

In Python, dictionaries are versatile data structures that store key-value pairs. When it comes to comparison, dictionaries can be compared for equality. Two dictionaries are considered equal if they have the same set of keys and the corresponding values for each key are equal.

Here's an example:

- dict1 = {'a': 1, 'b': 2, 'c': 3}

```
dict2 = {'a': 1, 'b': 2, 'c': 3}
```

```
if dict1 == dict2: print("Dictionaries are equal.")
```

```
else:
```

```
print("Dictionaries are not equal.")
```

Since in dict1 and dict2, the corresponding values for each key are equal therefore the output comes out to be Dictionaries are equal.

Two tuples are considered equal if their elements are equal in order. Here's an example:

- # Comparing tuples

```
tuple1 = (1, 2, 3)
```

```
tuple2 = (1, 2, 3)
```

```
if tuple1 == tuple2: print("Tuples are equal.")
```

```
else:
```

```
print("Tuples are not equal.")
```

Given that tuple1 and tuple2 have identical elements in the same order, the output of this code would be:

- Tuples are equal

consider the scenario where the positions of elements in tuple2 are changed to (3, 2, 1):

- # Comparing tuples

- tuple1 = (1, 2, 3)

```
tuple2 = (3, 2, 1)
```

```
if tuple1 == tuple2: print("Tuples are equal.")
```

```
else:
```

```
print("Tuples are not equal.")
```

In this case, the output would be:

Tuples are not equal.

This happens because the order of elements in tuple2 no longer matches the order in tuple1. The first element of tuple1 is 1, but the first element of tuple2 is 3. Since the

elements are compared in order, and there is a mismatch at the first position, the tuples are considered not equal. The order of elements is crucial when comparing tuples for equality.

The scenario where the order of elements matters, as discussed in the context of tuples and lists, is different when it comes to sets.

Sets in Python are unordered collections, meaning the elements are not stored in any specific order. Consequently, when comparing sets, the order of elements is not a factor.

Here we are creating two sets with the same elements but in different orders

- set1 = {1,2, 3}
- set2 = {3, 2, 1}

- # Comparing sets

- if set1 == set2
- print("Sets are equal.")
- else:
- print("Sets are not equal.")

The output to this code will be:

Sets are equal.

So the equality comparison for sets focuses on whether they contain the same elements, not on the order in which the elements are specified.

So, for lists, tuples, and strings, elements are required to be in the same order for equality; changing the order of elements in them results in inequality.

In contrast, for sets and dictionaries, the order of elements is not a determining factor for equality; they are considered equal as long as their contents or key-value pairs match, respectively, regardless of the order in which they are specified.

Python's logical operators, including and, or, and not, enable the combination of multiple conditions in code, allowing for the creation of more intricate decision-making criteria.

So in our first example, the message is printed only if the age is greater than 18 and the person is not a student.

- age = 25

```
is_student = False
```

```
if age > 18 and not is_student: print("Eligible for voting.")
```

Here, two variables are defined: age is assigned the value 25, and is\_student is assigned the boolean value False.

Then the if statement checks a condition. The condition is composed of two parts connected by the “and” logical operator:

Where the first part is age > 18 and this part checks whether the value of the age variable is greater than 18.

The second part is not is\_student: This part uses the “not” logical operator to negate the value of is\_student, so it checks whether the person is not a student.

The “and” operator requires both conditions on its sides to be true for the overall condition to be True. Therefore, the code inside the if block will be executed only if the person is both older than 18 and not a student.

In this case, since the age is 25 (greater than 18) and is\_student is False, both conditions are satisfied. The message "Eligible for voting." will be printed.

Now let's consider an example where the user enters a day, and the program checks whether it's the weekend or not:

- day = input("Enter a day of the week: ")

if

    day == Saturday

    or

    day == Sunday

    print("It's the weekend!") else:

    print("Not a weekend.")

After running this code, let's say the user enters "Monday":

Since the input is "Monday," the condition day == "Saturday" or day == "Sunday" evaluates to False because the entered day is neither Saturday nor Sunday. Therefore, the program prints "Not a weekend."

Now let's say the user enters 'saturday' (type it with a lowercase s'), the output would be "Not a weekend."

The reason for this is although "saturday" and "Saturday" may be considered the same in English language usage, but Python is a case-sensitive language, meaning it distinguishes between uppercase and lowercase letters. Therefore, 's' and 'S' are considered distinct.

So it doesn't match the condition day == "Saturday" or day == "Sunday". Consequently, the program prints 'Not a weekend.'

Now let's say the user enters 'Saturday'. The condition day == "Saturday" or day == "Sunday" evaluates to True because "Saturday" satisfies the first part of the condition. The or operator means that if at least one of the conditions is True, the entire expression is True.

So the program prints "It's the weekend!"

These logical operators can also be combined to create more sophisticated conditions. For example:

- age = 25

```
is_student = False has_job = True  
if age > 18 and not is_student or has_job:  
    print("Eligible for some opportunities.")
```

In this code, the if statement checks a condition using logical operators. The condition is: age > 18 and not is\_student or has\_job.

age > 18: Checks if the value of age is greater than 18.

not is\_student: Negates the value of is\_student, checking if the person is not a student. has\_job: Checks if the person has a job.

So due to order of precedence, the not operation is evaluated first, followed by “and”, and finally “or”.

Now let's evaluate the conditions step by step:

age > 18: True (since 25 is greater than 18).

not is\_student: True (because not False is True).

age > 18 and not is\_student: True (both conditions are True). has\_job: True.

Now, the entire expression is a combination of the evaluated conditions using “and” and “or”.

Therefore, the code block within the if statement will be executed, and the output will be:

- Eligible for some opportunities

Now, let's explore the simplicity of the ternary operator in Python. It condenses if-else statements into a single line, making your code more straightforward and readable.

The basic syntax of the ternary operator in Python is as follows:

- result\_if\_true

```
if  
condition  
else  
result_if_false
```

Here condition is the expression that is evaluated. If it's true, the ternary operator returns result\_if\_true; otherwise, it returns result\_if\_false.

Let's understand this with an example

- age = 20

```
eligibility_status = "Eligible" if age >= 18 else "Not eligible"  
print(eligibility_status)
```

In this example, the condition age  $\geq 18$  is true, so the variable eligibility\_status is assigned the value "Eligible" and consequently "Eligible" is printed.

The ternary operator can also be used for nested conditions. The syntax remains the same, but you can include another ternary operator within the expressions for result\_if\_true and result\_if\_false. Here's an example:

- score = 75

```
result = "Pass" if score >= 50 else ("Retake" if score >= 40 else "Fail")  
print(result)
```

In this example:

If the score is greater than or equal to 50, the result is "Pass."

If the score is between 40 and 49 (inclusive), the result is "Retake." Otherwise, the result is "Fail."

And here the score is 75, which is greater than 50. So, the first part of the ternary operator is chosen, and the result becomes "Pass."

It's important to keep the nested ternary operators readable, and in some cases, using regular if-else statements might be more appropriate for complex conditions.

Weekend activity planner! In this code, we've created a simple Python script that engages with the user to plan an exciting weekend. Now, let's dive into the details of this code and explore how it engages with user input to create a personalized and interactive experience. Here I have written the code beforehand and will walk you through it.

(In Python)

```
name = input("Enter your name: ")  
print("Hello, " + name + "! Let's plan an exciting weekend activity.")  
print("You have a free weekend ahead. How would you like to spend it?") print("1: Go for a hike in the mountains")  
print("2: Attend a music concert in the city") print("3: Have a relaxing day at the beach")  
choice = input("Enter the number corresponding to your preferred activity (1, 2, or 3): ") if choice == "1":  
    print("What difficulty level of hike do you prefer?") print("a: easy")  
    print("b: moderate") print("c: challenging")  
    difficulty_level = input("Enter the number corresponding to your difficulty level (a,b, or c")  
    ")  
    if difficulty_level == "a":  
        print("You enjoyed a scenic hike with breathtaking views.") elif difficulty_level == "b":  
        print("The moderate hike provided a good balance of challenge and enjoyment.") elif difficulty_level == "c":  
        print("You conquered a challenging hike and felt a great sense of accomplishment.") else:  
        print("Invalid difficulty level choice.") elif choice == "2":
```

```
genre = input("What music genre do you prefer? ('rock', 'pop', 'jazz'): ")
print(f"You attended a {genre} concert and had a fantastic time.")

elif choice == "3":
    print("You spent a relaxing day at the beach, enjoying the sun, sand,
and waves.") else:
    print("Not a valid choice. Please enter either '1', '2', or '3' based on
your preferred activity.")

print("Thank you for planning your weekend with us, " + name + "!")
```

The program begins by asking for the user's name and then presents a menu of weekend activities, allowing the user to choose between a mountain hike, a city concert, or a beach day.

If the user opts for a mountain hike, they are prompted to select the difficulty level of the hike, with options ranging from easy to challenging. Depending on their choice, the program provides a corresponding outcome, capturing the essence of the hiking experience.

Alternatively, if the user selects a music concert in the city, they are asked to specify their preferred music genre (rock, pop, or jazz). The program then acknowledges their choice and expresses that they had a fantastic time at the concert.

For those who prefer a more relaxed weekend by the beach, the program provides a description of a serene day spent enjoying the sun, sand, and waves.

The code concludes by thanking the user for planning their weekend and mentioning their name.

Output 1:

Here we have entered " Enter your name: Arya" Hello, Arya! Let's plan an exciting weekend activity.

You have a free weekend ahead. How would you like to spend it? 1: Go for a hike in the mountains

2: Attend a music concert in the city 3: Have a relaxing day at the beach

Enter the number corresponding to your preferred activity (1, 2, or 3):  
1 (Entering 1 ) What difficulty level of hike do you prefer?

a: easy

b: moderate c: challenging

Enter the number corresponding to your difficulty level (a,b, or c) b  
(Entering b) The moderate hike provided a good balance of challenge  
and enjoyment.

Thank you for planning your weekend with us, Arya!

Output 2:(Entering value as Ayush) Enter your name: Ayush

Hello, Ayush! Let's plan an exciting weekend activity.

You have a free weekend ahead. How would you like to spend it? 1:  
Go for a hike in the mountains

2: Attend a music concert in the city

3: Have a relaxing day at the beach

Enter the number corresponding to your preferred activity (1, 2, or 3):  
2 (Entering 2) What music genre do you prefer? ('rock', 'pop', 'jazz'):  
jazz (Entering jazz)

You attended a jazz concert and had a fantastic time. Thank you for  
planning your weekend with us, Ayush!

Let's understand the last concept in this segment which is truthy and  
falsy values in Python. In simple terms, a truthy value is one that is  
considered logically true when used in a boolean expression, and a  
falsy value is one that is considered logically false.

Let's understand this with an easy example:

- a = 1

if a:

```
print(a)
```

In this case, a is assigned the value 1. When used in the if statement,  
Python considers 1 to be truthy, so the code inside the if block is  
executed. The output will be:

1

Now, let's suppose we change the value of a to 0:

- a = 0

if a:

```
print(a)
```

In this case, 0 is considered falsy in Python. So, when used in the if statement, the condition evaluates to false, and the code inside the if block is not executed. Therefore, there is no output.

In Python, values that are generally considered falsy include

### **1. Sequences and Collections:**

Empty lists [] Empty tuples ()

Empty dictionaries {} Empty sets set() Empty strings " " Empty ranges range(0)

### **2. Numbers: Zero of any numeric type**

Integer: 0

Float: 0.0

Complex: 0j

### **3. Constants:**

None False

Truthy Values Include:

Non-empty sequences or collections (lists, tuples, strings, dictionaries, sets). Numeric values that are not zero.

Constant: True