القسم	المرحلة	المحاضرة	المادة	عنوان المحاضرة
علوم الحاسبات	الثانية	السابعة	برمجة – لغة بايثون	Data Structures in Python

University of Anbar

College of Computer Science and Information Technology

م.م. محمد عبدالرزاق رجب

PYTHON PROGRAMMING LANGUAGE

القسم	المرحلة	المحاضرة	المادة	عنوان المحاضرة
علوم الحاسبات	الثانية	السابعة	برمجة – لغة بايثون	Data Structures in Python

Lecture seven

Data Structures in Python

1. String in Python

- Strings in Python are a sequence of characters enclosed in single quotes ('), double quotes ("), or triple quotes (''' or """). They are used to represent text data. Here are some key features of strings in Python, along with examples:
- **Creating Strings:** You can create strings using single, double, or triple quotes:

```
# Single quotes
string1 = 'Hello, World!'

# Double quotes
string2 = "Python is fun!"

# Triple quotes (for multi-line strings)
string3 = '''This is a
multi-line string.'''

print(string1)
print(string2)
print(string3)
```

- **Accessing Characters in a String:** You can access individual characters in a string using **indexing (zero-based)**:

```
string = "Python"

# Accessing characters
print(string[0])  # Output: 'P'
print(string[1])  # Output: 'y'
print(string[-1]) # Output: 'n' (last character)
print(string[-2]) # Output: 'o (last character)
print(string[:2])  # Output: 'Py'
print(string[:-2])  # Output: 'Pyth'
```

- **String Slicing:** You can extract a substring using slicing:

```
string = "Hello, World!"


# Slicing the string
```

```
print(string[0:5]) # Output: 'Hello' (characters from index 0 to 4)
print(string[7:])  # Output: 'World!' (characters from index 7 to the end)
print(string[:5])  # Output: 'Hello' (characters from the start to index 4)
print(string[-6:]) # Output: 'World!' (last 6 characters)
```

- **String Length:** You can find the length of a string using the **len()** function**:**

```
string = "Hello, World!"
print(len(string))  # Output: 13
```

- **String Concatenation:** You can combine strings using the + operator**:**

```
string1 = "Hello"
string2 = "World"
combined = string1 + ", " + string2 + "!"
print(combined)  # Output: 'Hello, World!'
```

- **String Repetition:** You can repeat strings using the * operator**:**

```
string = "Python! "
print(string * 3)  # Output: 'Python! Python! Python! '
```

- **String Methods:** Python provides many built-in methods for string manipulation. Here are some common ones:
  - **Changing Case**

```
string = "Hello, World!"
print(string.upper())   # Output: 'HELLO, WORLD!'
print(string.lower())   # Output: 'hello, world!'
```

  - **Finding Substrings**

```
string = "Hello, World!"
print(string.find("World"))  # Output: 7 (starting index of the substring)
print(string.find("Python"))  # Output: -1 (not found)
```

  - Replacing Substrings

```
string = "Hello, World!"
new_string = string.replace("World", "Python")
print(new_string)  # Output: 'Hello, Python!'
```

  - **Splitting and Joining Strings**

```
string = "Hello, World!"
words = string.split(", ")  # Split into a list
print(words)  # Output: ['Hello', 'World!']
```

```
joined_string = " ".join(words) # Join the list back into a string
print(joined_string)  # Output: 'Hello World!'
```

➢ **String Formatting:** You can format strings using **f-strings, str.format( ),** or the **%** operator**:**

- **Using f-strings**

```
name = "Alice"
age = 30
greeting = f"My name is {name} and I am {age} years old."
print(greeting)  # Output: 'My name is Alice and I am 30 years old.'
```

- **Using str.format( )**

```
name = "Alice"
age = 30
greeting = "My name is {} and I am {} years old.".format(name, age)
print(greeting)  # Output: 'My name is Alice and I am 30 years old.'
```

- **Using the % operator:**

```
name = "Alice"
age = 30
greeting = "My name is %s and I am %d years old." % (name, age)
print(greeting)# Output: 'My name is Alice and I am 30 years old.'
```

Example 1// Write a program that asks the user for a string and prints out the location of each 'a' in the string.

```
# Ask the user for a string
s = input('Enter some text: ')

# Iterate through the string to get the index of a character
for i in range(len(s)):
    if s[i]=='a':
        print(i)
```

Example 2// Write a program that asks the user for a string and creates a new string that doubles each character of the original string. For instance, if the user enters **Hello**, the output should be **HHeellloo**.

```
# Ask the user for a string
string = input('Enter some text: ')
doubled_s = ''
for char in string:
```

```
    doubled_s = doubled_s + char*2

print(doubled_s)
```

Example 3// People often forget closing parentheses when entering formulas. Write a program that asks the user to enter a formula and prints out whether the formula has the same number of opening and closing parentheses.

```
# Program to check for matching parentheses in a formula

# Ask the user to enter a formula
formula = input("Enter a formula: ")

# Initialize counters for opening and closing parentheses
opening_count = 0
closing_count = 0

# Loop through each character in the formula
for char in formula:
    if char == '(':
        opening_count += 1  # Increment count for opening
parentheses
    elif char == ')':
        closing_count += 1  # Increment count for closing
parentheses

# Check if the counts are the same
if opening_count == closing_count:
    print("The formula has matching parentheses.")
else:
    print("The formula does not have matching parentheses.")
```

H.W.//

Question 1// Write a program that asks the user to enter a string. The program should then print the following:

a) The total number of characters in the string.
b) The string repeated **10** times.
c) The first character of the string (remember that string indices start at **0**)
d) The first three characters of the string.
e) The last three characters of the string.
f) The seventh character of the string if the string is long enough and a message otherwise.
g) The string with its first and last characters removed.
h) The string in all capital letters.
i) The string with every **a** replaced with an **e**.

عنوان المحاضرة	المادة	المحاضرة	المرحلة	القسم
Data Structures in Python	برمجة – لغة بايثون	السابعة	الثانية	علوم الحاسبات

Question 2// Write a program that asks the user to enter a word and prints out whether that word contains any vowels.

Question 3// Write a program that asks the user to enter a word and determines whether the word is a palindrome (e.g. radar) or not. A palindrome is a word that reads the same backwards as forwards.

2. Lists in Python

- In Python, a **list** is a mutable, ordered collection of items, which allows you to store multiple elements in a **single variable**. Lists are one of the most versatile data structures in Python because they can hold items of **different data types,** including integers, strings, and even other lists. Lists are defined by square brackets **[ ]**, and their elements are separated by **commas**.
- **Key Features of Lists:**
  - **Ordered:** Elements in a list are stored in a specific order, and their position can be accessed using an index (starting from 0).
  - **Mutable:** You can change, add, or remove elements after the list has been created.
  - **Heterogeneous:** A list can contain elements of different types.
  - **Dynamic:** The size of a list can change dynamically as you add or remove elements.
- **Creating Lists:**

```
my_list = [1, 2, 3, "hello", True]
```

- **Accessing Elements in a list:**

```
my_list = [1, 2, 3, "hello", True]

# Accessing the first element
print(my_list[0])  # Output: 1

# Accessing the last element
print(my_list[-1])  # Output: True
```

- **List Slicing:** You can extract a portion of a list using slicing.

```
my_list = [10, 20, 30, 40, 50]

# Extracting elements from index 1 to 3 (exclusive of 4)
print(my_list[1:4])  # Output: [20, 30, 40]
```

- **Modifying a List**: Since lists are mutable, you can modify them after creation.

```
my_list = [10, 20, 30, 40, 50]
```

```
# Changing the second element
my_list[1] = 25
print(my_list)  # Output: [10, 25, 30, 40, 50]

# Appending an element
my_list.append(60)
print(my_list)  # Output: [10, 25, 30, 40, 50, 60]
# Inserting an element at a specific position
my_list.insert(2, 35)
print(my_list)  # Output: [10, 25, 35, 30, 40, 50, 60]
```

- **Removing Elements:** You can remove elements from a list using various methods.

```
my_list = [10, 20, 30, 40, 50]

# Removing an element by value
my_list.remove(30)
print(my_list)  # Output: [10, 20, 40, 50]

# Removing an element by index
del my_list[1]
print(my_list)  # Output: [10, 40, 50]

# Removing the last element
my_list.pop()
print(my_list)  # Output: [10, 40]
```

- **List Operations:** Lists support various operations such as **concatenation**, **repetition**, and **checking membership**

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Concatenating two lists
concatenated_list = list1 + list2
print(concatenated_list)  # Output: [1, 2, 3, 4, 5, 6]

# Repeating a list
repeated_list = list1 * 2
print(repeated_list)  # Output: [1, 2, 3, 1, 2, 3]

# Checking if an element is in the list
```

```
print(2 in list1)   # Output: True
print(10 in list1)   # Output: False
```

- **List Comprehension:** List comprehension provides a concise way to create lists. It is a powerful tool to transform or filter lists.

```
# Creating a list of squares using a loop
squares = [x**2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]

# Creating a list of even numbers using a condition
evens = [x for x in range(1, 11) if x % 2 == 0]
print(evens)  # Output: [2, 4, 6, 8, 10]
```

- **Common List Methods:** Here are some commonly used list methods:
  - **append ( ):** Adds an item to the end of the list.
  - **extend ( ):** Adds all items of another list to the current list.
  - **insert ( ):** Inserts an item at a given position.
  - **remove ( ):** Removes the first occurrence of an item.
  - **pop ( ):** Removes and returns the item at the given index (or **the last item if no index is specified**).
  - **sort ( ):** Sorts the list in ascending or descending order.
  - **reverse ( ):** Reverses the order of the elements in the list.
  - **index ( ):** Returns the index of the first occurrence of an item.
  - **count ( ):** Returns the number of occurrences of an item in the list.

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]

# Sorting the list
my_list.sort()
print(my_list)  # Output: [1, 1, 2, 3, 4, 5, 5, 6, 9]

# Reversing the list
my_list.reverse()
print(my_list)  # Output: [9, 6, 5, 5, 4, 3, 2, 1, 1]

# Counting occurrences of an element
print(my_list.count(5))  # Output: 2
```

القسم	المرحلة	المحاضرة	المادة	عنوان المحاضرة
علوم الحاسبات	الثانية	السابعة	برمجة – لغة بايثون	Data Structures in Python

- **Built-in functions:** There are several built-in functions that operate on lists. Here are some useful ones:

Function	Description
`len`	returns the number of items in the list
`sum`	returns the sum of the items in the list
`min`	returns the minimum of the items in the list
`max`	returns the maximum of the items in the list

For example, the following computes the average of the values in a list **L**:

```
average = sum(L)/len(L)
```

- **Making copies of a list:** Another place that lists and strings differ is when we try to make copies. Consider the following code:

```
s = 'Hello'
copy = s
s = s + '!!!'
print('s is now:', s, 'Copy:', copy)
```

```
s is now: Hello!!!    Copy: Hello
```

In the code above we make a copy of **s** and then change **s**. Everything works as we would intuitively expect. Now look at similar code with lists:

```
L = [1,2,3]
copy = L
L[0]=9
print('L is now:', L, 'Copy:', copy)
```

```
L is now: [9, 2, 3]    Copy: [9, 2, 3]
```

We can see that the list code did not work as we might have expected. When we changed **L**, the **copy** got changed as well. The proper way to make a copy of **L** is **copy = L[:]**.

القسم	المرحلة	المحاضرة	المادة	عنوان المحاضرة
علوم الحاسبات	الثانية	السابعة	برمجة – لغة بايثون	Data Structures in Python

Example 1// Write a program that generates a list **L** of **50** random numbers between **1** and **100**.

```
from random import randint
L = []
for i in range(50):
    L.append(randint(1,100))

print('the created list is:', L)
```

Example 2// Write a Python program asking the user to enter a list and replacing each element in the entered list with its square.

```
# Ask the user to input a list of numbers
input_list = input("Enter a list of numbers separated by spaces:
")

# Convert the input string into a list of integers
numbers = list(map(int, input_list.split()))

# Replace each element with its square using list comprehension
squared_numbers = [x**2 for x in numbers]
# Print the squared list
print("List after replacing each element with its square:")
print(squared_numbers)
```

Enter a list of numbers separated by spaces: 1 2 3 4 5
List after replacing each element with its square: [1, 4, 9, 16, 25]

Example 3// Write a Python program asking the user to enter a list and counting how many items in the entered list are greater than 50.

```
# Ask the user to input numbers separated by spaces
user_input = input("Enter numbers separated by spaces: ")

# Convert the input string into a list of integers
num_list = list(map(int, user_input.split()))

# Initialize a counter for numbers greater than 50
count = 0

# Iterate through the list and count numbers greater than 50
```

```
for num in num_list:
    if num > 50:
        count += 1

# Print the result
print(f"Number of items greater than 50: {count}")
```

Example 4// Write a program that prints out the two largest and two smallest elements of a list called scores.

```
# Sample list of scores
scores = [85, 23, 90, 67, 45, 77, 91, 33, 100, 58, 99]

# Ensure the list has at least 2 elements
if len(scores) < 2:
    print("The list must contain at least 2 elements.")
else:
    # Sort the list in ascending order
    sorted_scores = sorted(scores)

    # Print out the sorted list
    print('The sorted list is:',sorted_scores)

    # Get the two smallest elements (first two elements in the
sorted list)
    smallest_two = sorted_scores[:2]

    # Get the two largest elements (last two elements in the
sorted list)
    largest_two = sorted_scores[-2:]

    # Print the results
    print(f"Two smallest elements: {smallest_two}")
    print(f"Two largest elements: {largest_two}")
```

Example 5// Write a python program to play a simple quiz game using a list of questions and their answers. Then check how many times the user gets the correct answer.

```
# List of questions
questions = [
    "What is the capital of France?",
    "What is 5 + 7?",
    "Who wrote 'Hamlet'?",
    "What is the boiling point of water in Celsius?",
    "What is the largest planet in our solar system?"]

# List of corresponding answers
answers = ["Paris", "12", "Shakespeare", "100", "Jupiter"]
```

```
# Variable to count the number of correct answers
correct_answers = 0

# Loop through the questions using their index
for i in range(len(questions)):
    # Ask the user the question and get their input
    user_answer = input(questions[i] + " ")

    # Check if the user's answer is correct (case insensitive
comparison)
    if user_answer.lower() == answers[i].lower():
        print("Correct!")
        correct_answers += 1
    else:
        print(f"Incorrect. The correct answer is: {answers[i]}")

# Print the total number of correct answers
print(f"\nYou got {correct_answers} out of {len(questions)}
questions correct!")
```

Example 6// Write a program that rotates the elements of a list so that the element at the first index moves to the second index, the element in the second index moves to the third index, etc., and the element in the last index moves to the first index.

```
# Function to rotate the list
def rotate_list(lst):
    if len(lst) == 0:
        return lst  # Return the list if it's empty

    # Rotate by taking the last element and placing it at the
start, followed by the rest of the list
    return [lst[-1]] + lst[:-1]

# Example list
numbers = [10, 20, 30, 40, 50]

# Print the original list
print("Original list:", numbers)

# Rotate the list
rotated_list = rotate_list(numbers)
```

القسم	المرحلة	المحاضرة	المادة	عنوان المحاضرة
علوم الحاسبات	الثانية	السابعة	برمجة – لغة بايثون	Data Structures in Python

```
# Print the rotated list
print("Rotated list:", rotated_list)
```

H.W//

Question 1// Write a program that asks the user to enter a list of integers. Do the following:

(a) Print the total number of items in the list.
(b) Print the last item in the list.
(c) Print the list in reverse order.
(d) Print Yes if the list contains a 5 and No otherwise.
(e) Print the number of fives in the list.
(f) Print how many integers in the list are less than 5.

Question 2// Write a program that generates a list of 20 random numbers between 1 and 100.

(a) Print the list.
(b) Print the average of the elements in the list.
(c) Print the largest and smallest values in the list.
(d) Print the second largest and second smallest entries in the list
(e) Print how many even numbers are in the list.

Question 3// Ask the user to enter a list containing numbers between 1 and 12. Then replace all of the entries in the list that are greater than 10 with 10.

Question 4// Write a program that removes any repeated items from a list so that each item appears at most once. For instance, the list [1,1,2,3,4,3,0,0] would become [1,2,3,4,0].

3. Tuple in Python

- In Python, a tuple is a built-in data structure that is used to store an ordered collection of items. Tuples are similar to lists, but they have a few key differences:
  1. **Immutability:** Tuples are immutable, meaning that once a tuple is created, its contents cannot be changed. You cannot add, remove, or modify elements.
  2. **Syntax:** Tuples are defined using parentheses **( )** rather than square brackets **[ ]** (used for lists).
- **Creating Tuples:** You can create a tuple by placing items inside parentheses, separated by commas:

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)  # Output: (1, 2, 3, 4, 5)

# A tuple can contain different data types
```

```
mixed_tuple = (1, "Hello", 3.14, True)
print(mixed_tuple)  # Output: (1, 'Hello', 3.14, True)

# An empty tuple
empty_tuple = ()
print(empty_tuple)  # Output: ()
```

- **Accessing Tuple Elements:** You can access elements of a tuple using indexing (like lists):

```
my_tuple = (10, 20, 30, 40, 50)

# Accessing elements
print(my_tuple[0])  # Output: 10
print(my_tuple[1])  # Output: 20
print(my_tuple[-1]) # Output: 50 (last element)
print(my_tuple[-2]) # Output: 40 (before the last element)
```

- **Slicing Tuples:** You can slice a tuple to get a subset of elements:

```
my_tuple = (1, 2, 3, 4, 5)

# Slicing the tuple
print(my_tuple[1:4])  # Output: (2, 3, 4)
print(my_tuple[:3])   # Output: (1, 2, 3)
print(my_tuple[2:])   # Output: (3, 4, 5)
```

- **Tuple Immutability:** Since tuples are immutable, you cannot change their content:

```
my_tuple = (1, 2, 3)
my_tuple(0) = 10  # This will raise a TypeError
```

- **Nested Tuples:** You can create tuples that contain other tuples:

```
nested_tuple = ((1, 2), (3, 4), (5, 6))
print(nested_tuple)     # Output: ((1, 2), (3, 4), (5, 6))
print(nested_tuple[0])     # Output: (1, 2)
print(nested_tuple[0][1])  # Output: 2
print(nested_tuple[1][1])  # Output: 4
```

- **Tuple Packing and Unpacking:** Tuples can be packed and unpacked, which means you can assign multiple values to a tuple in one go and unpack them back into variables:

```
# Packing
my_tuple = (1, "Hello", 3.14)

# Unpacking
a, b, c = my_tuple
print(a)  # Output: 1
print(b)  # Output: Hello
print(c)  # Output: 3.14
```

4. Sets in Python

- In Python, a set is a built-in data structure that is used to store an unordered collection of unique items. Sets are useful when you want to ensure that there are no duplicate values and when the order of the elements does not matter.
- **Key Features of Sets:**
  1. **Unordered**: Sets do not maintain any specific order for their elements.
  2. **Unique Elements**: A set cannot contain duplicate values. If you attempt to add a duplicate, it will be ignored.
  3. **Mutable**: You can add or remove elements from a set after it has been created.
- **Creating Sets:** You can create a set using curly braces **{ }** or the **set( )** constructor.

```
# Creating a set using curly braces
my_set = {1, 2, 3, 4, 5}
print(my_set)  # Output: {1, 2, 3, 4, 5}

# Creating a set using the set() constructor
another_set = set([1, 2, 3, 4, 5])
print(another_set)  # Output: {1, 2, 3, 4, 5}

# Sets automatically remove duplicates
duplicate_set = {1, 2, 2, 3, 4}
print(duplicate_set)  # Output: {1, 2, 3, 4}
```

- **Accessing Set Elements:** Since sets are unordered, you cannot access elements by index. However, you can iterate over the elements in a set:

```
my_set = {1, 2, 3, 4, 5}

# Iterating over a set
```

```
for item in my_set:
    print(item)
```

- **Adding and Removing Elements:** You can add elements to a set using the **add ( )** method and remove them using the **remove ( )** or **discard ( )** methods:

```
my_set = {1, 2, 3}

# Adding an element
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}

# Removing an element
my_set.remove(2)
print(my_set)  # Output: {1, 3, 4}

# Discarding an element (does not raise an error if the
element is not found)
my_set.discard(5)  # No error even though 5 is not in the set
print(my_set)  # Output: {1, 3, 4}
```

- **Set Operations:** Sets support various operations, such as **union, intersection, difference, and symmetric difference.**

Operator	Description	Example
|	union	{1,2,3} | {3,4} → {1,2,3,4}
&	intersection	{1,2,3} & {3,4} → {3}
-	difference	{1,2,3} - {3,4} → {1,2}
^	symmetric difference	{1,2,3} ^ {3,4} → {1,2,4}
in	is an element of	3 in {1,2,3} → True

```
# Union: Combines two sets.
set_a = {1, 2, 3}
set_b = {3, 4, 5}
union_set = set_a | set_b  # or set_a.union(set_b)
print(union_set)  # Output: {1, 2, 3, 4, 5}

# Intersection: Finds common elements in both sets.
intersection_set = set_a & set_b  # or set_a.intersection(set_b)
print(intersection_set)  # Output: {3}

# Difference: Finds elements in one set that are not in the
```

```
other.
difference_set = set_a - set_b  # or set_a.difference(set_b)
print(difference_set)  # Output: {1, 2}

# Symmetric Difference: Finds elements in either set but not in
both.
symmetric_difference_set = set_a ^ set_b  # or
set_a.symmetric_difference(set_b)
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

5. Dictionaries in Python

- In Python, a dictionary is a built-in data structure that stores data in key-value pairs. Dictionaries are unordered, mutable (modifiable), and indexed by keys, which are unique and immutable. The keys can be of various data types, such as strings, numbers, or tuples, while values can be of any data type.
- **Key Features of Dictionaries:**
  1. **Unordered:** The items in a dictionary are not stored in any particular order.
  2. **Mutable:** You can change the values in a dictionary, but the keys must be unique and immutable (e.g., strings or numbers).
  3. **Key-Value Pairs:** Dictionaries store data as pairs of keys and values.
- **Creating a Dictionary:** You can create a dictionary by enclosing key-value pairs in curly braces { }, with the key and value separated by a colon :.

```
# Example of a simple dictionary
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
print(my_dict)
# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

- **Accessing Dictionary Values:** You can access values in a dictionary by referring to their keys:

```
# Example of creating a simple dictionary
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
```

عنوان المحاضرة	المادة	المحاضرة	المرحلة	القسم
Data Structures in Python	برمجة – لغة بايثون	السابعة	الثانية	علوم الحاسبات

```
# Accessing values by key
print(my_dict["name"])  # Output: Alice
print(my_dict["age"])   # Output: 25
```

- **Adding and Modifying Items:** You can add a new key-value pair or modify an existing one by assigning a value to a key:

```
my_dict = {"name": "Alice", "age": 25}

# Adding a new key-value pair
my_dict["city"] = "New York"
print(my_dict)
# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Modifying an existing value
my_dict["age"] = 26
print(my_dict)
# Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

- **Removing Items:** You can remove items from a dictionary using the **del** statement or the **pop ( )** , or **popitem ( )** methods:

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Using del to remove a key-value pair
del my_dict["age"]

# Using pop() to remove a key-value pair and return its value
city = my_dict.pop("city")

# Using popitem() to remove and return the last inserted key-value pair
last_item = my_dict.popitem()

print(my_dict)        # Output: {}
print(city)           # Output: New York
print(last_item)      # Output: ('name', 'Alice')
```

- **Dictionary Methods:** Python provides several methods for working with dictionaries:
  - **keys ( ):** Returns a view object containing all keys.
  - **values ( ):** Returns a view object containing all values.
  - **items ( ):** Returns a view object containing key-value pairs.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

```
# Getting all keys
keys = my_dict.keys()
print(keys)  # Output: dict_keys(['name', 'age', 'city'])

# Getting all values
values = my_dict.values()
print(values)
# Output: dict_values(['Alice', 25, 'New York'])

# Getting all key-value pairs
items = my_dict.items()
print(items)
# Output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New
York')])
```

- **Iterating Over a Dictionary:** You can iterate through the keys, values, or key-value pairs in a dictionary:

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Iterating over keys
for key in my_dict:
    print(key)

# Iterating over values
for value in my_dict.values():
    print(value)

# Iterating over key-value pairs
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

- **Dictionary with Mixed Data Types:** The keys and values in a dictionary can be of different types, including **numbers**, **strings**, **lists**, or even other **dictionaries**:

```
mixed_dict = {
    1: "apple",
    "age": 25,
    "items": [1, 2, 3],
    "info": {"name": "Bob", "job": "developer"}
}

print(mixed_dict)
# Output: {1: 'apple', 'age': 25, 'items': [1, 2, 3], 'info':
{'name': 'Bob', 'job': 'developer'}}
```

عنوان المحاضرة	المادة	المحاضرة	المرحلة	القسم
Data Structures in Python	برمجة – لغة بايثون	السابعة	الثانية	علوم الحاسبات

- **Checking if a Key Exists:** You can check if a key exists in a dictionary using the in keyword:

```
my_dict = {"name": "Alice", "age": 25}

# Check if a key exists
if "name" in my_dict:
    print("Name is a key in the dictionary")

# Check if a key does not exist
if "city" not in my_dict:
    print("City is not a key in the dictionary")
```

Example 1// Write a program that repeatedly asks the user to enter product names and prices. Store all of these in a dictionary whose keys are the product names and whose values are the prices. When the user is done entering products and prices, allow them to repeatedly enter a product name and print the corresponding price or a message if the product is not in the dictionary.

```
# Initialize an empty dictionary to store products and prices
products = {}

# Step 1: Enter product names and prices
while True:
    # Ask the user for a product name
    product_name = input("Enter product name (or 'done' to stop):
").strip()

    # Break the loop if the user is done entering products
    if product_name.lower() == 'done':
        break

    # Ask the user for the product price
    try:
        price = float(input(f"Enter price for {product_name}: "))
        products[product_name] = price  # Store product name and
price in the dictionary
    except ValueError:
        print("Please enter a valid number for the price.")

# Print out the created dictionary
print(products)

# Step 2: Allow the user to look up product prices
while True:
    # Ask the user for a product name to look up
    search_product = input("Enter a product name to get its price
```

```
(or 'exit' to stop): ").strip()

    # Break the loop if the user wants to stop
    if search_product.lower() == 'exit':
        break

    # Print the corresponding price or a message if the product
is not found
    if search_product in products:
        print(f"The price of {search_product} is:
${products[search_product]:.2f}")
    else:
        print(f"{search_product} is not in the product list.")
```

Example Output:

```
Enter product name (or 'done' to stop): Apple
Enter price for Apple: 1.2
Enter product name (or 'done' to stop): Banana
Enter price for Banana: 0.5
Enter product name (or 'done' to stop): Milk
Enter price for Milk: 2.49
Enter product name (or 'done' to stop): done

Enter a product name to get its price (or 'exit' to stop): Apple
The price of Apple is: $1.20
Enter a product name to get its price (or 'exit' to stop): Milk
The price of Milk is: $2.49
Enter a product name to get its price (or 'exit' to stop): Bread
Bread is not in the product list.
Enter a product name to get its price (or 'exit' to stop): exit
```

Example 2// Write a Python program to create a dictionary whose keys are month names and whose values are the number of days in the corresponding months.

(a) Ask the user to enter a month name and use the dictionary to tell them how many days are in the month.

(b) Print out all of the keys in alphabetical order.

(c) Print out all of the months with 31 days.

```
# Dictionary of month names and the number of days in each month
months = {
    "January": 31, "February": 28, "March": 31, "April": 30,
    "May": 31, "June": 30, "July": 31, "August": 31,
    "September": 30, "October": 31, "November": 30, "December":
```

```
31
}

# (a) Ask the user to enter a month name and use the dictionary
to tell them how many days are in the month
month_name = input("Enter a month name: ").capitalize()  #
Capitalizing to match the dictionary keys
if month_name in months:
    print(f"{month_name} has {months[month_name]} days.")
else:
    print("Invalid month name.")

# (b) Print out all of the keys in alphabetical order
print("\nMonths in alphabetical order:")
for month in sorted(months.keys()):
    print(month)

# (c) Print out all of the months with 31 days
print("\nMonths with 31 days:")
for month, days in months.items():
    if days == 31:
        print(month)
```

H.W.//

Question 1// Using the dictionary created in the **example 1**, allow the user to enter a dollar amount and print out all the products whose price is less than that amount.

Question 2// Write a program that uses a dictionary that contains ten user names and passwords. The program should ask the user to enter their username and password. If the username is not in the dictionary, the program should indicate that the person is not a valid user of the system. If the username is in the dictionary, but the user does not enter the right password, the program should say that the password is invalid. If the password is correct, then the program should tell the user that they are now logged in to the system.