Dictionary

Dictionary

In Python, a dictionary is a built-in data type that represents a collection of key-value pairs. It is also known as an associative array or a hash map in other programming languages. Dictionaries are unordered, mutable, and indexed by keys.

The key-value pairs in a dictionary are enclosed in curly braces {} and separated by commas. Each key is associated with a value using a colon (:). The keys in a dictionary must be unique and immutable objects, such as strings or numbers, while the values can be of any data type.

Here's an example of a dictionary in Python:

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}
```

In this example, `"name"`, `"age"`, and `"city"` are the keys, and `"Vasan"`, `25`, and `"Vellore"` are their corresponding values.

You can access the values in a dictionary by specifying the corresponding key in square brackets ([]):

```
print(mydict["name"])  # Output: Vasan
print(mydict["age"])   # Output: 25
```

Dictionaries are widely used for efficient data retrieval and storage when the data can be associated with unique keys. They are often used for tasks such as caching, indexing, and mapping data.

Dictionaries are mutable, so you can modify the values associated with keys:

```
mydict["age"] = 25
print(mydict["age"])  # Output: 25
```

If you try to access a key that doesn't exist in the dictionary, it will raise a `KeyError`. To avoid this, you can use the `get()` method, which allows you to specify a default value to return if the key is not found:

```
print(mydict.get("address", "Unknown"))  # Output: Unknown
```

You can also check if a key exists in a dictionary using the `in` keyword:

```
if "name" in mydict:
    print("Name exists in the dictionary")
```

Dictionaries offer various methods and operations to manipulate their contents, such as adding or removing key-value pairs, iterating over keys or values, merging dictionaries, and more.

Dictionaries are commonly used for tasks involving data mapping, lookup tables, and organizing data with descriptive labels. They provide an efficient way to retrieve values based on unique keys, making them a powerful tool in Python programming.

Access item in dictionary

To access an item (value) in a dictionary, you can use the key associated with that item. Here are a few methods to access items in a dictionary in Python:

1. Using square brackets: You can use square brackets `[]` along with the key to access the corresponding value.

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

print(mydict["name"])  # Output: Vasan
print(mydict["age"])   # Output: 25
print(mydict["city"])  # Output: Vellore
```

2. Using the `get()` method: The `get()` method allows you to access a value based on the key. It takes the key as an argument and returns the corresponding value. If the key doesn't exist, you can specify a default value to return.

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

print(mydict.get("name"))           # Output: Vasan
print(mydict.get("age"))            # Output: 25
print(mydict.get("address"))        # Output: None (key doesn't
exist)
print(mydict.get("address", ""))    # Output: "" (default value
specified)
```

3. Using the `keys()` and `values()` methods: The `keys()` method returns a list-like object containing all the keys in the dictionary, while the `values()` method returns a list-like object containing all the values. You can use these methods to iterate over the keys or values and access them individually.

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

 # Accessing keys
 for key in mydict.keys():
     print(key)  # Output: name, age, city

 # Accessing values
```

```
        for value in mydict.values():
           print(value)  # Output: Vasan, 25, Vellore
```

These methods provide different ways to access items in a dictionary based on your specific requirements. Choose the one that suits your needs best.

Change item in dictionary

To change an item (value) in a dictionary, you can use the key associated with that item and assign a new value to it. Here's how you can change an item in a dictionary in Python:

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

mydict["age"] = 26
print(mydict)  # Output: {'name': 'Vasan', 'age': 26, 'city': 'Vellore'}
```

In this example, the value associated with the key `"age"` is changed from `25` to `26`.

If the key you specify doesn't exist in the dictionary, a new key-value pair will be added instead of modifying an existing one. For example:

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

mydict["address"] = "123 Main St"
print(mydict)  # Output: {'name': 'Vasan', 'age': 25, 'city': 'Vellore',
'address': '123 Main St'}
```

In this case, the key `"address"` doesn't exist in the dictionary, so it's added with the corresponding value `"123 Main St"`.

By changing the value associated with a specific key, you can update the content of the dictionary. Remember that dictionaries are mutable, meaning you can modify their values after they are created.

Add item in dictionary

To add an item (key-value pair) to a dictionary in Python, you can assign a value to a new key or an existing key. Here's how you can add items to a dictionary:

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}
```

```
mydict["address"] = "123 Main St"
print(mydict)
```

Output:

```
{'name': 'Vasan', 'age': 25, 'city': 'Vellore', 'address': '123 Main St'}
```

In this example, the key `"address"` is added to the `mydict` dictionary with the corresponding value `"123 Main St"`. If the key already exists in the dictionary, assigning a new value to that key will update its value.

You can add multiple items to a dictionary at once by using the `update()` method. This method takes another dictionary or an iterable of key-value pairs as an argument and adds them to the original dictionary.

```
mydict = {
    "name": "Vasan",
    "age": 25
}

mydict.update({"city": "Vellore", "address": "123 Main St"})
print(mydict)
```

Output:

```
{'name': 'Vasan', 'age': 25, 'city': 'Vellore', 'address': '123 Main St'}
```

In this example, the `update()` method is used to add two key-value pairs (`"city": "Vellore"` and `"address": "123 Main St"`) to the `mydict` dictionary.

Adding items to a dictionary is a common operation when you need to expand the dictionary with new information or dynamically build its content based on your program's logic.

Remove items

To remove items from a dictionary in Python, you can use the `del` statement or the `pop()` method. Here's how you can remove items from a dictionary:

1. Using the `del` statement: You can use the `del` statement followed by the key to delete the corresponding item from the dictionary.

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

del mydict["age"]
print(mydict)  # Output: {'name': 'Vasan', 'city': 'Vellore'}
```

In this example, the `del` statement is used to remove the item with the key `"age"` from the `mydict` dictionary.

2. Using the `pop()` method: The `pop()` method allows you to remove an item from the dictionary and retrieve its value. It takes the key as an argument and returns the corresponding value. If the key doesn't exist, you can specify a default value to return.

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

removed_age = mydict.pop("age")
print(mydict)        # Output: {'name': 'Vasan', 'city': 'Vellore'}
print(removed_age)    # Output: 25
```

In this example, the `pop()` method is used to remove the item with the key `"age"` from the `mydict` dictionary and retrieve its value.

It's important to note that using `del` or `pop()` to remove a non-existent key from a dictionary will raise a `KeyError`. To avoid this, you can check the existence of the key using the `in` keyword or use the `pop()` method with a default value.

Removing items from a dictionary allows you to update its content or eliminate unnecessary data based on your program's requirements.

Copy Dictionary

To create a copy of a dictionary in Python, you can use the `copy()` method or the dictionary constructor. Here's how you can make a copy of a dictionary:

1. Using the `copy()` method: The `copy()` method creates a shallow copy of the dictionary, which means it creates a new dictionary with the same key-value pairs. However, if the values in the original dictionary are mutable objects (like lists or dictionaries), the new dictionary will still reference those objects.

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

new_dict = mydict.copy()
print(new_dict)  # Output: {'name': 'Vasan', 'age': 25, 'city': 'Vellore'}
```

In this example, the `copy()` method is used to create a new dictionary `new_dict` that is a copy of `mydict`.

2. Using the dictionary constructor: You can pass the original dictionary as an argument to the dictionary constructor `{}` to create a copy.

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

new_dict = dict(mydict)
print(new_dict)  # Output: {'name': 'Vasan', 'age': 25, 'city': 'Vellore'}
```

In this example, the `dict()` constructor is used with `mydict` as an argument to create a new dictionary `new_dict`.

Both methods create a new dictionary that is independent of the original dictionary, allowing you to modify one dictionary without affecting the other.

It's important to note that creating a copy of a dictionary using either method results in a shallow copy. If the values in the original dictionary are mutable objects and you modify them in the copied dictionary, the changes will be reflected in both dictionaries. If you want to create a deep copy, where the values are also copied and not referenced, you can use the `copy` module's `deepcopy()` function.

Loop in Dictionary

In Python, you can use loops to iterate over the elements of a dictionary. There are several ways to loop through a dictionary, depending on whether you want to access the keys, values, or both. Here are a few common methods for looping through a dictionary:

1. Looping through keys using `for` loop:

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

for key in mydict:
    print(key)  # Output: name, age, city
```

In this example, the `for` loop iterates over the keys of the `mydict` dictionary. You can access the corresponding values using the keys inside the loop, like `mydict[key]`.

2. Looping through keys and values using `.items()` method:

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

for key, value in mydict.items():
    print(key, value)  # Output: name Vasan, age 25, city Vellore
```

The `.items()` method returns a list-like object containing the key-value pairs of the dictionary. By using this method, you can loop through both the keys and values simultaneously.

3. Looping through values using `.values()` method:

```
mydict = {
    "name": "Vasan",
    "age": 25,
    "city": "Vellore"
}

for value in mydict.values():
    print(value)  # Output: Vasan, 25, Vellore
```

The `.values()` method returns a list-like object containing only the values of the dictionary. By iterating over these values, you can access and process them individually.

These are some common approaches for looping through a dictionary in Python. Choose the method that best suits your needs based on whether you want to access keys, values, or both simultaneously.

Nested Dictionary

A nested dictionary is a dictionary that contains another dictionary (or dictionaries) as its values. This allows you to create a hierarchical structure where you can access values using multiple keys. Here's an example of a nested dictionary in Python:

```
mydict = {
    "person1": {
        "name": "Vasan",
        "age": 25,
        "city": "Vellore"
    },
    "person2": {
        "name": "Alice",
        "age": 30,
        "city": "London"
    }
}
```

In this example, the `mydict` dictionary contains two key-value pairs, where each value is a dictionary itself. The keys `"person1"` and `"person2"` are associated with the nested dictionaries containing information about individuals.

To access values in a nested dictionary, you can use multiple keys to navigate through the levels. Here's how you can access values from the example above:

```
print(mydict["person1"]["name"])     # Output: Vasan
print(mydict["person2"]["age"])      # Output: 30
```

You can access the inner dictionary by specifying the corresponding key, followed by another key to access the value within that nested dictionary.

Nested dictionaries are commonly used to represent structured or hierarchical data, such as organizing data in a tree-like structure, representing multi-level configurations, or modeling complex relationships between objects.

Dictionary methods

Here's a table summarizing some commonly used dictionary methods in Python, along with their descriptions and examples:

Method	Description	Example
`clear()`	Removes all key-value pairs from the dictionary.	`mydict.clear()`
`copy()`	Creates a shallow copy of the dictionary.	`new_dict = mydict.copy()`
`get(key, default)`	Returns the value associated with the given key. If the key is not found, it returns the default value.	`mydict.get("name", "Unknown")`
`items()`	Returns a list-like object containing key-value pairs as tuples.	`mydict.items()`
`keys()`	Returns a list-like object containing all the keys in the dictionary.	`mydict.keys()`
`values()`	Returns a list-like object containing all the values in the dictionary.	`mydict.values()`
`pop(key, default)`	Removes the item with the specified key and returns its value. If the key is not found, it returns the default value.	`mydict.pop("age", 0)`
`popitem()`	Removes and returns a key-value pair from the dictionary in LIFO (Last In, First Out) order.	`mydict.popitem()`
`update(other_dict)`	Updates the dictionary with key-value pairs from another dictionary or iterable.	`mydict.update({"city": "London"})`
`setdefault(key, default)`	Returns the value associated with the given key. If the key is not found, it inserts the key-value pair with the default value.	`mydict.setdefault("name", "Unknown")`

Example 1 :

Problem: Given a list of fruits, create a dictionary where the keys are the fruit names, and the values are their respective colors.

Solution:

```
fruits = ["apple", "banana", "orange", "grape"]
fruit_colors = {"apple": "red", "banana": "yellow", "orange": "orange",
"grape": "purple"}

# Accessing dictionary values
print(fruit_colors["apple"])  # Output: "red"
print(fruit_colors["banana"])  # Output: "yellow"
print(fruit_colors["grape"])  # Output: "purple"

# Modifying dictionary values
fruit_colors["orange"] = "green"
print(fruit_colors["orange"])  # Output: "green"

# Adding new key-value pairs
fruit_colors["watermelon"] = "green"
print(fruit_colors["watermelon"])  # Output: "green"

# Removing key-value pairs
del fruit_colors["banana"]
print(fruit_colors.get("banana"))  # Output: None
```

In this example, we have a list of fruits and a dictionary called `fruit_colors`. We initialize the dictionary with key-value pairs where the keys are fruit names and the values are their respective colors.

We can access the values in the dictionary by providing the corresponding key, such as `fruit_colors["apple"]`, which returns the color `"red"`. Similarly, we can modify the values by assigning a new value to a specific key, as shown with `fruit_colors["orange"] = "green"`.

To add a new key-value pair, we can simply assign a value to a new key, like `fruit_colors["watermelon"] = "green"`. Conversely, we can remove a key-value pair using the `del` keyword, as demonstrated with `del fruit_colors["banana"]`.

Note that dictionaries are mutable objects in Python, allowing us to modify their values, add new entries, and remove existing entries as needed.

Example 2:

Problem: Given a list of students' names and their corresponding scores, find the student with the highest score.

Solution:

```
student_scores = {
    "Alice": 85,
    "Bob": 92,
    "Charlie": 78,
    "David": 95,
    "Eve": 88
}

highest_score = max(student_scores.values())  # Find the highest score
highest_scorer = max(student_scores, key=student_scores.get)  # Find the
student with the highest score

print("Student with the highest score:")
print("Name:", highest_scorer)
print("Score:", highest_score)
```

Output:

```
Student with the highest score:
Name: David
Score: 95
```

In this example, we have a dictionary called `student_scores`, where the keys are the students' names and the values are their corresponding scores.

To find the student with the highest score, we use the `max()` function. By calling `max(student_scores.values())`, we obtain the highest score among all the values in the dictionary. Next, we use `max(student_scores, key=student_scores.get)` to find the student name associated with that highest score. The `key` parameter specifies that we want to compare the values of the dictionary when determining the maximum.

Finally, we print the name and score of the student with the highest score using the obtained values.

This example demonstrates how dictionaries can be useful for storing and accessing data, as well as performing operations based on the values or keys in the dictionary.

Example 3

Problem: Given a list of words, create a dictionary where the keys are the words, and the values are the lengths of the words.

Solution:

```
words = ["apple", "banana", "orange", "grape"]
word_lengths = {word: len(word) for word in words}

# Accessing dictionary values
print(word_lengths["apple"])  # Output: 5
print(word_lengths["banana"])  # Output: 6
print(word_lengths["grape"])  # Output: 5
```

In this example, we have a list of words and we use a dictionary comprehension to create a dictionary called `word_lengths`. The keys of the dictionary are the words from the input list, and the values are the lengths of the corresponding words.

By iterating over each word in the `words` list and using the syntax `{word: len(word)}`, we create key-value pairs in the `word_lengths` dictionary. The word is used as the key, and the length of the word is used as the value.

We can then access the values in the dictionary by providing the corresponding key, such as `word_lengths["apple"]`, which returns the length of the word "apple" as 5. Similarly, we can access the lengths of other words in the same manner.

This example demonstrates how dictionary comprehensions can be used to quickly create dictionaries based on a given iterable, in this case, a list of words and their lengths.

Example 4

Problem: Given a string, count the frequency of each character and store the results in a dictionary.

Solution:

```
string = "Hello, World!"
character_frequency = {}

for char in string:
    if char in character_frequency:
        character_frequency[char] += 1
    else:
        character_frequency[char] = 1

print(character_frequency)
```

Output:

```
{'H': 1, 'e': 1, 'l': 3, 'o': 2, ',': 1, ' ': 1, 'W': 1, 'r': 1, 'd': 1,
'!': 1}
```

In this example, we have a string and an empty dictionary called `character_frequency`. We iterate over each character in the string using a `for` loop. For each character, we check if it is already a key in the `character_frequency` dictionary. If it is, we increment the corresponding value by 1. Otherwise, we add a new key-value pair to the dictionary with the character as the key and an initial value of 1.

After iterating through all the characters in the string, we print the `character_frequency` dictionary, which contains the frequency of each character in the input string.

This example demonstrates how dictionaries can be used to efficiently count the occurrence of each character in a string or any other iterable.

Example 5

Problem: Given a list of words, count the frequency of each word and store the results in a dictionary.

Solution:

```
def count_word_frequency(words):
    frequency = {}
    for word in words:
        if word in frequency:
            frequency[word] += 1
        else:
            frequency[word] = 1
    return frequency

# Example usage
word_list = ["apple", "banana", "apple", "orange", "banana", "grape",
"banana"]
word_frequency = count_word_frequency(word_list)
print(word_frequency)
```

Output:

```
{'apple': 2, 'banana': 3, 'orange': 1, 'grape': 1}
```

In this example, we define a function `count_word_frequency` that takes a list of words as input. We initialize an empty dictionary called `frequency` to store the word frequencies. Then, for each word in the input list, we check if the word is already a key in the `frequency` dictionary. If it is, we increment the corresponding value by 1. Otherwise, we add a new key-value pair to the `frequency` dictionary with the word as the key and an initial value of 1. Finally, we return the `frequency` dictionary.

In the example usage, we provide a list of words and call the `count_word_frequency` function to get the frequency of each word. The resulting dictionary is then printed, showing the frequency of each word in the input list.

Example 6 :

Real-world Dictionary Problem: You have a list of students and their corresponding grades for multiple subjects. You need to calculate the average grade for each student and determine their overall performance.

Solution:

```
# Function to calculate average grade
def calculate_average_grade(grades):
    total_grades = sum(grades)
    num_grades = len(grades)
    average_grade = total_grades / num_grades
    return average_grade


# Function to determine performance based on average grade
def determine_performance(average_grade):
    if average_grade >= 90:
        return "Excellent"
    elif average_grade >= 80:
        return "Good"
    elif average_grade >= 70:
        return "Average"
    else:
        return "Below Average"


# Dictionary representing student grades
student_grades = {
    "Alice": [80, 85, 90],
    "Bob": [90, 92, 88],
    "Charlie": [70, 75, 72],
    "Diana": [95, 88, 92]
}

# Dictionary to store average grades and performance
student_performance = {}

# Calculate average grade and determine performance for each student
for student, grades in student_grades.items():
    average_grade = calculate_average_grade(grades)
    performance = determine_performance(average_grade)
    student_performance[student] = {
        "average_grade": average_grade,
        "performance": performance
    }

# Display the results
for student, data in student_performance.items():
    print(f"Student: {student}")
    print(f"Average Grade: {data['average_grade']}")
    print(f"Performance: {data['performance']}")
    print()
```

Output:

```
Student: Alice
```

```
Average Grade: 85.0
Performance: Good

Student: Bob
Average Grade: 90.0
Performance: Excellent

Student: Charlie
Average Grade: 72.33333333333333
Performance: Below Average

Student: Diana
Average Grade: 91.66666666666667
Performance: Excellent
```

In this real-world example, we define two functions: `calculate_average_grade` and `determine_performance`. The `calculate_average_grade` function takes a list of grades as input, calculates the average grade, and returns it. The `determine_performance` function takes the average grade as input, determines the performance category (e.g., Excellent, Good, Average, Below Average), and returns the result.

We have a dictionary `student_grades` that represents the grades of different students. We iterate over the items of the `student_grades` dictionary and calculate the average grade for each student using the `calculate_average_grade` function. Then, we determine the performance category for each student using the `determine_performance` function. The results, including the average grade and performance, are stored in the `student_performance` dictionary.

Finally, we display the results by iterating over the items of the `student_performance` dictionary and printing the student's name, average grade, and performance.

This example demonstrates how dictionaries can be used to solve real-world problems, such as calculating grades and determining performance categories for students.

Example 7

Real-world Dictionary Problem: You have an inventory of products in a store, and you need to keep track of the available quantity for each product. You also want to be able to add new products, update their quantities, and check the availability of a specific product.

Solution:

```
# Dictionary representing the inventory
inventory = {
    "apple": 10,
    "banana": 15,
    "orange": 20,
    "grape": 5
}

# Function to add a new product to the inventory
def add_product(product, quantity):
    if product in inventory:
        print("Product already exists in the inventory.")
    else:
        inventory[product] = quantity
        print(f"{product} added to the inventory.")

# Function to update the quantity of a product in the inventory
def update_quantity(product, quantity):
    if product in inventory:
        inventory[product] += quantity
        print(f"Quantity of {product} updated to {inventory[product]}.")
    else:
        print("Product does not exist in the inventory.")

# Function to check the availability of a product
def check_availability(product):
    if product in inventory:
        quantity = inventory[product]
        if quantity > 0:
            print(f"{product} is available. Quantity: {quantity}.")
        else:
            print(f"{product} is out of stock.")
    else:
        print("Product does not exist in the inventory.")

# Adding a new product
add_product("pear", 8)

# Updating the quantity of a product
update_quantity("banana", 5)

# Checking the availability of a product
check_availability("orange")
```

Output:

```
pear added to the inventory.
Quantity of banana updated to 20.
orange is available. Quantity: 20.
```

In this real-world example, we have a dictionary `inventory` that represents the available quantity of different products in a store. Each key-value pair in the dictionary represents a product and its quantity.

We define three functions: `add_product`, `update_quantity`, and `check_availability`.

- The `add_product` function allows adding a new product to the inventory if it doesn't already exist.
- The `update_quantity` function updates the quantity of a product in the inventory if the product exists.
- The `check_availability` function checks the availability of a product in the inventory and displays the quantity.

We demonstrate the usage of these functions by adding a new product "pear" with a quantity of 8, updating the quantity of the "banana" to 20, and checking the availability of the "orange".

This example showcases how dictionaries can be used to manage an inventory system, including adding products, updating quantities, and checking availability.

Example 8

Real-world Dictionary Problem: You need to implement a user authentication system where users can register, log in, and access their account information. You want to store user data in a dictionary and perform various operations such as user registration, login verification, and retrieval of account information.

Solution:

```
# Dictionary representing user data
user_data = {}

# Function to register a new user
def register_user(username, password, email):
    if username in user_data:
        print("Username already exists.")
    else:
        user_data[username] = {
            "password": password,
            "email": email,
            "account_info": {}
        }
        print("User registration successful.")

# Function to verify user login
def verify_login(username, password):
    if username in user_data:
        if user_data[username]["password"] == password:
            print("Login successful.")
            return True
        else:
            print("Incorrect password.")
    else:
        print("Username not found.")
    return False

# Function to retrieve account information
def get_account_info(username):
    if username in user_data:
        account_info = user_data[username]["account_info"]
        print(f"Account information for {username}: {account_info}")
    else:
        print("Username not found.")

# Registering a new user
register_user("Vasan123", "password123", "Vasan@example.com")

# Verifying user login
verify_login("Vasan123", "password123")

# Retrieving account information
get_account_info("Vasan123")
```

Output:

```
User registration successful.
Login successful.
Account information for Vasan123: {}
```

In this real-world example, we use a dictionary `user_data` to store user information. Each key in the dictionary represents a username, and the corresponding value is a nested dictionary containing the user's password, email, and account information.

We define three functions: `register_user`, `verify_login`, and `get_account_info`.

- The `register_user` function allows registering a new user by adding their username as a key in the `user_data` dictionary with the corresponding password, email, and an empty dictionary for account information.
- The `verify_login` function verifies the login credentials provided by the user. It checks if the username exists in the `user_data` dictionary and compares the password with the stored password for that user.
- The `get_account_info` function retrieves the account information for a given username from the `user_data` dictionary.

We demonstrate the usage of these functions by registering a new user, verifying their login credentials, and retrieving their account information.

This example illustrates how dictionaries can be used to implement a basic user authentication system, storing user data and performing operations such as registration, login verification, and retrieval of account information.

Example 9

Real-world Dictionary Problem: You want to create a simple voting system where people can vote for their favorite candidate. You need to keep track of the votes received by each candidate and determine the winner based on the highest number of votes.

Solution:

```
# Dictionary representing the candidates and their initial votes
candidates = {
    "Alice": 0,
    "Bob": 0,
    "Charlie": 0
}

# Function to cast a vote for a candidate
def cast_vote(candidate):
    if candidate in candidates:
        candidates[candidate] += 1
        print(f"Vote cast for {candidate}.")
    else:
        print("Invalid candidate.")

# Function to determine the winner
def determine_winner():
    winner = max(candidates, key=candidates.get)
    max_votes = candidates[winner]
    print(f"The winner is {winner} with {max_votes} votes.")

# Casting votes
cast_vote("Alice")
cast_vote("Bob")
cast_vote("Charlie")
cast_vote("Bob")

# Determining the winner
determine_winner()
```

Output:

```
Vote cast for Alice.
Vote cast for Bob.
Vote cast for Charlie.
Vote cast for Bob.
The winner is Bob with 2 votes.
```

In this real-world example, we use a dictionary `candidates` to store the candidates and their corresponding vote counts. Each key in the dictionary represents a candidate's name, and the value represents the number of votes received by that candidate.

We define two functions: `cast_vote` and `determine_winner`.

- The `cast_vote` function allows casting a vote for a specific candidate. It checks if the candidate is valid (exists in the `candidates` dictionary) and increments their vote count by one.
- The `determine_winner` function determines the winner by finding the candidate with the highest number of votes using the `max` function with a custom key.

We demonstrate the usage of these functions by casting votes for different candidates and determining the winner based on the highest vote count.

Example 10

Problem: Given a list of names, group the names by their first letter into a dictionary.

Solution:

```
names = ["Alice", "Bob", "Charlie", "David", "Eve", "Alex", "Daniel"]
name_groups = {}

for name in names:
    first_letter = name[0]
    if first_letter in name_groups:
        name_groups[first_letter].append(name)
    else:
        name_groups[first_letter] = [name]

print(name_groups)
```

Output:

```
{'A': ['Alice', 'Alex'], 'B': ['Bob'], 'C': ['Charlie'], 'D': ['David',
'Daniel'], 'E': ['Eve']}
```

In this example, we have a list of names and an empty dictionary called `name_groups`. We iterate over each name in the list using a `for` loop. For each name, we extract the first letter using indexing (`name[0]`). We then check if the first letter is already a key in the `name_groups` dictionary. If it is, we append the current name to the list of names associated with that key. Otherwise, we create a new key-value pair in the dictionary, where the key is the first letter and the value is a list containing the current name.

After processing all the names, we print the `name_groups` dictionary, which groups the names by their first letter.

This example demonstrates how dictionaries can be used to group data based on a specific criterion, in this case, the first letter of the names.

Exercise Problem

1. Phonebook Directory: Design a phonebook directory where users can add contacts, look up phone numbers, and update contact information. Use a dictionary to store the contact names as keys and their corresponding phone numbers as values.
2. Word Frequency Counter: Write a program that reads a text document and counts the frequency of each word. Store the word frequencies in a dictionary, where the words are the keys and the frequencies are the values.
3. Product Inventory: Create an inventory management system for a store. Implement functions to add products, update their quantities, and check the availability of a specific product. Use a dictionary to store the product names as keys and their quantities as values.
4. Student Grades: Build a program to calculate and manage student grades. Allow users to add students, input their grades for multiple subjects, and calculate their average grades. Store the student names as keys in a dictionary and their corresponding grades as values (nested dictionaries).
5. Employee Database: Develop an employee database where users can add new employees, update their information (e.g., salary, department), and retrieve employee details. Use a dictionary to store employee IDs as keys and their information as values (nested dictionaries).
6. Weather Forecast: Create a weather forecast program that retrieves weather information for different cities. Store the city names as keys in a dictionary and their corresponding weather details (temperature, humidity, etc.) as values (nested dictionaries).
7. Library Catalog: Implement a library catalog system where users can add books, search for books by title or author, and check the availability of a book. Use a

dictionary to store book titles as keys and their corresponding information (author, publication year, availability) as values.

8. Movie Ratings: Create a program to collect and analyze movie ratings. Allow users to enter movie titles and their ratings. Store the movie titles as keys in a dictionary and their ratings as values (lists or nested dictionaries). Implement functions to calculate the average rating for each movie and find the highest-rated movie.
9. Word Game Score: Design a word game where users input words, and each word earns a certain score based on its length and letter values. Store the words as keys in a dictionary and their corresponding scores as values. Implement functions to calculate the total score for a given word and find the highest-scoring word.
10. Employee Attendance: Build a program to track employee attendance. Store the names of employees as keys in a dictionary and their attendance records as values (lists or nested dictionaries). Allow users to mark attendance, check the attendance status of a specific employee, and calculate the overall attendance percentage.
11. Online Store Cart: Develop a shopping cart system for an online store. Use a dictionary to store the items selected by the user as keys and their corresponding quantities as values. Allow users to add items, remove items, and calculate the total cost of the items in the cart.
12. Language Translator: Create a language translator program where users can input a word or phrase in one language and get its translation in another language. Use a dictionary to store the translations, with the words or phrases in the original language as keys and their translations as values.
13. Student Management System: Implement a student management system where users can add student records, retrieve student information, and calculate their average grades. Use a dictionary to store student IDs as keys and their information (name, grades) as values.
14. Stock Portfolio: Build a program to track a stock portfolio. Allow users to add stocks, update their prices, and calculate the total value of the portfolio. Use a dictionary to store the stock symbols as keys and their corresponding prices and quantities as values.
15. Social Media Analytics: Create a program to analyze social media data. Allow users to input posts or tweets and count the frequency of words used. Store the words as keys in a dictionary and their frequencies as values. Implement functions to find the most common words and calculate the overall sentiment score.
16. Election Voting System: Develop an election voting system where users can vote for different candidates. Store the candidate names as keys in a dictionary and their vote counts as values. Implement functions to cast votes, determine the winner, and display the results.
17. Hotel Reservation System: Design a hotel reservation system where users can book rooms, check room availability, and calculate the total cost of their stay. Use a dictionary to store room numbers as keys and their availability and price information as values.

18. Password Manager: Create a password manager program that stores and retrieves user passwords. Use a dictionary to store usernames as keys and their corresponding passwords as values. Implement functions to add new passwords, retrieve passwords, and generate strong passwords.