Chapter 7 Lists

Opening Problem

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

Objectives

- To describe why lists are useful in programming (§7.1).
- To create lists (§7.2.1).
- To invoke list's append, insert, extend, remove, pop, index, count, sort, reverse methods (§7.2.2).
- To use the len, min/max, sum, and random.shuffle functions for a list (§7.2.3).
- To access list elements using indexed variables (§7.2.4).
- To obtain a sublist using the slicing operator [start:end] (§7.2.5).
- To use +, *, and in/not in operators on lists (§7.2.6).
- To traverse elements in a list using a for-each loop (§7.2.7).
- To create lists using list comprehension (§7.2.8).
- To compare lists using comparison operators (§7.2.9).
- To split a string to a list using the str's split method (§7.2.10).
- To use lists in the application development (§§7.3–7.5).
- To copy contents from one list to another (§7.6).
- To develop and invoke functions with list arguments and return value (§7.7–7.9).
- To search elements using the linear (§7.7.1) search algorithm.
- To sort a list using the selection sort (§7.11.1)

Creating Lists

Creating list using the list class

```
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 Methods

list
append(x: object): None
insert(index: int, x: object): None
remove(x: object): None
index(x: object): int
count(x: object): int
sort(): None
reverse(): None
extend(l: list): None
pop([i]): object

Add an item x to the end of the list.

Insert an item x at a given index. Note that the first element in the list has index 0.

Remove the first occurrence of the item x from the list.

Return the index of the item x in the list.
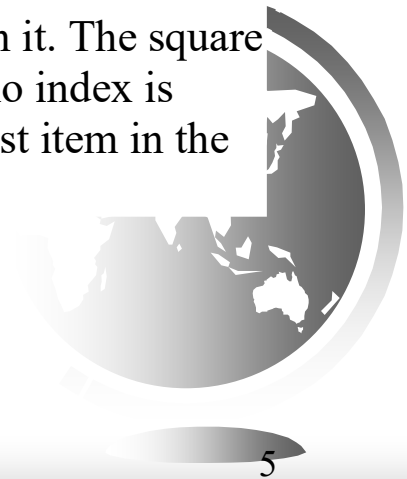
Return the number of times item x appears in the list.

Sort the items in the list.

Reverse the items in the list.

Append all the items in L to the list.

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.

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]

myList reference →	myList[0]	5.6
list reference variable ↑	myList[1]	4.5
	myList[2]	3.3
	myList[3]	13.2
	myList[4]	4.0
list element at index 5 →	myList[5]	34.33 ← Element value
	myList[6]	34.0
	myList[7]	45.45
	myList[8]	99.993
	myList[9]	11123
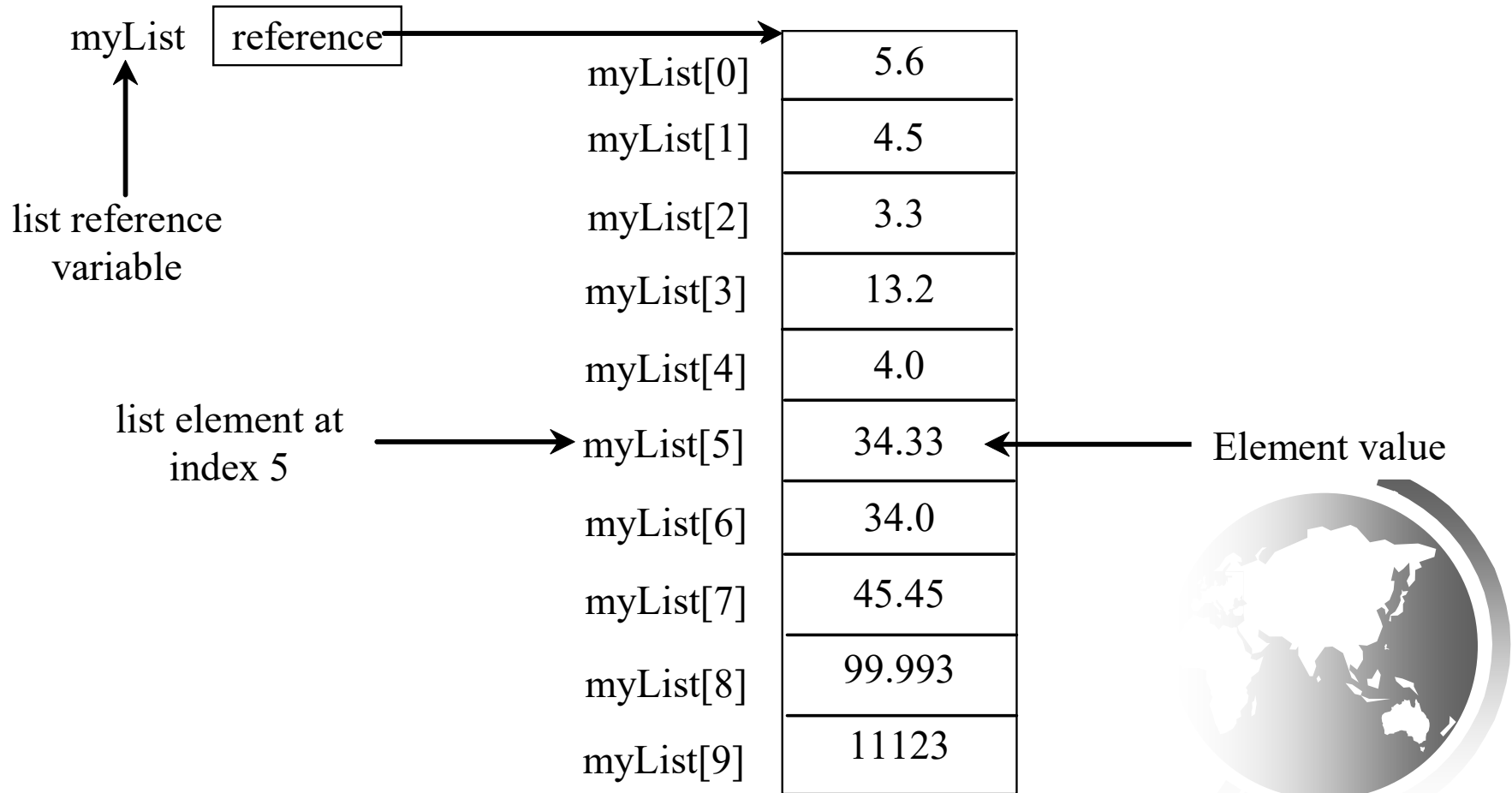
The +, *, [:], and in Operators

```
>>> 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
```

off-by-one Error

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

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]
```

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
```

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']

Analyze Numbers

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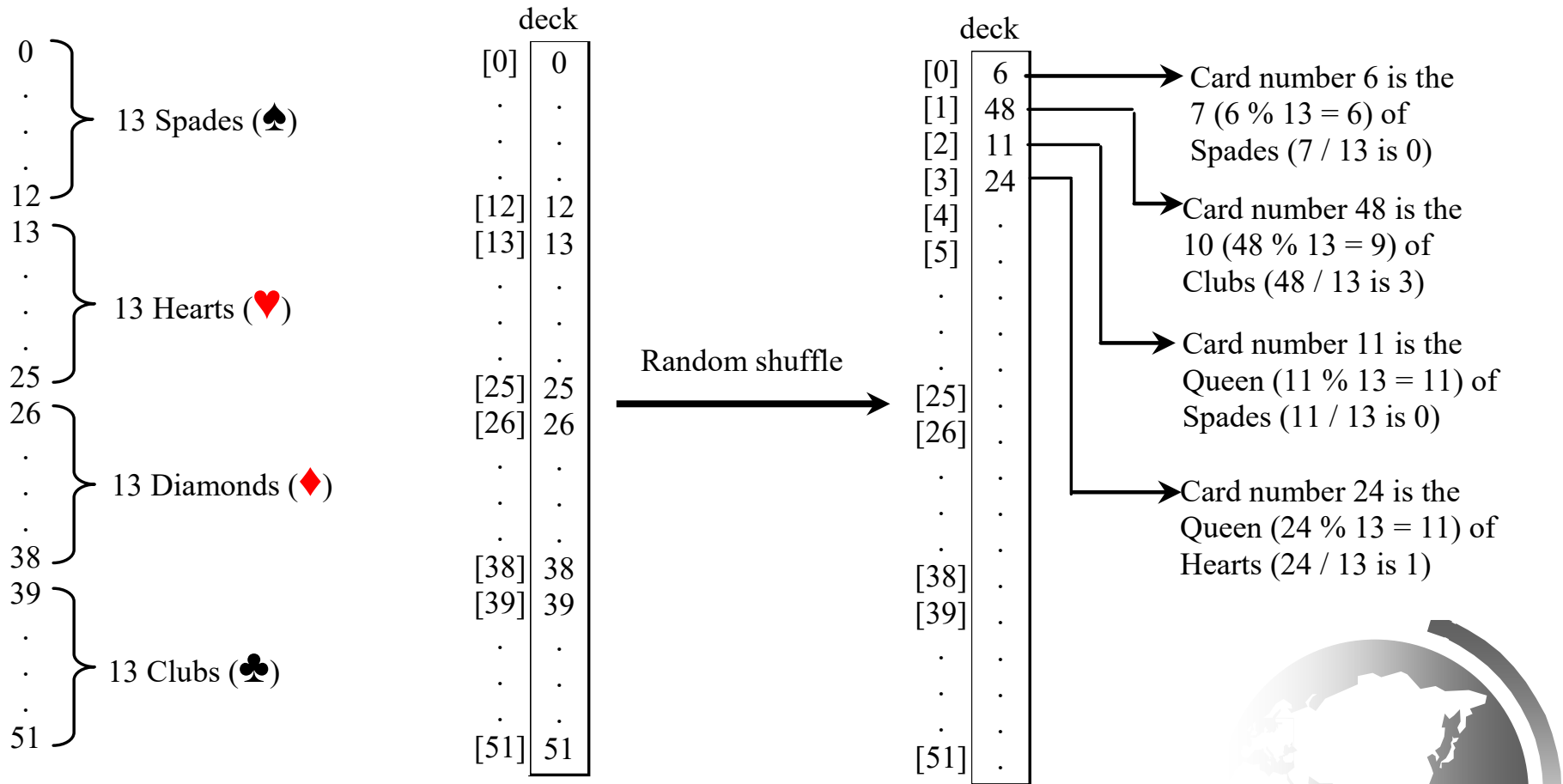
AnalyzeNumbers

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

Problem: Deck of Cards, cont.

0 ⎫
· ⎬ 13 Spades (♠)
· ⎪
· ⎪
12 ⎭

13 ⎫
· ⎪
· ⎬ 13 Hearts (♥)
· ⎪
25 ⎭

26 ⎫
· ⎪
· ⎬ 13 Diamonds (♦)
· ⎪
38 ⎭

39 ⎫
· ⎪
· ⎬ 13 Clubs (♣)
· ⎪
51 ⎭

deck

[0]	0
·	·
·	·
·	·
·	·
[12]	12
[13]	13
·	·
·	·
·	·
[25]	25
[26]	26
·	·
·	·
·	·
[38]	38
[39]	39
·	·
·	·
·	·
[51]	51

Random shuffle →

deck

[0]	6
[1]	48
[2]	11
[3]	24
[4]	·
[5]	·
·	·
·	·
·	·
[25]	·
[26]	·
·	·
·	·
·	·
[38]	·
[39]	·
·	·
·	·
·	·
[51]	·

Card number 6 is the 7 (6 % 13 = 6) of Spades (7 / 13 is 0)

Card number 48 is the 10 (48 % 13 = 9) of Clubs (48 / 13 is 3)

Card number 11 is the Queen (11 % 13 = 11) of Spades (11 / 13 is 0)

Card number 24 is the Queen (24 % 13 = 11) of Hearts (24 / 13 is 1)

Problem: Deck of Cards, cont.

cardNumber / 13 = $\left\{ \begin{array}{l} 0 \longrightarrow \text{Spades} \\ 1 \longrightarrow \text{Hearts} \\ 2 \longrightarrow \text{Diamonds} \\ 3 \longrightarrow \text{Clubs} \end{array} \right.$

cardNumber % 13 = $\left\{ \begin{array}{l} 0 \longrightarrow \text{Ace} \\ 1 \longrightarrow 2 \\ . \\ . \\ 10 \longrightarrow \text{Jack} \\ 11 \longrightarrow \text{Queen} \\ 12 \longrightarrow \text{King} \end{array} \right.$

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;

Before the assignment
list2 = list1;

list1 → Contents of list1

list2 → Contents of list2

After the assignment
list2 = list1;

list1 → Contents of list1

list2 →

Garbage → Contents of list2
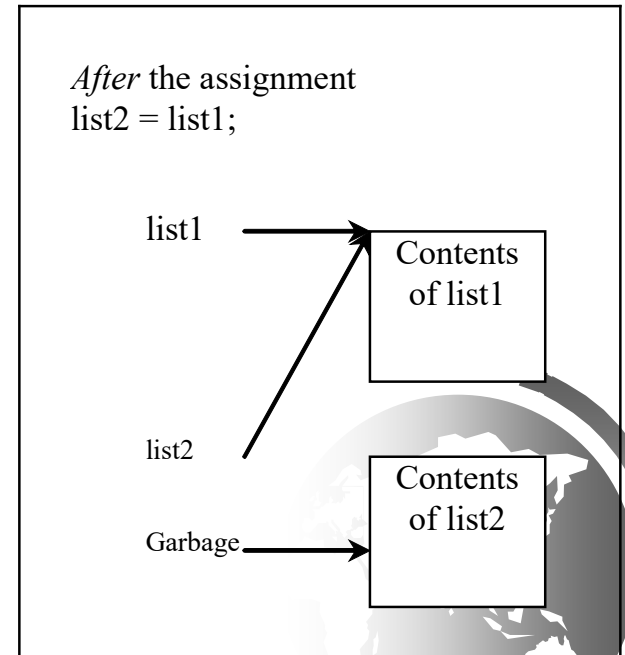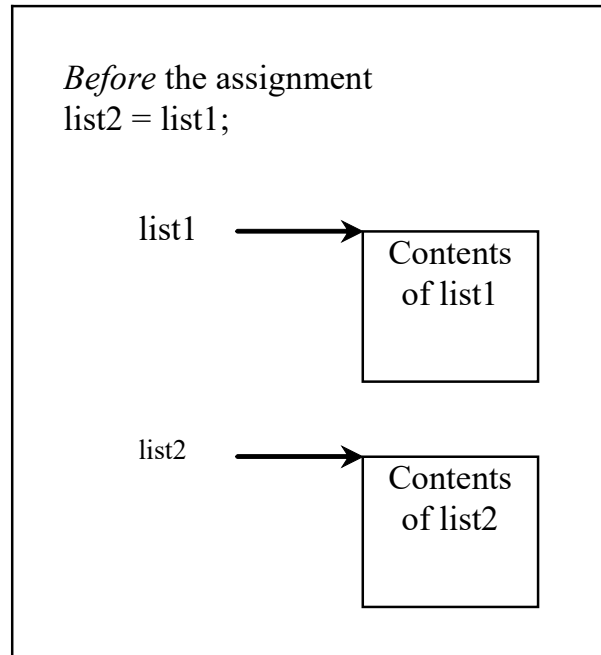
Passing Lists to Functios

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

```python
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()
```

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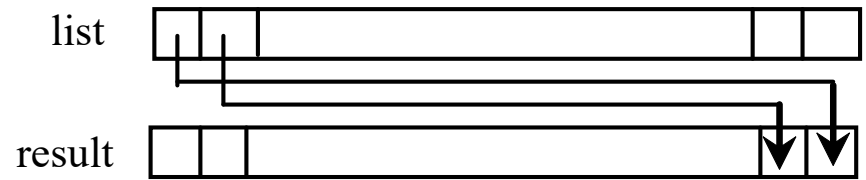[11, 12, 13, 14]
[1, 2, 4]

Returning a List from a Function

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

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

    return result
```

list

result

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

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

Issues: List as a Default Argument

DefaultListArgument

DefaultNoneListArgument

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.

chars[0]	
chars[1]	
...	...
...	...
chars[98]	
chars[99]	

counts[0]	
counts[1]	
...	...
...	...
counts[24]	
counts[25]	

CountLettersInList

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.

Linear Search Animation

https://liveexample.pearsoncmg.com/dsanimation/LinearSearcheBook.html

Linear Search Animation

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*.

```python
# 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
```

[0] [1] [2] ...
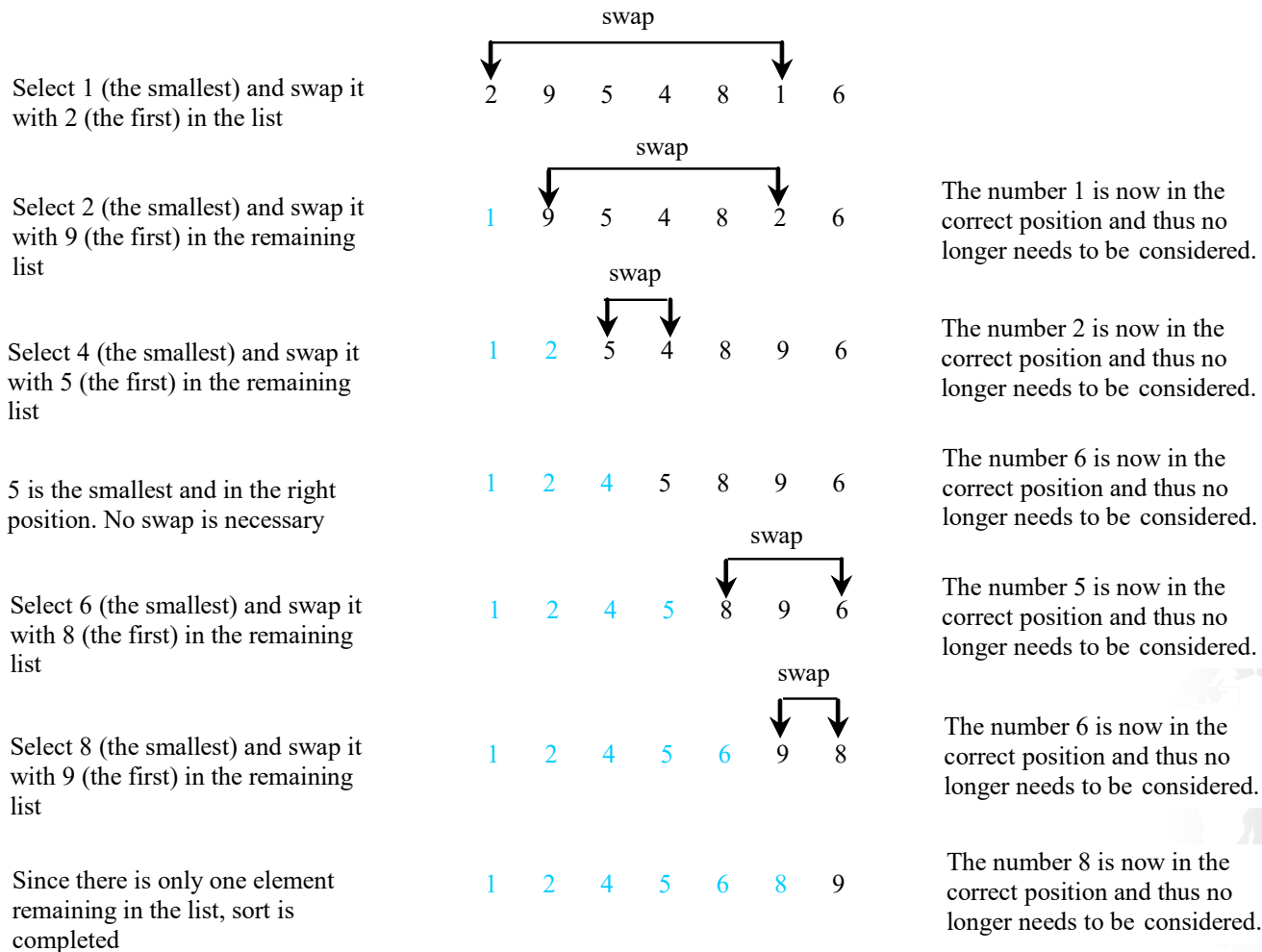
lst

key   Compare key with lst[i] for i = 0, 1, …

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.

swap

Select 1 (the smallest) and swap it with 2 (the first) in the list

2   9   5   4   8   1   6

swap

Select 2 (the smallest) and swap it with 9 (the first) in the remaining list

1   9   5   4   8   2   6

The number 1 is now in the correct position and thus no longer needs to be considered.

swap

Select 4 (the smallest) and swap it with 5 (the first) in the remaining list

1   2   5   4   8   9   6

The number 2 is now in the correct position and thus no longer needs to be considered.

5 is the smallest and in the right position. No swap is necessary

1   2   4   5   8   9   6

The number 6 is now in the correct position and thus no longer needs to be considered.

swap

Select 6 (the smallest) and swap it with 8 (the first) in the remaining list

1   2   4   5   8   9   6

The number 5 is now in the correct position and thus no longer needs to be considered.

swap

Select 8 (the smallest) and swap it with 9 (the first) in the remaining list

1   2   4   5   6   9   8

The number 6 is now in the correct position and thus no longer needs to be considered.

Since there is only one element remaining in the list, sort is completed

1   2   4   5   6   8   9

The number 8 is now in the correct position and thus no longer needs to be considered.

Selection Sort Animation

https://liveexample.pearsoncmg.com/dsanimation/Selection SortNew.html

From Idea to Solution

```
for i in range(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 = min(lst[i : ])
```

Note: this is a new version much simpler
than the one in the book.

```
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 = min(lst[i : ])

currentMinIndex = i + lst[i : ].index(currentMin)

# Swap lst[i] with lst[currentMinIndex] if necessary
if currentMinIndex != i:
    lst[currentMinIndex], lst[i] = lst[i], currentMin
```

Wrap it in a Function

```python
# The function for sorting elements in ascending order
def selectionSort(lst):
    for i in range(len(lst) - 1):
        # Find the minimum in the lst[i : len(lst)]
        currentMin = min(lst[i : ])
        currentMinIndex = i + lst[i: ].index(currentMin)

        # Swap lst[i] with lst[currentMinIndex] if necessary
        if currentMinIndex != i:
            lst[currentMinIndex], lst[i] = lst[i], currentMin
```

Invoke it

selectionSort(yourList)