Chapter 10 Lists

Opening Problem

Read one hundred numbers, compute their average, and find out how many numbers are above the average.

Solution

DataAnalysis

Run

Objectives

- To describe why lists are useful in programming (§10.1).
- To create lists (§10.2.1).
- To invoke list's append, insert, extend, remove, pop, index, count, sort, reverse methods (§10.2.2).
- To use the len, min/max, sum, and random.shuffle functions for a list (§10.2.3).
- To access list elements using indexed variables (§10.2.4).
- To obtain a sublist using the slicing operator [start:end] (§10.2.5).
- To use +, *, and in/not in operators on lists (§10.2.6).
- To traverse elements in a list using a for-each loop (§10.2.7).
- To create lists using list comprehension (§10.2.8).
- To compare lists using comparison operators (§10.2.9).
- To split a string to a list using the str's split method (§10.2.10).
- To use lists in the application development (§§10.3–10.5).
- To copy contents from one list to another (§10.6).
- To develop and invoke functions with list arguments and return value (§10.7–10.9).
- To search elements using the linear (§10.10.1) or binary (§10.10.2) search algorithm.
- To sort a list using the selection sort (§10.11.1)
- To sort a list using the insertion sort (§10.11.2).
- To develop the bouncing ball animation using a list (§10.12).

Python Collections (Arrays)

- There are four collection data types in the Python programming language:
  - **List** is a collection which is ordered and changeable. Allows duplicate members.
  - **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
  - **Set** is a collection which is unordered and unindexed. No duplicate members.
  - **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members. It stores key-value pairs.

Creating Lists

Creating list using the list class with the list() constructor

```
list1 = list() # Create an empty list
list2 = list([2, 3, 4]) # Create a list with elements 2, 3, 4
list3 = list(["red", "green", "blue"]) # Create a list with strings
list4 = list(range(3, 6)) # Create a list with elements 3, 4, 5
list5 = list("abcd") # Create a list with characters a, b, c
```

For convenience, you may create a list using the following syntax:

```
list1 = [] # Same as list()
list2 = [2, 3, 4] # Same as list([2, 3, 4])
list3 = ["red", "green"] # Same as list(["red", "green"])
```

List Is a Sequence Type

- Strings and lists are sequence types in Python.
  - A string is a sequence of characters,
  - while a list is a sequence of any elements.

The common operations for sequences

TABLE 10.1 Common Operations for Sequence s

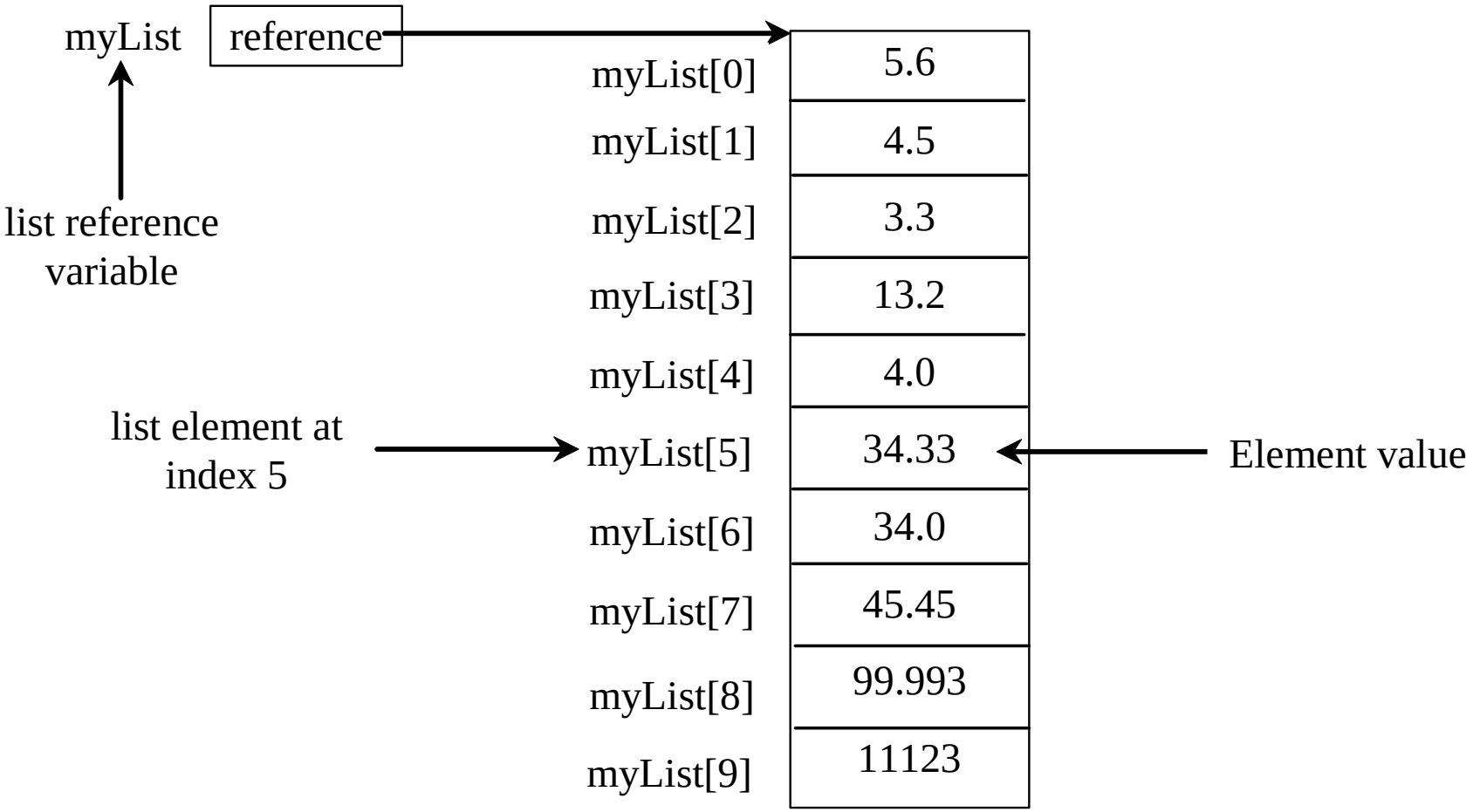
Operation	*Description*
`x in s`	True if element x is in sequence s.
`x not in s`	True if element x is not in sequence s.
`s1 + s2`	Concatenates two sequences s1 and s2.
`s * n, n * s`	n copies of sequence s concatenated.
`s[i]`	ith element in sequence s.
`s[i : j]`	Slice of sequence s from index i to j − 1.
`len(s)`	Length of sequence s, i.e., the number of elements in s.
`min(s)`	Smallest element in sequence s.
`max(s)`	Largest element in sequence s.
`sum(s)`	Sum of all numbers in sequence s.
`for loop`	Traverses elements from left to right in a `for` loop.
`<, <=, >, >=, =, !=`	Compares two sequences.

Functions for lists

```
>>> list1 = [2, 3, 4, 1, 32]
>>> len(list1)
5
>>> max(list1)
32
>>> min(list1)
1
>>> sum(list1)
42
>>> import random
>>> random.shuffle(list1) # Shuffle the items in the list
>>> list1
[4, 1, 2, 32, 3]
```

Indexer Operator []

myList = [5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123]



The +, *, [:], and in Operators

```
>>> list1 = [2, 3]
>>> list2 = [1, 9]
>>> list3 = list1 + list2
>>> list3
[2, 3, 1, 9]
>>> list3 = 2 * list1
>>> list3
[2, 3, 2, 3, 2, 3]
>>> list4 = list3[2 : 4]
>>> list4
[2, 3]
```

The +, *, [:], and in Operators

```
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> list1[-1]
21
>>> list1[-3]
2
```

```
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> 2 in list1
True
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> 2.5 in list1
False
```

List Slicing [start : end]

- The index operator allows you to select an element at the specified index.
- The slicing operator returns a slice of the list using the syntax list[start : end]. The slice is a sublist from index start to index end – 1.

List Slicing [start : end]

- The starting index or ending index may be omitted. In this case, the starting index is 0 and the ending index is the last index.
- You can use a negative index in slicing if you want to start the search from the end of the list.
- If start >= end, list[start : end] returns an empty list. If end specifies a position beyond the end of the list, Python will use the length of the list for end instead.

```
1  >>> list1 = [2, 3, 5, 2, 33, 21]
2  >>> list1[ : 2]
3  [2, 3]
4  >>> list1[3 : ]
5  [2, 33, 21]
6  >>>
```

```
1  >>> list1 = [2, 3, 5, 2, 33, 21]
2  >>> list1[1 : -3]
3  [3, 5]
4  >>> list1[-4 : -2]
5  [3, 5]
6  >>>
```

Comparing Lists

```
>>>list1 = ["green", "red", "blue"]
>>>list2 = ["red", "blue", "green"]
>>>list2 == list1
False
>>>list2 != list1
True
>>>list2 >= list1
False
>>>list2 > list1
False
>>>list2 < list1
True
>>>list2 <= list1
True
```

- The comparison uses lexicographical ordering:
  - the first two elements are compared, and if they differ this determines the outcome of the comparison;
  - if they are equal, the next two elements are compared, and so on, until either list is exhausted.

List Comprehension

List comprehensions provide a concise way to create items from sequence. A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression. Here are some examples:

```
>>> list1 = [x for x range(0, 5)] # Returns a list of 0, 1, 2, 4
>>> list1
[0, 1, 2, 3, 4]
>>> list2 = [0.5 * x for x in list1]
>>> list2
[0.0, 0.5, 1.0, 1.5, 2.0]
>>> list3 = [x for x in list2 if x < 1.5]
>>> list3
[0.0, 0.5, 1.0]
```

off-by-one Error

```
i = 0
while i <= len(lst):
    print(lst[i])
    i += 1
```

Programmers often mistakenly execute a loop one time more or less than intended. This kind of mistake is commonly known as the off-by-one error.

```
lst = [2,5,8,99,11,77]

i = 0
while i <= len(lst):
    print(i, lst[i])
    i += 1

0 2
1 5
2 8
3 99
4 11
5 77

---------------------------------
IndexError
<ipython-input-6-307dd5a426b6> in <m
      1 i = 0
      2 while i <= len(lst):
----> 3     print(i, lst[i])
      4     i += 1

IndexError: list index out of range
```

Loop Through a List

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

```
apple
banana
cherry
```

Append, insert, remove

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

['apple', 'banana', 'cherry', 'orange']

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

['apple', 'orange', 'banana', 'cherry']

The `remove()` method removes the specified item:

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

['apple', 'cherry']

Pop, clear, del

The `pop()` method removes the specified index, (or the last item if index is not specified):

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

['apple', 'banana']

The `clear()` method empties the list:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

[]

The `del` keyword removes the specified index:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

['banana', 'cherry']

The `del` keyword can also delete the list completely:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Splitting a String to a List

```
items = "Welcome to the US".split()
print(items)
['Welcome', 'to', 'the', 'US']


items = "34#13#78#45".split("#")
print(items)
['34', '13', '78', '45']
```

list Methods

list	
append(x: object): None	Add an item x to the end of the list.
insert(index: int, x: object): None	Insert an item x at a given index. Note that the first element in the list has index 0.
remove(x: object): None	Remove the first occurrence of the item x from the list.
index(x: object): int	Return the index of the item x in the list.
count(x: object): int	Return the number of times item x appears in the list.
sort(): None	Sort the items in the list.
reverse(): None	Reverse the items in the list.
extend(l: list): None	Append all the items in L to the list.
pop([i]): object	Remove the item at the given position and return it. The square bracket denotes that parameter is optional. If no index is specified, list.pop() removes and returns the last item in the list.

Problem: Lotto Numbers

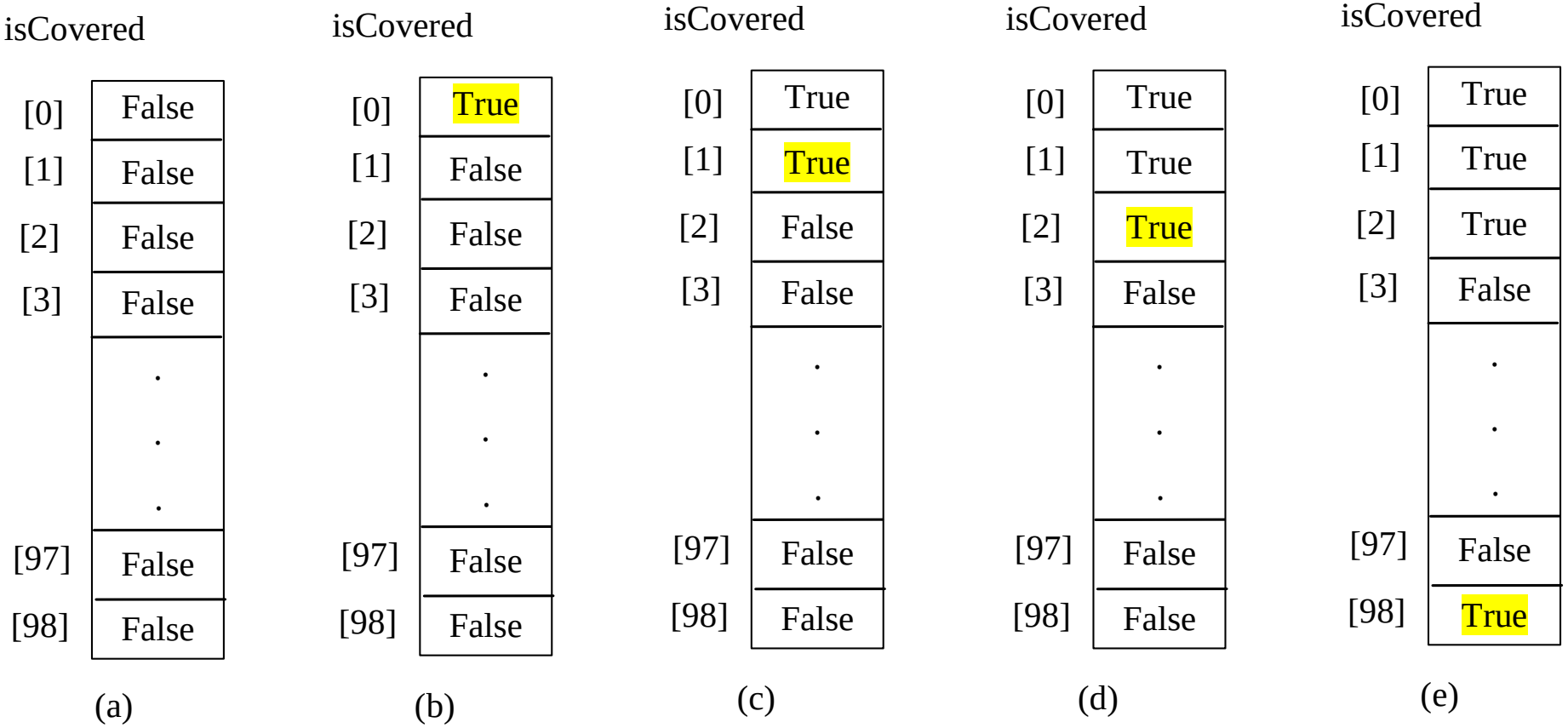
Suppose you play the Pick-10 lotto. Each ticket has 10 unique numbers ranging from 1 to 99. You buy a lot of tickets. You like to have your tickets to cover all numbers from 1 to 99. Write a program that reads the ticket numbers from a file and checks whether all numbers are covered. Assume the last number in the file is 0.

Lotto Numbers Sample Data

LottoNumbers

Run

Problem: Lotto Numbers



Problem: Deck of Cards

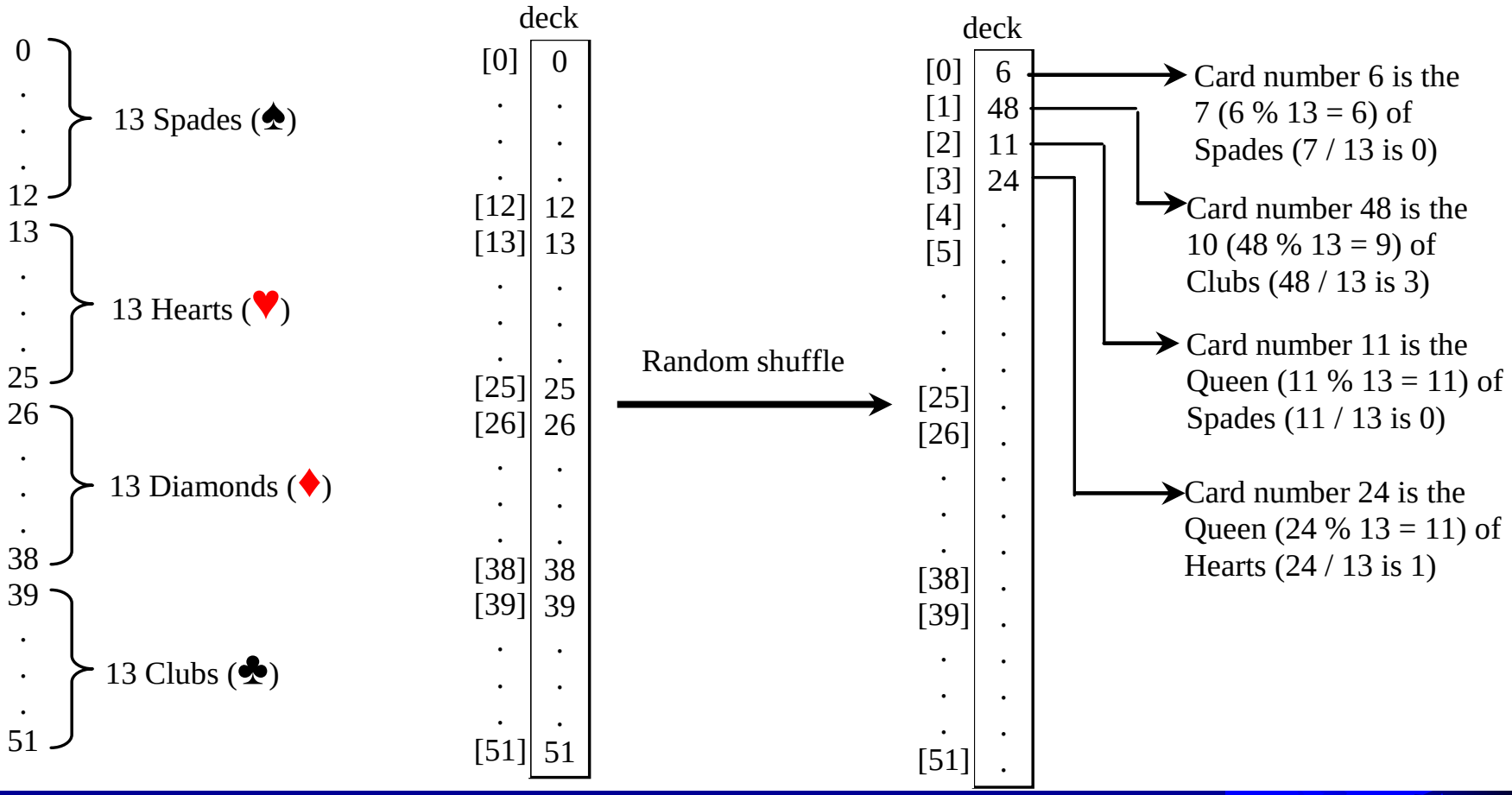
The problem is to write a program that picks four cards randomly from a deck of 52 cards. All the cards can be represented using a list named deck, filled with initial values 0 to 51, as follows:

```
deck = [x for x in range(0, 52)]
```

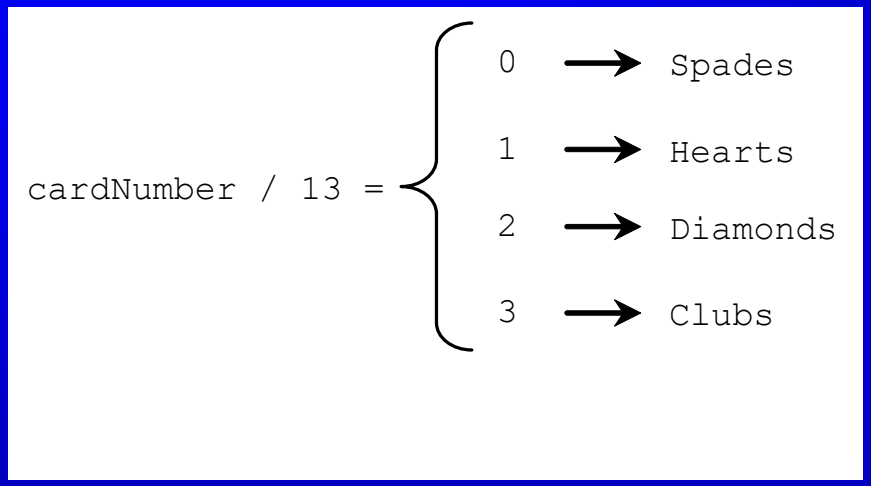
DeckOfCards

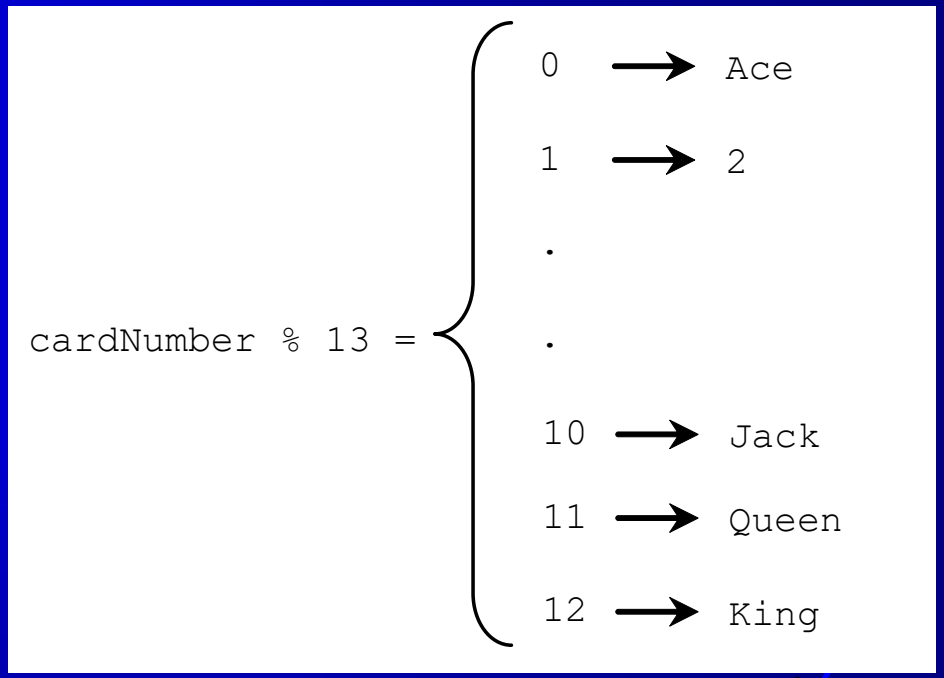
Run

Problem: Deck of Cards, cont.



Problem: Deck of Cards, cont.





DeckOfCards

Run

GUI: Deck of Cards

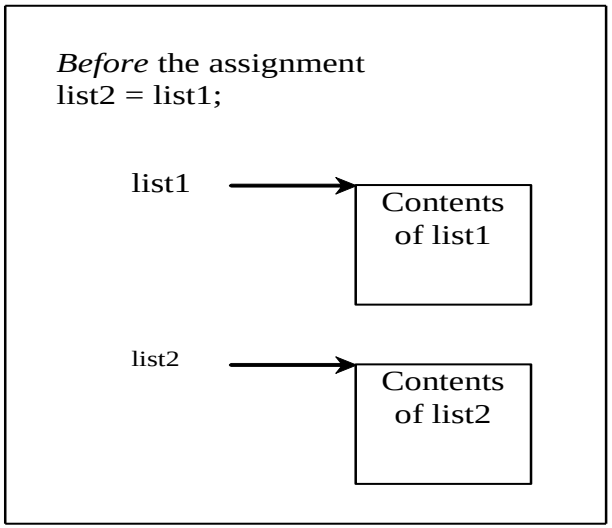
DeckOfCardsGUI

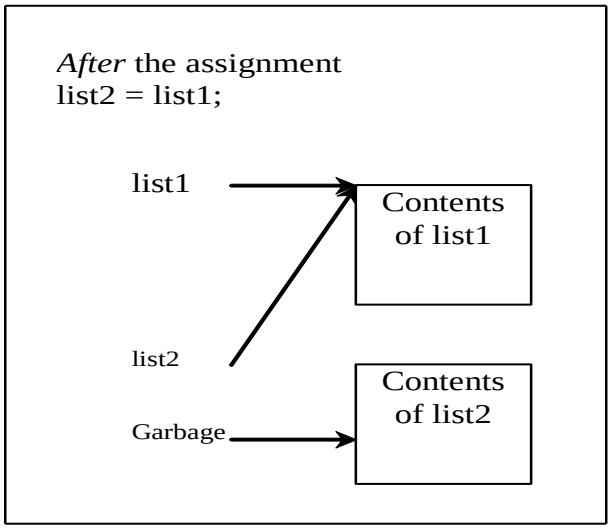
DeckOfCards

Copying Lists

Often, in a program, you need to duplicate a list or a part of a list. In such cases you could attempt to use the assignment statement (=), as follows:

```
list2 = list1;
```





You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`

Copy a List

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

['apple', 'banana', 'cherry']

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

['apple', 'banana', 'cherry']

Join Two Lists

Join two list:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

['a', 'b', 'c', 1, 2, 3]

Append list2 into list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

['a', 'b', 'c', 1, 2, 3]

Use the `extend()` method to add list2 at the end of list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

['a', 'b', 'c', 1, 2, 3]

Passing Lists to Functios

```
def printList(lst):
    for element in lst:
        print(element)
```

```
Invoke the function

lst = [3, 1, 2, 6, 4, 2]
printList(lst)
```

```
Invoke the function
printList([3, 1, 2, 6, 4, 2])
```

Anonymous list

Pass By Value

- Python uses *pass-by-value* to pass arguments to a function.
- There are important differences between passing the values of variables of numbers and strings and passing lists.
  - Immutable objects
  - Changeable objects

Pass By Value (Immutable objects)

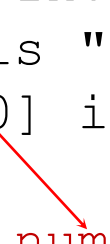
- For an argument of a number or a string, the original value of the number and string outside the function is not changed,
- because numbers and strings are immutable in Python.

Pass By Value (changeable objects)

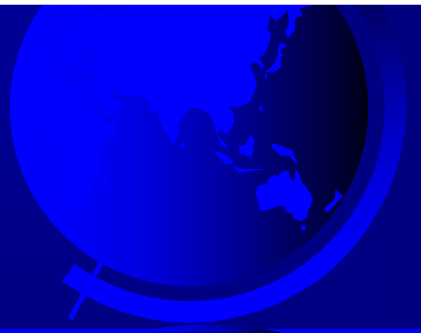
- For an argument of a list, the value of the argument is a reference to a list;
- this reference value is passed to the function.
- Semantically, it can be best described as *pass-by-sharing*, i.e., the list in the function is the same as the list being passed.
- So if you change the list in the function, you will see the change outside the function.

Simple Example

```
def main():
    x = 1 # x represents an int value
    y = [1, 2, 3] # y represents a list
    m(x, y) # Invoke f with arguments x and y
    print("x is " + str(x))
    print("y[0] is " + str(y[0]))

def m(number, numbers):
    number = 1001 # Assign a new value to number
    numbers[0] = 5555 # Assign a new value to numbers[0]

main()
```




```
x is 1
y[0] is 5555
```



Subtle Issues Regarding Default Arguments

```
def add(x, lst = []):
    if not(x in lst):
        lst.append(x)
    return lst
list1 = add(1)
print(list1)
list2 = add(2)
print(list2)
list3 = add(3, [11, 12, 13, 14])
print(list3)
list4 = add(4)
print(list4)
```

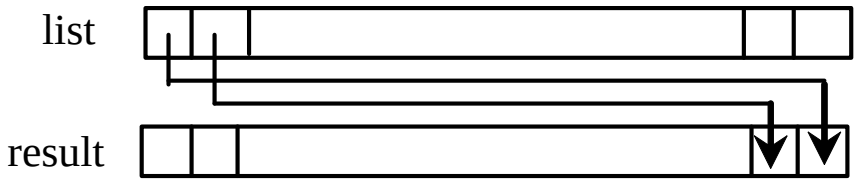
default value is created only once.

Output

```
[1]
[1, 2]
[11, 12, 13, 14]
[1, 2, 4]
```

Returning a List from a Function

```
def reverse(list):
    result = []

    for element in list:
        result.insert(0, element)

    return result
```

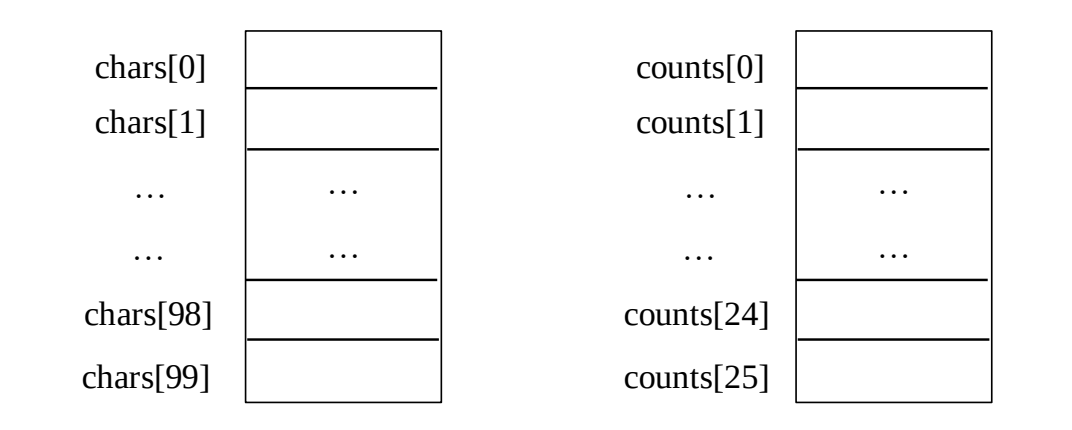


list1 = [1, 2, 3, 4, 5, 6]
list2 = reverse(list1)

Note that list already has the reverse method list.reverse()

Problem: Counting Occurrence of Each Letter

- Generate 100 lowercase letters randomly and assign to a list of characters.
- Count the occurrence of each letter in the list.



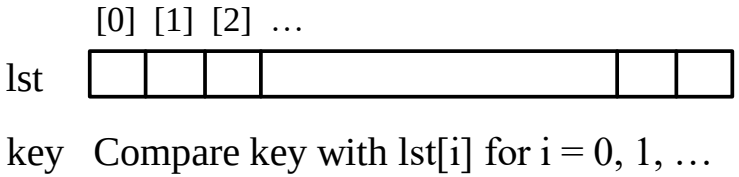
CountLettersInList

Run

Searching Lists

Searching is the process of looking for a specific element in a list; for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. There are many algorithms and data structures devoted to searching. In this section, two commonly used approaches are discussed, *linear search* and *binary search*.

```
# The function for finding a key in the list
def linearSearch(lst, key):
    for i in range(0, len(lst)):
        if key == lst[i]:
            return i

    return -1
```



Linear Search

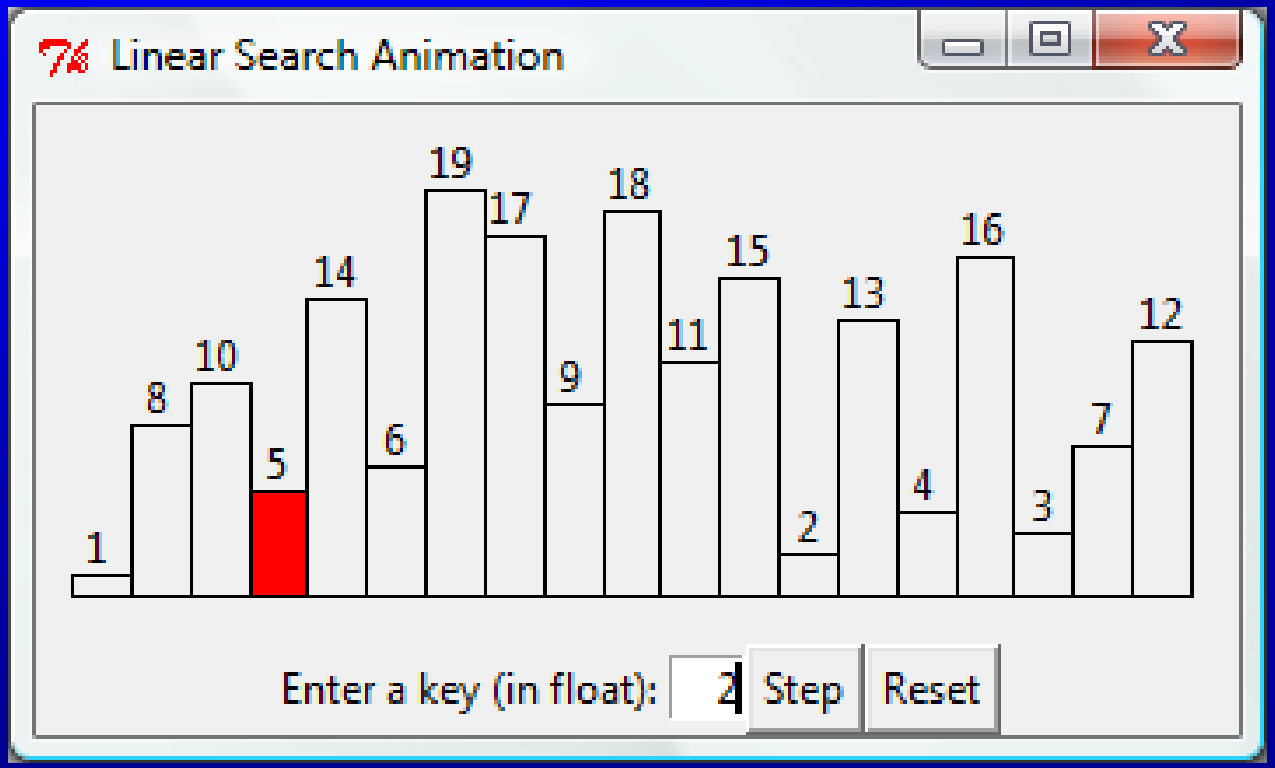
The linear search approach compares the key element, key, *sequentially* with each element in list. The method continues to do so until the key matches an element in the list or the list is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the list that matches the key. If no match is found, the search returns -1.

animation

Linear Search Animation

Linear Search Animation

http://www.cs.armstrong.edu/liang/animation/LinearSearchAnimation.html



Run

Binary Search

For binary search to work, the elements in the list must already be ordered. Without loss of generality, assume that the list is in ascending order.

e.g., 2 4 7 10 11 45 50 59 60 66 69 70 79

The binary search first compares the key with the element in the middle of the list.

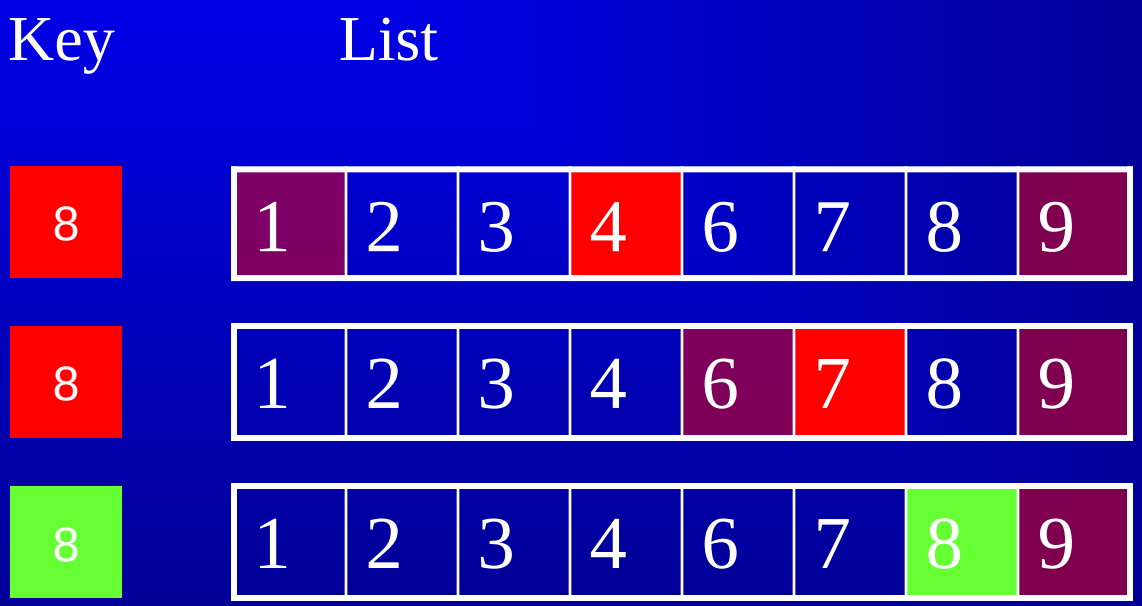
Binary Search, cont.

Consider the following three cases:

- If the key is less than the middle element, you only need to search the key in the first half of the list.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you only need to search the key in the second half of the list.

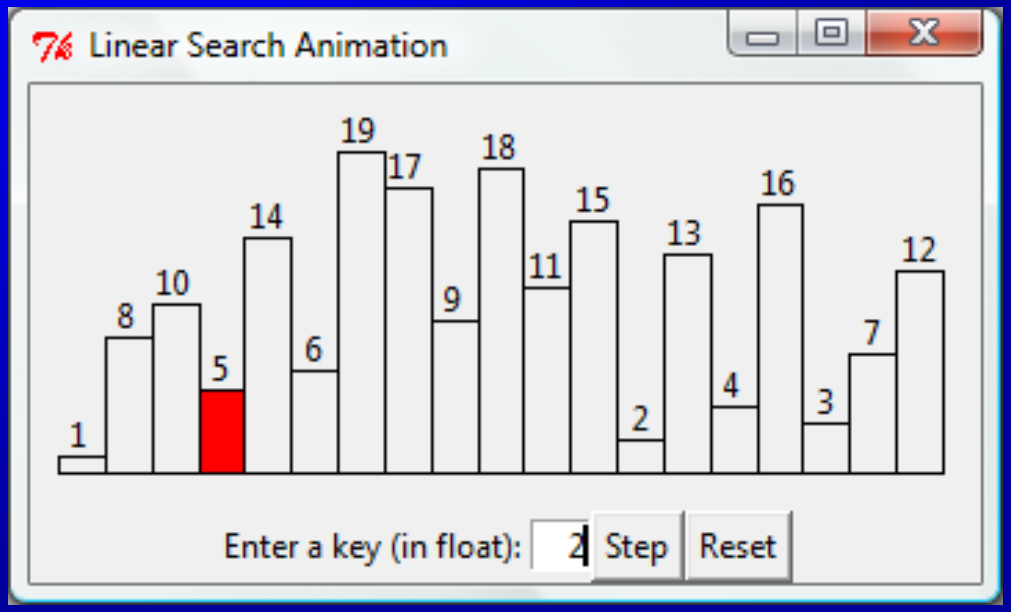
animation

Binary Search



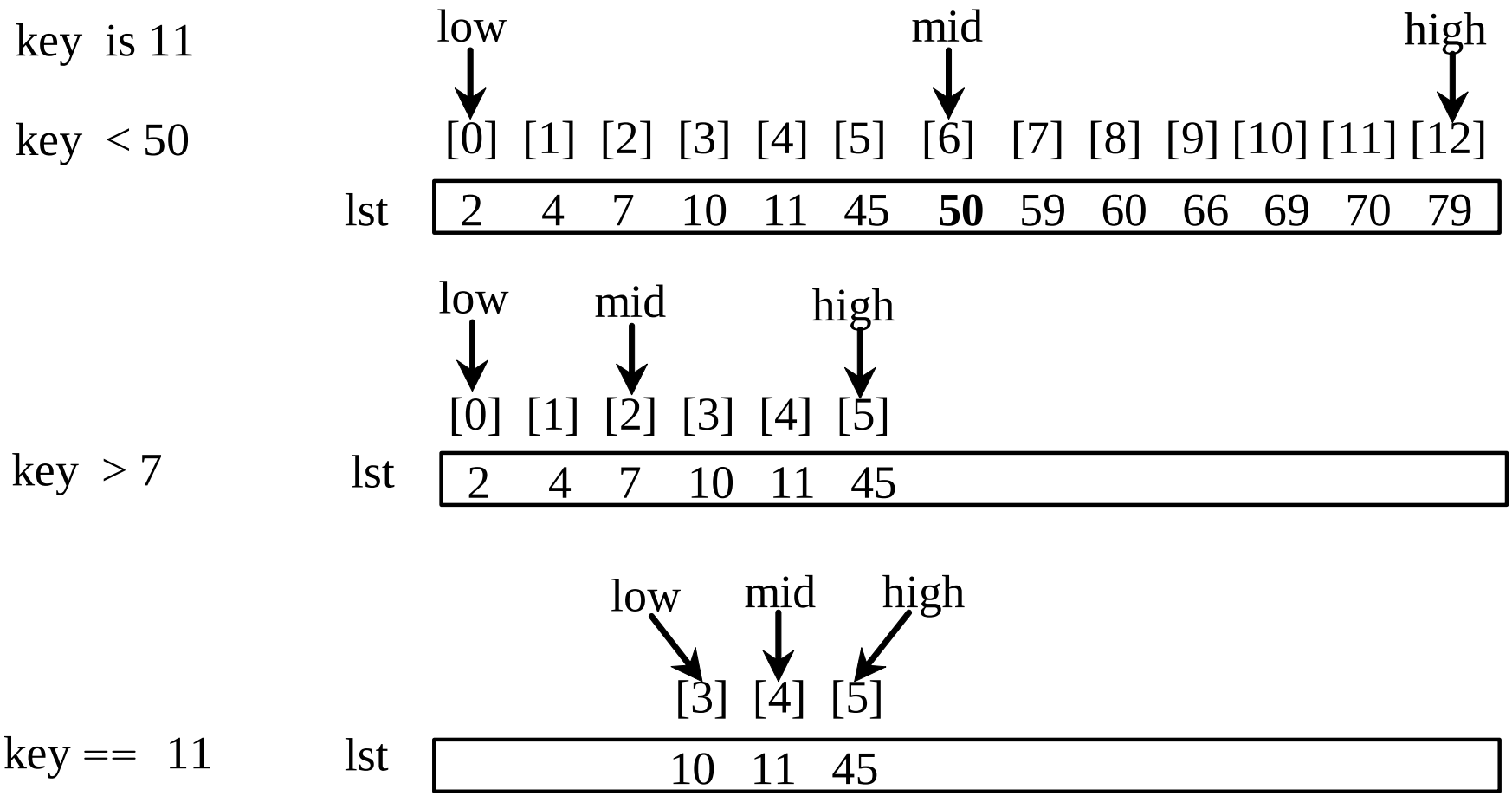
Binary Search Animation

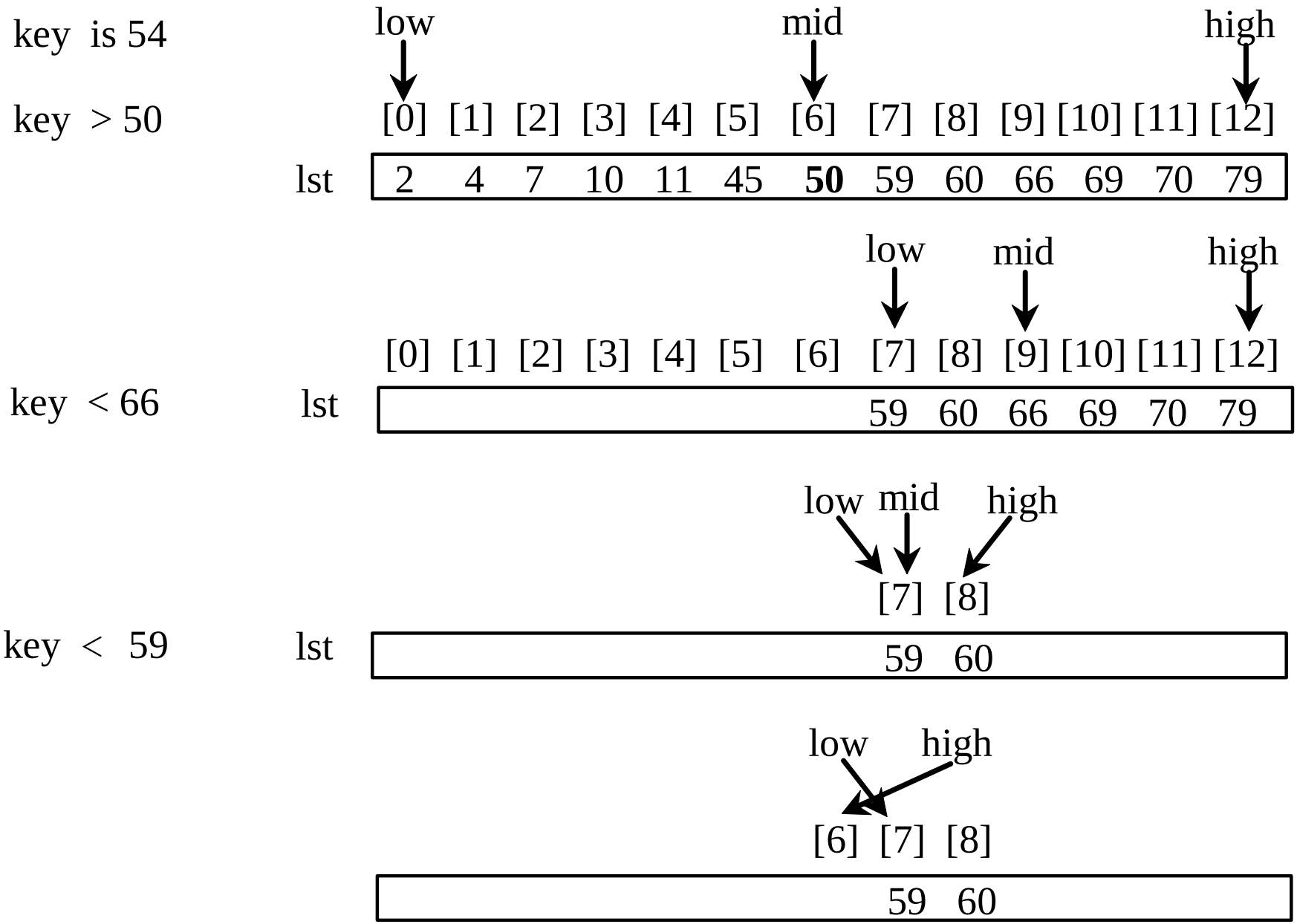
http://www.cs.armstrong.edu/liang/animation/BinarySearchAnimation.html



Run

Binary Search, cont.





Binary Search, cont.

The binarySearch method returns the index of the element in the list that matches the search key if it is contained in the list. Otherwise, it returns

- insertion point - 1.

The insertion point is the point at which the key would be inserted into the list.

From Idea to Soluton

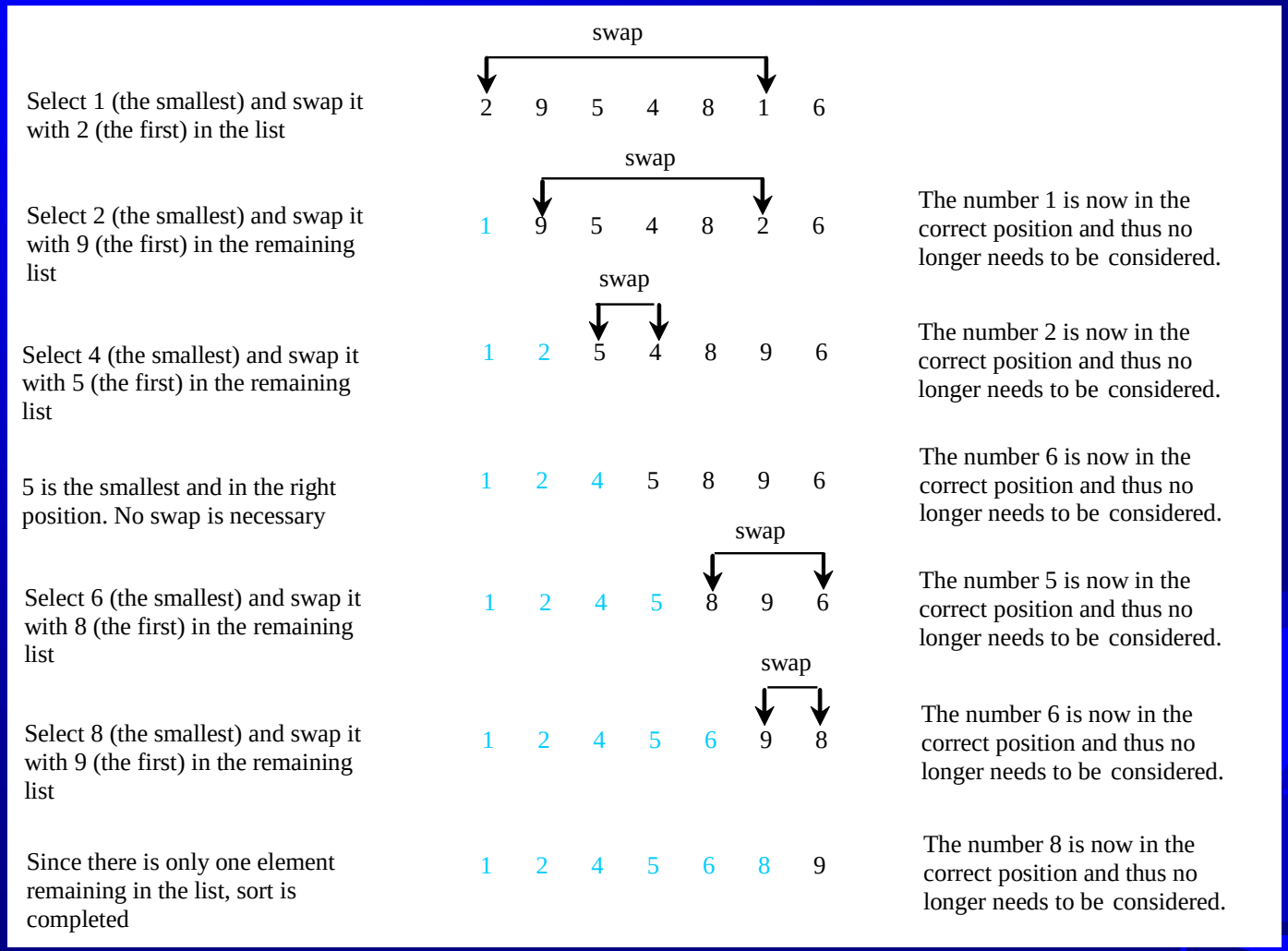
```
# Use binary search to find the key in the list
def binarySearch(lst, key):
    low = 0
    high = len(lst) - 1

    while high >= low:
        mid = (low + high) // 2
        if key < lst[mid]:
            high = mid - 1
        elif key == lst[mid]:
            return mid
        else:
            low = mid + 1

    return –low - 1 # Now high < low, key not found
```

Sorting Lists

Sorting, like searching, is also a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces two simple, intuitive sorting algorithms: *selection sort* and *insertion sort*.

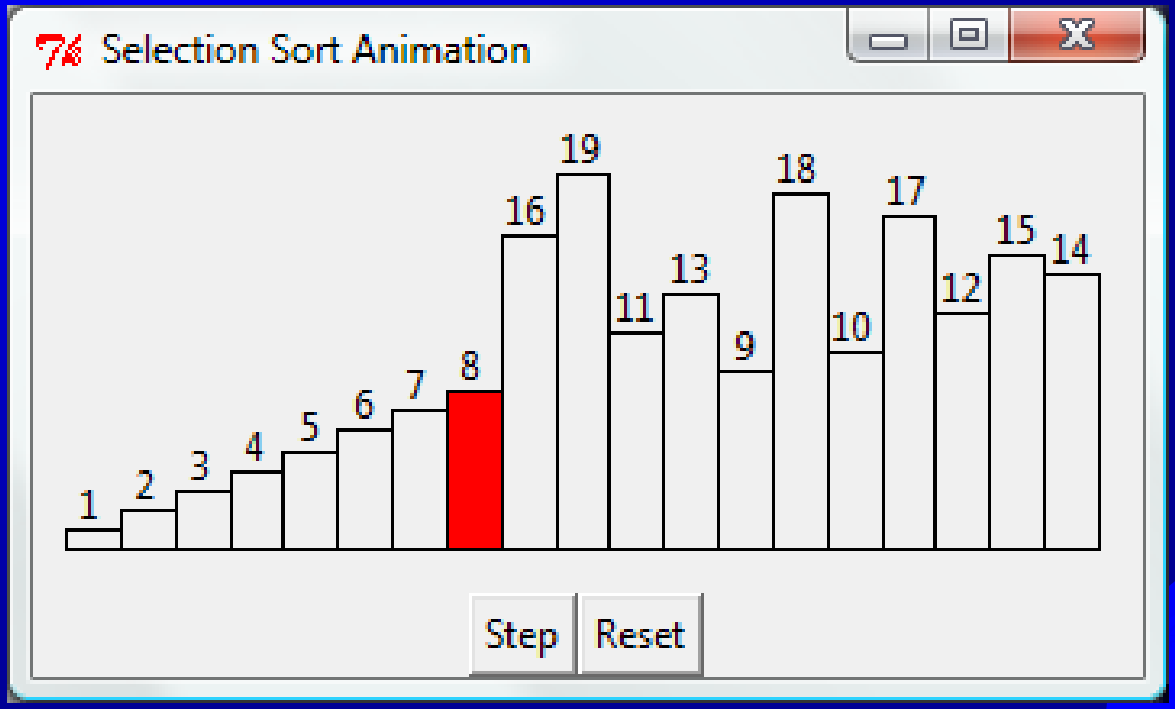
Selection Sort

Selection sort finds the largest number in the list and places it last. It then finds the largest number remaining and places it next to last, and so on until the list contains only a single number. Figure 6.17 shows how to sort the list {2, 9, 5, 4, 8, 1, 6} using selection sort.



Selection Sort Animation

http://www.cs.armstrong.edu/liang/animation/SelectionSortAnimation.html



Run

From Idea to Solution

for i in range(0, len(lst)):
    select the smallest element in lst[i.. len(lst)-1]
    swap the smallest with lst[i], if necessary
    # lst[i] is in its correct position.
    # The next iteration apply on lst[i+1..len(lst)-1]

```
lst[0] lst[1] lst[2] lst[3] ...                 lst[10]

lst[0] lst[1] lst[2] lst[3] ...                 lst[10]

lst[0] lst[1] lst[2] lst[3] ...                 lst[10]

lst[0] lst[1] lst[2] lst[3] ...                 lst[10]

lst[0] lst[1] lst[2] lst[3] ...                 lst[10]

                            ...

lst[0] lst[1] lst[2] lst[3] ...                 lst[10]
```

```
for i in range(0, len(lst)):
    select the smallest element in lst[i.. len(lst)-1]
    swap the smallest with lst[i], if necessary
    # lst[i] is in its correct position.
    # The next iteration apply on lst[i+1..len(lst)-1]
```

Expand

```
currentMin = lst[i]

    for j in range(i + 1, len(lst)):
        if currentMin > lst[j]:
            currentMin = lst[j]
```

```
for i in range(0, len(lst)):
    select the smallest element in lst[i.. len(lst)-1]
    swap the smallest with lst[i], if necessary
    # lst[i] is in its correct position.
    # The next iteration apply on lst[i+1..len(lst)-1]
```

Expand

```
# Find the minimum in the lst[i..len(lst)-1]
    currentMin = lst[i]
    currentMinIndex = i
    for j in range(i + 1, len(lst)):
        if currentMin > lst[j]:
            currentMin = lst[j]
            currentMinIndex = j
    # Swap lst[i] with lst[currentMinIndex] if necessary
    if currentMinIndex != i:
        lst[currentMinIndex] = lst[i]
        lst[i] = currentMin
```

Wrap it in a Function

```
# The function for sorting the numbers
def selectionSort(lst):
    for i in range(0, len(lst) - 1):
        # Find the minimum in the lst[i..len(lst)-1]
        currentMin = lst[i]
        currentMinIndex = i
        for j in range(i + 1, len(lst)):
            if currentMin > lst[j]:
                currentMin = lst[j]
                currentMinIndex = j
        # Swap lst[i] with lst[currentMinIndex] if necessary
        if currentMinIndex != i:
            lst[currentMinIndex] = lst[i]
            lst[i] = currentMin
```

Invoke it

selectionSort(yourList)

Insertion Sort

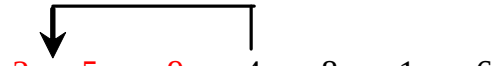
myList = [2, 9, 5, 4, 8, 1, 6] # Unsorted

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

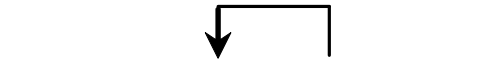
Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step2: The sorted sublist is [2, 9]. Insert 5 into the sublist.

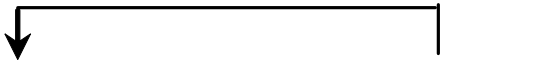
Step 3: The sorted sublist is [2, 5, 9]. Insert 4 into the sublist.



Step 4: The sorted sublist is [2, 4, 5, 9]. Insert 8 into the sublist.



Step 5: The sorted sublist is [2, 4, 5, 8, 9]. Insert 1 into the sublist.

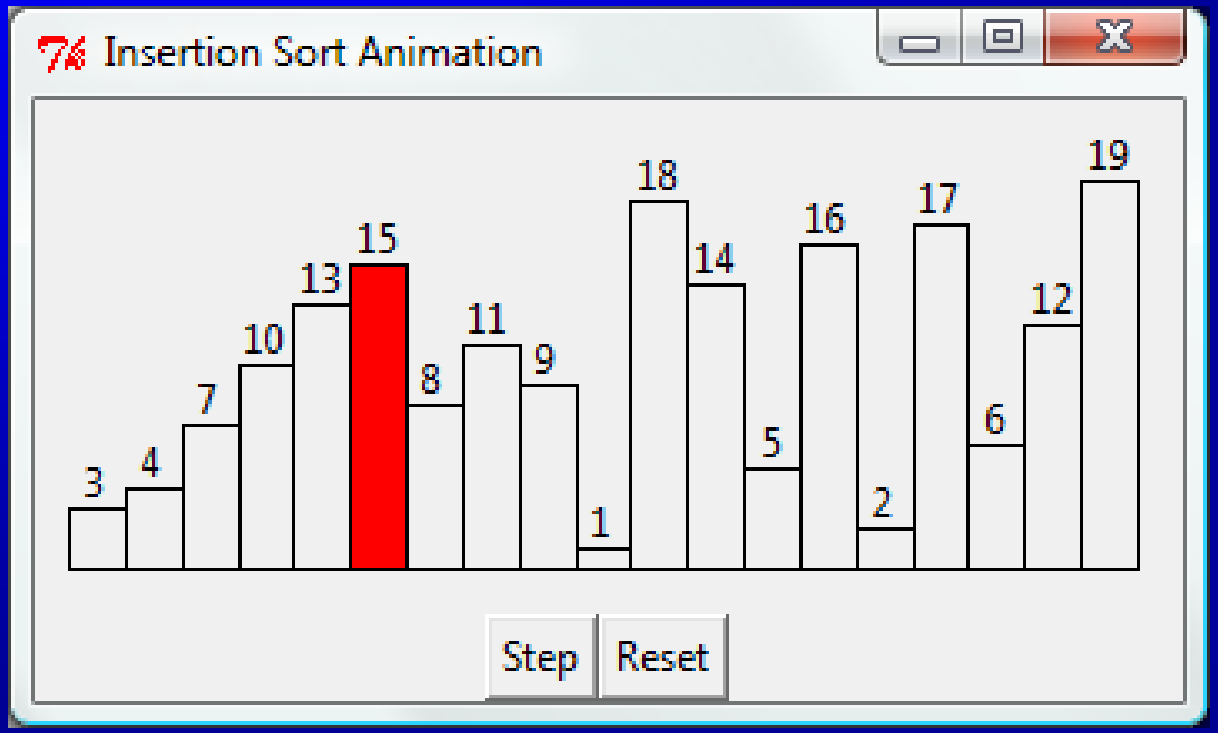


Step 6: The sorted sublist is [1, 2, 4, 5, 8, 9]. Insert 6 into the sublist.

Step 7: The entire list is now sorted

Insertion Sort Animation

http://www.cs.armstrong.edu/liang/animation/InsertionSortAnimation.html

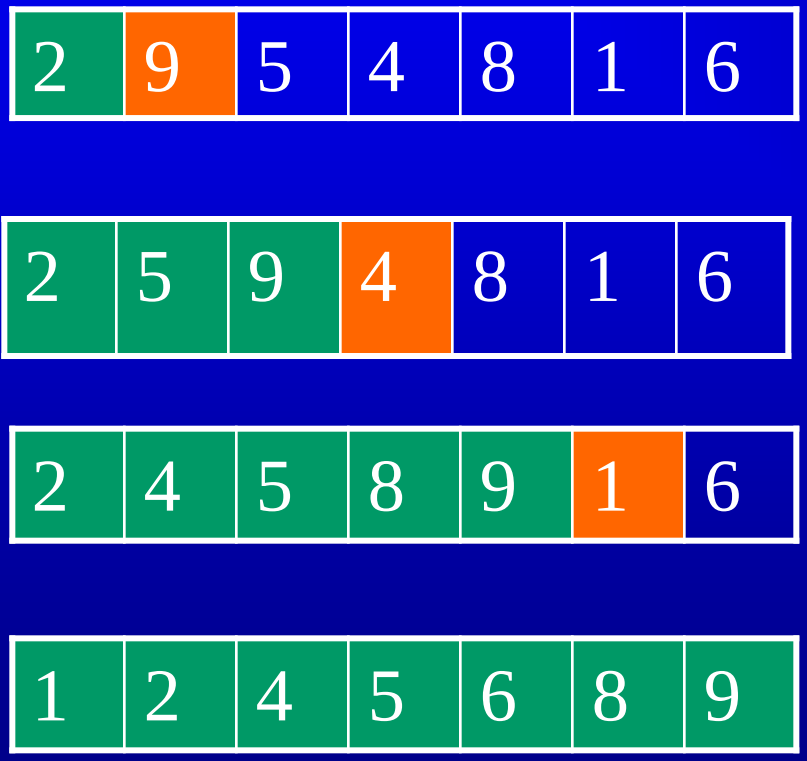


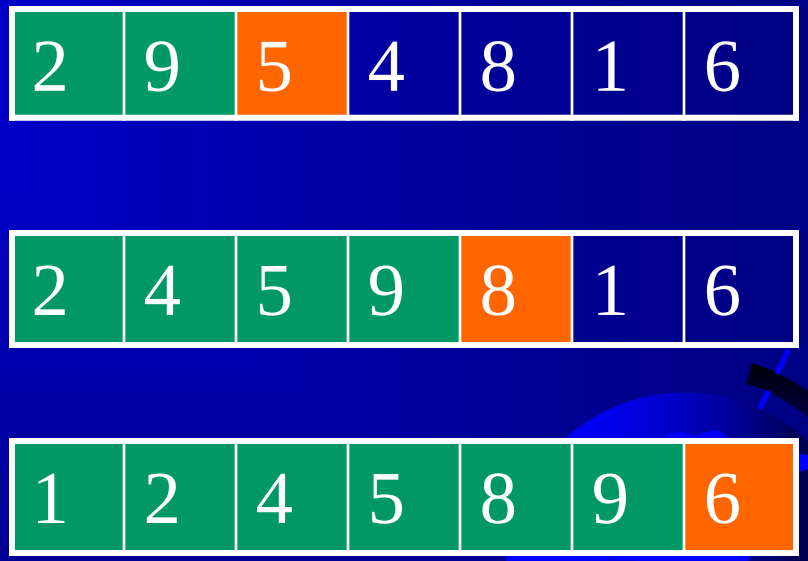
Run

animation

Insertion Sort

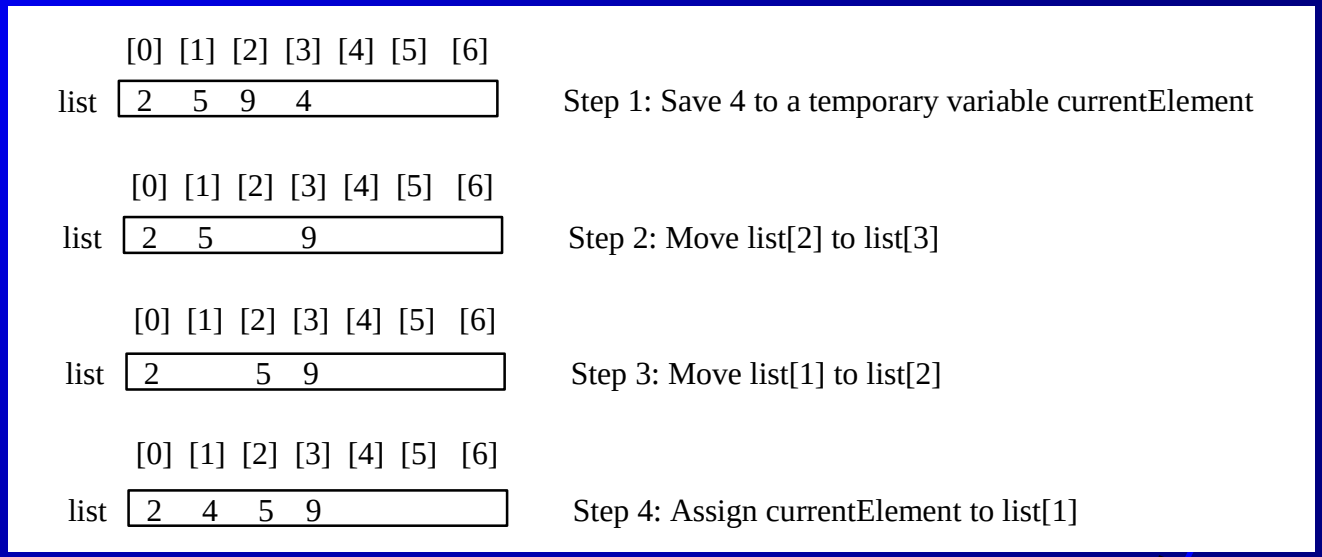
myList = [2, 9, 5, 4, 8, 1, 6] # Unsorted





How to Insert?

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.



From Idea to Solution

```
for i in range(1, len(lst)):
    insert lst[i] into a sorted sublist lst[0..i-1] so that
    lst[0..i] is sorted.
```

```
lst[0]
```

```
lst[0] lst[1]
```

```
lst[0] lst[1] lst[2]
```

```
lst[0] lst[1] lst[2] lst[3]
```

```
lst[0] lst[1] lst[2] lst[3] ...
```

From Idea to Solution

```
for i in range(1, len(lst)):
    insert lst[i] into a sorted sublist lst[0..i-1] so that
    lst[0..i] is sorted.
```

Expand

```
k = i - 1
while k >= 0 and lst[k] > currentElement:
            lst[k + 1] = lst[k]
            k -= 1
# Insert the current element into lst[k + 1]
lst[k + 1] = currentElement
```

InsertSort

Case Studies: Bouncing Balls

Ball
x: int
y: int
dx: int
dy: int
color: Color
radius: int

The x-, y-coordinates for the center of the ball. By default, it is (0, 0).

dx and dy are the increment for (x, y).

The color of the ball.

The radius of the ball.

BouncingBalls

Run