7 Array, List and Dictionary

Learning Outcome

Programming Elements and Constructs Understand the different data structures: array, list and dictionary; initialize arrays (1-dimensional and 2-dimensional)

7.1 Array

An array provides a convenient structure for storing data items of the same data type. As you will see, arrays allow for both easy reading of the data into memory and efficient accessing of the data for processing.

We will consider two standard types of situations in which arrays are useful:
(1) when a data list must be processed **more than once** and
(1) when a large number of related summing or counting variables are needed.

7.1.1 One-Dimensional Array

So far, all the variables that we have considered have been a simple data type. A variable of **simple data type** consists of a single memory cell that can hold only one value at a time. By contrast, a variable of **structured data type** consists of a collection of memory cells.

An **array** consists of a collection of memory cells for storing a list of values that are all of the **same data type** -- for example, a list of integers, a list of real numbers, a list of characters, or a list of boolean values. The entire list is given a name.

An array can be compared to an apartment building. The name of the array represents the name of the building. The **elements** of the array represent the building's apartments, each of which has a **subscript** or **index**, which corresponds to an apartment number.

Assuming that the index starts with **1**, **Riser[1]** is used to specify the first apartment in the building called Riser, while **Riser[4]** refers to apartment 4 in the building called Riser

Riser[1]	Riser[2]	Riser[3]	Riser[4]

In order to specify an individual element of an array, you must give **both** the name of the array and the subscript. For example, for an array called **scores**, the statement

scores[4] ← 86

assigns the value 86 to the memory cell **scores[4]**. Similarly,

OUTPUT scores[6]

outputs the contents of the memory cell **scores[6]**.
The following pseudocode assigns values to be stored in an array's elements.

```
// program drill
BEGIN
      DECLARE X: ARRAY[1:4] of INTEGER
      x[1]←83
      x[2]←59
      x[3]←88
      x[4]←72

      OUTPUT x[4]
      OUTPUT x[2]
      OUTPUT x[2+1]
      OUTPUT x[2]+1
END
```

In programming languages where static arrays are used, a declaration of the array would require the data-type, and the number of elements you want in the array. By specifying these information, memory cells memory cells are set aside for the array declared.

In the above example, 4 memory cells, **x[1]**, **x[2]**, and **x[3], x[4]** are set aside, each enough storage for an integer value. By the time the four assignment statements have been executed, four numbers will be stored in array **x** as follows:

x[1]	x[2]	x[3]	x[4]
83	**59**	**88**	**72**

The output will be

72
59
88
60

Note (Array index out of range):

In the above example, if 4 elements were assumed i.e. **x[1..4],** You should not access/assign values to **x[5].** This may be accepted in some programming languages, however, it may cause unpleasant things to happen in the computer memory because the array ranges from **x[1]** to **x[4].**

Using a for Loop to read (access, write) values into the array

The numbers 83, 59, 88, and 72 could be assigned interactively to the array **x** using a **for** loop.

```
FOR i ← 1 TO 4
      OUTPUT "Enter Element:"
      INPUT x[i]
ENDFOR
```

An important use of arrays is in processing a data list more than once.

Example Suppose a program reads in list of up to 25 non-negative numbers. For example, the user may input

enter list of non-negative integer: 41
enter list of non-negative integer: 68
enter list of non-negative integer: 32
enter list of non-negative integer: 74
enter list of non-negative integer: 55
enter list of non-negative integer: -999

The following program, in pseudocode, will print the numbers in their original order and then in reverse order. The printout will be

original order: 41 68 32 74 55
reverse order: 55 74 32 68 41

```
BEGIN
    DECLARE numbs: ARRAY[1:25] of INTEGER
    DECLARE count: INTEGER
    DECLARE num: INTEGER
    count ← 0 // keeps track of the no. of input integers
    // read list of up to 25 non-negative integers into
    // array numbs
    // using input -999 to end list
    OUTPUT "enter non-negative integer:"
    INPUT num

    WHILE(num <> -999 AND count < 25)
        count ← count + 1
        numbs[count] ← num
        OUTPUT "enter non-negative integer:"
        INPUT num
    ENDWHILE

    IF(count>0)
        //prints original order
        OUTPUT "original order:"
        FOR i ← 1 TO count
            OUTPUT numbs[i]
        ENDFOR

        // prints the reverse order
        OUTPUT "reverse order:"
        FOR i ← count DOWN TO 1
            OUTPUT numbs[i]
        ENDFOR
    ENDIF
END
```

Remark

(1) The **numbs** array was declared to hold up to 25 integers. In the **while** loop that reads values into the **numbs** array, **count** is used as the array subscript.

(2) In the **for** loop that takes a second look at the array, **count** is not a subscript but rather the upper limit of the loop.

(3) You may also assume that the array index starts from 0 ie. **numbs[0..24]**. In this case, the pseudocode above, would need to be updated accordingly.

7.1.2 Array of Counting Variables

Arrays are particularly useful when you are tallying a number of related quantities. By using the array index, you can directly increment the appropriate counter instead of performing a tedious multiway selection.

Example (Vote Counting) Suppose that in a recently held election, there were four candidates 1, 2, 3 and 4. Suppose the votes are entered such that 1 represents a vote for candidate 1, 2 a vote for candidate 2, 3 a vote for candidate 3, and 4 a vote for candidate 4. –999 will end the list.

That is, the input may consists of a list like

1 3 1 4 2 1 2 3 etc

We wish to write a program to process the input list. The printout should be of the form

CANDIDATE	**NO. OF VOTES**
1	**17**
2	**38**
3	**24**
4	**32**

We will use the array of **voteCnts** to keep track of the votes for each of the four candidates.

By the time all the votes have been counted, **voteCnts** will have the following values:

	voteCnts
[1]	**17**
[2]	**38**
[3]	**24**
[4]	**32**

At the start of the program, the four memory boxes of **voteCnts** should be initialized to 0.

Here is the draft pseudocode.

1. initialize **voteCnts** to zeros
2. use a loop to read in and process each of the votes; for example, a vote of 3 will increase **voteCnts[3]** by 1.
3. use a **for** loop to print the final values for each of the memory boxes of **voteCnts**

Processing a Single Vote
The statement

read vote

will read in a vote from the user. The variable **vote** will have the value 1, 2, 3 or 4.

The value of **vote** (1, 2, 3 or 4) gives the subscript for the element of **voteCnts** that should be increased by 1. So we can just use a single line to count the votes.

voteCnts[vote] ← voteCnts[vote] + 1

Here is the detailed pseudocode.

```
//  program voting;
//  processes the votes and prints each candidate's tally

BEGIN
        CONSTANT VOTERANGE = 4
        DELARE vote: INTEGER
        DECLARE voteCnts: ARRAY [1: VOTERANGE] of INTEGER
        // initialize voteCnts to 0
        FOR i ← 1 TO VOTERANGE
                voteCnts[i] ← 0

        // process votes by reading in individual vote and updating // appropriate
        // counter
        OUTOUT "enter vote or -999 to end:"
        INPUT  vote
        WHILE(vote != -999)
                IF (vote >= 1 AND vote <=4)
                        voteCnts[vote] ← voteCnts[vote] + 1
                ELSE
                        OUTPUT "Invalid vote"
                ENDIF
                OUTPUT "enter vote or -999 to end:"
                INPUT vote
        ENDWHILE
        // prints the number of votes for each candidate
        OUTPUT "CANDIDATE     ", "NO. OF VOTES"
        FOR i ← 1 TO VOTERANGE
                OUTPUT i, voteCnts[i]
        ENDFOR
END
```

7.1.3 Parallel Array

Suppose we have a list consisting of the names of students in a class and their respective grade:

Jones
92
Johnson
88
Cohen
92

Write a program, in pseudocode, to read in the above data, find the highest grade achieved and then print the names of everyone who earned it. There might be one such person, or there might be more than one. For the above input, the printout would be

Highest grade 92
Achieved by:
Jones
Cohen

Note that the list of scores will have to be processed twice. A first pass will determine what the highest score is, and a second pass will print the names of those who share it.

The program will use the ***parallel arrays*** **names** and **scores**.

In the parallel arrays, a given student's name and score will be contained in memory boxes with the same subscript. That is, the element **scores[i]** will contain the score of the student whose name is in **name[i]**.

Here is how the parallel arrays will be set up for the preceding data.

	Names
[1]	**Jones**
[2]	**Johnson**
[3]	**Cohen**

	scores
[1]	**92**
[2]	**88**
[3]	**92**

Here is the detailed pseudocode.

```
// program HighScorers;
// prints names of students with highest score
BEGIN
    CONSTANT MAXSIZE = 40
    DECLARE size: INTEGER
    DECLARE maxScore: INTEGER
    DECLARE name:STRING
    DECLARE score:INTEGER

    DECLARE names: ARRAY [1: MAXSIZE] of STRING
    DECLARE scores: ARRAY [1: MAXSIZE] of INTEGER
    size ← 0      // keeps track of the no. of valid
```

```
                    //names
        // read all the data into parallel arrays
        // using input 'xxx' for name to end input
        OUTPUT "enter name or xxx to end input:"
        INPUT name

        WHILE( (name <> "xxx")  AND (size < MAXSIZE) )
            OUTPUT "enter score of student:"
            INPUT score

            size ← size + 1
            names[size] ← name
            scores[size] ← score

            OUTPUT "enter name or xxx to end input:"
            INPUT name
        ENDWHILE

        IF(size > 0)
            // find the highest score
            maxScore ← scores[1]
            FOR i ← 2 TO size
                IF (scores[i] > maxScore)
                    maxScore ← scores[i]
                ENDIF
            ENDFOR
            OUTPUT "Highest grade: ", maxScore

        // prints names of those achieving the highest score
            OUTPUT "Achieved by: "
            FOR i ← 1 TO size
                IF (scores[i] = maxScore)
                    OUTPUT names[i]
                ENDIF
            ENDFOR
        ENDIF
END
```

7.1.4 Two-Dimensional Array

So far, all the arrays we have considered have been one dimensional. Thus, only one subscript has been needed to specify the desired element. When information fits naturally into a rectangular table with several rows and columns, however, it is often advantageous to store it in a **two-dimensional array**, also called a **matrix**. Two subscripts are necessary to specify an element in a matrix – a row subscript and a column subscript.

Often, the data being processed can be organized as a table with several rows and columns. Such an array is called a two-dimensional array or a matrix.

Assuming array index starts from 1 and **sales** is an array containing five-day sales figures for the 18 employees of ABC Company. The first row gives the week's sales figures for salesperson 1, the second row gives the figures for salesperson 2, and so on. For the moment, we will not concern ourselves with how these values were placed into this array.

sales

	1	**2**	**3**	**4**	**5**
1	25	31	29	40	30
2	41	39	38	42	33
3	48	58	62	47	40
.	.	.	.	.	.
.	.	.	.	.	.
18	30	30	32	34	28

In order to access a particular cell from this array, we must specify the name of the array followed by the row and column of the desired cell. Thus **print sales[2, 3]** would cause the computer to output **38** -- that is, the contents of the cell in the row 2 and column 3. The statement **sales[2,4] = 0** will reassign the content at row 2, column 4 to 0 – that is, the value 42 will change to 0.

Using Data from a Two-Dimensional Array

Suppose we use an array, **sales,** that can store five-day figures for up to 30 salesperson. We can do so as follows, in pseudocode:

```
CONSTANT MAXSIZE = 30
CONSTANT WEEK = 5
DECLARE sales [1:MAXSIZE, 1:WEEK] of INTEGER
```

Let us write a program fragment that will use the contents of the array **sales** to output a table giving each individual salesperson's weekly total. Here is the printout.

Salesperson	**5-day total**
1	**155**
2	**193**
.	**.**
18	**154**

We will assume that the number of salespeople is stored by the variable **size**.
Here is the pseudocode:

```
// print table heading
OUTPUT "Salesperson          5-day total"

FOR salesPerson ← 1 TO size
    // find salesperson's weekly total
    sum ← 0
    FOR day ← 1 TO 5
        sum = sum + sales[salesperson,day]
    ENDFOR
    OUTPUT salesperson, sum
ENDFOR
```

Reading Data into a Two-Dimensional Array

Write a program, in pseudocode, that reads in sales figures of a maximum of 30 salespeople from Monday to Friday and output the contents of **names** and **sales** in a matrix format as below:

```
Jack   25  31  29  40  30

Jill   41  39  38  42  33
.
.
John   30  30  32  34  28
```

Here is the program, in pseudocode.

```
CONSTANT MAXSIZE = 30
CONSTANT WEEK = 5
DECLARE salesman: INTEGER
DECLARE day: INTEGER
DECLARE name:STRING
DECLARE names:ARRAY[1: MAXSIZE] of STRING
DECLARE sales:ARRAY [1: MAXSIZE, 1: WEEK] of INTEGER
salesman ← 0

// read input of names and sales, xxx to end input list
OUTPUT "enter xxx to end or name of salesman "
INPUT name

WHILE( (name <> "xxx") AND (salesman < MAXSIZE) )
    salesman ← salesman + 1
    names[salesman]  name

    OUTPUT "enter sales from Monday to Friday: "

    FOR day ← 1 TO WEEK
        INPUT sales[salesman, day]
    ENDFOR
```

```
        OUTPUT "enter xxx to end or name of salesman "
        INPUT name

    ENDWHILE

    // output names and sales in matrix form
    FOR salesPerson ← 1 TO salesman

        OUTPUT names[salesPerson]

        FOR day ← 1 TO WEEK
            OUTPUT sales[salesperson, day]
        ENDFOR

    ENDFOR
```

7.1.5 Case Study: Dropping the Lowest Score

Professor Fairchild gives four exams and then determines each student's letter mark by taking the average of the student's three best scores. An A is at least 90, a B is at least 80 but under 90, a C is at least 70 but under 80, a D is at least 60 but under 70, and an F is under 60.

Suppose the program reads in a maximum of 30 students' name and four exam scores. Entering "xxx" as the student's name will end the input list. A sample input data:

```
Smith  80  70  79  83
Jones  90  92  80  91
.
.
Mike   70  75  66  70
```

Sample output:

```
Names       Grades
===============
Smith       B
Jones       A
    .
    .
Mike        C
```

For each student, the sum of the three top scores can be computed as

(sum of all four scores) - (lowest score)

In preparation for dropping each student's lowest score, we will find each student's lowest score and store it in an array called **lows**.

In this program, we will use the following parallel data structures:

	names	**scores**				**lows**	**best3avgs**
1	Smith	80	70	79	83	70	80.67
2	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
size	.	.	.	.	.	.	.

Here is the pseudocode.

- read all the data into **names** and **scores**.
- find each student's low grade and stores these low grades in **lows**.
- find each student's average for the best three grades.
- print each student's letter grade.

```
// program LetterGrade
// drops lowest grade in computing students' letter grade

BEGIN
        CONSTANT MAXSIZE = 30
        CONSTANT MAXTEST = 4
        DECLARE size : INTEGER
        DECLARE test : INTEGER
        DECLARE student : INTEGER
        DECLARE studMin: INTEGER
        DECLARE name: STRING


        DECLARE names:ARRAY [1: MAXSIZE] of STRING
        DECLARE scores:ARRAY [1: MAXSIZE, 1: MAXTEST] of INTEGER
        DECLARE lows:ARRAY [1: MAXSIZE] of INTEGER
        DECLARE avgs:ARRAY[1:MAXSIZE] of FLOAT

        size ← 0

        // read input names and scores into respective arrays
        OUTPUT "enter xxx to end or name of student ", size
        INPUT name
        WHILE( (name<> "xxx")  AND (size < MAXSIZE) )
                size ← size + 1
                names[size] ← name
```

```
            OUTPUT "enter the 4 test scores: "
            FOR test ← 1 TO MAXTEST
                  INPUT scores[size, test]

            ENDFOR

            OUTPUT "enter xxx to end input or name of student "
            INPUT name
      ENDWHILE

      // find low grade of each student and store it in an array
      FOR student ← 1 TO size
            studMin ← scores[student,1]
            FOR test ← 2 TO MAXTEST
                  IF (scores[student, test] < studMin)
                        studMin ← scores[student, test]
                  ENDIF
            lows[student] = studMin
      ENDFOR

      // find average of each student's 3 highest scores by dropping lowest score
      FOR student ← 1 TO size
            sum ← 0
            FOR test ← 1 TO MAXTEST
                  sum = sum +  scores[student, test]
            avgs[student] ← (sum - lows[student]) / 3
      ENDFOR

      // assigns a letter grade to each student and prints it
      OUTPUT "Names           Grades"
      OUTPUT "================"
      FOR student ← 1 TO size
            OUTPUT names[student]
            IF (avgs[student] >= 90)
                  OUTPUT "A"
            ELSE IF (avgs[student] >=80)
                  OUTPUT "B"
            ELSE IF (avgs[student] >= 70)
                  OUTPUT "C"
            ELSE IF (avgs[student] >= 60)
                  OUTPUT "D"
            ELSE
                  OUTPUT "F"
            ENDIF
      ENDFOR
END
```

7.1.6 Initiating Arrays in Python

To initiate a one-dimensional array, we can assign default values or strings as its content.

For example,
score = [0] * 10 initiated an array **score** of size 10, and each cell has an initial value of 0.
name = [''] * 10 initiated an array **name** of size 10, and each cell has an initial empty string.

Different from pseudocode, the index **always starts with 0**.

Two-dimensional array requires a for loop. The code below initiated an array of size 3x4.

```
row = 3
col = 4

score = [0]*row

for i in range(row):
      score[i] = [0]*col
```

Tutorial 7A

1. A basketball team with six players has played four games. Write a program, in pseudocode, that accepts each player's name followed by the points scored by them in each of the four games. A typical input data is given below for one player.

   Smith
   12 14 7 10

   Then, print the raw data entered in a table form. Include in the table, each player's scoring average, and the number of points scored by the team for each game.

2. Write a program, in pseudocode, to read the name and the closing prices of several stocks for each weekday of last week into two parallel arrays. For each stock, the program should print the maximum and minimum price and the day it was achieved.

3. A class has at most 40 students. Write a program, in pseudocode, that will read in the scores of an exam for each student in the class, finds and prints the class average and then prints each of the scores that are above the class average. For example, if the input scores are as follow:

   70
   80
   71
   58
   79
   92

   the output would be

   class average: 75.0
   scores above average: 80 79 92

7.2 Lists

- List: Sequence of data values (**items** or **elements**)
- Some examples:
  - To-do list
  - Recipe, which is a list of instructions
  - Text document, which is a list of lines
  - Words in a dictionary
- Each item in a list has a unique **index** that specifies its position (from 0 to length – 1)

7.2.1 List Literals and Basic Operators

- Some examples:

  ['apples', 'oranges', 'cherries']

  [[5, 9], [541, 78]]

- When an element is an expression, its value is included in the list:

```
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
```

OPERATOR OR FUNCTION	WHAT IT DOES
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L + L`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range `0` through `upper - 1`.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns `True` if all the results are true, or `False` otherwise.
`for <variable> in L:` `  <statement>`	Iterates through the list, binding the variable to each element.
`<any value> in L`	Returns `True` if the value is in the list or `False` otherwise.

- Lists of integers can be built using **range**:

```
>>> first = [1, 2, 3, 4]
>>> second = list(range(1, 5))
>>> first
[1, 2, 3, 4]
>>> second
[1, 2, 3, 4]
>>>
```

- **len**, **[]**, +, and == work on lists as expected:

```
>>> len(first)
4
>>> first[2:4]
[3, 4]
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
>>> first == second
True
```

- To print the contents of a list:

```
>>> print("1234")
1234
>>> print([1, 2, 3, 4])
[1, 2, 3, 4]
>>>
```

- **in** detects the presence of an element:

```
>>> 0 in [1, 2, 3]
False
```

7.2.2 Replacing Elements in a List

A list is **mutable**:

- Elements can be inserted, removed, or replaced
- The list itself maintains its identity, but its **state**—its length and its contents—can change

Subscript operator is used to replace an element. Subscript is used to reference the **target** of the assignment, which is not the list but an element's position within it.

```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
```

```
>>> numbers = range(6)
>>> numbers
[0, 1, 2, 3, 4, 5]
>>> numbers[0:3] = [11, 12, 13]
>>> numbers
[11, 12, 13, 3, 4, 5]
```

We can also use list to assign values to multiple items:

```
>>> Name, CG, Gender = ['Ryan', 'S6A','M']
>>> Name
'Ryan'
>>> CG
'S6A'
>>> Gender
'M'
```

7.2.3 Inserting and Removing Elements in a List

LIST METHOD	WHAT IT DOES
`L.append(element)`	Adds **element** to the end of **L**.
`L.extend(aList)`	Adds the elements of **L** to the end of **aList**.
`L.insert(index, element)`	Inserts **element** at **index** if **index** is less than the length of **L**. Otherwise, inserts **element** at the end of **L**.
`L.pop()`	Removes and returns the element at the end of **L**.
`L.pop(index)`	Removes and returns the element at **index**.

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
```

```
>>> example = [1, 2]
>>> example.append(10)
>>> example
[1, 2, 10]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop()
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0)
1
```

Notice that append takes an element and extend takes a list. If we swap the input, the output is very different.

```
>>> L = [1,2,3]
>>> L.append([1,2])
>>> L
[1, 2, 3, [1, 2]]
```

```
>>> L = [1,2,3]
>>> L.extend(5)
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    L.extend(5)
TypeError: 'int' object is not iterable
>>> L.extend([5])
>>> L
[1, 2, 3, 5]
```

7.2.4 Searching a List

- **in** determines an element's presence or absence, but does not return position of element
- Use method **index** to locate an element's position in a list. It raises an error when the target element is not found.

```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

7.2.5 Sorting a List

- A list's elements are always ordered by position, but you can impose a **natural ordering** on them. For example, in alphabetical order.
- When the elements can be related by comparing them <, >, and ==, they can be sorted The method **sort** mutates a list by arranging its elements in ascending order

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

7.2.6 Mutator Methods and the Value None

- All of the functions and methods examined in previous chapters return a value that the caller can then use to complete its work
- **Mutator** methods usually return no value of interest to caller. Python automatically returns the special value **None.**

```
>>> aList = aList.sort()

>>> print(aList)
None
```

Notice that **append**, **extend**, **insert**, and **sort** are all mutator methods.

7.2.7 Aliasing and Side Effects

Mutable property of lists leads to interesting phenomena:

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
[10, 99, 30]
>>> second
[10, 99, 30]
```

first and second are aliases (refer to the exact same list object)

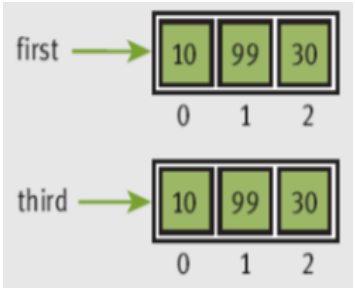
In this case, if we change any element of first, second will make the same change as well. To prevent aliasing, copy contents of object.

```
>>> third = []
>>> for element in first:
        third.append(element)

>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
```

Alternativ

```
>>> third = first[:]
```

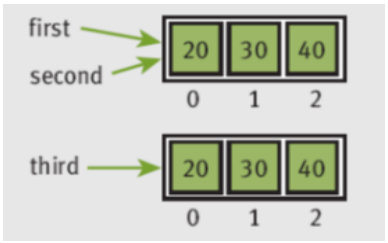


Recall that in Section 6.2, we learn that string is immutable. It will be different if we try the above example on strings.

```
>>> first = 'sample'
>>> second = first
>>> first += 's'
>>> first
'samples'
>>> second
'sample'
```

7.2.8 Equality: Object Identity and Structural Equivalence

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = [20, 30, 40]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```



7.2.9 Tuples

A **tuple** resembles a list, but is immutable. It is indicated by enclosing its elements in ( ).

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

Most of the operators and functions used with lists can be used in a similar fashion with tuples.

Tutorial 7B

1. Assume that the variable **data** refers to the list **[5, 3, 7]**. Write the values of the following expressions:
    (a) **data[2]**
    (b) **data[-1]**
    (c) **len(data)**
    (d) **data[0:2]**
    (e) **0 in data**
    (f) **data + [2, 10, 5]**
    (g) **tuple(data)**

2. Assume that the variable **data** refers to the list **[5, 3, 7].** Write the expression that perform the following tasks:

    (a) Replace the value at position 0 in **data** with that value's negation.

    (b) Add the value 10 to the end of **data**.

    (c) Insert the value 22 at position 2 in **data**.

    (d) Remove the value at position 1 in **data**.

    (e) Add the values in the list **newData** to the end of the **data**.

    (f) Locate the index of the value 7 in **data**, safely.

    (g) Sort the values in **data**.

3. Write a loop that accumulates the sum of all the numbers in a list named **data**.

4. Assume that **data** refers to a list of numbers, and **result** refers to an empty list. Write a loop that adds the nonzero values in **data** to the **result** list.

5. Write a loop that replaces each number in a list named **data** with its absolute value.

7.3 Dictionaries

- A dictionary organizes information by **association**, not position

Example: When you use a dictionary to look up the definition of "mammal," you don't start at page 1; instead, you turn directly to the words beginning with "M"

- Data structures organized by association are also called **tables** or **association lists**
- In Python, a **dictionary** associates a set of **keys** with data values

7.3.1 Dictionary Literals

A Python dictionary is written as a sequence of key/value pairs separated by commas

– Pairs are sometimes called **entries**
– Enclosed in curly braces **{** and **}**
– A colon (**:**) separates a key and its value
– Keys can be data of any immutable types, including other data structures

{'Sarah':'476-3321', 'Nathan':'351-7743'}	A Phone book
{'Name':'Molly', 'Age':18}	Personal information
{}	An empty dictionary

7.3.2 Adding Keys and Replacing Values

Add a new key/value pair to a dictionary using **[]**:

```
<a dictionary>[<a key>] = <a value>
```

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name': 'Sandy', 'occupation': 'hacker'}
```

Use **[]** also to replace a value at an existing key:

```
>>> info["occupation"] = "manager"
>>> info
{'name': 'Sandy', 'occupation': 'manager'}
```

7.3.3 Accessing Values

- Use **[]** to obtain the value associated with a key
- If key is not present in dictionary, an error is raised

```
>>> info["name"]
'Sandy'
>>> info["job"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'job'
```

- If the existence of a key is uncertain, test for it using the method **get**

```
>>> print(info.get("job", None))
None
>>>
```

7.3.4 Removing Keys

- To delete an entry from a dictionary, remove its key using the method **pop**
- **pop** expects a key and an optional default value as arguments

```
>>> print(info.pop("job", None))
None
>>> print(info.pop("occupation"))
manager
>>> info
{'name': 'Sandy'}
>>>
```

7.3.5 Traversing a Dictionary

We can use a **for** loop to print all of the keys and their values:

```
for key in info:
   print(key, info[key])
```

Alternatively, we can use **list** and dictionary methods to print all the keys, values, or both.

```
>>> grades={90:'A',80:'B',70:'C'}
>>> list(grades.keys())
[90, 80, 70]
>>> list(grades.values())
['A', 'B', 'C']
>>> list(grades.items())
[(90, 'A'), (80, 'B'), (70, 'C')]
```

Notice that when we print items, all entries are represented as tuples within the list.

DICTIONARY OPERATION	WHAT IT DOES
`len(d)`	Returns the number of entries in **d**.
`aDict[key]`	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
`d.get(key [, default])`	Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`d.pop(key [, default])`	Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`d.clear()`	Removes all the keys.
`for key in d:`	**key** is bound to each key in **d** in an unspecified order.

Tutorial 7C

1. Assume that the variable **data** refers to the dictionary **{"b":20, "a":35}.** Write the values of the following expressions:
   (a) **data["a"]**
   (b) **data.get("c", None)**
   (c) **len(data)**
   (d) **list(data.keys())**
   (e) **list(data.values())**
   (f) **data.pop("b")**
   (g) **data** # After the pop above

2. Assume that the variable **data** refers to the dictionary **{"b":20, "a":35}.** Write the expressions that perform the following tasks:

   (a) Replace the value at the key **"b"** in **data** with that value's negation.
   (b) Add the key/value pair **"c":40** to **data**.
   (c) Remove the value at key **"b"** in **data**, safely.
   (d) Print the keys in **data** in alphabetical order.

3. Make a dictionary where the keys are the names of weight lifting exercises, and the values are the number of times you did that exercise. Use a for loop to print out a series of statements such as "I did 10 bench presses".

4. What are the similarities and difference among One-Dimensional Array, Parallel Array, Two-Dimensional Array, List and Dictionary?

Assignment 7

1. Write a program that compute the median and mode of a set of numbers. It reads a list of numbers from a text file and print their median and mode.

2. Write a program that allows the user to navigate the lines of text in a file. The program should prompt the user for a filename and input the lines of text into a list. The program then prints the number of lines in the file and prompts the user for a line number. Actual line numbers range from 1 to the number of lines in the file, If the input is zero, the program quits. Otherwise, the program prints the line associated with that number.

3. Assume a file contains words separated by newlines.

   (a) Write a program that print all of the unique words in the file in alphabetical order
   (b) Write a program that output the unique words and their frequencies in alphabetical order
   (c) Write a program that output the most frequent word(s) in alphabetical order