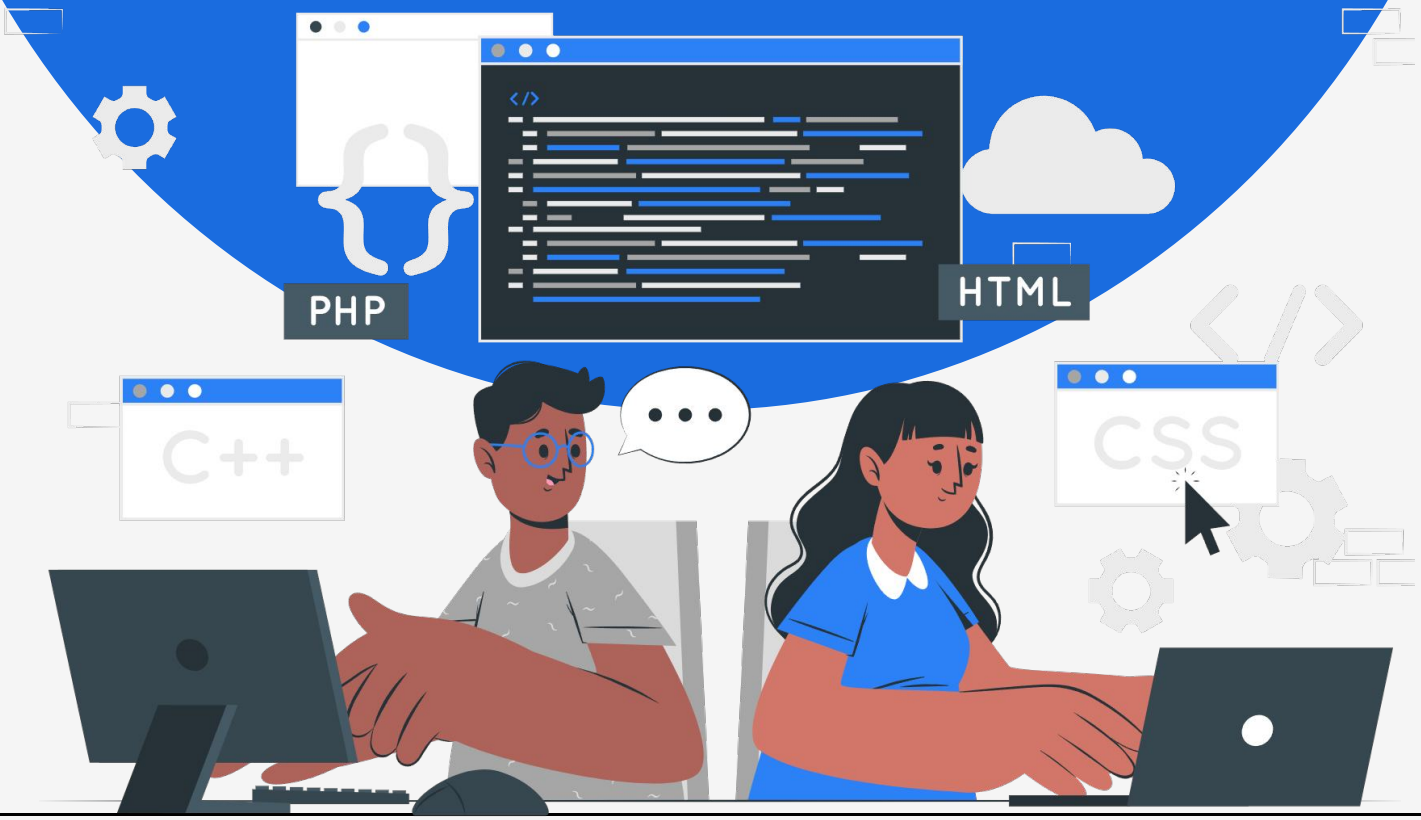
Lesson Plan:

Data Structures in Python



PW SKILLS

Data Structures in Python					
List	Tuple	Set	Dictionaries	Array	Stack & Queue

Introduction

- Data structures play a pivotal role in computer science and programming, as they are essential for storing, organizing, and manipulating data efficiently.
- Python, a versatile and popular programming language, provides a wide array of data structures to accommodate various needs.
  - Data Handling: Programming often involves working with data, which can be numbers, text, or more complex information. Data structures are like specialized containers that help us manage this data efficiently.
  - Organization: Just as you use different shelves, drawers, and containers at home to organize various items, in programming, you use different data structures to organize different types of data.
  - Efficiency: The choice of data structure can significantly impact how quickly you can access, manipulate, and process your data. Picking the right data structure is crucial for writing efficient code.
  - Flexibility: Different data structures serve different purposes. Some are great for storing lots of data, while others are optimized for quick searches or data retrieval.
  - Built-in Tools: Python provides a rich set of built-in data structures that make it easier for programmers to handle common data management tasks without having to build everything from scratch.
  - Real-World Analogy: You can think of data structures as tools in your toolbox. For example, if you want to organize your books, you might use a bookshelf (analogous to a list). If you want to store unique items, like a collection of rare coins, you'd use a display case (analogous to a set).
  - Custom Data Structures: In addition to built-in data structures, Python allows you to create custom data structures to suit your specific needs. This is akin to crafting your own unique container for a particular purpose.

Data Structure in Python

- Data structures in Python are specialized formats or containers used to store and organize data efficiently.
- They provide various ways to store, access, and manipulate data according to specific needs.

Here are the key data structures in Python:

- **List**
  - A list is one of the most versatile and commonly used data structures in Python. It is an ordered collection of elements that can be of any data type, such as integers, strings, or other lists.
  - Lists are mutable, which means you can change their contents after creation.
  - You can access elements by their index, and you can perform various operations like adding, removing, or sorting elements.

Example 1:

```
# Creating a list of numbers
numbers = [1, 2, 3, 4, 5]

# Accessing and printing the third element (index 2)
print("The third number is:", numbers[2])
```

Output:

The third number is: 3

Example 2:

```
# Creating a list of phone
Phones = ["Realme", "Mi", "Samsung", "Iphone"]

# Adding a new phone to the list
Phones.append("Oppo")

# Removing a phone from the list
Phones.remove("Mi")

# Printing the updated list
print("Updated list of phones:", Phones)
```

Output:

Updated list of phones: ['Realme', 'Samsung', 'Iphone', 'Oppo']

Example 3:

```
# Creating a list with mixed data types
info = ["Julie", 21, 1.75, True]

# Accessing and printing elements
name = info[0]
age = info[1]

print("Name:", name)
print("Age:", age
```

Output:

Name: Julie
Age: 21

Example 4:

```
# Creating a list of lists
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Accessing and printing a specific element
element = matrix[1][2]

print("Element at row 1, column 2:", element)
```

Output:

Element at row 1, column 2: 6

Real-World Analogy: Grocery Shopping

- Imagine a shopping list as a Python list.
- It's like having a piece of paper where you jot down items you need to buy at the grocery store. Each item is a list element.
- You can add new items, remove items, or check if a specific item is on the list.

Code:

```
# Creating a shopping list
shopping_list = ["Apples", "Bananas", "Milk", "Bread"]

# Checking if an item is on the list
item_to_check = "Milk"

if item_to_check in shopping_list:
    print(item_to_check, "is on the shopping list.")
else:
    print(item_to_check, "is not on the shopping list.")
```

Output:

Milk is on the shopping list.

- **Tuples**
  - Tuples are similar to lists, but they are immutable.
  - Once you create a tuple, you cannot change its content. This immutability makes tuples useful for situations where you want to ensure that the data remains constant.
  - They are often used to represent collections of related values.

Example 1:

```
# Tuple to store coordinates (x, y)
point = (3, 4)
print("X-coordinate:", point[0])
print("Y-coordinate:", point[1])
```

Output:

X-coordinate: 3
Y-coordinate: 4

Example 2:

```
# Packing values into a tuple
person = ("Arjun", 28, "Bengaluru")

# Unpacking values from the tuple

name, age, city = person

print(f"Name: {name}, Age: {age}, City: {city}")
```

Output:

Name: Arjun, Age: 28, City: Bengaluru

Example 3:

```
def get_student_info():
    # Simulating a function that returns multiple values
    name = "Arun"
    age = 21
    grade = "A"
    return name, age, grade

student_info = get_student_info()
name, age, grade = student_info
print(f"Name: {name}, Age: {age}, Grade: {grade}")
```

Output:

Name: Arun, Age: 21, Grade: A

Example 4:

```
# Combining two tuples
colors1 = ("red", "green", "blue")
colors2 = ("yellow", "orange")

combined_colors = colors1 + colors2
print("Combined Colors:", combined_colors)
```

Output:

Combined Colors: ('red', 'green', 'blue', 'yellow', 'orange')

Real World Analogy:

- Imagine you have different boxes, each containing a variety of chocolates. You want to represent this situation using tuples:

Code:

```
# Define tuples to represent boxes of chocolates
box1 = ("Dark Chocolate", "Milk Chocolate", "White Chocolate")
box2 = ("Caramel Chocolate", "Hazelnut Chocolate")
box3 = ("Raspberry Chocolate", "Coconut Chocolate")
# Combine the boxes into a larger bag
chocolate_bag = (box1, box2, box3)

# Access the chocolates in the bag
for box in chocolate_bag:
    print("Box contents:")
    for chocolate in box:
        print(" - " + chocolate)
```

Output:

Box contents:
- Dark Chocolate
- Milk Chocolate
- White Chocolate

Box contents:
- Caramel Chocolate
- Hazelnut Chocolate

Box contents:
- Raspberry Chocolate
- Coconut Chocolate

Sets

- Sets are unordered collections of unique elements. They are useful when you need to work with distinct items and perform set operations like union, intersection, and difference.
- Sets are mutable, so you can add and remove elements, but the elements themselves must be immutable.

Example 1:

```
# Create a set of favorite colors
favorite_colors = {"red", "blue", "green", "yellow"}
print(favorite_colors)
```

Output:

{'green', 'yellow', 'blue', 'red'}

Example 2:

```
# Create an empty set for shopping items
shopping_list = set()
# Add items to the shopping list
shopping_list.add("apples")
shopping_list.add("bananas")
shopping_list.add("milk")
print(shopping_list)
```

Output:

{'apples', 'bananas', 'milk'}

Example 3:

```
# Create a set of courses you've passed
passed_courses = {"Math", "History", "Biology", "English"}
# Remove a course you didn't pass
passed_courses.remove("Math")
print(passed_courses)
```

Output:

{'English', 'History', 'Biology'}

Example 4:

```
# Create sets of fruits
fruits_set_1 = {"apple", "banana", "cherry"}
fruits_set_2 = {"banana", "orange", "strawberry"}
# Combine both sets to get a unique list of fruits
all_fruits = fruits_set_1.union(fruits_set_2)
print(all_fruits)
```

Output:

{'apple', 'banana', 'cherry', 'orange', 'strawberry'}

Real-World Analogy

- Think of a set as your shopping list. You want to buy specific items at the grocery store. Your shopping list contains unique items, and you can add or remove items as needed. You don't want duplicates on your list, and you can quickly check whether an item is on your list or not. If you decide not to buy an item, you remove it from the list.

Code:

```
shopping_list = set()

# Add items to your shopping list
shopping_list.add("apples")
shopping_list.add("bananas")
shopping_list.add("milk")

# Check if an item is on your list
if "apples" in shopping_list:
    print("You need to buy apples.")
else:
    print("You don't need to buy apples.")

# Remove an item from your list if you change your mind
shopping_list.discard("milk")

# Check if an item is on your list again
if "milk" in shopping_list:
    print("You need to buy milk.")
else:
    print("You don't need to buy milk.")

# Print the final shopping list
print("Your shopping list contains:", shopping_list)
```

Output:

You need to buy apples.
You don't need to buy milk.
Your shopping list contains: {'apples', 'bananas'}

Dictionaries

- Dictionaries are key-value pairs that allow you to store and retrieve data based on a unique key. The keys must be immutable, such as strings or numbers.
- Dictionaries are useful for storing data that needs to be accessed quickly and efficiently.

Example 1:

```
my_dict = {'Course': 'PWIOI', 'Duration': 2, 'city': 'Banglore'}
print(my_dict)
type(my_dict)
```

Output:

{'Course': 'PWIOI', 'Duration': 2, 'city': 'Banglore'}

Example 2

```
phonebook = {
    'PW Customer care': 1234567890,
    'PW Team': 9876543210,
    'PW Enquiry': 5555555555,
    'PW Admmision Dep': 3333333333,
    'PWIOI Teacher': 8888888888
}
```

Output:

{'PW Customer care': 1234567890, 'PW Team': 9876543210, 'PW Enquiry': 5555555555, 'PW Admmision Dep': 3333333333, 'PWIOI Teacher': 8888888888}

Example 3

```
student_grades = {
    'Jeni': 85,
    'jimmy': 92,
    'Sindu': 78,
    'Akil': 95,
    'Chaitu': 89
}
```

Output:

{'Jeni': 85, 'jimmy': 92, 'Sindu': 78, 'Akil': 95, 'Chaitu': 89}

Example 4

```
movie_ratings = {
    'Inception': 4.5,
    'The Shawshank Redemption': 4.7,
    'Pulp Fiction': 4.2,
    'The Godfather': 4.6,
    'Forrest Gump': 4.4
}
movie_ratings
```

Output:

{'Inception': 4.5, 'The Shawshank Redemption': 4.7, 'Pulp Fiction': 4.2, 'The Godfather': 4.6, 'Forrest Gump': 4.4}

Real-World Analogy

- Let's consider a real-world analogy for a dictionary using a Product Catalog. Imagine you're running an online store, and you need to keep track of the prices of various products in your catalog. Here's a Python code example along with a real-world analogy.

Code:

```
product_catalog = {
    'iPhone 13': 999.99,
    'Samsung Galaxy S22': 849.99,
    'Sony PlayStation 5': 499.99,
    'MacBook Air': 1199.99,
    'LG 55-inch 4K TV': 799.99
}

# Accessing the price of an iPhone 13
iphone_price = product_catalog['iPhone 13']
print(f"The price of the iPhone 13 is ${iphone_price:.2f}")
```

Output:

The price of the iPhone 13 is $999.99

Arrays

- In Python, arrays are similar to lists, but they are more efficient for working with numerical data. To use 'arrays', you need to import the array module from the 'array' library.
- Arrays provide a memory-efficient way to work with large datasets of the same data type.

Example 1

```
#Temperature Readings:
#Imagine you're collecting daily temperature readings for a
month.
#You can use an array to store these values in a structured way.
from array import array
temperature_readings = array('i', [75, 76, 74, 72, 71, 73, 77])
```

Output:

array('i', [75, 76, 74, 72, 71, 73, 77])

Example 2

```
""" Student Scores:
Suppose you're keeping track of test scores for a class of
students.
An array can be used to store these scores."""
from array import array
student_scores = array('i', [85, 90, 78, 92, 88, 76])
student_scores
```

Output:

array('i', [85, 90, 78, 92, 88, 76])

Example 3

```
from array import array
stock_prices = array('i', [150, 152, 155, 158, 160, 157])
stock_prices
```

Output:

array('i', [150, 152, 155, 158, 160, 157])

Example 4

```
from array import array
employee_ages = array('i', [30, 35, 28, 45, 32, 40])
employee_ages
```

Output:

array('i', [30, 35, 28, 45, 32, 40])

Real World Analogy:
Code:

```
"""Imagine you're preparing a grocery shopping list.
Your list will consist of various items that you need to buy at
the store.
Each item can be thought of as an integer representing the
quantity of that item you plan to purchase."""
from array import array

# Create an array to represent your grocery shopping list
shopping_list = array('i', [3, 2, 1, 4, 2])

# The array contains quantities of items: apples, bananas, bread,
milk, and eggs

# Let's print and process the shopping list
print("Your Grocery Shopping List:")
print("Apples: ", shopping_list[0])
print("Bananas: ", shopping_list[1])
print("Bread: ", shopping_list[2])
print("Milk: ", shopping_list[3])
print("Eggs: ", shopping_list[4])

# You can also update the quantities, for example, if you decide
to buy more milk:
shopping_list[3] = 2  # Change the quantity of milk to 2
```

```
# Print the updated shopping list
print("\nUpdated Shopping List:")
print("Apples: ", shopping_list[0])
print("Bananas: ", shopping_list[1])
print("Bread: ", shopping_list[2])
print("Milk: ", shopping_list[3])
print("Eggs: ", shopping_list[4])
```

Output:

Your Grocery Shopping List:
Apples: 3
Bananas: 2
Bread: 1
Milk: 4
Eggs: 2

Updated Shopping List:
Apples: 3
Bananas: 2
Bread: 1
Milk: 2
Eggs: 2

Introduction To Lists

- Lists in Python are ordered collections of items, where each item can be of any data type.
- Lists are defined by enclosing a comma-separated sequence of items within square brackets '[]'.
- Lists in Python are similar to arrays in other programming languages but are more versatile because they can store different data types in the same list.

Creating a List

- You can create a list by simply assigning a sequence of values to a variable using square brackets.
- Lists can contain a mix of data types, such as integers, strings, and booleans.

Code:

```
#Example-1
# Imagine creating a grocery shopping list in Python.
grocery_list = ['apples', 'bananas', 'milk', 'eggs', 'bread']
print(grocery_list)
```

Output:

['apples', 'bananas', 'milk', 'eggs', 'bread']

```
#Example-2
#Representing a to-do list of tasks to complete.
todo_list = ['workout', 'meeting', 'buy groceries', 'pay bills']
print(todo_list)
```

Output:

['workout', 'meeting', 'buy groceries', 'pay bills']

```
#Example-3
#Analogy: Creating a list of students in a classroom.
students = ['Arun', 'Arjun', 'Charle', 'David', 'Eva']
print(students)
```

Output:

['Arun', 'Arjun', 'Charle', 'David', 'Eva']

Accessing Elements

Indexing and slicing

- Lists are zero-indexed, meaning the first element is at index 0, the second at index 1, and so on.
- You can access elements by using square brackets and the index of the element.

Code:

```
#Example-1
#Analogly: Imagine a list as a book with pages. Indexing is like
turning to a specific page, and slicing is like reading a range
of pages.
book_pages = ["Introduction", "Chapter 1", "Chapter 2", "Chapter
3", "Conclusion"]
first_page = book_pages[0]
chapter_1_to_3 = book_pages[1:4]
print(chapter_1_to_3)
```

Output:

['Chapter 1', 'Chapter 2', 'Chapter 3']

```
#Example-2
#Analogy: Think of a list as a music playlist. Indexing is like
selecting a particular song, and slicing is like creating a sub-
playlist.
playlist = ["Song 1", "Song 2", "Song 3", "Song 4", "Song 5"]
song_2 = playlist[1]
sub_playlist = playlist[2:5]
print(sub_playlist)
```

Output:

['Song 3', 'Song 4', 'Song 5']

```
#Example-4
#Analogy: Picture a list as a calendar with events. Indexing is
like focusing on one event, and slicing is like highlighting a
period of time.
calendar = ["Meeting", "Lunch", "Conference", "Appointment",
"Presentation"]
appointment = calendar[3]
busy_schedule = calendar[1:4]
print(appointment)
print(busy_schedule)
```

Output:

Appointment
['Lunch', 'Conference', 'Appointment']

```
#Example-5
#Analogy: Imagine a list as a collection of movies. Indexing is
like selecting a specific movie, and slicing is like creating a
marathon of movies.
movies = ["Action Movie", "Comedy Movie", "Horror Movie", "Sci-Fi
Movie", "Drama Movie"]
selected_movie = movies[2]
movie_marathon = movies[0:4]
print(selected_movie)
print(movie_marathon)
```

Output:

Horror Movie ['Action Movie', 'Comedy Movie', 'Horror Movie', 'Sci-Fi Movie']

Negative indexing

- Negative indexing allows you to access elements from the end of the list.
- '-1' represents the last element, '-2' the second-to-last, and so on.

Code:

```
#Example-1:
#Analogy: Imagine a book where the last page is -1, the second-
to-last is -2, and so on. Negative indexing allows you to easily
flip to pages from the end.
pages = ["Title Page", "Chapter 1", "Chapter 2", "Conclusion",
"Index"]
last_page = pages[-1]
second_last_page = pages[-2]
print(last_page)
print(second_last_page)
```

Output:

Index
Conclusion

```
#Example-2
#Analogy:  In a queue at a supermarket, you can count people
starting from the last person (the person at the end of the
queue) as -1, the second-to-last as -2, and so on.
queue = ["Arjun", "Krish", "Karan", "David"]
last_person = queue[-1]
second_last_person = queue[-2]
print(last_person)
print(second_last_person)
```

Output:

David
Karan

```
#Example-3
#Analogy: In a playlist of songs, the last song can be thought of
as -1, the second-to-last as -2, and so on. Negative indexing
helps you play songs from the end.
playlist = ["Hipop", "Classic", "Sal-sa", "Katakali"]
last_song = playlist[-1]
second_last_song = playlist[-2]
print(last_song)
print(second_last_song)
```

Output:

Katakali
Sal-sa

```
#Example-4
#Analogy: On a film reel, the final frame can be seen as -1, the
second-to-last as -2, and so on. Negative indexing helps you view
frames from the end of the film.
film_reel = ["Frame 1", "Frame 2", "Frame 3", "Frame 4"]
last_frame = film_reel[-1]
second_last_frame = film_reel[-2]
print(last_frame)
print(second_last_frame)
```

Output:

Frame 4
Frame 3

```
#Example-5
#Analogy: In a calendar, you can count days starting from the
last day of the month as -1, the second-to-last as -2, and so on.
Negative indexing helps you navigate days in reverse order.
days_of_month = ["Day 1", "Day 2", "Day 3", "Day 4", "Last Day"]
last_day = days_of_month[-1]
second_last_day = days_of_month[-2]
print(last_day)
print(second_last_day)
```

Output:

Last Day
Day 4

Modifying Lists

Append method

- The append() method is used to add an element to the end of a list.
- It modifies the original list by adding a new element.
- It is a common operation when you want to add new data or elements to an existing list.

Code:

```
#Example-1
#Analogy: Imagine a shopping cart in a supermarket. You can add
items to it one by one. Similarly, in Python, you can use the
append() method to add elements to a list one at a time.
shopping_cart = []
shopping_cart.append("apples")
shopping_cart.append("bananas")
print(shopping_cart)
```

Output:

['apples', 'bananas']

```
#Example-2
#Analogy: Building a tower by adding one brick at a time. You
append bricks to the top of the tower just like you append
elements to the end of a list in Python.
tower = []
tower.append("brick1")
tower.append("brick2")
print(tower)
```

Output:

['brick1', 'brick2']

```
#Example-3
#Analogy: Creating a music playlist by adding songs one by one.
Each song is appended to the end of the playlist, just like using
append() for a list.
playlist = []
playlist.append("song1")
playlist.append("song2")
print(playlist)
```

Output:

['song1', 'song2']

```
#Example-4
#Analogy: Think of a bookshelf where you add books to the end.
Similarly, in Python, you use append() to add items to the end of
a list.
bookshelf = []
bookshelf.append("book1")
bookshelf.append("book2")
print(bookshelf)
```

Output:

['book1', 'book2']

```
#Example-5
#Analogy: Managing an inventory system where you add items to the
available stock one at a time. The append() method in Python can
be compared to adding items to the inventory.
inventory = []
inventory.append("item1")
inventory.append("item2")
print(inventory)
```

Output:

['item1', 'item2']

Insert method

- The insert() method allows you to add an element at a specified index in the list.
- It also modifies the original list.

Code:

```
#Example-1
#Analogy: Think of a list as a recipe, and you want to insert a
new ingredient (element) at a specific step. You use the insert()
method to add it at the desired location in the recipe.
recipe = ["flour", "sugar", "eggs", "milk"]
recipe.insert(2, "baking powder")
recipe
```

Output:

['flour', 'sugar', 'baking powder', 'eggs', 'milk']

```
#Example-2
#Analogy: Imagine a bus (list) where passengers (elements) are
seated in specific seats. You can use the insert() method to add
a new passenger to a particular seat (index).
bus_seats = ["Alice", "Bob", "Charlie"]
bus_seats.insert(1, "David")
bus_seats
```

Output:

['Alice', 'David', 'Bob', 'Charlie']

```
#Example-3
#Analogy: Think of a bookshelf (list) with books (elements) on
each shelf. You can use insert() to place a new book on a
specific shelf (index) on the bookshelf.
bookshelf = ["Science", "Fiction", "Mystery"]
bookshelf.insert(0, "Biography")
bookshelf
```

Output:

['Biography', 'Science', 'Fiction', 'Mystery']

```
#Example-4
#Analogy: Consider a deck of cards (list), and you want to insert
a new card at a particular position in the deck using the
insert() method.
deck = ["Ace", "2", "3", "4", "5"]
deck.insert(2, "King of Spades")
deck
```

Output:

['Ace', '2', 'King of Spades', '3', '4', '5']

```
#Example-5
#Analogy: Visualize a salad (list) with various ingredients
(elements). You can use the insert() method to include a new
ingredient at a specific location in the salad.
salad = ["Lettuce", "Tomatoes", "Cucumbers"]
salad.insert(1, "Bell Peppers")
salad
```

Output:

['Lettuce', 'Bell Peppers', 'Tomatoes', 'Cucumbers']

Extend method()

- The extend() method is used to append elements from another iterable (e.g., a list or tuple) to the end of the list.
- It modifies the original list by adding the new elements.

```
#Example-1
#Analogy: Imagine you have a shopping list, and your friend has
another list. You want to combine both lists to make a single,
comprehensive list before going to the store.
my_list = ["apples", "bananas"]
friend_list = ["milk", "bread"]
my_list.extend(friend_list)
my_list
```

Output:

['apples', 'bananas', 'milk', 'bread']

```
#Example-2
#Analogy: If you have a notebook with some pages filled, and you
receive more pages separately, you can extend your notebook by
adding those additional pages to the end.
notebook = ["Page 1", "Page 2"]
additional_pages = ["Page 3", "Page 4"]
notebook.extend(additional_pages)
notebook
```

Output:

['Page 1', 'Page 2', 'Page 3', 'Page 4']

```
#Example-3
#Analogy: In data analysis, you may have different reports or
datasets, and you want to combine them into a single dataset for
analysis.
report1_data = [10, 20, 30]
report2_data = [40, 50, 60]
combined_data = []
combined_data.extend(report1_data)
combined_data.extend(report2_data)
combined_data
```

Output:

[10, 20, 30, 40, 50, 60]

```
#Example-4
#Analogy: Imagine you have several bookshelves with books, and
you want to create a collection by extending it with books from
other shelves.
my_books = ["Book A", "Book B"]
additional_shelf = ["Book C", "Book D"]
my_books.extend(additional_shelf)
my_books
```

Output:

['Book A', 'Book B', 'Book C', 'Book D']

```
#Example-5
#Analogy: When cooking, you may have some ingredients in one bowl
and others in a different bowl. To make a complete recipe, you
extend one bowl with the ingredients from the other.
mixing_bowl1 = ["Flour", "Sugar"]
mixing_bowl2 = ["Eggs", "Milk"]
mixing_bowl1.extend(mixing_bowl2)
mixing_bowl1
```

Output:

['Flour', 'Sugar', 'Eggs', 'Milk']

List Operations:

Concatenation operation:

- You can concatenate two or more lists using the + operator.
- This creates a new list without modifying the original lists.

Code:

```
#Example-1
#Analogy: Imagine you have two lists of ingredients for different
recipes, and you want to create a new list containing all the
ingredients from both recipes.
recipe1_ingredients = ["flour", "sugar", "eggs"]
recipe2_ingredients = ["milk", "butter", "chocolate"]
combined_ingredients = recipe1_ingredients + recipe2_ingredients
combined_ingredients
```

Output:

['flour', 'sugar', 'eggs', 'milk', 'butter', 'chocolate']

```
#Example-2
#Analogy: Consider two separate lists of books available in two
different library branches. You can concatenate them to create a
unified catalog.
library_branch1_books = ["Science Fiction", "Mystery",
"Biography"]
library_branch2_books = ["History", "Cooking", "Self-Help"]
combined_catalog = library_branch1_books + library_branch2_books
combined_catalog
```

Output:

['Science Fiction', 'Mystery', 'Biography', 'History', 'Cooking', 'Self-Help']

```
#Example-3
#Analogy: When shopping online, you can combine items from
different carts into a single order before making a purchase.
cart1 = ["item1", "item2", "item3"]
cart2 = ["item4", "item5"]
combined_cart = cart1 + cart2
combined_cart
```

Output:

['item1', 'item2', 'item3', 'item4', 'item5']

```
#Example-4
#Analogy: Imagine two different attendance lists for a class, one
for the morning session and another for the afternoon session.
You can concatenate them to get the full-day attendance.
morning_session = ["student1", "student2", "student3"]
afternoon_session = ["student4", "student5"]
full_day_attendance = morning_session + afternoon_session
full_day_attendance
```

Output:

['student1', 'student2', 'student3', 'student4', 'student5']

```
#Example-5
#Analogy: You have two playlists of your favorite songs, one for
pop music and another for rock music. You want to create a new
playlist that combines both genres.
pop_playlist = ["song1", "song2", "song3"]
rock_playlist = ["song4", "song5"]
combined_playlist = pop_playlist + rock_playlist
combined_playlist
```

Output:

['song1', 'song2', 'song3', 'song4', 'song5']

Repetition Operation:

- You can repeat a list by using the * operator.
- This creates a new list with repeated elements without modifying the original list.

Code:

```
#Example-1
#Analogy: Imagine you want to print a pattern of stars or any
other character. You can use repetition to create the pattern.
pattern = "*" * 5
print(pattern)
```

Output:

*****

```
#Example-2
#Analogy: Suppose you want to create a string with a repeated
sequence of characters, like a border for a text box.
border = "-" * 20
print(border)
```

Output:

```
#Example-3
#Analogy: You can create a list with repeated elements, which is
useful for initializing data structures or creating sequences.
zeros = [0] * 10
print(zeros)
```

Output:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

```
#Example-4
#Analogy: You can generate a sequence of numbers by using
repetition to create increments or decrements.
sequence = list(range(1, 6)) * 3
print(sequence)
```

Output:

[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

```
#Example-5
#Analogy: Repetition can be used to send repeated messages, like
in messaging or chat applications.
message = "Reminder: Your appointment is tomorrow." + "\n"
reminders = message * 3
print(reminders)
```

Output:

Reminder: Your appointment is tomorrow.
Reminder: Your appointment is tomorrow.
Reminder: Your appointment is tomorrow.

Membership testing (in and not in):

- Python allows you to check if an element is present in a list using the in and not in operators.
- These operators return a Boolean value, indicating whether the element is found in the list or not.

Code:

```
#Example-1
#Analogy: Imagine your shopping list as a Python list. You can
use in to check if a specific item is on the list, or not in to
see if it's missing.
shopping_list = ["apples", "bananas", "milk", "bread"]
item = "bananas"

if item in shopping_list:
    print(f"{item} is on the shopping list.")
else:
    print(f"{item} is not on the shopping list.")
```

Output:

bananas is on the shopping list.

```
#Example-2
#Analogy: Consider a library catalog as a Python list. You can
use in to see if a book is available for borrowing or not in to
check if it's checked out.
library_catalog = ["Python Programming", "Machine Learning",
"Data Science"]
book = "Machine Learning"

if book in library_catalog:
    print(f"{book} is available for borrowing.")
else:
    print(f"{book} is checked out or not in the catalog.")
```

Output:

Machine Learning is available for borrowing.

```
#Example-3
#Analogy: Think of a list of event attendees. You can use in to
check if a person is attending or not in to see if they haven't
registered.
event_attendees = ["Alice", "Bob", "Charlie", "Eve"]
person = "Charlie"

if person in event_attendees:
    print(f"{person} is attending the event.")
else:

print(f"{person} hasn't registered for the event.")
```

Output:

Charlie is attending the event.

```
#Example-4
#Analogy: When managing a list of approved email addresses, you
can use in to validate if an email is in the approved list or not
in to check if it's unauthorized.
approved_emails = ["user1@example.com", "user2@example.com",
"user3@example.com"]
email = "user2@example.com"

if email in approved_emails:
    print(f"{email} is an approved email address.")
else:
    print(f"{email} is not authorized to access the system.")
```

Output:

user2@example.com is an approved email address.

```
#Example-5
#Analogy: In a warehouse, you can use in to check if a specific
product is in stock or not in to determine if it's out of stock.
warehouse_inventory = ["Laptops", "Smartphones", "Tablets"]
product = "Laptops"

if product in warehouse_inventory:
    print(f"{product} is in stock.")
else:
    print(f"{product} is currently out of stock.")
```

Output:

Laptops is in stock.

List Slicing and Copying:

Slicing to create sublists:

- Slicing is a way to create sublists from a larger list by specifying a range of indices.
- It doesn't modify the original list.

Code:

```
#Example-1
#Analogy: Think of a pizza as a list, and slicing allows you to
take a slice of pizza.
pizza = ["cheese", "pepperoni", "vegetarian", "mushroom"]
slice_of_pizza = pizza[1:3]
slice_of_pizza
```

Output:

['pepperoni', 'vegetarian']

```
#Example-2
#Analogy: Imagine a book with numbered pages. Slicing is like
selecting specific pages from the book.
book = list(range(1, 101))  # Pages 1 to 100
selected_pages = book[20:30]
selected_pages
```

Output:

['song2', 'song3', 'song4']

```
#Example-4
#Analogy: Consider the aisles in a supermarket as a list. Slicing
allows you to select items from specific aisles.
supermarket = ["produce", "dairy", "meat", "canned goods",
"bakery"]
shopping_list = supermarket[2:4]
shopping_list
```

Output:

['meat', 'canned goods']

```
#Example-5
#Analogy: Time can be thought of as a list of hours or days.
Slicing lets you pick a range of time periods.
hours_of_day = list(range(0, 24))
work_hours = hours_of_day[9:17]
work_hours
```

Output:

[9, 10, 11, 12, 13, 14, 15, 16]

Copying lists (shallow copy vs. deep copy):

- Creating a shallow copy of a list creates a new list, but the elements themselves are still references to the original elements.
- Creating a deep copy creates a completely independent duplicate of the list and its elements.

Code:

```
#Exapmle-1
#Analogy: Think of a shallow copy like making a photocopy of a
book. The pages are the same as the original book, but if you
highlight or make notes on a page in the copy, it won't affect
the original book.
import copy

original_book = ["Page 1", "Page 2", "Page 3"]
photocopy = copy.copy(original_book)
photocopy.append("Page 4")
photocopy
```

Output:

['Page 1', 'Page 2', 'Page 3', 'Page 4']

```
#Example-2
#Analogy: A deep copy is like creating an entirely new painting
that looks identical to the original. If you make changes to the
copy, it doesn't affect the original artwork.
import copy

original_painting = {"color": "blue", "size": "large"}
duplicate_painting = copy.deepcopy(original_painting)
duplicate_painting["color"] = "red"
duplicate_painting
```

Output:

{'color': 'red', 'size': 'large'}

```
#Example-3
#Analogy: Think of a shallow copy as sharing an address book with
someone. Both you and the other person have access to the same
list of addresses. If they add a new address, you can see it too.
import copy

your_address_book = ["Alice", "Bob", "Charlie"]
friend_address_book = your_address_book
friend_address_book.append("David")
friend_address_book
```

Output:

['Alice', 'Bob', 'Charlie', 'David']

```
#Example-4
#Analogy: When you clone a pet, you get an entirely separate
animal that looks and acts like the original. Any changes made to
the clone won't affect the original pet.
import copy

original_pet = {"name": "Fido", "species": "Dog"}
cloned_pet = copy.deepcopy(original_pet)
cloned_pet["name"] = "Buddy"
cloned_pet
```

Output:

{'name': 'Buddy', 'species': 'Dog'}

```
#Example-5
#Analogy: Consider a shallow copy like sharing digital photos
with a friend. You both have access to the same set of photos. If
your friend deletes a photo, it's also gone from your collection.
import copy

your_photos = ["Photo1.jpg", "Photo2.jpg", "Photo3.jpg"]
friend_photos = your_photos
del friend_photos[1]
friend_photos
```

Output:

['Photo1.jpg', 'Photo3.jpg']

List Sorting

Sorting Lists in Python

- Python provides several methods for sorting lists, the most common being the sort() method and the sorted() function.
- These methods arrange the elements of a list in ascending order.
- You can customize the sorting behavior by using key functions.
- For instance, to sort a list of strings by their lengths, you can use the len function as the key.
- To sort a list in descending order, you can use the reverse parameter or the 'reverse()' method.

Code:

```
#Example-1
#Analogy: Imagine you have a list of students and their exam
scores, and you want to sort them based on their scores.
students = [("Alice", 85), ("Bob", 92), ("Charlie", 78),
("David", 95)]
students.sort(key=lambda x: x[1])
print(students)
```

Output:

[('Charlie', 78), ('Alice', 85), ('Bob', 92), ('David', 95)]

```
#Example-2
#Analogy: Think of a library where books are in disarray. You can
sort the books on the shelves based on their titles in
alphabetical order.
books = ["Python Programming", "Algorithms", "Web Development",
"Data Science"]
sorted_books = sorted(books)
print(sorted_books)
```

Output:

['Algorithms', 'Data Science', 'Python Programming', 'Web Development']

```
#Example-3
#Analogy: You have a collection of movies, and you want to
arrange them in chronological order based on their release years.
movies = [("Inception", 2010), ("The Matrix", 1999), ("Avatar",
2009)]
movies.sort(key=lambda x: x[1])
print(movies)
```

Output:

[('The Matrix', 1999), ('Avatar', 2009), ('Inception', 2010)]

```
#Example-4
#Analogy: If you're managing an e-commerce website, you can sort
a list of products by their prices to help customers find the
best deals.
products = [
    {"name": "Laptop", "price": 800},
    {"name": "Smartphone", "price": 600},
    {"name": "Tablet", "price": 300},
]
products.sort(key=lambda x: x["price"])
print(products)
```

Output:

[{'name': 'Tablet', 'price': 300}, {'name': 'Smartphone', 'price': 600}, {'name': 'Laptop', 'price': 800}]

```
#Example-5
#Analogy: Consider a to-do list where each task has a priority
level. You can sort the list based on the priority to tackle
high-priority tasks first.
tasks = [
    {"task": "Buy groceries", "priority": 2},
    {"task": "Finish project", "priority": 1},
    {"task": "Read a book", "priority": 3},
]
tasks.sort(key=lambda x: x["priority"])
print(tasks)
```

Output:

[{'task': 'Finish project', 'priority': 1}, {'task': 'Buy groceries', 'priority': 2}, {'task': 'Read a book', 'priority': 3}]

List Sorting

- List indexing allows you to access individual elements of a list using their positions (indices).
- The first element has an index of 0, the second has an index of 1, and so on.
- Python also supports negative indexing, which allows you to access elements from the end of the list. -1 refers to the last element, -2 to the second-to-last, and so on.

Code:

```
#Example-1
#Analogy: Think of a list as a shelf in a supermarket, and each
item on the shelf has an index. You can grab an item from the
shelf by specifying its index.
supermarket_shelf = ["apples", "bananas", "oranges", "grapes"]
selected_item = supermarket_shelf[2]
print(selected_item)
```

Output:

oranges

```
#Example-2
#Analogy: Imagine a book as a list of pages, where each page has
a page number. You can open the book to a specific page by
referring to its page number, just like you access an element in
a list by its index.
book_pages = ["Introduction", "Chapter 1", "Chapter 2",
"Conclusion"]
current_page = book_pages[3]
print(current_page)
```

Output:

Conclusion

```
#Example-3
#Analogy: Consider a music playlist as a list of songs, and each
song has a track number. You can play a specific song by
selecting its track number.
playlist = ["Song 1", "Song 2", "Song 3", "Song 4"]
now_playing = playlist[1]
print(now_playing)
```

Output:

Song 2

```
#Example-4
#Analogy: In your computer's file system, files in a folder are
similar to elements in a list. You can open a particular file by
specifying its position or index in the list.
folder_files = ["document.txt", "image.png", "data.csv",
"presentation.ppt"]
open_file = folder_files[2]
print(open_file)
```

Output:

data.csv

```
#Example-5
#Analogy: Consider a recipe as a list of ingredients, where each
ingredient has a specific order. You can list the ingredients you
need by referring to their order in the list.
recipe_ingredients = ["Flour", "Eggs", "Milk", "Sugar"]
needed_ingredient = recipe_ingredients[0]
print(needed_ingredient)
```

Output:

Flour

Introduction to Stacks and Queues

- A stack is a linear data structure that operates based on the Last-In-First-Out (LIFO) principle. This means that the most recently added element is the first to be removed.
- Think of it as a collection of items stacked on top of each other, like a stack of plates, where you can only add or remove items from the top.
- A queue is another linear data structure that follows the First-In-First-Out (FIFO) principle. In a queue, the element that has been in the queue the longest is the first to be removed.
- An analogy for a queue is a line of people waiting for a service, such as a bus. The person who arrived first is the first to be served.

Real-World Application and Example

- **Stack**: To better understand stacks, imagine a physical stack of plates. You can only add a new plate on top of the existing ones, and when you want to take a plate, you start from the top. The last plate placed is the first one you can use.
- **Queue**: Visualize people waiting in line at a grocery store or bus stop. The person who arrived first is the first to be served. Similarly, in a queue data structure, the element that has been in the queue the longest is the first to be removed. This is essential in situations where order matters, such as scheduling tasks or processing requests.

Stack

What is Stack?

- A stack is a fundamental data structure in computer science and programming.
- It is a linear data structure that follows the Last-In-First-Out (LIFO) principle, which means that the last element added to the stack is the first one to be removed.
- Stacks are used to store and manage a collection of elements.

LIFO (Last-In-First-Out) Principle:

- Stacks adhere to the LIFO principle, which means that the last element added to the stack is the first one to be removed.
- This behavior is similar to a physical stack of items, like a stack of plates, books, or trays. The last one you put on top is the first one you can take off.

Code:

```
#Example-1
#Analogy: Think of a stack of plates. The last plate you place on
top of the stack is the first one you remove.
stack_of_plates = []

# Adding plates to the stack (push)
stack_of_plates.append("Plate 1")
stack_of_plates.append("Plate 2")
stack_of_plates.append("Plate 3")

# Removing plates from the stack (pop)
removed_plate = stack_of_plates.pop()  # Last plate added (Plate
3) is removed first
print("Removed plate:", removed_plate)
```

Output:

Removed plate: Plate 3

```
#Example-2
#Analogy: When you click the "back" button in a web browser, it
takes you to the last page you visited.
browsing_history = []

# Visiting web pages (push)
browsing_history.append("Homepage")
browsing_history.append("About Us")
browsing_history.append("Contact Us")

# Going back (pop)
last_page_visited = browsing_history.pop()
print("Last page visited: ",  last_page_visited)
```

Output:

Last page visited: Contact Us

```
#Example-3
#Analogy: When you undo an action in a text editor, the most
recent action is reversed first.
undo_stack = []

# Performing actions (push)
undo_stack.append("Typed 'Hello'")
undo_stack.append("Deleted last character")

# Undoing actions (pop)
last_action = undo_stack.pop()  # Reverses the last action
("Deleted last character" undone)\
print("last action: ", last_action)
```

Output:

last action: Deleted last character

```
#Example-4
#Analogy: In programming, the call stack manages function calls.
The last function called is the first one to return.
def function_a():
    print("Function A")

def function_b():
    print("Function B")

# Function calls (push)
function_a()
function_b()
```

Output:

Function A
Function B

```
#Example-5
#Analogy: When dealing with a deck of cards, the last card placed
on the deck is the first one you pick.
deck_of_cards = ["Ace", "2", "3", "4", "5", "6", "7", "8", "9",
"10", "Jack", "Queen", "King"]

# Taking the top card (pop)
top_card = deck_of_cards.pop()
print(top_card)
```

Output:

King

Push and Pop Operations:

- **Push:** The "push" operation is used to add an element to the top of the stack. When you push an element onto the stack, it becomes the new top element.
- **Pop**: The "pop" operation is used to remove the top element from the stack. After a pop operation, the element just below the top becomes the new top element.

Code:

```
#Example-1
#Real-World Analogy: Think of a stack of plates. You add plates
to the top and remove them from the top, just like in a stack
data structure.
stack_of_plates = []  # Creating an empty stack
stack_of_plates.append("Plate 1")
stack_of_plates.append("Plate 2")
stack_of_plates.append("Plate 3")

popped_plate = stack_of_plates.pop()
print("Popped Plate:", popped_plate)
```

Output:

Popped Plate: Plate 3

```
#Example-2
#Analogy: When you visit a webpage, it's like pushing that
webpage onto a stack. When you click the "Back" button, it's like
popping the previous webpage from the stack.
history_stack = []  # Create a history stack
history_stack.append("Homepage")
history_stack.append("Page 1")
previous_page = history_stack.pop()
print("Previous Page:", previous_page)
```

Output:

Previous Page: Page 1

```
#Example-3
#Analogy: When a function is called, it's like pushing that
function onto the call stack. When the function returns, it's
like popping the function from the call stack.
def foo():
    print("Inside foo")

def bag():
    print("Inside bag")
    foo()  # Call the foo function

bag()  # Call the bar function, pushing functions onto the call
stack
```

Output:

Inside bag
Inside foo

```
#Example-4
#Analogy: Each edit action, like typing a letter, is like pushing
that edit onto the undo stack. When you press "Undo," it's like
popping the last edit to revert it.
undo_stack = []  # Create an undo stack
text = "Hello, World!"
undo_stack.append(text)  # Push the initial text
text = "Hello, World! AI"  # Make an edit
undo_stack.append(text)  # Push the edited text
text = undo_stack.pop()  # Pop the edit to undo it
print("Undo:", text)
```

Output:

Undo: Hello, World! AI

```
#Example-5
#Analogy: When a customer arrives, it's like enqueueing them in a
queue. When the cashier serves a customer, it's like dequeuing
the next customer from the front of the queue.
from collections import deque

checkout_queue = deque()  # Create a checkout queue
checkout_queue.append("Customer 1")  # Enqueue "Customer 1"
checkout_queue.append("Customer 2")  # Enqueue "Customer 2"
served_customer = checkout_queue.popleft()  # Dequeue the next
customer
print("Served:", served_customer)
```

Output:

Served: Customer 1

Queue:

- A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. This means that the first element added to the queue is the first one to be removed.
- Queues are used to manage and process items in an orderly, sequential manner.

Properties of Queues:

- **FIFO Principle:** The most fundamental property of a queue is its adherence to the FIFO principle. This ensures that the item that has been in the queue the longest is the first to be processed.
- **Two Ends:** A queue typically has two ends: the front and the rear. New elements are enqueued (added) at the rear, and elements are dequeued (removed) from the front.
- **Sequential Processing:** Queues are designed for sequential processing. Items are processed one after the other in the order in which they were added to the queue.

Code:

```
#Example-1
#Analogy: In a supermarket, customers join a queue at the
checkout counter. The first customer to arrive is the first to be
served.
from collections import deque

# Create a queue using a deque
checkout_queue = deque()

# Customers join the queue
checkout_queue.append("Customer 1")
checkout_queue.append("Customer 2")
checkout_queue.append("Customer 3")

# Serve customers in the order they arrived
while checkout_queue:
    customer = checkout_queue.popleft()
    print(f"Serving {customer}")
```

Output:

Serving Customer 1
Serving Customer 2
Serving Customer 3

```
#Example-2
#Analogy: In a printer, print jobs are queued up, and the first
print job submitted is the first to be printed.
from queue import Queue

# Create a print job queue
print_queue = Queue()

# Add print jobs to the queue
print_queue.put("Print Job 1")
print_queue.put("Print Job 2")
print_queue.put("Print Job 3")
# Print jobs in the order they were submitted
while not print_queue.empty():
    print_job = print_queue.get()
    print(f"Printing: {print_job}")
```

Output:

Printing: Print Job 1
Printing: Print Job 2
Printing: Print Job 3

```
#Example-3
#Analogy: In an operating system, tasks are added to a queue for
execution, and they are processed in the order they were
scheduled.
from queue import Queue

# Create a task scheduling queue
task_queue = Queue()

# Schedule tasks to be executed
task_queue.put("Task 1")
task_queue.put("Task 2")
task_queue.put("Task 3")

# Execute tasks in the order they were scheduled
while not task_queue.empty():
    task = task_queue.get()
    print(f"Executing task: {task}")
```

Output:

Executing task: Task 1
Executing task: Task 2
Executing task: Task 3

```
#Example-4
#Analogy: In a call center, incoming customer service requests
are handled in the order they are received.
from queue import Queue

# Create a call center queue
call_queue = Queue()
# Handle incoming customer calls
call_queue.put("Customer 1")
call_queue.put("Customer 2")
call_queue.put("Customer 3")

# Assist customers in the order they called
while not call_queue.empty():
    customer = call_queue.get()
    print(f"Assisting customer: {customer}")
```

Output:

Assisting customer: Customer 1
Assisting customer: Customer 2
Assisting customer: Customer 3

```
#Example-5
#Analogy:In distributed systems, messages are placed in a queue
to ensure reliable and ordered delivery between components.
#Note: In this example, we'll use the queue.Queue class from
Python's standard library to represent a message queue.
from queue import Queue

# Create a message queue
message_queue = Queue()

# Send messages to the queue
message_queue.put("Message 1")
message_queue.put("Message 2")
message_queue.put("Message 3")

# Retrieve and process messages in order
while not message_queue.empty():
    message = message_queue.get()
    print(f"Processing message: {message}")
```

Output:

Processing message: Message 1
Processing message: Message 2
Processing message: Message 3

Introduction List Comprehensions

- A list comprehension is a concise and elegant way to create new lists by applying an expression to each item in an existing iterable (e.g., a list, tuple, or range).
- It's a fundamental feature of Python that allows you to generate lists with minimal code.
- List comprehensions are used to simplify the process of creating new lists from existing ones.
- They are particularly handy for transforming, filtering, or processing data in a clear and readable manner. List comprehensions make your code more efficient and expressive.

List Comprehensions Syntax

- List comprehensions are known for their compact and readable syntax.
- They replace multiple lines of code with a single line, making your code more elegant and easier to understand.
- List comprehensions are enclosed in square brackets, which signal the creation of a new list.

Simple List Comprehensions

- The list comprehension achieves the same result as the for loop but in a more concise and readable way.

```
#Example-1
#Analogy: Imagine you have a list of prices, and you want to
double each price to calculate the new prices.
prices = [10, 20, 30, 40, 50]
doubled_prices = [price * 2 for price in prices]
doubled_prices
```

Output:

[20, 40, 60, 80, 100]

```
#Example-2
#Analogy: Suppose you have a list of names, and you want to
capitalize the first letter of each name.
names = ['arun', 'bob', 'chals', 'dave']
capitalized_names = [name.capitalize() for name in names]
capitalized_names
```

Output:

['Arun', 'Bob', 'Chals', 'Dave']

```
#Example-3
#Analogy: If you have a list of numbers and you want to calculate
the square of each number.
numbers = [1, 2, 3, 4, 5]
squares = [num ** 2 for num in numbers]
squares
```

Output:

[1, 4, 9, 16, 25]

```
#Example-4
#Analogy: Imagine you have a list of numbers, and you want to
create a new list containing only the odd numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
odd_numbers = [num for num in numbers if num % 2 =≠ 0]
odd_numbers
```

Output:

[1, 3, 5, 7, 9]

```
#Example-5
#Analogy: Suppose you have a list of file names, and you want to
extract the file extensions from them.
file_names = ['document.txt', 'image.jpg', 'presentation.pptx',
'code.py']
extensions = [file.split('.')[-1] for file in file_names]
extensions
```

Output:

['txt', 'jpg', 'pptx', 'py']

Conditional List Comprehensions

- Conditional list comprehensions allow you to filter and transform data within a list comprehension based on specified conditions.

Code:

```
#Example-1
#Analogy: Selecting students who passed an exam.
grades = {'Ali': 85, 'Bob': 60, 'Chals': 75, 'Dravid': 90}
passed_students = [student for student, grade in grades.items()
if grade ⩾ 70]
Passed_students
```

Output:

['Ali', 'Chals', 'Dravid']

```
#Example-2
#Analogy: Grouping products in a store by category.
products = [{'name': 'Laptop', 'category': 'Electronics'},
           {'name': 'Shirt', 'category': 'Apparel'},
           {'name': 'Toothbrush', 'category': 'Personal Care'}]
electronics = [product['name'] for product in products if
product['category'] == 'Electronics']
electronics
```

Output:

['Laptop']

```
#Example-3
#Analogy: Collecting email addresses from a particular domain.
email_addresses = ['user1@example.com', 'user2@gmail.com',
'user3@example.com', 'user4@yahoo.com']
example_emails = [email for email in email_addresses if
email.endswith('@example.com')]
example_emails
```

Output:

['user1@example.com', 'user3@example.com']

```
#Example-4
#Analogy: Sorting numbers into even and odd categories.
numbers = [12, 7, 23, 6, 14, 8]
even_numbers = [num for num in numbers if num % 2 == 0]
odd_numbers = [num for num in numbers if num % 2 ≠ 0]
even_numbers
```

Output:

[12, 6, 14, 8]

```
#Example-5
#Analogy: Selecting news articles that match specific keywords.
news_articles = ['Tech Giant' ,'New Product Released', 'Economy
Grows in Q3', 'Python Conference Announced']
keyword = 'Python'
relevant_articles = [article for article in news_articles if
keyword in article]
relevant_articles
```

Output:

['Python Conference Announced']

Nested List Comprehensions

- Nested list comprehensions allow you to create lists of lists and perform operations on nested data structures.

Code:

```
#Example-1
#Analogy: Imagine you have a list of lists representing two
matrices, and you want to perform matrix multiplication.
matrix_A = [[1, 2], [3, 4]]
matrix_B = [[5, 6], [7, 8]]

result = [[sum(a * b for a, b in zip(row_A, col_B)) for col_B in
zip(*matrix_B)] for row_A in matrix_A]
result
```

Output:

[[19, 22], [43, 50]]

```
#Example-2
#Analogy: Suppose you have a list of (x, y) coordinates, and you
want to translate each point by a given vector.
points = [(1, 2), (3, 4), (5, 6)]
translation_vector = (10, 10)

translated_points = [(x + dx, y + dy) for (x, y), (dx, dy) in
zip(points, [translation_vector] * len(points))]
translated_points
```

Output:

[(11, 12), (13, 14), (15, 16)]

```
#Example-3
#Analogy: You have a list of lists representing different
categories, and you want to create a flat list of all items.
categories = [['apple', 'banana', 'cherry'], ['dog', 'elephant'],
['car', 'bus', 'bike']]

items = [item for sublist in categories for item in sublist]
items
```

Output:

['apple', 'banana', 'cherry', 'dog', 'elephant', 'car', 'bus', 'bike']

```
#Example-4
#Analogy: You have a list of lists representing rows and columns
of CSV data, and you want to extract a specific column.
csv_data = [
    ['Name', 'Age', 'City'],
    ['Arun', '25', 'New York'],
    ['Bob', '30', 'Los Angeles'],
    ['Chals', '22', 'Chicago']
]
# Extract the 'Name' column
names = [row[0] for row in csv_data[1:]]
names
```

Output:

['Arun', 'Bob', 'Chals']

```
#Example-5
#Analogy: You have a list of lists representing courses taken by
students, and you want to create a flat list of courses taken by
seniors (students with more than 3 years of experience)
students = [
    ['Alice', 'Physics', 2],
    ['Bob', 'Math', 4],
    ['Charlie', 'Physics', 3],
    ['David', 'Chemistry', 5]
]

senior_courses = [course for student in students if student[2] >
3 for course in [student[1]]]
senior_courses
```

Output:

['Math', 'Chemistry']

Advantages of List Comprehensions

- A list comprehension is a concise way to create lists in Python. Here's a more detailed explanation of the advantages of list comprehension:
  - **Readability**: List comprehensions are often more readable and intuitive than traditional for loops, especially for simple operations. They express the intended operation directly.
  - **Conciseness:** List comprehensions allow you to achieve the same result with fewer lines of code. This reduces code clutter and makes your code more compact.
  - **Performance:** List comprehensions are generally faster than traditional for loops because they are optimized for performance by Python's interpreter.
  - **Expressiveness:** List comprehensions promote a more functional style of programming in Python, allowing you to map, filter, and transform data in a clear and expressive manner.
  - **Reduced Bugs:** With list comprehensions, there are fewer chances for off-by-one errors or other common loop-related bugs, as the structure is more straightforward.

Tuples

Introduction to Tuples

- Tuples are ordered collections of elements, often heterogeneous, enclosed within parentheses.
- They are immutable, meaning once created, their elements cannot be changed.
- Tuples are created using parentheses '()' and separating elements with commas ','.

```
# Example Tuple.
tuplee = ('pw','skills',1,2,'pwioi')
print(tuplee)
```

Output:

('pw', 'skills', 1, 2, 'pwioi')

Accessing and Slicing Tuples

- **Indexing**: Elements in a tuple are accessed using their index positions, starting from 0. Example: my_tuple[0] accesses the first element.
- **Slicing:** Sections of tuples can be retrieved using slicing operations, denoted by [start:stop:step]. Example: my_tuple[1:3] retrieves elements at index 1 and 2.
- **Nested Tuples:** Tuples can contain other tuples as elements, allowing for nested structures. Example: nested_tuple = ((1, 2), ('a', 'b'), (True, False))

Example

```
#Example-1
#Analogy: Accessing a specific page in a book
book_pages = ('Introduction', 'Chapter 1', 'Chapter 2', 'Chapter
3', 'Conclusion')

# Accessing a specific page using indexing
current_page = book_pages[2]
print(f"Currently reading: {current_page}")
```

Output:

Currently reading: Chapter 2

```
#Example-2
#Analogy: Selecting ingredients from a recipe list
recipe_ingredients = ('Flour', 'Sugar', 'Eggs', 'Milk',
'Vanilla')

# Accessing multiple ingredients using slicing
required_ingredients = recipe_ingredients[1:4]
print(f"Ingredients needed: {required_ingredients}")
```

Output:

Ingredients needed: ('Sugar', 'Eggs', 'Milk')

```
#Example-4
#Analogy: Retrieving specific tools from a toolbox
toolbox = ('Hammer', 'Screwdriver', 'Wrench', 'Pliers', 'Tape
Measure')

# Accessing a specific tool using indexing
tool_needed = toolbox[3]  # Accessing 'Pliers'
print(f"Tool needed: {tool_needed}")

# Accessing a selection of tools using slicing
essential_tools = toolbox[1:4]  # Accessing 'Screwdriver',
'Wrench', 'Pliers'
print(f"Essential tools: {essential_tools}")
```

Output:

Tool needed: Pliers
Essential tools: ('Screwdriver', 'Wrench', 'Pliers')

```
#Example-5
#Analogy: Obtaining specific songs from a playlist
playlist = ('Song1', 'Song2', 'Song3', 'Song4', 'Song5')

# Accessing a particular song using indexing
current_song = playlist[2]
print(f"Currently playing: {current_song}")

# Accessing a subset of songs using slicing
favourite_songs = playlist[1:4]
print(f"Favourite songs: {favourite_songs}")
```

Output:

Currently playing: Song3
Favourite songs: ('Song2', 'Song3', 'Song4')

Tuple Methods and Operations

- count(): Counts the occurrences of a specified element in the tuple.
- index(): Returns the index of the first occurrence of a specified element.
- Concatenation (+): Combining two tuples to form a new tuple.
- Repetition (*): Repeating a tuple's elements a specified number of times.

Code:

```
#Example-1
#Analogy: Inventory Tracking
# Tuple count() method analogy
inventory = ('Apple', 'Banana', 'Orange', 'Apple', 'Apple')
# Count the occurrences of 'Apple' in the inventory
apple_count = inventory.count('Apple')
print(f"Number of Apples in inventory: {apple_count}")
```

Output:

Number of Apples in inventory: 3

```
#Example-2
#Analogy: Student Grades
# Tuple index() method analogy
grades = ('A', 'B', 'C', 'A', 'B')
# Find the index of the first occurrence of 'A' in the grades
first_A_index = grades.index('A')
print(f"Index of first 'A' grade: {first_A_index}")
```

Output:

Index of first 'A' grade: 0

```
#Example-3
#Analogy: Monthly Expenses
# Tuple concatenation analogy
previous_month = (1000, 1200, 800)
current_month = (1100, 1300, 850)
# Combine expenses from both months
total_expenses = previous_month + current_month
print(f"Total expenses for both months: {total_expenses}")
```

Output:

Total expenses for both months: (1000, 1200, 800, 1100, 1300, 850)

```
#Example-4
#Analogy: Coordinates
# Tuple unpacking analogy
coordinates = (23.567, 45.678)
# Unpack the tuple into latitude and longitude variables
latitude, longitude = coordinates
print(f"Latitude: {latitude}, Longitude: {longitude}")
```

Output:

Latitude: 23.567, Longitude: 45.678

```
#Example-5
#Analogy: Menu Options
# Tuple repetition analogy
menu_options = ('Coffee', 'Tea')
# Duplicate menu options for a larger menu
extended_menu = menu_options * 2
print(f"Extended Menu: {extended_menu}")
```

Output:

Extended Menu: ('Coffee', 'Tea', 'Coffee', 'Tea')

Sets

Introduction

- Sets are collections of unique and unordered elements.
- They don't allow duplicate elements and are represented by curly braces {} in Python.
- Sets don’t maintain order, so elements aren’t indexed.
- They ensure each element is unique within the set.

Example code:

```
#Example
my_set = {1, 2, 3}
another_set = set([3, 4, 5])
print(my_set)
print(another_set)
```

Output:

{1, 2, 3}
{3, 4, 5}

Set Methods

- add(): Adds an element to the set
- remove(): Removes a specific element from the set. Raises an error if the element doesn’t exist.
- discard(): Removes a specific element from the set if it exists. Doesn't raise an error if the element is not present.
- clear(): Empties the set, making it empty.

Code:

```
#Example-1
#Analogy: Suppose you have two shopping carts with items, and you
want to find out what items are common in both carts.
cart1 = {'apple', 'banana', 'orange', 'grapes'}
cart2 = {'banana', 'grapes', 'watermelon', 'kiwi'}

# Finding common items in both carts using intersection
common_items = cart1.intersection(cart2)
print("Common items:", common_items)
```

Output:

Common items: {'grapes', 'banana'}

```
#Example-2
#Analogy: Consider two sets of courses to compare which courses
are available in both sets.
courses_offered = {'Math', 'Physics', 'Biology', 'Chemistry'}
my_courses = {'Biology', 'Chemistry', 'History', 'Geography'}

# Finding common courses using intersection
common_courses = courses_offered.intersection(my_courses)
print("Common courses:", common_courses)
```

Output:

Common courses: {'Chemistry', 'Biology'}

```
#Example-3
#Analogy: Let's say you have two sets of skills possessed by
employees and you want to know the unique skills each employee
has.
employee1_skills = {'Python', 'Java', 'SQL', 'C++'}
employee2_skills = {'Java', 'JavaScript', 'Python', 'HTML'}

# Finding unique skills for each employee using difference
unique_skills_employee1 =
employee1_skills.difference(employee2_skills)
unique_skills_employee2 =
employee2_skills.difference(employee1_skills)

print("Employee 1 unique skills:", unique_skills_employee1)
print("Employee 2 unique skills:", unique_skills_employee2)
```

Output:

Employee 1 unique skills: {'SQL', 'C++'}
Employee 2 unique skills: {'HTML', 'JavaScript'}

```
#Example-4
#Analogy: Imagine two sets representing friends on different
social media platforms and finding mutual friends.
facebook_friends = {'Alice', 'Bob', 'Charlie', 'David'}
twitter_friends = {'Alice', 'Charlie', 'Eve', 'Frank'}

# Finding mutual friends using intersection
mutual_friends = facebook_friends.intersection(twitter_friends)
print("Mutual friends:", mutual_friends)
```

Output:

Mutual friends: {'Alice', 'Charlie'}

```
#Example-5
#Analogy: Consider sets representing available and required
ingredients for a recipe.
available_ingredients = {'flour', 'sugar', 'eggs', 'milk'}
required_ingredients = {'sugar', 'butter', 'vanilla', 'eggs'}

# Finding missing ingredients using difference
missing_ingredients =
required_ingredients.difference(available_ingredients)
print("Missing ingredients:", missing_ingredients)
```

Output:

Missing ingredients: {'vanilla', 'butter'}

Set Operations

- **Union (|):** Combines elements from two sets, excluding duplicates.
- **Intersection (&):** Finds common elements between sets.
- **Difference (-):** Returns elements that are in the first set but not in the second.
- **Symmetric Difference (^):** Returns elements that are in either of the sets, but not in both.

Code:

```
#Example-1
#Analogy: Shared Interests among Friends
# Imagine friends with various interests
john_interests = {'hiking', 'reading', 'coding'}
alice_interests = {'coding', 'travelling', 'photography'}

# Finding common interests using set intersection
common_interests = john_interests & alice_interests
print("Common interests:", common_interests)
```

Output:

Common interests: {'coding'}

```
#Example-2
#Analogy: Course Enrollment Choices in a University
# Available courses for two students
student_A_courses = {'Math', 'Physics', 'Chemistry'}
student_B_courses = {'Physics', 'Biology', 'History'}

# Finding courses available for both students using set
intersection
common_courses = student_A_courses & student_B_courses
print("Courses available for both students:", common_courses)
```

Output:

Courses available for both students: {'Physics'}

```
#Example-3
#Analogy: Shopping List Comparison
# Shopping lists of two people
shopping_list_A = {'apples', 'bread', 'milk', 'eggs'}
shopping_list_B = {'bread', 'milk', 'yogurt', 'cheese'}

# Finding items available in both shopping lists using set
intersection
common_items = shopping_list_A & shopping_list_B
print("Common items in shopping lists:", common_items)
```

Output:

Common items in shopping lists: {'bread', 'milk'}

```
#Example-4
#Analogy: Employee Skill Proficiency
# Skills of employees in different departments
marketing_skills = {'analytics', 'social_media', 'copywriting'}
tech_skills = {'programming', 'data_analysis', 'analytics'}

# Finding skills common to both departments using set
intersection
common_skills = marketing_skills & tech_skills
print("Common skills between departments:", common_skills)
```

Output:

Common skills between departments: {'analytics'}

```
#Example-5
#Analogy: Overlapping Features in Products
# Features of two different software products
product_A_features = {'cloud_storage', 'encryption',
'file_sharing'}
product_B_features = {'file_sharing', 'version_control',
'encryption'}

# Finding common features in both products using set intersection
common_features = product_A_features & product_B_features
print("Common features in products:", common_features)
```

Output:

Common features in products: {'encryption', 'file_sharing'}

Dictionary

Introduction to dictionaries

- Dictionaries are a data structure in Python that store data as key-value pairs.
- They are unordered collections and are defined using curly braces {}. Keys are unique and immutable, while values can be of any data type.
- Keys are unique and immutable, while values can be of any data type.
- Dictionaries are created using {} and key-value pairs separated by colons.
- Elements are accessed by referencing their keys using square brackets [].

Advance Dictionary

- **Sorting Dictionaries:**
  - Dictionaries cannot be sorted directly, but they can be sorted based on keys or values using functions like sorted() or by converting them to lists first.
- **Nested Dictionaries:**
  - Dictionaries can contain other dictionaries as values, forming nested structures.
  - These nested dictionaries can be accessed and manipulated just like any other dictionary.
- **Removing Elements:**
  - pop(): Removes and returns the value associated with the specified key.
  - popitem(): Removes and returns the last inserted key-value pair as a tuple.

Code:

```
#Example-1
#Analogy: In a library, books are organized using a catalog
system where each book has a unique identification number and
related information.
# Dictionary representing a library catalog
library_catalog = {
    101: {'title': 'Python Crash Course', 'author': 'Eric
Matthes', 'genre': 'Programming'},
    102: {'title': 'The Alchemist', 'author': 'Paulo Coelho',
'genre': 'Fiction'},
    103: {'title': 'Principles', 'author': 'Ray Dalio', 'genre':
'Finance'}
}

# Accessing book information using the book ID
book_id = 102
print(f"Book '{library_catalog[book_id]['title']}' by
{library_catalog[book_id]['author']} is in the
{library_catalog[book_id]['genre']} genre.")
```

Output:

Book 'The Alchemist' by Paulo Coelho is in the Fiction genre.

```
#Example-2
#Analogy: In a school database, student records are stored with
unique IDs containing various details like name, grade, and
subjects.
# Dictionary representing a student database
student_database = {
    'S001': {'name': 'Alice', 'grade': 'A', 'subjects': ['Math',
'Science', 'English']},
    'S002': {'name': 'Bob', 'grade': 'B', 'subjects': ['History',
'Geography', 'Math']}
}

# Accessing student details using the student ID
student_id = 'S001'
print(f"{student_database[student_id]['name']} has a grade of
{student_database[student_id]['grade']}.")
print(f"{student_database[student_id]['name']} takes {',
'.join(student_database[student_id]['subjects'])} subjects.")
```

Output:

Alice has a grade of A.
Alice takes Math, Science, English subjects.

```
#Example-3
#Analogy: In a retail store, the inventory system contains
details of products, including their prices, quantities, and
categories.
# Dictionary representing a product inventory
product_inventory = {
    'P001': {'name': 'Laptop', 'price': 1200.00, 'quantity': 25,
'category': 'Electronics'},
    'P002': {'name': 'Backpack', 'price': 45.50, 'quantity': 50,
'category': 'Accessories'},
    'P003': {'name': 'Smartphone', 'price': 800.00, 'quantity':
30, 'category': 'Electronics'}
}
# Accessing product information using the product ID
product_id = 'P002'
print(f"Product '{product_inventory[product_id]['name']}' is in
the {product_inventory[product_id]['category']} category.")
print(f"The price of '{product_inventory[product_id]['name']}' is
${product_inventory[product_id]['price']}.")
```

Output:

Product 'Backpack' is in the Accessories category.
The price of 'Backpack' is $45.5.

```
#Example-4
#Analogy: In a company's HR system, employee details such as
name, department, and salary are stored for easy access.
# Dictionary representing employee records
employee_records = {
    'E001': {'name': 'John Doe', 'department': 'Engineering',
'salary': 75000},
    'E002': {'name': 'Jane Smith', 'department': 'Marketing',
'salary': 65000},
    'E003': {'name': 'David Lee', 'department': 'Finance',
'salary': 80000}
}

# Accessing employee details using the employee ID
employee_id = 'E001'
print(f"{employee_records[employee_id]['name']} works in the
{employee_records[employee_id]['department']} department.")
print(f"{employee_records[employee_id]['name']}'s salary is
${employee_records[employee_id]['salary']}.")
```

Output:

John Doe works in the Engineering department.
John Doe's salary is $75000.

```
#Example-5
#Analogy: In a restaurant, a menu consists of various dishes with
their respective prices and descriptions.
# Dictionary representing a restaurant menu
restaurant_menu = {
    'Dish1': {'name': 'Pasta Carbonara', 'price': 15.99,
'description': 'Creamy pasta with bacon and parmesan'},
    'Dish2': {'name': 'Chicken Caesar Salad', 'price': 12.50,
'description': 'Grilled chicken with romaine lettuce and Caesar
dressing'},
    'Dish3': {'name': 'Margherita Pizza', 'price': 14.00,
'description': 'Pizza topped with tomato, mozzarella, and basil'}
}

# Accessing dish details using the dish name
dish_name = 'Dish3'
print(f"'{restaurant_menu[dish_name]['name']}' costs
${restaurant_menu[dish_name]['price']}.")
print(f"Description: {restaurant_menu[dish_name]
['description']}")
```

Output:

'Margherita Pizza' costs $14.0.
Description: Pizza topped with tomato, mozzarella, and basil

Introduction to dictionary comprehension

- Dictionary comprehensions are a concise way to create dictionaries in Python by employing a compact and expressive syntax.
- They allow you to create dictionaries using a single line of code by iterating over an iterable (like lists, tuples, or other dictionaries) and generating key-value pairs.
- This construct is similar to list comprehensions but produces dictionaries instead of lists.
- In Python, the basic syntax of dictionary comprehension involves enclosing an expression within curly braces '{}', where each expression generates a key-value pair separated by a colon ':'.
- This is followed by an iterable and an iteration variable, often expressed within square brackets [] or parentheses ().
- The syntax typically follows this structure: {key: value for element in iterable}.

Benefits of Using Dictionary Comprehensions

- **Conciseness and readability:** They provide a more compact and readable way to create dictionaries compared to traditional methods like loops or for loops combined with if statements.
- **Expressiveness:** Dictionary comprehensions allow developers to express complex operations in a succinct manner, enhancing code clarity and maintainability.
- **Efficiency:** They often result in faster execution times and better performance, especially for smaller dictionaries.

Real-time Analogy

Code:

```
#Example-1
#Analogy: Suppose you have a list of students and their
corresponding grades, and you want to create a dictionary with
their names as keys and grades as values.
students = ['Arun', 'Bob', 'Charlie']
grades = [85, 92, 78]

# Creating a dictionary using dictionary comprehension
grade_dict = {student: grade for student, grade in zip(students,
grades)}
print(grade_dict)
```

Output:

{'Arun': 85, 'Bob': 92, 'Charlie': 78}

```
#Example-2
#Analogy: For a given sentence, you want to count the occurrences
of each character and create a dictionary with characters as keys
and their counts as values.
sentence = "Hello, Everyone Welcome to PW skills"

# Creating a character count dictionary using dictionary
comprehension
char_count = {char: sentence.count(char) for char in sentence if
char ≠ ' '}
print(char_count)
```

Output:

{'H': 1, 'e': 5, 'l': 5, 'o': 4, ',': 1, 'E': 1, 'v': 1, 'r': 1, 'y': 1, 'n': 1, 'W': 2, 'c': 1, 'm': 1, 't': 1, 'P': 1, 's': 2, 'k': 1, 'i': 1}

```
#Example-3
#Analogy: Given a dictionary of temperatures in Celsius, you want
to convert them to Fahrenheit using a formula and store the
results in a new dictionary.
celsius_temperatures = {'Monday': 25, 'Tuesday': 18, 'Wednesday':
30}

# Converting Celsius to Fahrenheit using dictionary comprehension
fahrenheit_temperatures = {day: (temp * 9/5) + 32 for day, temp
in celsius_temperatures.items()}
print(fahrenheit_temperatures)
```

Output:

{'Monday': 77.0, 'Tuesday': 64.4, 'Wednesday': 86.0}

```
#Example-4
#Analogy: Suppose you have a list of user IDs and corresponding
usernames, and you want to create a dictionary that maps each
username to its respective user ID.
user_ids = [101, 102, 103]
usernames = ['arun', 'ben', 'charls']

# Creating a dictionary mapping usernames to user IDs using
dictionary comprehension
user_id_mapping = {username: user_id for username, user_id in
zip(usernames, user_ids)}
print(user_id_mapping)
```

Output:

{'arun': 101, 'ben': 102, 'charls': 103}

```
#Example-5
#Analogy: Given a word, you want to count the occurrences of
vowels and create a dictionary with vowels as keys and their
counts as values.
word = 'encyclopedia'

# Counting vowels in a word using dictionary comprehension
vowel_count = {char: word.count(char) for char in word if
char.lower() in 'aeiou'}
print(vowel_count)
```

Output:

{'e': 2, 'o': 1, 'i': 1, 'a': 1}

Dictionary View Objects

- In Python, dictionary view objects provide dynamic, live views of a dictionary's contents. These views reflect changes made in the associated dictionary.
- They offer an interface to observe and interact with the keys, values, or key-value pairs within a dictionary without creating separate lists or sets.
- One crucial aspect of dictionary views is their dynamic nature. They are live representations of the original dictionary, meaning they reflect any changes made to the underlying dictionary in real time.
- If the dictionary changes (keys, values, or items are added, modified, or removed), these changes are immediately reflected in the view object.

Types of Dictionary Views

- **dict.keys():** This method returns a view object that displays all the keys present in the dictionary. It represents the keys as a set-like object, providing operations like membership tests (e.g., in operator) and iteration through keys.

- **dict.values()**: The values() method returns a view object that shows all the values present in the dictionary. Similar to dict.keys(), this view represents the values as a set-like object, allowing operations like iteration and membership tests.

- **dict.items():** The items() method returns a view object that displays all the key-value pairs present in the dictionary. Each item in the view is a tuple containing a key-value pair. This view is iterable and allows convenient access to both keys and values simultaneously.

Analogy Code:

```
#Example-1
#Imagine a library catalog with unique book IDs as keys. The
dict.keys() view can be likened to this catalog, providing a
listing of all the available book IDs.
# Library catalog (Dictionary representing books and their IDs)
library_books = {
    101: 'Introduction to Python',
    102: 'Data Structures and Algorithms',
    103: 'Machine Learning Basics'
}

# Get a view of book IDs
book_ids_view = library_books.keys()

# Iterating through book IDs
for book_id in book_ids_view:
    print(f"Book ID: {book_id}")
```

Output:

Book ID: 101
Book ID: 102
Book ID: 103

```
#Example-2
#Analogy: Suppose the library catalog has book titles stored as
values. The dict.values() view is analogous to a list of all book
titles available in the library.
# Get a view of book titles
book_titles_view = library_books.values()

# Iterating through book titles
for book_title in book_titles_view:
    print(f"Book Title: {book_title}")
```

Output:

Book Title: Introduction to Python
Book Title: Data Structures and Algorithms
Book Title: Machine Learning Basics

```
#Example-3
#Analogy: In the library catalog, pairs of book IDs and titles
are like the dict.items() view, providing a combined view of book
ID-title pairs.
# Get a view of book ID-title pairs
book_id_title_view = library_books.items()

# Iterating through book ID-title pairs
for book_id, book_title in book_id_title_view:
    print(f"Book ID: {book_id} - Book Title: {book_title}")
```

Output:

Book ID: 101 - Book Title: Introduction to Python
Book ID: 102 - Book Title: Data Structures and Algorithms
Book ID: 103 - Book Title: Machine Learning Basics

```
#Example-4
#Analogy: Recipe dictionary (Ingredients and their quantities)
recipe = {
    'flour': '2 cups',
    'sugar': '1 cup',
    'eggs': '2',
    'milk': '1/2 cup',
}

# Get a view of ingredients
ingredients_view = recipe.keys()

# Iterating through ingredients
for ingredient in ingredients_view:
    print(f"Ingredient: {ingredient}")
```

Output:

Ingredient: flour
Ingredient: sugar
Ingredient: eggs
Ingredient: milk

```
#Example-5
#Analogy: In the same recipe book scenario, the quantities of
ingredients stored as values correspond to the dict.values()
view, offering a glimpse of the amounts needed for different
recipes.
# Get a view of ingredient quantities
quantities_view = recipe.values()

# Iterating through ingredient quantities
for quantity in quantities_view:
    print(f"Quantity: {quantity}")
```

Output:

Quantity: 2 cups
Quantity: 1 cup
Quantity: 2
Quantity: 1/2 cup