

python unit 3



How are tuples created in Python? Explain with an example.



In Python, tuples are created by placing a comma-separated sequence of values inside parentheses. Tuples can contain elements of different data types, and they are immutable, meaning that once created, their contents cannot be changed.

Example:

Here's how to create a tuple:

python

```
# Creating a tuple
my_tuple = (1, 2, 3, 'apple', 'banana')

# Accessing elements in a tuple
print(my_tuple[0]) # Output: 1
print(my_tuple[3]) # Output: apple

# Trying to change an element (this will raise an error)
# my_tuple[1] = 4 # TypeError: 'tuple' object does not support item assignment
```

Key Points:

- Tuples are defined using parentheses `()`.
- Elements are accessed using indexing, similar to lists.
- Attempting to modify a tuple will result in a `TypeError`.

You can also create a tuple without parentheses, just by separating the values with commas:

python

```
another_tuple = 4, 5, 6
print(another_tuple) # Output: (4, 5, 6)
```



Explain any three basic operations that can be performed on a tuple with examples.



Here are three basic operations that can be performed on tuples in Python:

1. Accessing Elements

You can access individual elements of a tuple using indexing. The index starts from 0.

Example:

```
python

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

# Accessing elements
print(my_tuple[0]) # Output: 10
print(my_tuple[2]) # Output: 30
```

2. Slicing

You can create a new tuple by slicing the original tuple. This allows you to extract a range of elements.

Example:

```
python

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

# Slicing the tuple
sliced_tuple = my_tuple[1:4] # Elements from index 1 to 3
print(sliced_tuple) # Output: (2, 3, 4)
```

3. Concatenation

You can concatenate two tuples using the `+` operator to create a new tuple.

Example:

```
python

tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenating tuples
combined_tuple = tuple1 + tuple2
print(combined_tuple) # Output: (1, 2, 3, 4, 5, 6)
```

These operations demonstrate how to access and manipulate data within tuples, even though the tuples themselves remain immutable.



What are sets in Python? How do you create a set?



In Python, a **set** is an unordered collection of unique elements. Sets are useful when you want to store a group of items without duplicates and when the order of items does not matter. They support operations like union, intersection, and difference.

Creating a Set

You can create a set in two main ways:

1. Using curly braces `{}`:

python

```
my_set = {1, 2, 3, 'apple', 'banana'}
```

2. Using the `set()` constructor:

python

```
my_set = set([1, 2, 3, 'apple', 'banana'])
```

Example:

Here's an example demonstrating both methods of creating a set:

python

```
# Creating a set using curly braces
set1 = {1, 2, 3, 4, 5}
print(set1) # Output: {1, 2, 3, 4, 5}

# Creating a set using the set() constructor
set2 = set([1, 2, 3, 4, 5])
print(set2) # Output: {1, 2, 3, 4, 5}

# Demonstrating uniqueness
set3 = {1, 2, 2, 3} # Duplicates are ignored
print(set3) # Output: {1, 2, 3}
```

Key Points:

- Sets are unordered, so the items do not have a defined index.
- Sets automatically remove duplicate entries.
- You can perform various set operations, like union and intersection, easily with sets.



Give comparison between Tuple and List in python



Here's a comparison between tuples and lists in Python, highlighting their key differences:

Feature	Tuple	List
Syntax	Defined with parentheses `()`	Defined with square brackets `[]`
Mutability	Immutable (cannot be modified)	Mutable (can be modified)
Performance	Generally faster due to immutability	Slightly slower due to mutability
Use Case	Suitable for fixed collections of items	Suitable for dynamic collections that may change
Methods	Fewer built-in methods (e.g., count, index)	More built-in methods (e.g., append, remove, sort)
Nested Structures	Can contain nested lists or other tuples	Can contain nested tuples or other lists
Memory Consumption	Uses less memory due to immutability	Requires more memory for dynamic resizing
Hashability	Can be used as a key in dictionaries (hashable)	Cannot be used as a key in dictionaries (not hashable)

Examples:

Tuple Example:

python

```
my_tuple = (1, 2, 3)
# Accessing an element
print(my_tuple[1]) # Output: 2
# Attempting to modify (will raise an error)
# my_tuple[1] = 4 # TypeError
```

List Example:

python

```
my_list = [1, 2, 3]
# Accessing an element
print(my_list[1]) # Output: 2
# Modifying an element
my_list[1] = 4
print(my_list) # Output: [1, 4, 3]
```

Summary:

- Use tuples when you need a fixed collection of items that should not change.
- Use lists when you need a dynamic collection that may require modifications.



Given a tuple (10, 20, 30, 40), write a Python code to unpack the elements into four variables.



You can unpack the elements of a tuple into separate variables by assigning them directly. Here's how you can do it with the tuple `(10, 20, 30, 40)`:

python

```
# Given tuple
my_tuple = (10, 20, 30, 40)

# Unpacking the tuple into variables
a, b, c, d = my_tuple

# Printing the variables
print(a) # Output: 10
print(b) # Output: 20
print(c) # Output: 30
print(d) # Output: 40
```

In this code, the elements of `my_tuple` are unpacked into the variables `a`, `b`, `c`, and `d`, respectively.



Define key-value pairs in a dictionary and demonstrate access and modification.



In Python, a **dictionary** is a collection of key-value pairs. Each key is unique and is used to access its corresponding value. Dictionaries are mutable, meaning you can modify them after creation.

Creating a Dictionary

You can create a dictionary using curly braces `{}` or the `dict()` constructor.

Example:

python

```
# Creating a dictionary using curly braces
my_dict = {
    'name': 'Alice',
    'age': 30,
    'city': 'New York'
}
```

Accessing Values

You can access values in a dictionary by referring to their keys.

Example:

python

```
# Accessing values
print(my_dict['name']) # Output: Alice
print(my_dict['age'])  # Output: 30
```

Modifying Values

You can modify existing values by assigning a new value to a specific key.

Example:

python

```
# Modifying a value
my_dict['age'] = 31
print(my_dict['age']) # Output: 31
```

Adding New Key-Value Pairs

You can also add new key-value pairs to the dictionary.

Example:

python

```
# Adding a new key-value pair
my_dict['occupation'] = 'Engineer'
print(my_dict) # Output: {'name': 'Alice', 'age': 31, 'city': 'New York', 'occupation': 'Engineer'}
```

Complete Example:

Here's a complete example demonstrating creation, access, modification, and addition:

python

```
# Creating a dictionary
my_dict = {
    'name': 'Alice',
    'age': 30,
    'city': 'New York'
}

# Accessing values
print(my_dict['name']) # Output: Alice

# Modifying a value
my_dict['age'] = 31
print(my_dict['age']) # Output: 31

# Adding a new key-value pair
my_dict['occupation'] = 'Engineer'
print(my_dict) # Output: {'name': 'Alice', 'age': 31, 'city': 'New York', 'occupation': 'Engineer'}
```

This example demonstrates how to create a dictionary, access its values, modify existing values, and add new key-value pairs.



Differentiate between lists and dictionaries



Here's a comparison between lists and dictionaries in Python, highlighting their key differences:

Feature	List	Dictionary
Definition	Ordered collection of items	Unordered collection of key-value pairs
Syntax	Defined with square brackets `[]`	Defined with curly braces `{}` or `dict()`
Order	Maintains the order of elements	Does not maintain any specific order (as of Python 3.7, insertion order is preserved, but it's still considered unordered)
Access	Accessed by index (integer)	Accessed by key (hashable type)
Mutability	Mutable (can be modified)	Mutable (can be modified)
Duplicates	Allows duplicate values	Keys must be unique; values can be duplicated
Use Case	Suitable for ordered collections of items	Suitable for associative arrays or when you need a mapping from keys to values
Methods	Methods like append, remove, sort	Methods like get, keys, values, items
Memory Usage	Generally uses less memory	Generally uses more memory due to key-value structure

Examples:

List Example:

python

```
my_list = [1, 2, 3, 4, 5]
print(my_list[1]) # Output: 2
my_list.append(6) # Adding an element
print(my_list)    # Output: [1, 2, 3, 4, 5, 6]
```

Dictionary Example:

python

```
my_dict = {'name': 'Alice', 'age': 30}
print(my_dict['name']) # Output: Alice
my_dict['age'] = 31    # Modifying a value
print(my_dict)        # Output: {'name': 'Alice', 'age': 31}
```

Summary:

- Use **lists** when you need an ordered collection of items where duplicates are allowed and index-based access is required.
- Use **dictionaries** when you need to associate unique keys with values for quick access and retrieval.



Explain sets in Python and their unique properties



In Python, a **set** is an unordered collection of unique elements. Sets are a built-in data type and are particularly useful when you need to store multiple items without duplicates and when the order of elements does not matter.

Key Properties of Sets:

1. Unordered:

- Sets do not maintain any specific order. The elements may not appear in the order they were added.

2. Unique Elements:

- Sets automatically remove duplicate entries. Each element must be unique, meaning a set cannot contain the same value multiple times.

3. Mutable:

- Sets are mutable, meaning you can add or remove elements after creation.

4. Dynamic Size:

- Sets can grow and shrink as you add or remove elements.

5. Support for Mathematical Set Operations:

- Sets support various operations like union, intersection, difference, and symmetric difference, which are useful for mathematical computations.

6. Unhashable Types:

- The elements of a set must be hashable. This means you can use immutable data types like numbers, strings, and tuples but not lists or other sets as elements.

Creating a Set:

You can create a set using curly braces `{}` or the `set()` constructor.

Example:

python

```
# Creating a set using curly braces
my_set = {1, 2, 3, 'apple', 'banana'}

# Creating a set using the set() constructor
my_set2 = set([1, 2, 2, 3, 'apple']) # Duplicates will be removed
```

Basic Operations:

1. Adding Elements:

python

```
my_set.add('orange')
print(my_set) # Output: {1, 2, 3, 'apple', 'banana', 'orange'}
```

2. Removing Elements:

python

```
my_set.remove('banana') # Raises KeyError if 'banana' is not found
print(my_set) # Output: {1, 2, 3, 'apple', 'orange'}
```

3. Set Operations:

python

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

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

# Intersection
intersection_set = set_a & set_b # or set_a.intersection(set_b)
print(intersection_set) # Output: {3}

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

Summary:

Sets in Python provide a powerful way to handle unique collections of items, perform mathematical set operations, and manipulate data efficiently without worrying about duplicates. They are particularly useful in scenarios where membership testing, deduplication, and mathematical operations are required.



Differentiate between sets and dictionaries.



Here's a comparison between sets and dictionaries in Python, highlighting their key differences:

Feature	Set	Dictionary
Definition	Unordered collection of unique elements	Unordered collection of key-value pairs
Syntax	Defined with curly braces `{}`	Defined with curly braces `{}` or `dict()`

Feature	Set	Dictionary
Structure	Contains only values	Contains key-value pairs
Access	Elements are accessed by value	Values are accessed by key
Uniqueness	All elements must be unique	Keys must be unique; values can be duplicated
Mutability	Mutable (can add/remove elements)	Mutable (can add/remove key-value pairs)
Order	Unordered (as of Python 3.7, maintains insertion order but not guaranteed)	Unordered (as of Python 3.7, maintains insertion order but not guaranteed)
Use Case	Suitable for storing collections of items without duplicates	Suitable for associating unique keys with values for quick access
Methods	Methods like add, remove, union, intersection	Methods like get, keys, values, items
Memory Usage	Generally uses less memory	Generally uses more memory due to key-value structure

Examples:

Set Example:

python

```
my_set = {1, 2, 3, 'apple'}
print(my_set) # Output: {1, 2, 3, 'apple'}

# Adding an element
my_set.add('banana')
print(my_set) # Output: {1, 2, 3, 'apple', 'banana'}
```

Dictionary Example:

python

```
my_dict = {'name': 'Alice', 'age': 30}
print(my_dict) # Output: {'name': 'Alice', 'age': 30}

# Accessing a value
print(my_dict['name']) # Output: Alice

# Modifying a value
my_dict['age'] = 31
print(my_dict) # Output: {'name': 'Alice', 'age': 31}
```

Summary:

- Use **sets** when you need to store unique items and perform set operations like union and intersection.
- Use **dictionaries** when you need to map unique keys to values for efficient access and retrieval.



Differentiate List and Tuple in Python



Here's a comparison between lists and tuples in Python, highlighting their key differences:

Feature	List	Tuple
Definition	Ordered collection of items	Ordered collection of items
Syntax	Defined with square brackets `[]`	Defined with parentheses `()`
Mutability	Mutable (can be modified)	Immutable (cannot be modified)
Performance	Generally slower due to mutability	Generally faster due to immutability
Use Case	Suitable for dynamic collections that may change	Suitable for fixed collections of items
Methods	More built-in methods (e.g., append, remove, sort)	Fewer built-in methods (e.g., count, index)
Duplicates	Allows duplicate values	Allows duplicate values
Memory Consumption	Generally uses more memory due to dynamic resizing	Uses less memory due to fixed size
Hashability	Not hashable (cannot be used as a key in dictionaries)	Hashable (can be used as a key in dictionaries)

Examples:

List Example:

python

```
my_list = [1, 2, 3, 4, 5]
print(my_list[1]) # Output: 2
my_list.append(6) # Adding an element
print(my_list)    # Output: [1, 2, 3, 4, 5, 6]
```

Tuple Example:

python

```
my_tuple = (1, 2, 3)
print(my_tuple[1]) # Output: 2
# Attempting to modify (will raise an error)
# my_tuple[1] = 4 # TypeError
```

Summary:

- Use **lists** when you need a dynamic collection of items that may require modifications.
- Use **tuples** when you need a fixed collection of items that should not change, offering better performance and hashability.



Consider the list `lst=[9,8,7,6,5,4,3]`. Write the Python program which performs the following operation without using built-in methods.

- 1) Insert element 10 at beginning of the list.
 - 2) Insert element 2 at end of the list.
 - 3) Delete the element at index position 5.
- What is dictionary in Python? Explain with an example.



Here's how you can perform the specified operations on the list `lst = [9, 8, 7, 6, 5, 4, 3]` without using built-in methods:

Python Program

python

```
# Initial list
lst = [9, 8, 7, 6, 5, 4, 3]

# 1) Insert element 10 at the beginning
def insert_at_beginning(lst, element):
    # Create a new list with the element added at the start
    new_list = [element] + lst
    return new_list

lst = insert_at_beginning(lst, 10)

# 2) Insert element 2 at the end
def insert_at_end(lst, element):
    # Create a new list with the element added at the end
    new_list = lst + [element]
    return new_list

lst = insert_at_end(lst, 2)

# 3) Delete the element at index position 5
def delete_at_index(lst, index):
    # Create a new list excluding the element at the specified index
    new_list = lst[:index] + lst[index+1:]
    return new_list

lst = delete_at_index(lst, 5)

# Print the final list
print(lst) # Output: [10, 9, 8, 7, 6, 2]
```

Explanation of Operations:

1. **Insert at Beginning:** A new list is created by adding the new element at the start and concatenating it with the original list.
2. **Insert at End:** Similarly, a new list is created by concatenating the original list with a new list containing the element to be added.
3. **Delete at Index:** A new list is created by slicing the original list to exclude the element at the specified index.

What is a Dictionary in Python?

A **dictionary** is a built-in data type in Python that stores data in key-value pairs. Each key is unique and is used to access its corresponding value. Dictionaries are mutable, meaning you can add, remove, and modify key-value pairs.

Key Properties:

- **Unordered:** The order of elements is not guaranteed (as of Python 3.7, dictionaries maintain insertion order).
- **Mutable:** You can change the contents of a dictionary after it is created.
- **Keys must be unique:** Duplicate keys are not allowed; if a key is reused, its value is updated.

Creating a Dictionary:

You can create a dictionary using curly braces `{}` or the `dict()` constructor.

Example:

python

```
# Creating a dictionary
my_dict = {
    'name': 'Alice',
    'age': 30,
    'city': 'New York'
}

# Accessing a value
print(my_dict['name']) # Output: Alice

# Modifying a value
my_dict['age'] = 31
print(my_dict) # Output: {'name': 'Alice', 'age': 31, 'city': 'New York'}

# Adding a new key-value pair
my_dict['occupation'] = 'Engineer'
print(my_dict) # Output: {'name': 'Alice', 'age': 31, 'city': 'New York', 'occupation': 'Engineer'}
```

In this example, the dictionary `my_dict` stores information about a person, and you can access, modify, and add key-value pairs easily.



Describe any four common set methods with examples.



Here are four common set methods in Python, along with explanations and examples for each:

1. `add()`

The `add()` method adds an element to a set. If the element is already present, it does not add a duplicate.

Example:

python

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set) # Output: {1, 2, 3, 4}

# Adding a duplicate element
my_set.add(2)
print(my_set) # Output: {1, 2, 3, 4} (no change)
```

2. `remove()`

The `remove()` method removes a specified element from a set. If the element is not found, it raises a `KeyError`.

Example:

python

```
my_set = {1, 2, 3, 4}
my_set.remove(3)
print(my_set) # Output: {1, 2, 4}

# Trying to remove a non-existent element (will raise KeyError)
# my_set.remove(5) # Uncommenting this will raise KeyError
```

3. `discard()`

The `discard()` method also removes a specified element from a set, but it does not raise an error if the element is not found.

Example:

python

```
my_set = {1, 2, 3, 4}
my_set.discard(3)
print(my_set) # Output: {1, 2, 4}

# Trying to discard a non-existent element (no error)
my_set.discard(5)
print(my_set) # Output: {1, 2, 4} (no change)
```

4. `union()`

The `union()` method returns a new set that is the union of two sets, combining all unique elements from both sets.

Example:

python

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union of set_a and set_b
```

```
result = set_a.union(set_b)
print(result) # Output: {1, 2, 3, 4, 5}

# Using the | operator for union
result2 = set_a | set_b
print(result2) # Output: {1, 2, 3, 4, 5}
```

Summary of Common Set Methods:

- `add(element)`: Adds a single element to the set.
- `remove(element)`: Removes a specified element; raises `KeyError` if not found.
- `discard(element)`: Removes a specified element; does not raise an error if not found.
- `union(other_set)`: Returns a new set that is the union of the current set and another set.



Create a set `fruit_set` with the values {"apple", "banana", "cherry"}. Add the value "date" to `fruit_set`. Print the updated `fruit_set`. Remove the value "banana" from `fruit_set`. Print the modified `fruit_set`.



Here's how you can create a set called `fruit_set`, add the value "date" to it, and then remove "banana":

python

```
# Creating the initial set
fruit_set = {"apple", "banana", "cherry"}

# Adding "date" to the set
fruit_set.add("date")
print("Updated fruit_set:", fruit_set) # Output: Updated fruit_set: {'apple', 'banana', 'cherry', 'date'}

# Removing "banana" from the set
fruit_set.remove("banana")
print("Modified fruit_set:", fruit_set) # Output: Modified fruit_set: {'apple', 'cherry', 'date'}
```

Explanation:

1. **Creation:** The set `fruit_set` is created with the initial values "apple", "banana", and "cherry".
2. **Addition:** The `add()` method is used to include "date" in the set.
3. **Removal:** The `remove()` method is used to eliminate "banana" from the set. If "banana" is not present, it would raise a `KeyError`.



Create a tuple `fruit_tuple` with the values ("apple", "banana", "cherry"). Access and print the second element of `fruit_tuple` using indexing. Extract and print the substring containing "banana" and "cherry" using slicing.
Create another tuple `more_fruits` with the values ("date", "elderberry"). Concatenate `fruit_tuple`

and `more_fruits` into a new tuple `all_fruits` and print it. Use the `len()` function to print the number of elements in `all_fruits`.



Here's how you can create and manipulate the specified tuples in Python:

python

```
# Creating the initial tuple
fruit_tuple = ("apple", "banana", "cherry")

# Accessing and printing the second element
second_element = fruit_tuple[1]
print("Second element:", second_element) # Output: Second element: banana

# Extracting and printing the substring containing "banana" and "cherry" using slicing
substring = fruit_tuple[1:3] # Slicing from index 1 to 2 (3 is not included)
print("Sliced tuple:", substring) # Output: Sliced tuple: ('banana', 'cherry')

# Creating another tuple
more_fruits = ("date", "elderberry")

# Concatenating the tuples
all_fruits = fruit_tuple + more_fruits
print("All fruits:", all_fruits) # Output: All fruits: ('apple', 'banana', 'cherry', 'date', 'elderberry')

# Using len() to print the number of elements in all_fruits
number_of_fruits = len(all_fruits)
print("Number of elements in all_fruits:", number_of_fruits) # Output: Number of elements in all_fruits: 5
```

Explanation:

1. **Creation:** The tuple `fruit_tuple` is created with the values "apple", "banana", and "cherry".
2. **Accessing an Element:** The second element is accessed using indexing (`fruit_tuple[1]`).
3. **Slicing:** A slice is created from the second to the third element (`fruit_tuple[1:3]`).
4. **Concatenation:** The tuples `fruit_tuple` and `more_fruits` are concatenated to form `all_fruits`.
5. **Length:** The `len()` function is used to determine the number of elements in `all_fruits`.



Create a dictionary named `person_info` with the following keys: "name", "age", "occupation", and "hobbies". Populate the dictionary with values of your choice. "hobbies" should be a list of at least three hobbies. Print the `person_info` dictionary. Access and print the value associated with the "occupation" key. Update the "age" value to a new age and print the updated dictionary. Add a new key-value pair "email" with a value of your choice to the dictionary and print the updated dictionary.



Here's how you can create and manipulate the `person_info` dictionary in Python:

python

```
# Creating the dictionary
person_info = {
    "name": "Alice",
    "age": 30,
    "occupation": "Engineer",
    "hobbies": ["reading", "hiking", "painting"]
}

# Printing the person_info dictionary
print("Person Info:", person_info)

# Accessing and printing the value associated with the "occupation" key
occupation = person_info["occupation"]
print("Occupation:", occupation) # Output: Occupation: Engineer

# Updating the "age" value to a new age
person_info["age"] = 31
print("Updated Person Info:", person_info)

# Adding a new key-value pair "email"
person_info["email"] = "alice@example.com"
print("Updated Person Info with Email:", person_info)
```

Explanation:

1. **Creation:** The dictionary `person_info` is created with keys: "name", "age", "occupation", and "hobbies".
2. **Printing the Dictionary:** The entire dictionary is printed to show its contents.
3. **Accessing a Value:** The value associated with the "occupation" key is accessed and printed.
4. **Updating a Value:** The "age" value is updated to a new age, and the updated dictionary is printed.
5. **Adding a New Key-Value Pair:** A new key "email" is added to the dictionary, and the updated dictionary is printed again.

A dictionary in Python is a collection of key-value pairs where each key is unique. Dictionaries are defined using curly braces `{}` or the `dict()` constructor. You can access, add, modify, and delete elements in a dictionary.

Example of a Dictionary

Here's an example of a dictionary representing a person's information:

python

```
# Creating a dictionary
person_info = {
    "name": "Alice",
    "age": 30,
    "occupation": "Engineer",
    "hobbies": ["reading", "hiking", "cooking"]
}

# Printing the dictionary
print(person_info)
```

Accessing Individual Members

You can access individual members of a dictionary using their keys. Here are some methods to access members:

1. **Using Square Bracket Notation:** You can access values directly by using the key inside square brackets.

```
python

name = person_info["name"]
print("Name:", name) # Output: Name: Alice
```

2. **Using the `.get()` Method:** The `.get()` method is a safe way to access values. If the key does not exist, it returns `None` (or a default value if provided) instead of raising a `KeyError`.

```
python

age = person_info.get("age")
print("Age:", age) # Output: Age: 30

# Accessing a non-existent key
email = person_info.get("email", "Not found")
print("Email:", email) # Output: Email: Not found
```

3. **Accessing Nested Elements:** If a value is another dictionary or a list, you can access its elements using multiple keys or indices.

```
python

# Accessing the first hobby
first_hobby = person_info["hobbies"][0]
print("First Hobby:", first_hobby) # Output: First Hobby: reading
```

Modifying Dictionary Members

You can modify the values associated with existing keys:

```
python

# Updating the age
person_info["age"] = 31
print("Updated Age:", person_info["age"]) # Output: Updated Age: 31
```

Adding New Members

To add a new key-value pair to the dictionary:

```
python

# Adding a new key-value pair for email
person_info["email"] = "alice@example.com"
```

```
print("Updated Person Info:", person_info)
```

Summary

- **Creating a Dictionary:** Use curly braces or the `dict()` constructor.
- **Accessing Values:** Use square brackets or the `.get()` method.
- **Modifying and Adding Members:** You can update existing values or add new key-value pairs easily.

Dictionaries are versatile and widely used for storing and managing structured data in Python.

ChatGPT can make mistakes. Check important info.