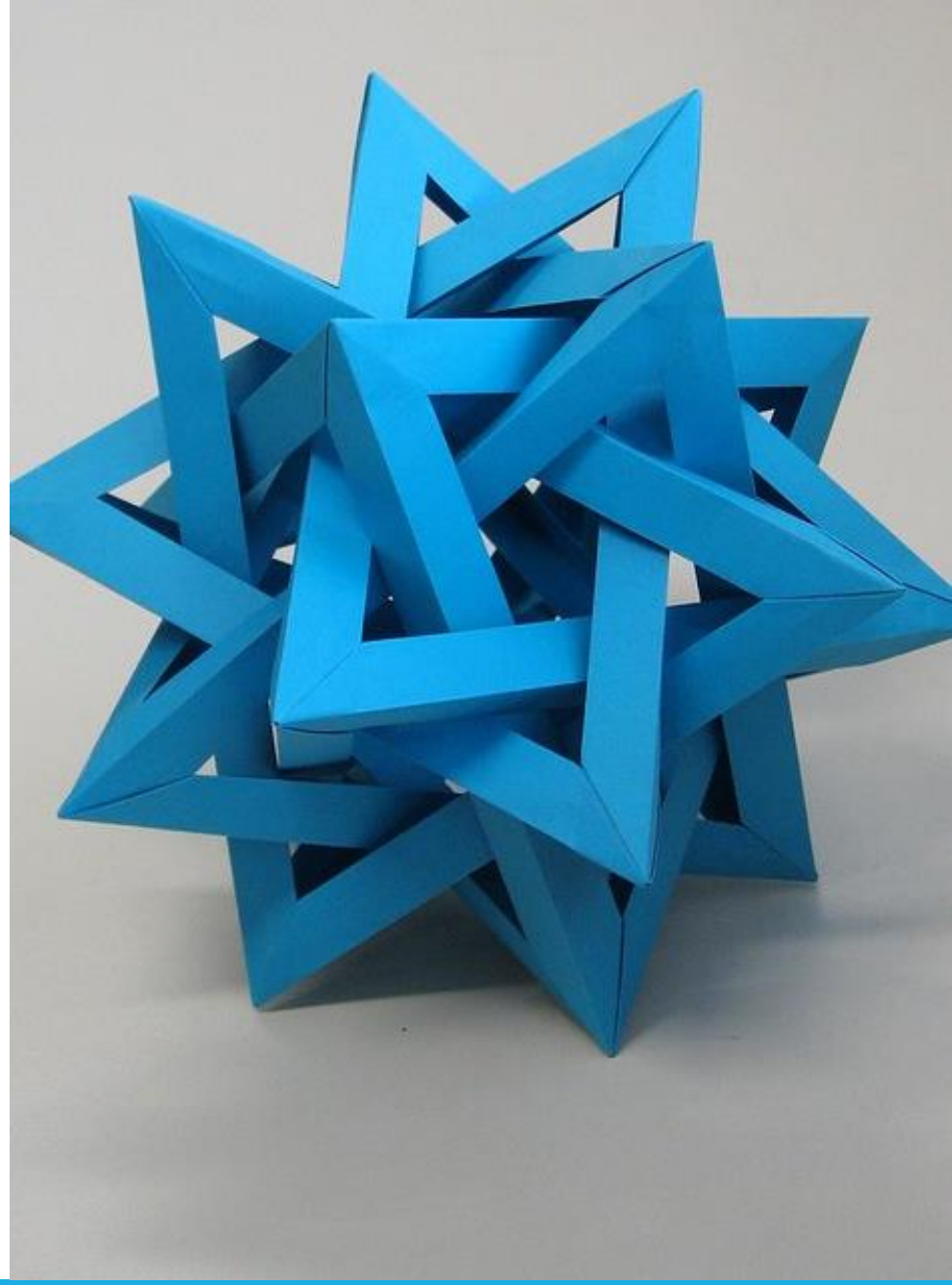
POLITECNICO DI TORINO
Dipartimento di Automatica e Informatica

Unit P6: Lists and Tables

LISTS, OPERATION ON LISTS, NESTED LISTS

Chapter 6



Unit 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

What is a List?

- A List is a versatile dynamic data structure, that stores a variable number of elements, of any type, that may be accessed by position (index)
- Functionally, it covers what other languages may call
  - List
  - Sequence
  - Array
  - Vector

Basic Properties of Lists

6.1

Creating a List

- Specify a list variable with the subscript operator [ ]

Syntax

To create a list:	[*value*$_1$, *value*$_2$, . . .]
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]
```

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)
- Indexes start at `0`

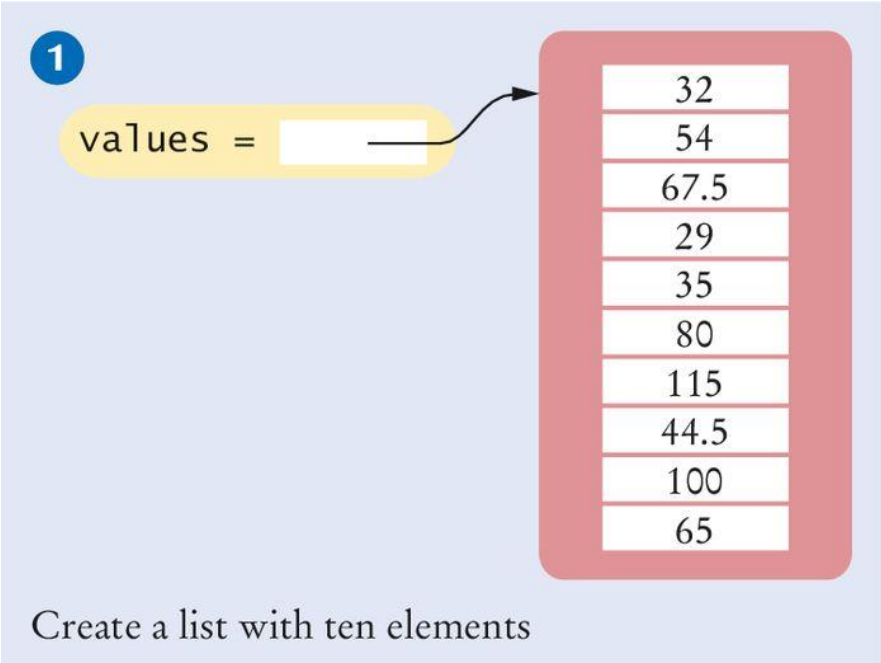
Accessing list elements

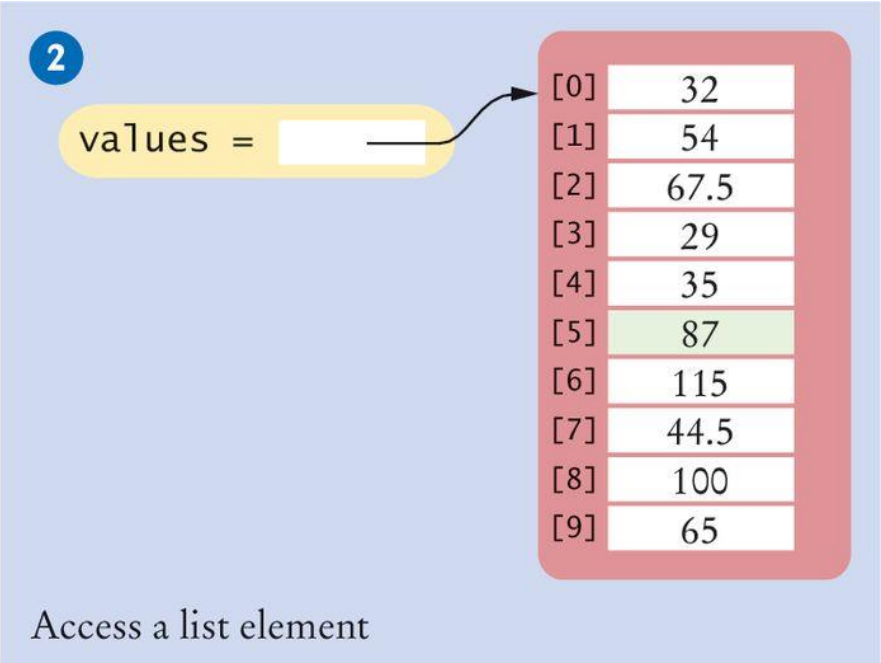
```
print(values[5])
```

Replacing list elements

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

Creating Lists/Accessing Elements





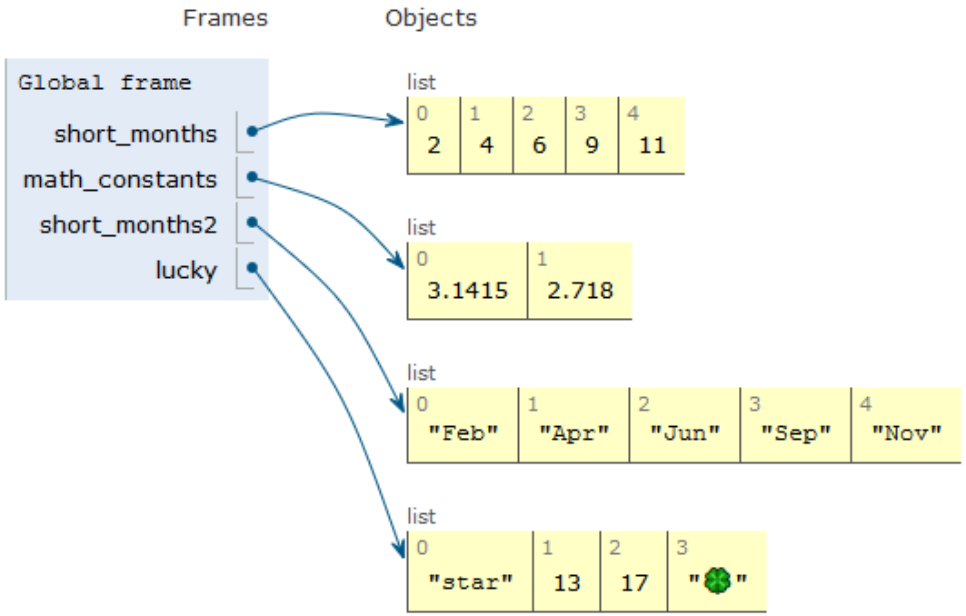

```
# 1: Creating a list
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]

# 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 – the values of an element may be updated, new elements may be added, elements may be deleted

Types of Elements in a List



- A list of integer values

  `short_months = [ 2, 4, 6, 9, 11 ]`

- A list of real values

  `math_constants = [ 3.1415, 2.718 ]`

- A list of strings

  `short_months2 = [ 'Feb', 'Apr', 'Jun', 'Sep', 'Nov' ]`

- A list with mixed values
  - `lucky = [ 'star', 13, 17, '🍀' ]`
  - Unless you have a good reason, avoid mixing types

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

Printing a list

- A list may be used with the `print()` function
- All the elements are printed, with a syntax resembling the list creation

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

Determining the List Length

- You can use the `len()` function to obtain the length of the list, i.e., 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 (i.e., they identify an element of the list)

```
values[4]
```

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

```
values = [4]
```

This statement creates a list with one element, the integer 4

Loop Over the Index Values

- Given a list named “values” that contains 10 elements, we want to to access each element of the list
- We may 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])
```

This statement does always 10 loops… but what if *values* is of a different length?

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

This statement works regardless *values* length… it checks the actual length at run-time

Loop Over the Index Values

- Given a list named “values” that contains 10 elements, we want to to access each element of the list
- If we don’t need the index `i`, we may directly loop over the **values**

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

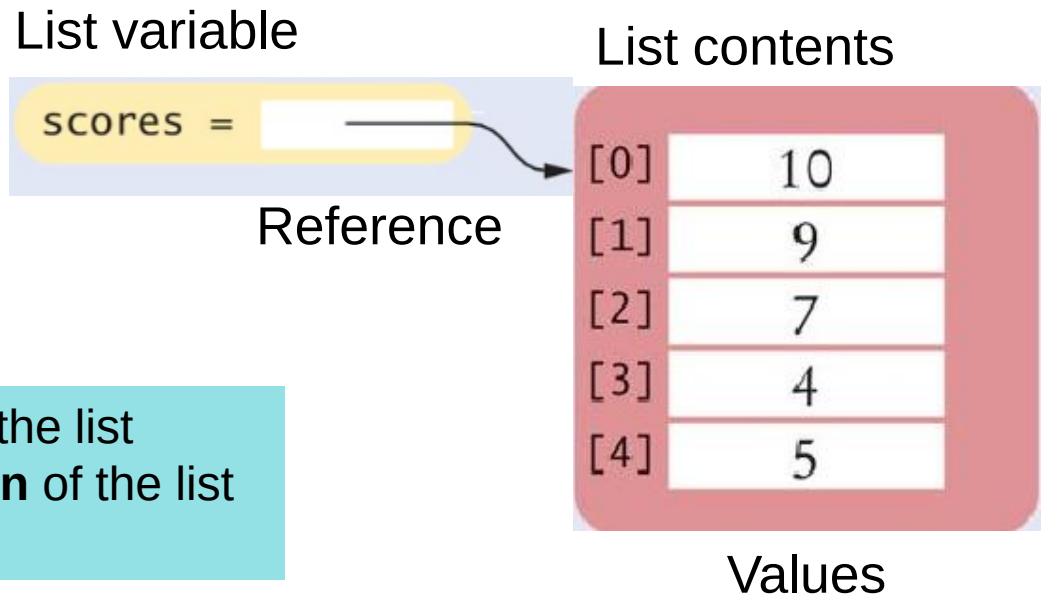
This statement is great if we do not need the index variable

- The for iterates directly over the list (`values`)
- At each iteration, `element` takes the value of every list element

List References

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

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



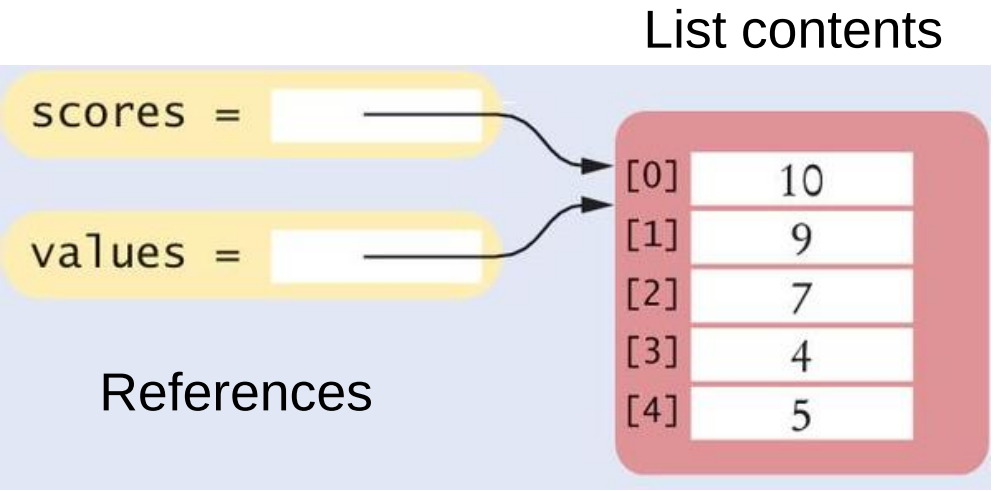
A list variable contains a **reference** to the list contents. The reference is the **location** of the list contents (in memory).

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

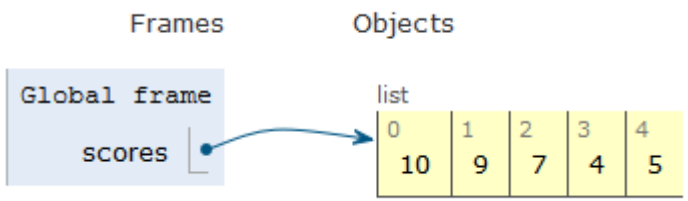
```
scores = [10, 9, 7, 4, 5]
values = scores      # Copying list reference
```

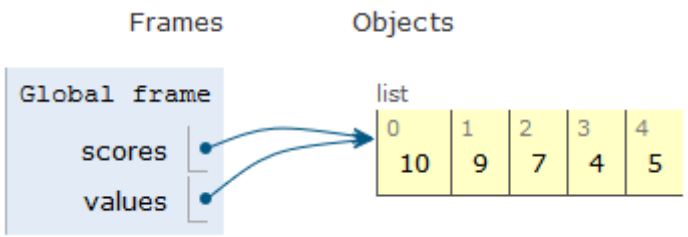
A list variable specifies the **location** of a list. Copying the reference yields a second **reference** to the **same** list.



Try it with PythonTutor

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

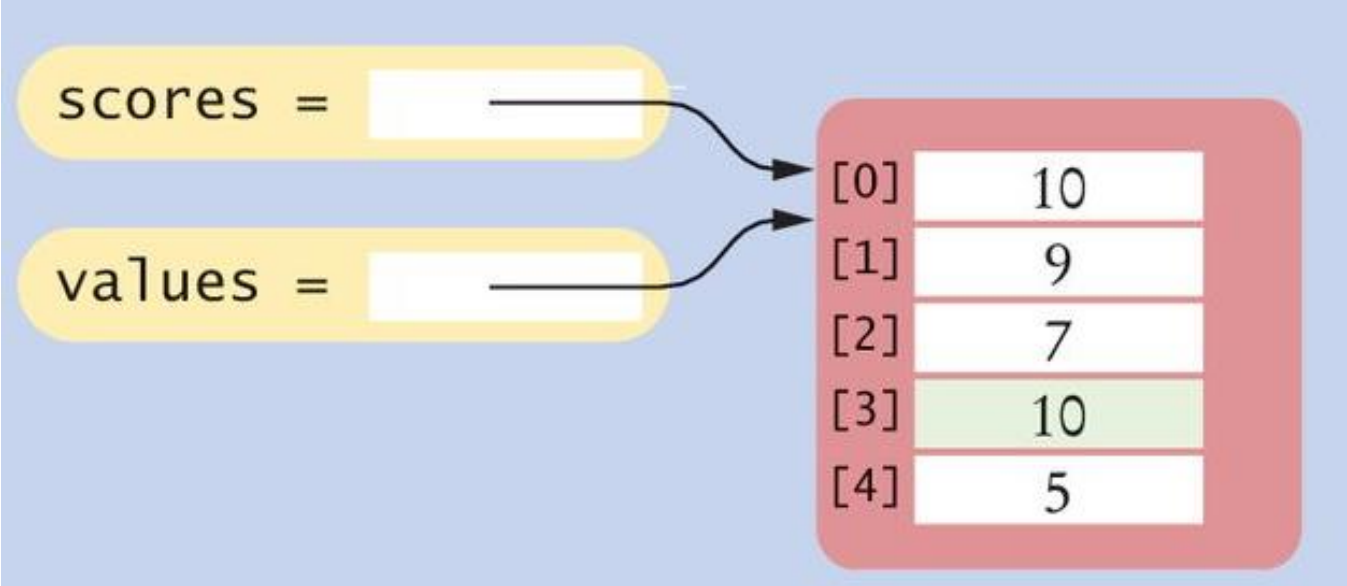




Modifying Aliased Lists

- You can modify the list through either of the variables
- You are in fact modifying the same list (accessed with two different names)

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

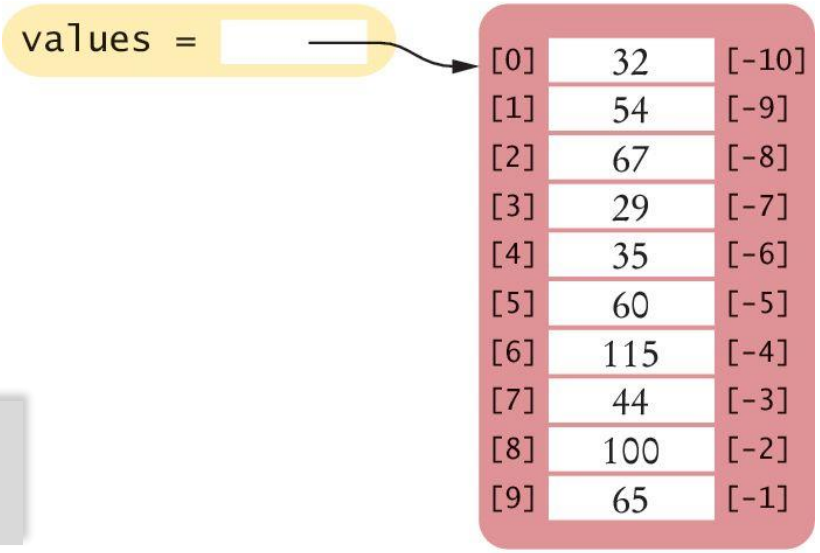


Negative/Reverse Subscripts

- Python, unlike other languages, allows 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)
```

```
last = values[len(values)-1]
# equivalent
```



List Operations

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

Try them with PythonTutor!

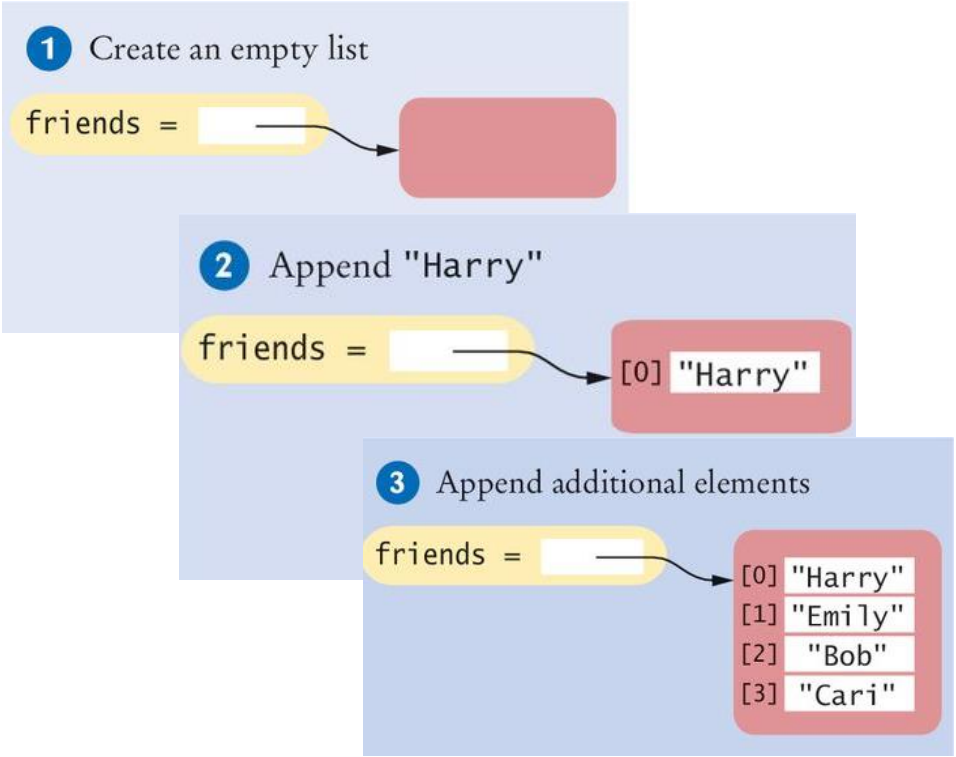
Appending Elements

- 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")
```

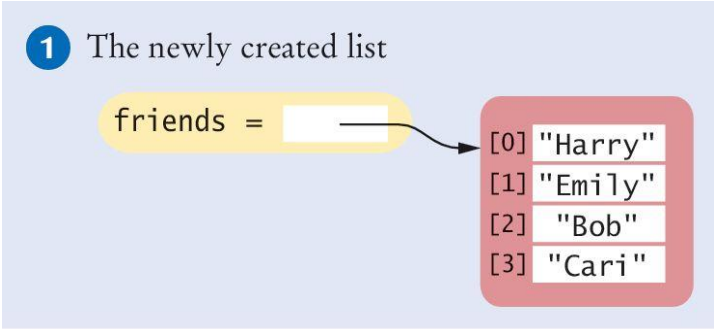


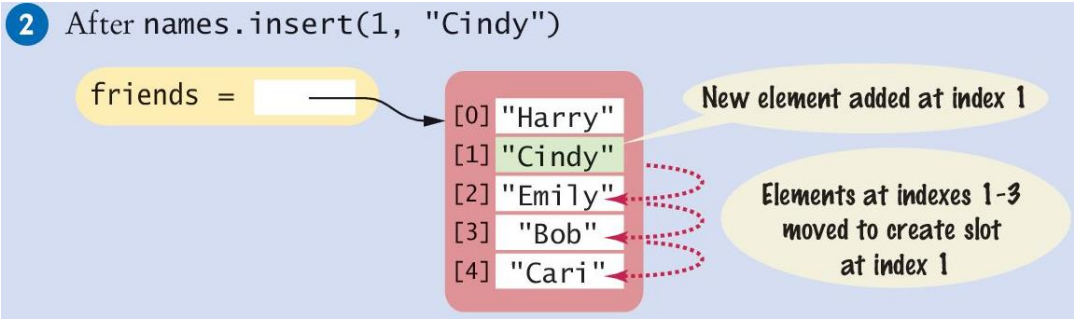
Inserting an Element

- 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
  - The other elements "move forward"

```
#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")
```

- The result is a Boolean value:
  - `True` if the element is contained in the list
  - `False` if the element is not contained in the list
  - Usually used in `while` or `if` statements

Finding an Element

- Often, you want to know the position at which an element occurs
  - The `index()` method yields the index of the first match
  - The method is "applied" (with a dot '`.`') to a list variable, and returns an integer value

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

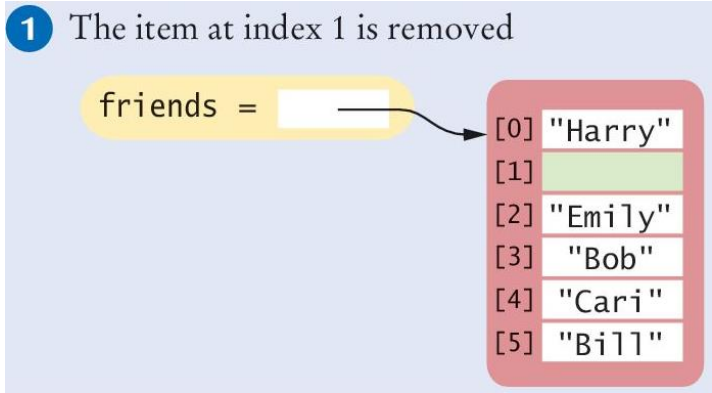
- If the element is not found in the list, a ValueError occurs

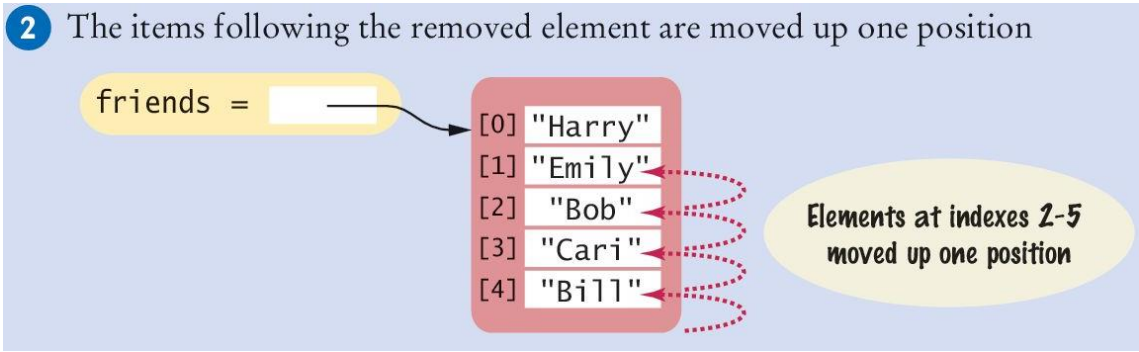
Removing an Element

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

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

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

Replication

- Replication of a list generates many copies of the elements of a list (similar to string replication)

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

  - Results in the list [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2 ,3]
- The integer specifies how many copies of the list should be concatenated
  - You can place the integer on either side of the "*" operator
- 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

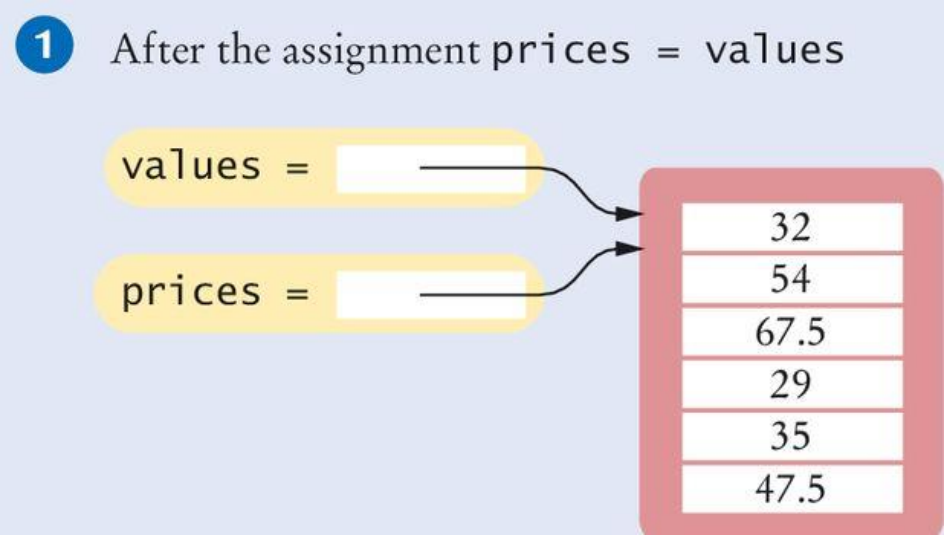
- The `sort()` method sorts a list of numbers or strings, from the smallest to the largest value
- The sort happens “in place”, within the same list

```
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:

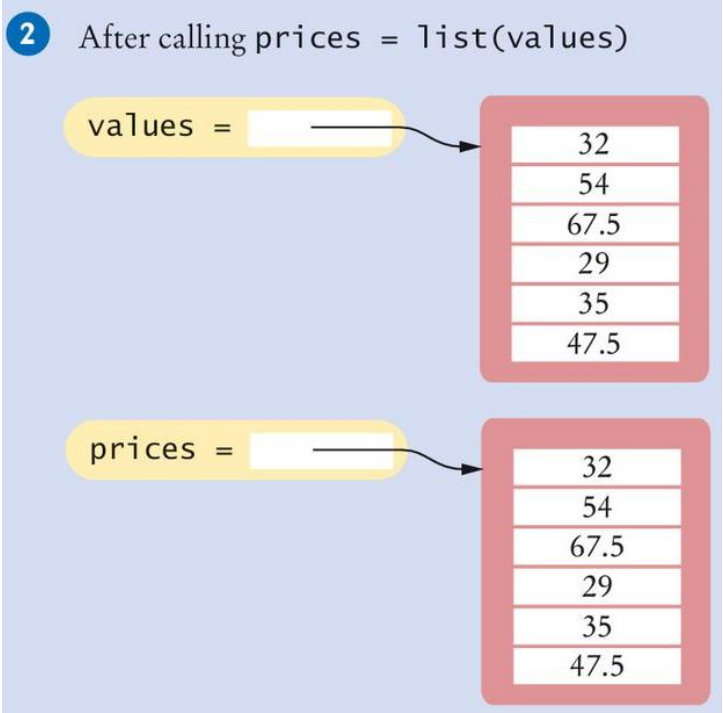
```
prices = values
```



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)
```

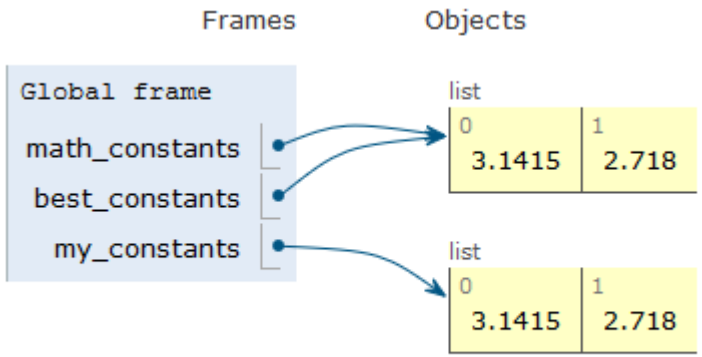


Try it with PythonTutor

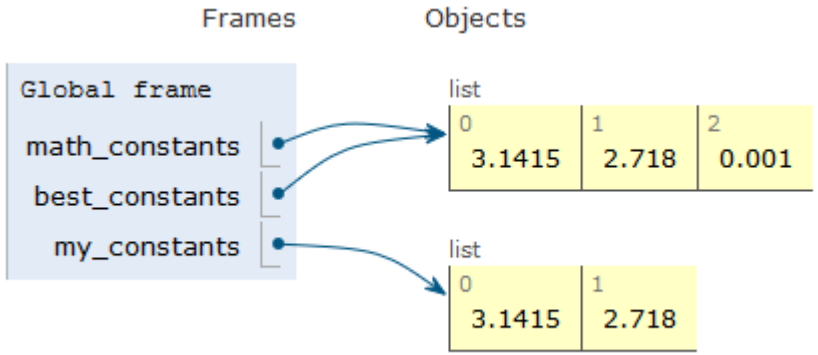
```
math_constants = [ 3.1415, 2.718 ]

best_constants = math_constants

my_constants =
list(math_constants)
```




```
math_constants.append(0.001)
```



Slices of a List

- Sometimes you want to extract 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
  - If the second index is omitted, all elements up to the end of the list are included
- Examples

  `temperatures[ : 6]`
  - includes all elements up to, but not including, position 6

  `temperatures[6 : ]`
  - includes all elements starting at position 6 to the end of the list

Slices (3)

- You can assign values to a slice:
  `temperatures[6 : 9] = [45, 44, 40]`
- Replaces the values in elements 6, 7, and 8
  - Other elements are unaffected
- Note: if the length of the replacement is different from the length of the slice, elements will be removed or added to the list

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.

List Methods

Python List Methods
append() - Add an element to the end of the list
extend() - Add all elements of a list to the another list
insert() - Insert an item at the defined index
remove() - Removes an item from the list
pop() - Removes and returns an element at the given index
clear() - Removes all items from the list
index() - Returns the index of the first matched item
count() - Returns the count of the number of items passed as an argument
sort() - Sort items in a list in ascending order
reverse() - Reverse the order of items in the list
copy() - Returns a shallow copy of the list

https://www.programiz.com/python-programming/list

Common List Algorithms

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

Try them with PythonTutor!

Try them on repl.it!

Filling a List

- This loop creates and fills a list with squares of integer numbers (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 need to have a string-type 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:

```
result = ''

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

print(result)
```

Element Separators (3)

- If you want to print values directly, without first adding them to a string, you may prevent the automatic ‘newline’:

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

Python Shortcut: `.join()`

- The `.join` method of strings automatically joins the elements of a list, using the string as a separator

  `separator_string.join(list)`

```
result2 = ', '.join(names)

print(result2)
```

Maximum and Minimum

- Here is the implementation of the max and min algorithms:

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

```
# equivalent to
largest = max(values)
```

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

```
# equivalent to
largest = min(values)
```

Linear Search

- Find 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
```

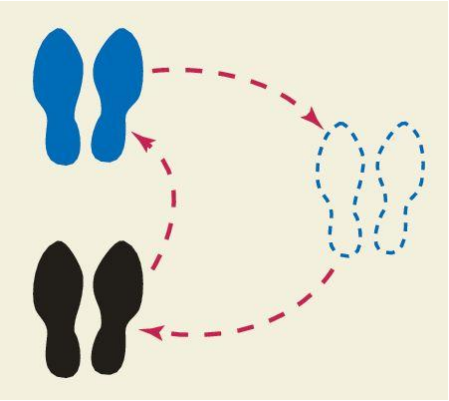
Removing Matches

- 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) # delete i-th element
        # do NOT increment i
    else :
        i = i + 1
```

Swapping Elements

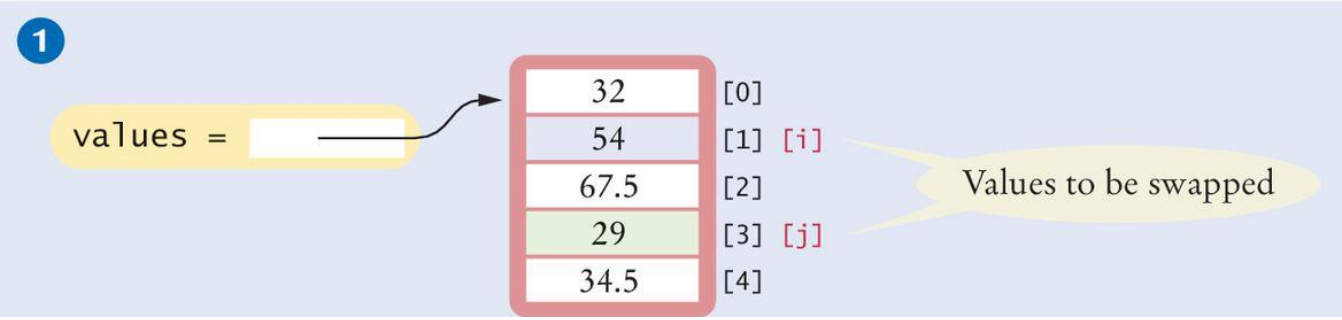
- For example, you can sort a list by repeatedly swapping elements that are not in order
- Swap the elements at positions `i` and `j` of a list values
- We'd like to set `values[i]` to `values[j]`. But that overwrites the value that is currently stored in `values[i]`, so we want to save that first:



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)

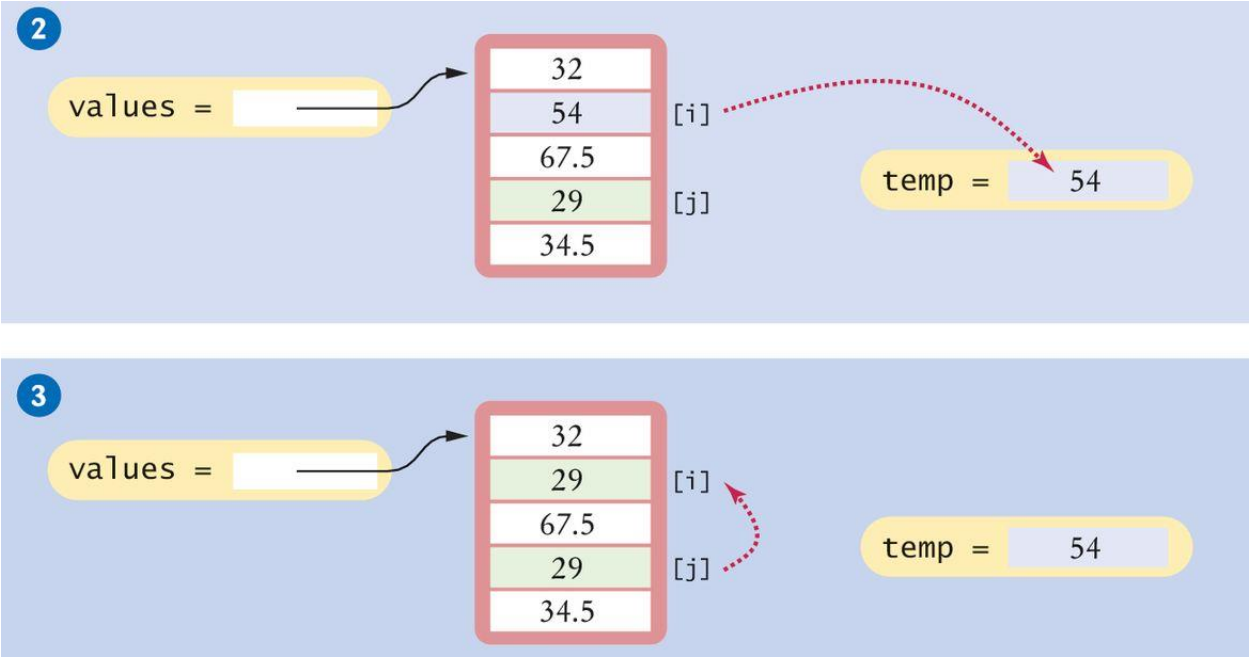
- Swapping elements `[1]` and `[3]`
  - This sets up the scenario for the actual code that will follow



Swapping Elements (3)

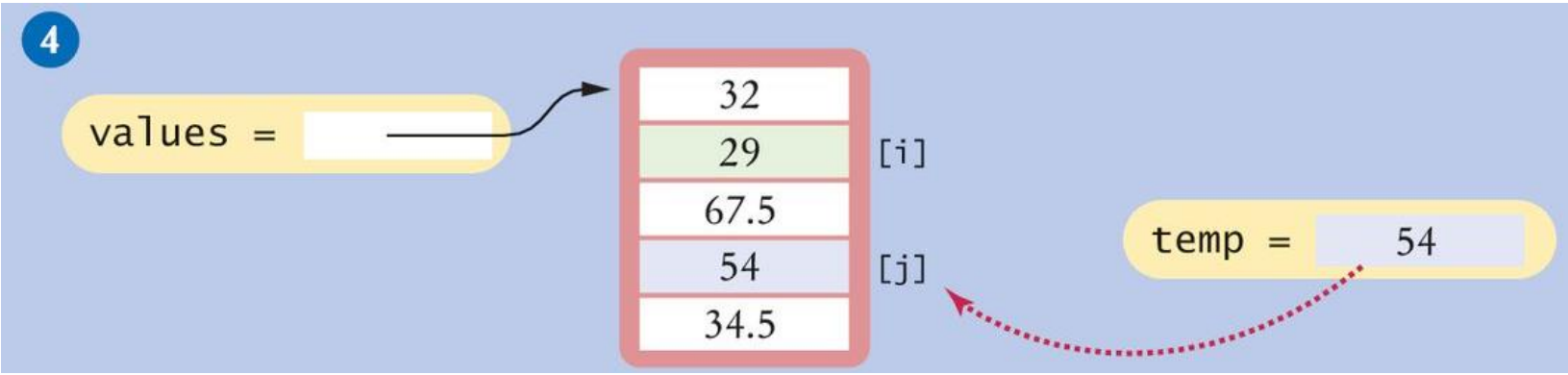
```
# Step 1
temp = values[i]

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



Swapping Elements (4)

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



Python Shortcut: multiple assignment

- More than one assignment may be done in parallel (and we avoid the temp variable) by using a 'tuple' syntax
  - `(a, b) = (3, 4)` is equivalent to a=3 and b=4, simultaneously
  - `(a, b) = (b, a)` effectively swaps a and b

```
# shortcut for swap

( values[1], values[3] ) = ( values[3], values[1] )
```

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

- Open the file largest.py

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
- 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

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 sumsq(values) :
  total = 0
  for element in values :
    total = total + element**2
  return total
```

Try it with PythonTutor!

Modifying List Elements

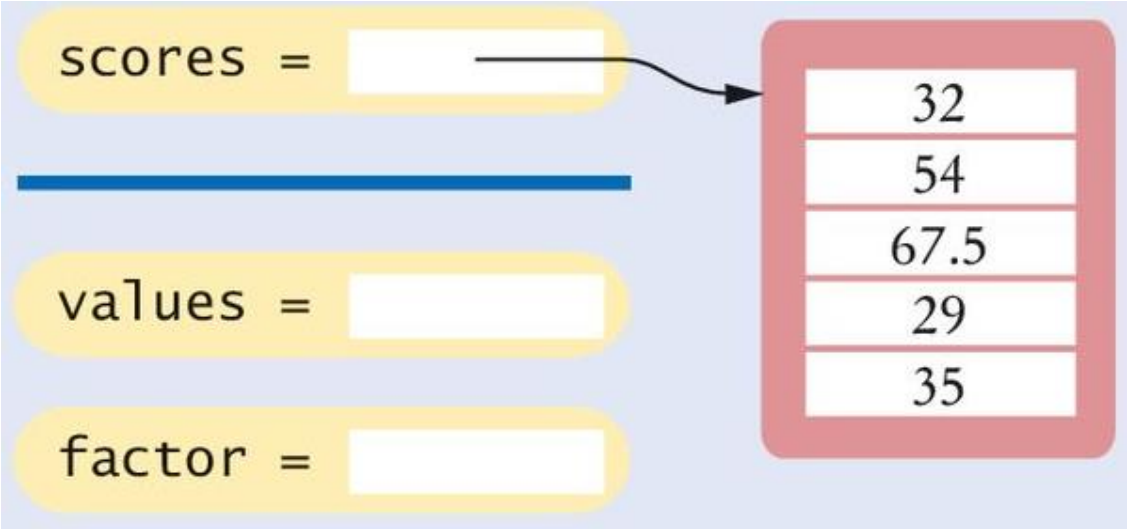
- The following function multiplies all elements of a list by a given factor
- The original list is modified

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

Try it with PythonTutor!

Example: Step 1

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

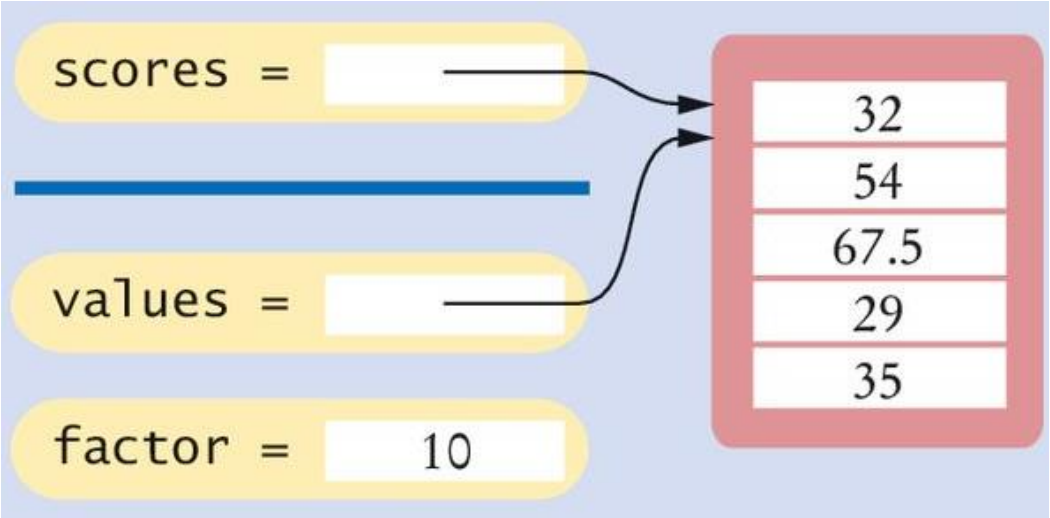


Example: Step 2

- The parameter variables are initialized with the arguments that are passed in the call

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

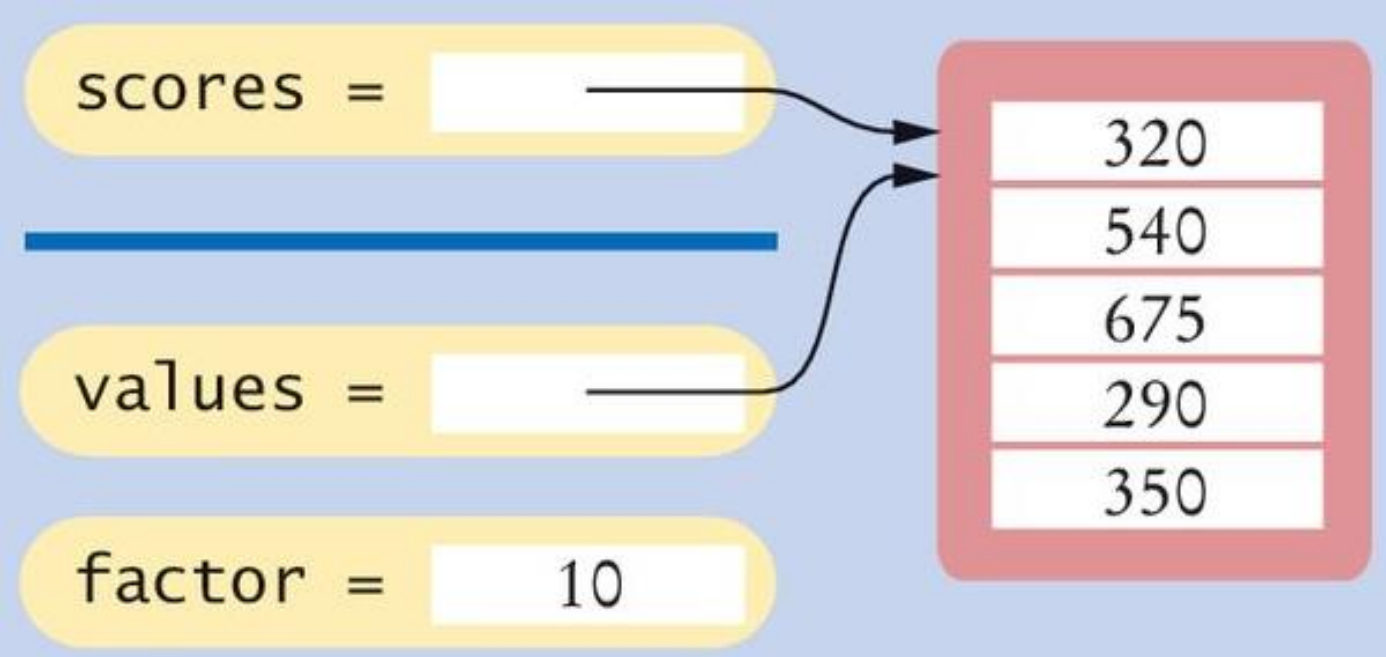
- In our case, `values` is set to `scores` and `factor` is set to `10`
  - Note that `values` and `scores` are references to the same list



Example: Step 3

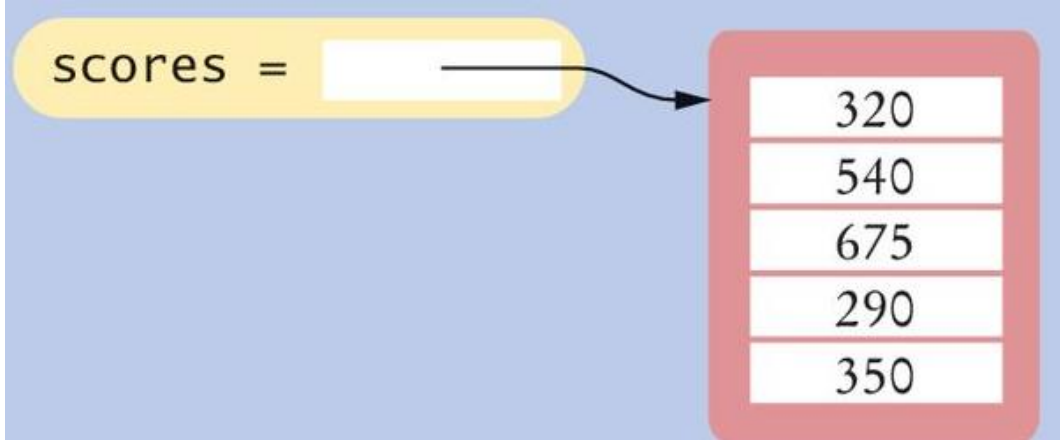
- The function multiplies all list elements by 10

```
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

- Open and study 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
  - The function receives a copy of the value (or the value is immutable)
- Call by reference
  - Function can change the values referenced by the arguments of a method call
  - A Python function can mutate the contents of a list when it receives a reference to it

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
```

- Note: tuples are an advanced data structure, with many more characteristics. They will not be studied in depth in this course.

Returning Multiple Values

- It is common practice in Python 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()
```

Adapting Algorithms

6.5

Adapting Algorithms

- 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

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

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

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

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
```

Problem 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 refinement is to identify the steps we need to process the data:
  - Read inputs
  - Remove the minimum
  - Remove the minimum again
  - 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.
  - We 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
8	Error	At least 2 scores are needed
(no inputs)	Error	That is not a legal input

scores.py

- Open the file scores.py

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 values 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 counters we can read and then discard the inputs
- Our next step in the stepwise refinement is to identify the steps we need to process the data:
  - Read input
  - For each input value:
    - Increment the corresponding counter
  - 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
```

- We therefore 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 a 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

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:

9	13	21	4	11	7	1	3

- Rearrange it to:

11	7	1	3	9	13	21	4

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

Manipulating Objects

- Visualize removing one object

Manipulating Objects

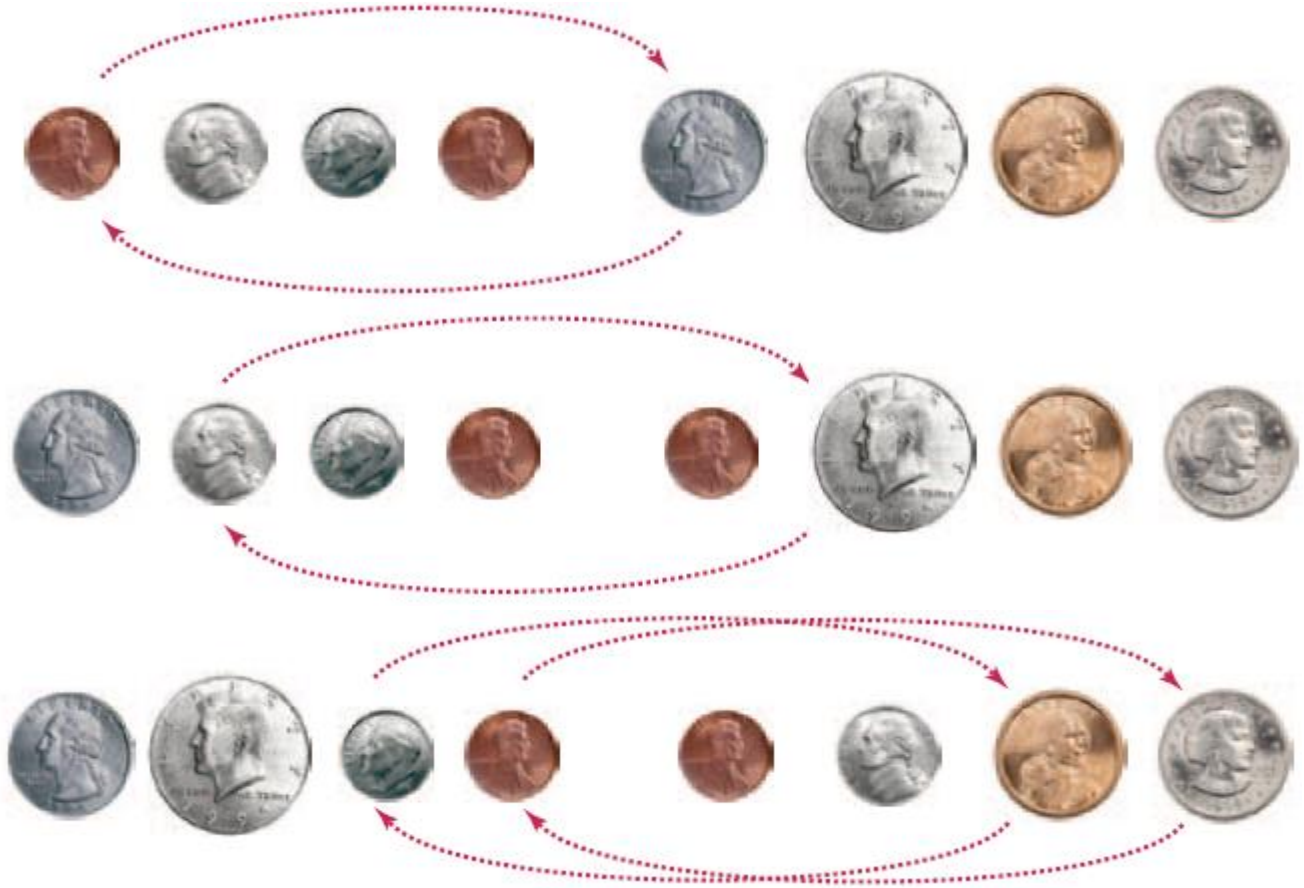
- Visualize inserting one object

Manipulating Objects

- 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
  - The first swap is with i=0
- How can j be set to handle any number of items?
  - … if size is 8, j is index 4…
- And when do we stop our loop?...


```
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
```

Develop an Algorithm

- Pick two locations (indexes) for the first swap and start a loop
  - The first swap is with i=0
- How can j be set to handle any number of items?
  - … if size is 8, j is index 4…
- And when do we stop our loop?...


```
i = 0
j = [size/2]
While ([size/2])
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

Develop an Algorithm

- Pick two locations (indexes) for the first swap a
  - The first swap is with i=0
- How can j be set to handle any number of item
  - … if size is 8, j is index 4…
- And when do we stop our loop?...

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


```
i = 0
j = [size/2]
While ([size/2])
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

swaphalves.py

- Open the file swaphalves.py

Tables

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

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

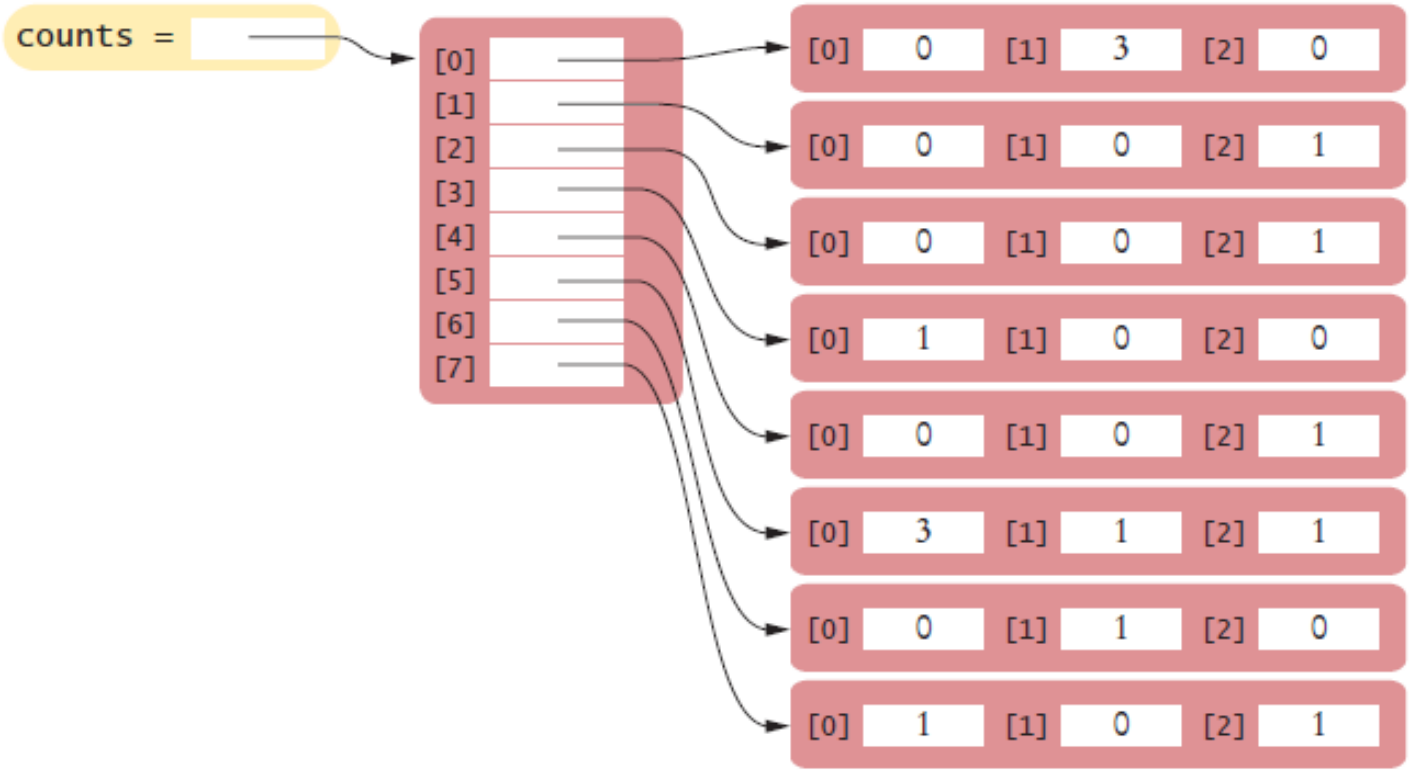
```
COUNTRIES = 8
MEDALS = 3

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 ]
    ]
```

Try it with PythonTutor!

Creating Tables (2)

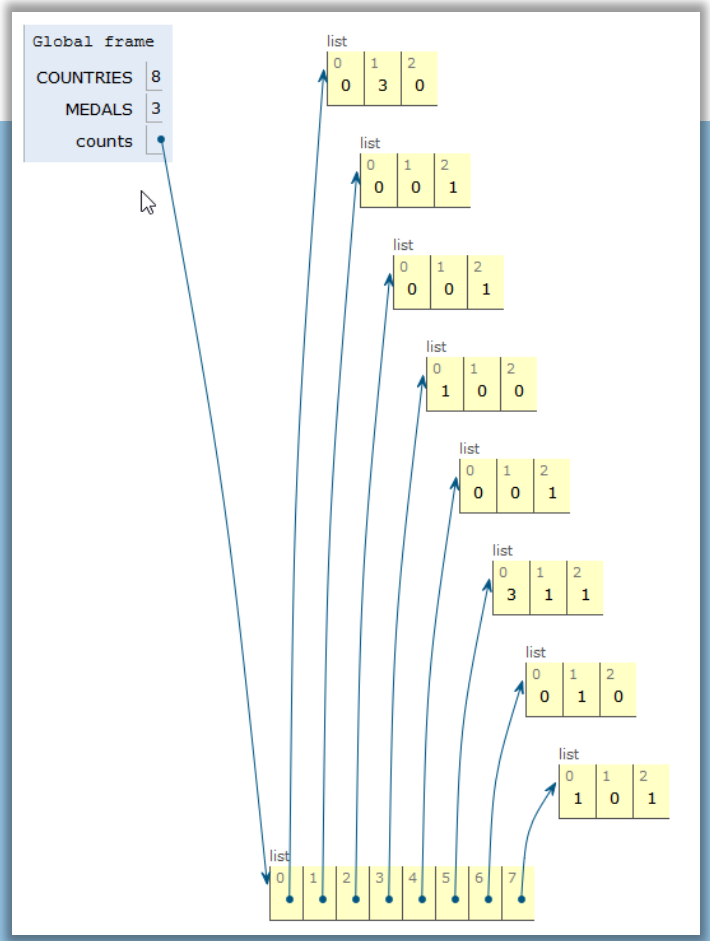
- It creates a list in which each element is itself another list:

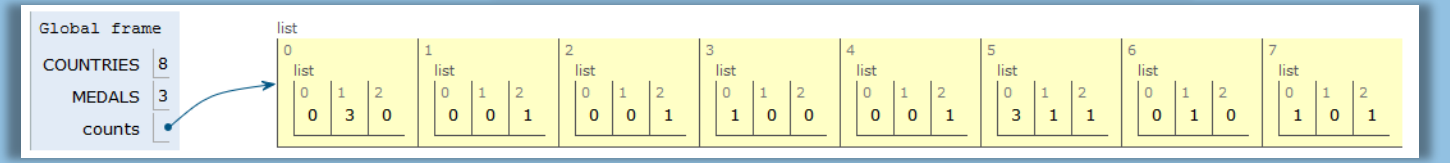


What really happens

```
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 (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)
```

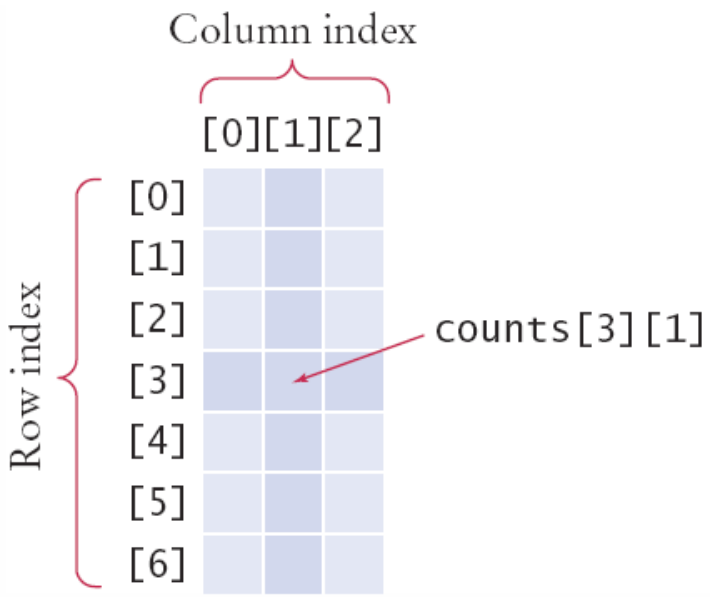
- The result is a table that consists of 5 rows and 20 columns

Accessing Elements

- Use two index values:
  - First the Row index, then the Column index

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

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




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

```
       0       3       0
       0       0       1
       0       0       1
       1       0       0
       0       0       1
       3       1       1
       0       1       0
       1       0       1
>
```

Locating Neighboring Elements

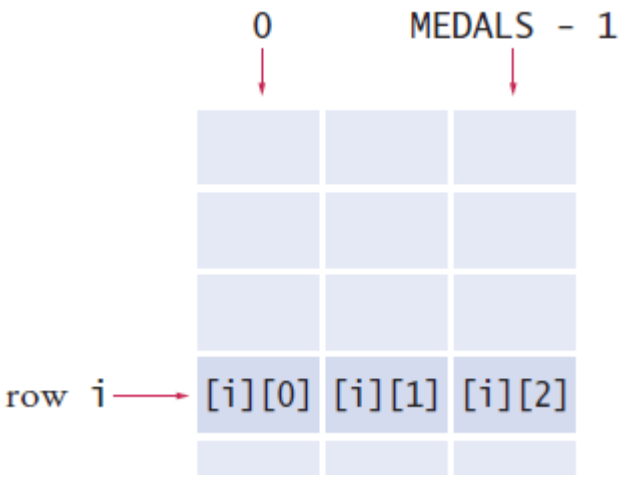
- Some programs that work with two-dimensional lists need to locate the elements that are adjacent to an element
- This task is particularly common in games
- You are at location `i, j`
- Watch out for edges!
  - No negative indexes!
  - Not off the ‘board’

[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]

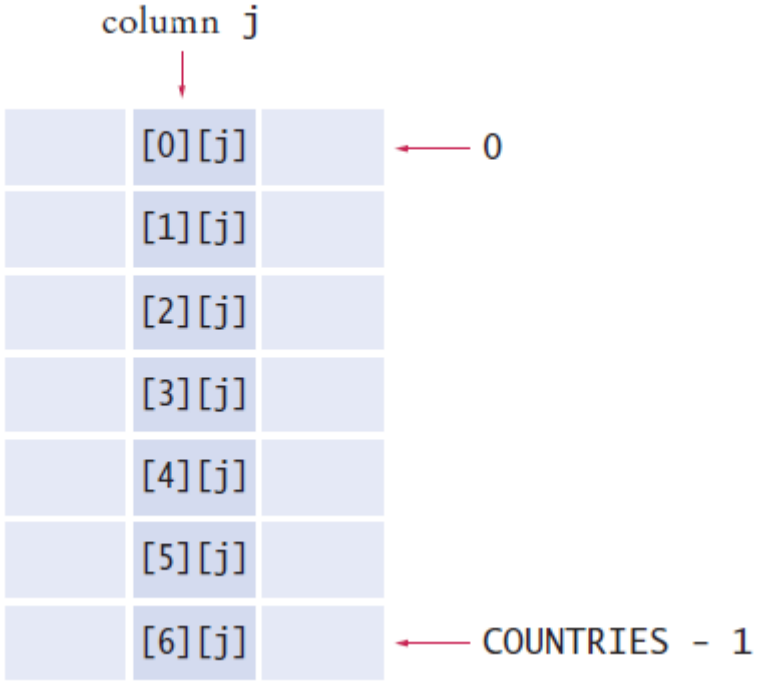
Adding Rows and Columns

Rows (x)

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



Columns (y)




```
coltotal = 0
for i in range(MEDALS):
    coltotal = coltotal + 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