A **nested dictionary** in Python is a dictionary within another dictionary. It's a way to store and organize complex data structures, where each key maps to a dictionary instead of a single value. This is useful for representing data that has multiple layers of information.

### Example 1: Basic Nested Dictionary

Here's a simple example where we have a dictionary of students, and each student has their own dictionary with more details.

~~~
students = {
    "John": {
        "age": 25,
        "major": "Computer Science",
        "grades": {"math": 90, "science": 88}
    },
    "Alice": {
        "age": 22,
        "major": "Physics",
        "grades": {"math": 85, "science": 92}
    }
}

# Accessing nested dictionary elements
print(students["John"]["age"])  # Output: 25
print(students["Alice"]["grades"]["science"])  # Output: 92
```
~~~

### Example 2: Nested Dictionary with Multiple Levels

You can also nest dictionaries deeper by adding more layers.

~~~
company = {
    "Department_A": {
        "Manager": "Bob",
        "Employees": {
            "Emp1": {"name": "Alice", "position": "Developer"},
            "Emp2": {"name": "Charlie", "position": "Designer"}
~~~

```
        }
    },
    "Department_B": {
        "Manager": "Sarah",
        "Employees": {
            "Emp1": {"name": "David", "position": "Analyst"},
            "Emp2": {"name": "Emma", "position": "Data Scientist"}
        }
    }
}


# Accessing elements
print(company["Department_A"]["Employees"]["Emp1"]["name"])  # Output: Alice
print(company["Department_B"]["Manager"])  # Output: Sarah
```

### Key Points:

- Each key in the main dictionary can point to another dictionary.
- You can access nested values by chaining keys (e.g., `dict1["key1"]["key2"]`).
- Nested dictionaries are useful when you need to store hierarchical or multi-level data.

To **update** or **delete** values in a nested dictionary, you can access the desired keys and then apply the appropriate dictionary methods.

### Updating Values in a Nested Dictionary

To update a value in a nested dictionary, you access the value by chaining keys and then assign the new value.

#### Example: Updating a Value

Let's update the age of a student in a nested dictionary.

```
students = {
    "John": {
        "age": 25,
        "major": "Computer Science",
        "grades": {"math": 90, "science": 88}
    },
    "Alice": {
        "age": 22,
        "major": "Physics",
        "grades": {"math": 85, "science": 92}
    }
}

# Updating John's age
students["John"]["age"] = 26

# Updating Alice's math grade
students["Alice"]["grades"]["math"] = 95

print(students)
```

Output:

```
{
    "John": {
        "age": 26,
        "major": "Computer Science",
        "grades": {"math": 90, "science": 88}
    },
    "Alice": {
        "age": 22,
        "major": "Physics",
        "grades": {"math": 95, "science": 92}
    }
}
```

### Deleting Elements from a Nested Dictionary

To delete an element in a nested dictionary, you use the `del` keyword or the `.pop()` method to remove an entry.

#### Example: Deleting a Key-Value Pair

1. **Using `del` to delete:**
```python
# Deleting John's grades
del students["John"]["grades"]

print(students)
```

Output:
```python
{
    "John": {
        "age": 26,
        "major": "Computer Science"
    },
    "Alice": {
        "age": 22,
        "major": "Physics",
        "grades": {"math": 95, "science": 92}
    }
}
```

2. **Using `.pop()` to delete:**

The `.pop()` method can also be used to delete a key, and it returns the value that was removed.

```python
# Removing Alice's major and storing the value
removed_major = students["Alice"].pop("major")

print(removed_major)  # Output: Physics
print(students)
```

Output:
```python
{
    "John": {
        "age": 26,
        "major": "Computer Science"
    },
    "Alice": {
        "age": 22,
        "grades": {"math": 95, "science": 92}
    }
}
```

### Recap of Key Operations:
- **Updating**: Use `dict[key] = new_value` to change values.
- **Deleting**: Use `del dict[key]` to remove a key-value pair, or use `dict.pop(key)` to remove and retrieve the value.

This allows you to dynamically modify nested dictionaries based on your needs.

A **tuple** in Python is an immutable, ordered collection of elements. Unlike lists, once a tuple is created, its elements cannot be changed (hence the term "immutable"). Tuples are defined by placing elements inside parentheses `()` and separated by commas.

### Example: Tuple in Python

my_tuple = (1, 2, 3, "hello", True)

print(my_tuple) # Output: (1, 2, 3, 'hello', True)

print(my_tuple[0]) # Accessing first element, Output: 1

print(my_tuple[-1]) # Accessing last element, Output: True

~~~

Characteristics of Tuples:

- **Immutable**: Elements cannot be changed or removed once a tuple is created.

- **Ordered**: The order in which you place items is preserved.

- **Can hold different data types**: Tuples can store integers, strings, booleans, or other types of data.

Creating a Tuple with One Value

To create a tuple with **one value**, you must include a trailing comma after the element; otherwise, Python will interpret it as a simple expression or the type of the value itself.

Example: Single-element Tuple

```python

# Without a comma, this is not a tuple

not_a_tuple = (5)

print(type(not_a_tuple))  # Output: <class 'int'>

# Correct way: With a comma, this is a tuple
~~~

single_value_tuple = (5,)

print(type(single_value_tuple))  # Output: <class 'tuple'>

### Why Use Tuples?

- **Immutability**: If you want a collection of values that should not be changed, tuples are ideal.

- **Memory Efficiency**: Tuples take less memory compared to lists.

- **Performance**: Tuples can be faster than lists when iterating over items or when used as keys in dictionaries (since tuples are hashable, but lists are not).

In short, a tuple can store multiple values, including a single value (with a comma for single-element tuples), and is immutable by nature.

**You can iterate over a tuple in Python in two main ways:**

1. **Using `range()` and `len()`**: This allows you to iterate over the tuple by index.

2. **Directly iterating**: This allows you to loop through the elements of the tuple directly without using indices.

### 1. Iterating Using `range()` and `len()`

When you use `range()` along with `len()`, you can iterate over the tuple using indices.

#### Example:

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

# Iterating using range() and len()
for i in range(len(my_tuple)):
    print(f"Index {i}: {my_tuple[i]}")
```

Output:

```
Index 0: 10

Index 1: 20

Index 2: 30

Index 3: 40
```

### 2. Iterating Directly Without `range()`

You can directly iterate over the elements of the tuple without needing `range()` or indices.

#### Example:

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

# Direct iteration over the tuple elements
for item in my_tuple:
    print(item)
```

Output:

```
10

20

30

40
```

### Summary:

- **Using `range()`**: Gives you access to both index and value.

- **Direct iteration**: Simplifies looping when you only need the values in the tuple.

Here are examples of how you can use the following tuple functions in Python: `min()`, `max()`, `count()`, `index()`, and `sum()`.

### Example Tuple:

```python
my_tuple = (5, 8, 2, 7, 5, 8, 1)
```

### 1. `min()` Function

The `min()` function returns the smallest element in the tuple.

```python
# Smallest element in the tuple
print(min(my_tuple))  # Output: 1
```

### 2. `max()` Function

The `max()` function returns the largest element in the tuple.

```python
# Largest element in the tuple
print(max(my_tuple))  # Output: 8
```

### 3. `count()` Function

The `count()` function returns how many times a specified value appears in the tuple.

```python
# Count how many times the number 5 appears in the tuple
print(my_tuple.count(5))  # Output: 2
```

### 4. `index()` Function

The `index()` function returns the first index of the specified value in the tuple. If the value is not found, it raises a `ValueError`.

```python
# Find the index of the first occurrence of the number 7
print(my_tuple.index(7))  # Output: 3

# Find the index of the number 8
print(my_tuple.index(8))  # Output: 1
```

### 5. `sum()` Function

The `sum()` function returns the sum of the elements in the tuple. You can optionally provide a second argument, which will be added to the result.

#### Example with One Argument:

```python
# Sum of all elements in the tuple
print(sum(my_tuple))  # Output: 36
```

#### Example with Two Arguments:

In this example, we add a second argument `10` to the sum of elements in the tuple.

```python
# Sum of all elements in the tuple, with an additional 10 added to the result
print(sum(my_tuple, 10))  # Output: 46
```

### Summary of Functions:

- **`min(tuple)`**: Returns the smallest element.

- **`max(tuple)`**: Returns the largest element.

- **`count(value)`**: Returns how many times `value` appears in the tuple.

- **`index(value)`**: Returns the index of the first occurrence of `value`.

- **`sum(tuple)`**: Returns the sum of elements (optional second argument adds a value to the sum).

==

### **What is a Set in Python?**

A **set** in Python is an unordered collection of unique elements. Sets are mutable, meaning you can modify them after creation (add or remove elements). They do not allow duplicates and are defined using curly braces `{}` or the `set()` function.

### **Creating a Set**

```python

# Example of creating a set

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

print(my_set)  # Output: {1, 2, 3, 4, 5}


# Using the set() function to create a set

another_set = set([1, 2, 2, 3, 4])  # Duplicates are removed

print(another_set)  # Output: {1, 2, 3, 4}
```

### **How to Iterate Over a Set**

Since sets are unordered, you can iterate over them directly using a `for` loop.

```python

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

# Iterating through a set
for item in my_set:
    print(item)
```

**Output:**
```
1
2
3
4
5

### **Set Functions and Methods**

Here are some important functions and methods used with sets:

#### 1. **`set()`**: Creating a Set
The `set()` function is used to create a set from an iterable (like a list or tuple), eliminating duplicate elements.

```python
# Creating a set from a list
my_list = [1, 2, 2, 3, 4]
my_set = set(my_list)
print(my_set)  # Output: {1, 2, 3, 4}
```

#### 2. **`add()`**: Adding an Element to a Set
The `add()` method adds a single element to the set.

```python
my_set = {1, 2, 3}
my_set.add(4)  # Adding the number 4 to the set
print(my_set)  # Output: {1, 2, 3, 4}
```

#### 3. **`pop()`**: Removing and Returning an Arbitrary Element

The `pop()` method removes and returns an arbitrary element from the set. Since sets are unordered, you cannot predict which element will be removed.

```python
my_set = {1, 2, 3, 4, 5}
removed_item = my_set.pop()
print(removed_item)  # Output: Could be any item, e.g., 1
print(my_set)  # Output: The set with one element removed
```

#### 4. **`remove()`**: Removing a Specific Element

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

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

# Trying to remove an element that is not present raises an error:
# my_set.remove(5)  # This will raise a KeyError
```

#### 5. **`discard()`**: Removing a Specific Element (No Error if Not Found)

The `discard()` method also removes a specific element, but it **does not raise an error** if the element is not present.

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

# Does not raise an error if the element is not found:
my_set.discard(5)  # No error, does nothing
```

#### 6. **`clear()`**: Removing All Elements from the Set

The `clear()` method removes all elements from the set, leaving it empty.

```python
my_set = {1, 2, 3, 4}
my_set.clear()  # Removes all elements
print(my_set)  # Output: set()  (an empty set)
```

#### 7. **`update()`**: Adding Multiple Elements to a Set

The `update()` method is used to add multiple elements (from another set, list, or any iterable) to the set.

```python
my_set = {1, 2, 3}
my_set.update([4, 5, 6])  # Adding multiple elements from a list
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}

# You can also update using another set:
my_set.update({7, 8})
```

```
print(my_set)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

### **Summary of Set Methods**

- **`set(iterable)`**: Creates a set from an iterable (like a list or tuple).

- **`add(element)`**: Adds a single element to the set.

- **`pop()`**: Removes and returns an arbitrary element from the set.

- **`remove(element)`**: Removes a specific element from the set (raises `KeyError` if not found).

- **`discard(element)`**: Removes a specific element from the set (no error if the element is not found).

- **`clear()`**: Removes all elements from the set.

- **`update(iterable)`**: Adds multiple elements from an iterable to the set.

Sets are especially useful when you need to store unique items and perform set operations like union, intersection, and difference.

### **What is a Function in Python?**

A **function** in Python is a reusable block of code that performs a specific task. Functions allow you to divide your program into smaller, modular pieces, making the code more organized and reusable. A function can accept inputs (called **arguments**) and return a result using the **`return`** statement.

### **How to Create a Function in Python**

To create a function in Python, you use the `def` keyword followed by the function name, parentheses `()`, and a colon `:`. The code inside the function is indented.

#### Basic Structure of a Function

```python
def function_name(parameters):
    # Code block (function body)
    # Optionally return a value
    return result
```

### Example 1: Simple Function Without Return

This is a function that prints a greeting message.

```python
def greet(name):
    print(f"Hello, {name}!")

# Calling the function
greet("Alice")  # Output: Hello, Alice!
```

### Example 2: Function With Parameters and Return

This function takes two numbers as arguments, adds them, and returns the result using the `return` statement.

```python
def add_numbers(a, b):
    return a + b

# Calling the function
result = add_numbers(5, 10)
print(result)  # Output: 15
```

### Why Do We Use `return` in Python?

The **`return`** statement is used to exit a function and pass a value back to the caller. It allows a function to produce an output that can be used elsewhere in the program.

#### Reasons for Using `return`:

1. **To pass back the result of a computation**: If you want to get a value from a function after some calculations or operations, you use `return`.

2. **To end the function's execution**: Once a function encounters a `return`, it stops executing further code.

### Example 1: Using `return` to Pass Back a Value

Here’s an example where a function calculates the area of a rectangle and returns the result:

```python
def calculate_area(length, width):
    area = length * width
    return area

# Using the returned value
rectangle_area = calculate_area(5, 3)
print(rectangle_area)  # Output: 15
```

In this example, the function **returns** the area, which is then stored in the variable `rectangle_area`.

### Example 2: Using `return` to Control Flow

Here’s a function that checks if a number is even or odd. It uses `return` to immediately return the result.

```python
def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False

# Using the returned value
result = is_even(10)
print(result)  # Output: True
```

In this case, the function returns `True` if the number is even and `False` if it’s odd. The `return` statement helps the function provide an output that can be used in further logic.

### Summary:

- **Functions** allow code reuse and organization.

- Use `def` to define a function.

- The **`return`** statement allows a function to send back a result or value to the caller, which is useful for further calculations or decision-making.

### What is a User-Defined Function in Python?

A **user-defined function** is a function that is created by the programmer to perform a specific task. Unlike built-in functions (e.g., `print()`, `len()`, etc.) which come predefined in Python, user-defined functions are created to handle custom logic based on the needs of your program.

User-defined functions can take arguments, perform operations, and return results using the `return` statement. They help in organizing code, making it reusable, and reducing redundancy.

### How to Create a User-Defined Function?

You define a function in Python using the `def` keyword, followed by the function name, parentheses (which may contain parameters), and a colon. The code inside the function is indented, and the function can optionally return a value.

### Structure of a User-Defined Function

```python
def function_name(parameters):
    # Code block (function body)
    # Optionally return a value
    return result
```

### Example 1: A Simple User-Defined Function

This is a user-defined function that greets a person by name.

```python
def greet(name):
    print(f"Hello, {name}!")

# Calling the function
greet("Alice")  # Output: Hello, Alice!
```

### Example 2: User-Defined Function with a Return Value

Here’s a user-defined function that calculates the square of a number and returns the result.

```python
def square(number):
    return number * number

# Calling the function
result = square(5)
print(result)  # Output: 25
```

### Why Use User-Defined Functions?

1. **Code Reusability**: Once a function is defined, it can be called multiple times without rewriting the same code.

2. **Modularity**: Functions help break a complex problem into smaller, manageable parts.

3. **Readability**: Functions make your code cleaner and easier to read.

4. **Debugging**: Functions can be tested separately, making debugging easier.

### Key Points About User-Defined Functions:

- **Parameters**: Functions can accept input values (parameters) to perform operations on them.

- **Return Value**: Functions can return a value using the `return` statement.

- **Calling a Function**: To execute the function, you "call" it by using its name followed by parentheses, possibly passing arguments.

By defining your own functions, you can tailor Python to your specific needs and create a more efficient and organized program.

In Python, **call by value** and **call by reference** work differently than in some other programming languages. Python uses a mechanism called **call by object reference** (or **call by sharing**). This means that:

- **Immutable objects** (like integers, strings, and tuples) behave as if they are passed by **value**.

- **Mutable objects** (like lists, dictionaries, and sets) behave as if they are passed by **reference**.

Let’s look at examples for both behaviors.

### **Call by Value (for Immutable Objects)**

In the case of **immutable objects** like integers, strings, or tuples, modifying the value inside the function does **not** affect the original variable outside the function.

#### Example 1: Integer (Immutable)

```python

def modify_value(num):
```

    num = 10 # Modifying the number inside the function
    print("Inside function:", num)

x = 5
modify_value(x)
print("Outside function:", x) # Original value remains unchanged
```
**Output:**
```
Inside function: 10
Outside function: 5

#### Example 2: String (Immutable)
```python
def modify_string(s):
    s = "Hello, World!"  # Modifying the string inside the function
    print("Inside function:", s)

my_str = "Hello"
modify_string(my_str)
print("Outside function:", my_str)  # Original string remains unchanged
```
Output:
```
Inside function: Hello, World!
Outside function: Hello
```

#### Example 3: Tuple (Immutable)
```python
```

```python
def modify_tuple(tup):
    tup = (4, 5, 6)  # Modifying the tuple inside the function
    print("Inside function:", tup)


my_tuple = (1, 2, 3)
modify_tuple(my_tuple)
print("Outside function:", my_tuple)  # Original tuple remains unchanged
```
Output:
```
Inside function: (4, 5, 6)
Outside function: (1, 2, 3)
```

### **Call by Reference (for Mutable Objects)**

For **mutable objects** like lists, dictionaries, and sets, the changes made inside the function affect the original object outside the function because they share the same reference.

#### Example 1: List (Mutable)
```python
def modify_list(lst):
    lst.append(4)  # Modifying the list inside the function
    print("Inside function:", lst)


my_list = [1, 2, 3]
modify_list(my_list)
print("Outside function:", my_list)  # The original list is modified
```
Output:
```
```

Inside function: [1, 2, 3, 4]

Outside function: [1, 2, 3, 4]

#### Example 2: Dictionary (Mutable)

```python
def modify_dict(d):
    d['new_key'] = 'new_value'  # Modifying the dictionary inside the function
    print("Inside function:", d)

my_dict = {'a': 1, 'b': 2}
modify_dict(my_dict)
print("Outside function:", my_dict)  # The original dictionary is modified
```

Output:

```
Inside function: {'a': 1, 'b': 2, 'new_key': 'new_value'}
Outside function: {'a': 1, 'b': 2, 'new_key': 'new_value'}
```

#### Example 3: Set (Mutable)

```python
def modify_set(s):
    s.add(4)  # Modifying the set inside the function
    print("Inside function:", s)

my_set = {1, 2, 3}
modify_set(my_set)
print("Outside function:", my_set)  # The original set is modified
```

Output:

```
Inside function: {1, 2, 3, 4}
Outside function: {1, 2, 3, 4}
```

### **Key Takeaways:**

- For **immutable objects** (integers, strings, tuples), changes inside the function do not affect the original value outside the function.

- For **mutable objects** (lists, dictionaries, sets), changes made inside the function will affect the original object because they share the same reference.