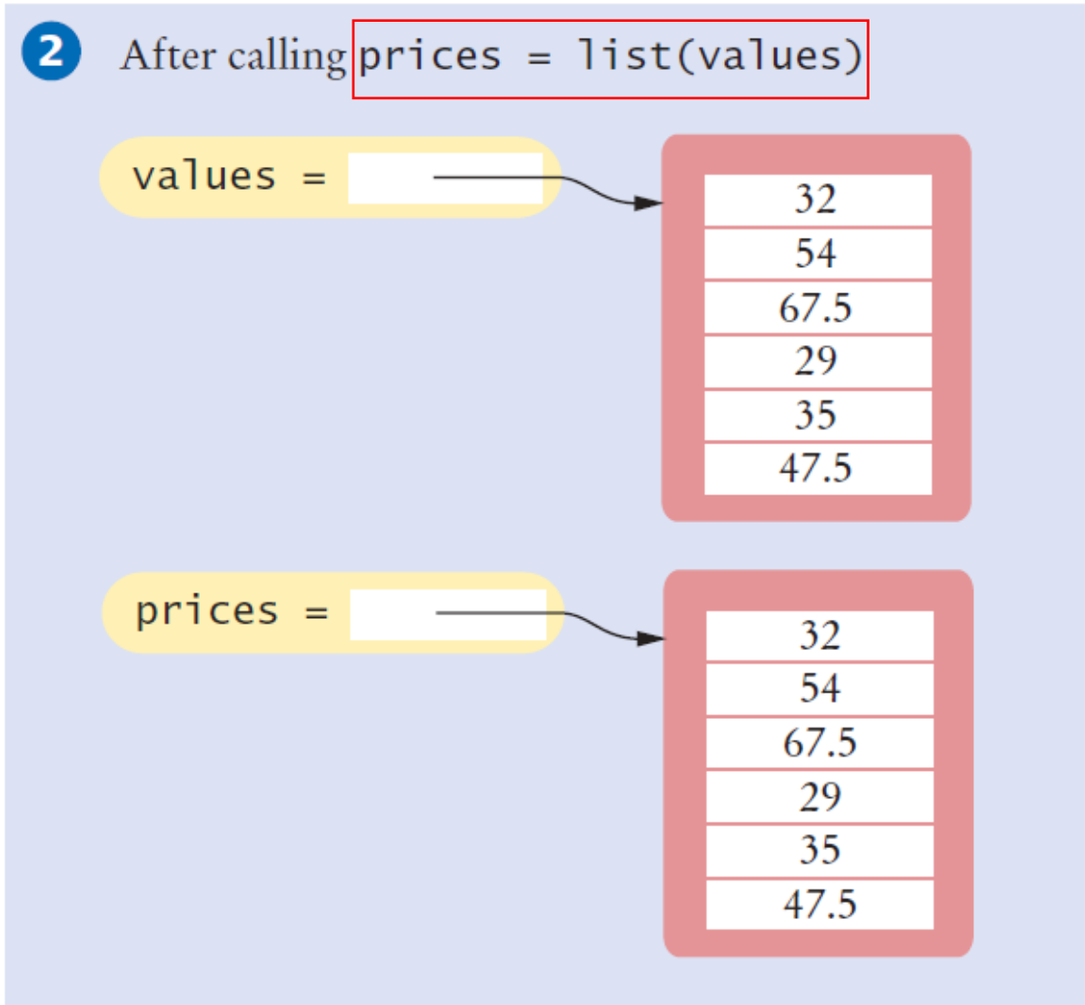
- The sort method sorts a list of numbers or strings. For example,

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

Lists Operations

- **Copying Lists:**





Lists Operations

- **Creating Sub-lists:** Sometimes you want to look at a part of a list.

```
temperatures = [18, 21, 24, 28, 33, 39, 40, 39, 36, 30, 22, 18]
```

- You are interested in the temperatures only for the third quarter:

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

```
temperatures[ : 6]
```

- contains all elements up to (but not including) position 6

```
temperatures[6 : ]
```

- includes all elements from index 6 to the end of the list

Lists Operations

- **Creating Sub-lists:**
- If you omit both index values, **temperatures [:]**, you get a copy of the list.
- You can even assign values to a slice or sub-list. For example, the assignment

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

- The size of the slice and the replacement don't have to match:

```
friends[ : 2] = ["Peter", "Paul", "Mary"]
```

- replaces the first two elements of friends with three new elements, increasing the length of the list.

```
greeting = "Hello, World!"
greeted = greeting[7 : 12]    # The substring "World"
```

Common Functions and Operators Used with Lists

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 Functions and Operators Used with Lists

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

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

- **Filling:** 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:** If you want to compute the sum of a list of numbers or concatenate a list of strings:

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

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

Common List Algorithms

- **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
```

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

- If you want to print values without adding them to a string, you need to adapt the algorithm slightly. Suppose we want to print a list of numbers like this:

```
32 | 54 | 67.5 | 29 | 35
```

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

Common List Algorithms

- **Maximum and Minimum:**

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

- Note that the loop starts at 1 because we initialize largest with values[0].

- To compute the smallest element, reverse the comparison.

Common List Algorithms

- **Linear Search:** A linear search inspects elements in sequence until a match is found.

- **Exercise:** Finding the first value in a list that is > 100.

Common List Algorithms

- **Linear Search:** A linear search inspects elements in sequence until a match is found.

- **Exercise:** Finding the first value in a list that is > 100.
- values = [1, 4, 235, 65, 40]

- The first value that is > 100 is at position 2.

Common List Algorithms

- **Collecting and Counting Matches:**
- Here, we collect all values that are > 100:

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

- Sometimes you just want to know how many matches there are without collecting them.

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

Common List Algorithms

- **Removing Matches:** A common processing task is to remove all elements that match a particular condition.

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

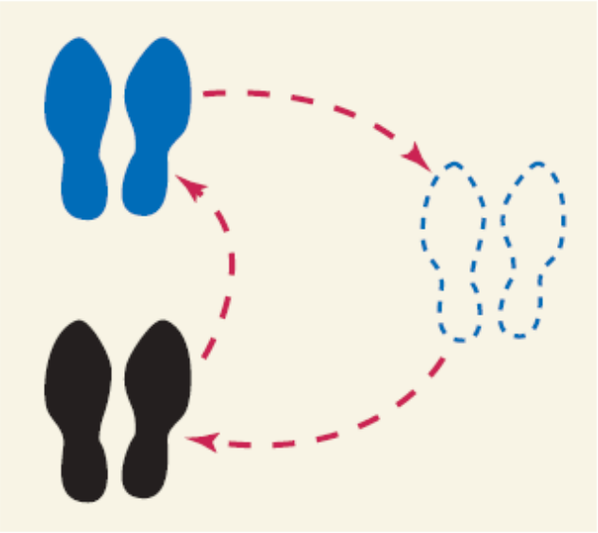
- Can you see why a **for loop** is not suitable for this algorithm?

```
for i in range(len(words)) :
    word = words[i]
    if len(word) < 4 :
        Remove the element at index i.
```

Common List Algorithms

- **Swapping Elements:** You often need to swap elements of a list. For example, you can sort a list by repeatedly swapping elements that are not in order.

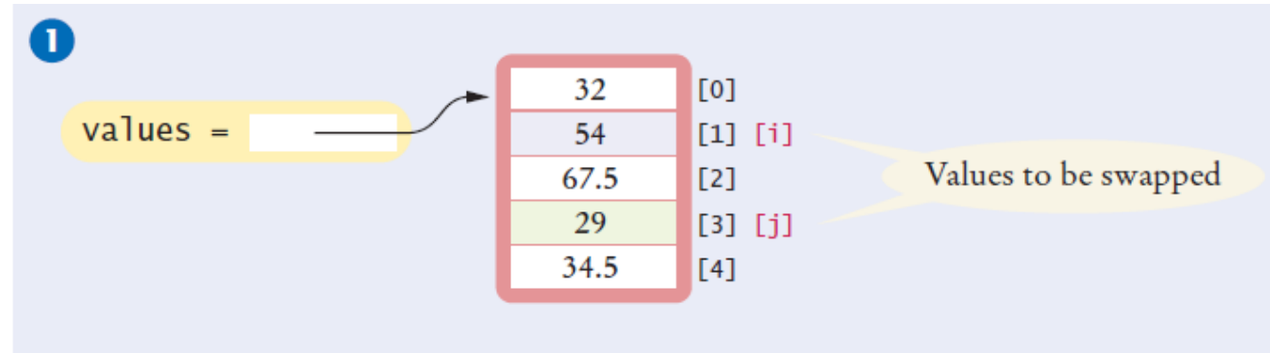
Use a temporary variable when swapping two elements.

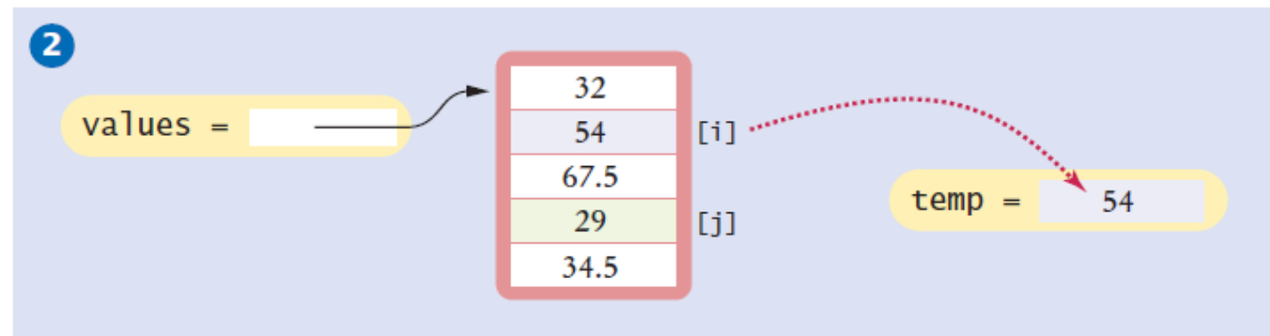


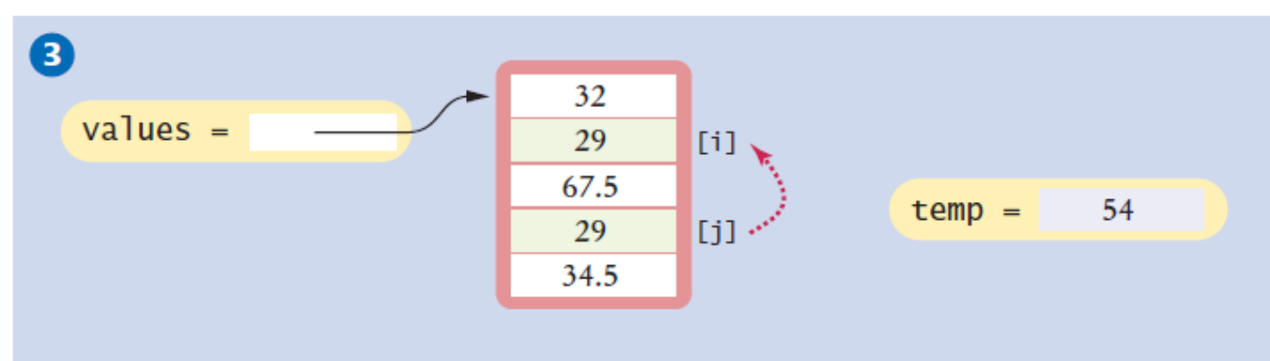
To swap two elements, you need a temporary variable.

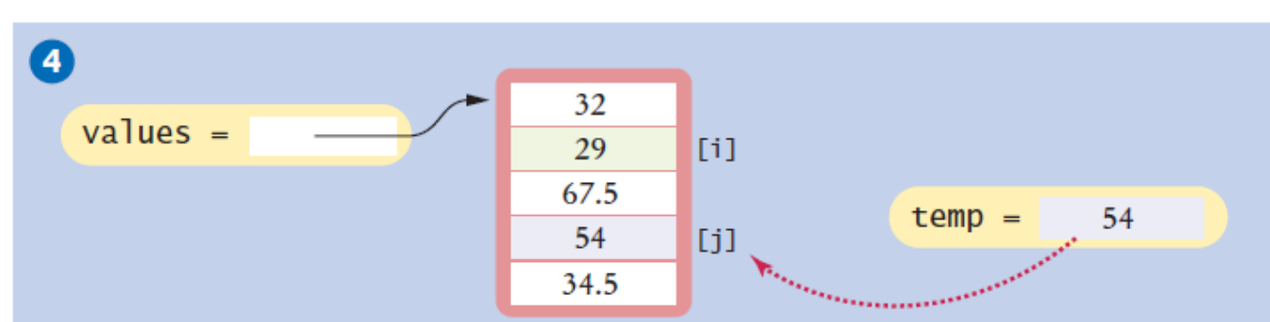
Common List Algorithms

- **Swapping Elements:**









Use a temporary variable when swapping two elements.

Common List Algorithms

- **Reading Input:** It is very common to read input from a user and store it in a list for later processing. Start with an empty list and, as each value is read, append the value to the end of the list:

```
values = []
print("Please enter values, Q to quit:")
userInput = input("")    # Get user input without a prompt.
while userInput.upper() != "Q" :
   values.append(float(userInput))
   userInput = input("")
```

Using Lists with Functions

- A function can accept a list as an argument.

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

   return total
```

When calling a function with a list argument, the function receives a list reference, not a copy of the list.

- This function visits the list elements, but it does not modify them. It is also possible to modify the elements of a list. 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
```

Tuples

- Python provides a data type for **immutable sequences** of arbitrary data. A **tuple** is very similar to a list, but once created, its contents cannot be modified. A tuple is created by specifying its contents as a comma-separated sequence. You can enclose the sequence in parentheses:

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

A tuple is created as a comma-separated sequence enclosed in parentheses.

Tuples

- Conveniently used to swap variable values.

```
x = y
y = x
```

NO

```
temp = x
x = y
y = temp
```

OK

```
(x, y) = (y, x)
```

Using Tuples -- OK

Tuples

- Used to return more than one value from a function.

```
def quotient_and_remainder (x, y):
        q = x // y
        r = x % y
        return (q, r)
```

```
(quot, rem) = quotient_and_remainder (4, 5)
```

Tuples

- **Functions with a Variable Number of Arguments:**

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

    return total
```

```
a = sum(1, 3)          # Sets a to 4
b = sum(1, 7, 2, 9)    # Sets b to 19
```

- A function can also be defined to receive a fixed number of arguments followed by a variable number of arguments:

```
def studentGrades(idNum, name, *grades) :
```

Tuples

- **Returning Multiple Values with Tuples:**

```
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.
```

```
(month, day, year) = readDate()
```

End of Chapter 6