Let's now delve into the significance of the order of arithmetic operations in determining the accurate results of mathematical expressions.

In Python, arithmetic operators perform various mathematical operations on numeric values.

You can add, subtract, multiply, and divide numbers using these operators. Now, here's the interesting part: the order in which you use these tools matters.

Imagine you have a set of instructions for a math problem. It's like following a recipe. You can't just mix everything together; there's a specific order.

The same goes for math in Python!

Let's say you have an expression like 5 + 3 * 2. Should you add first or multiply first? The order matters!

The order of arithmetic operations in Python is similar to the rules you learned in mathematics, where certain operations take precedence over others. The acronym PEMDAS is often used to remember the order of operations:

And PEMDAS stands for:

- P: Parenthesesfirsti.e.Operationsinsideparenthesesareperformedfirst.
- E for:Exponentslikepowersandsquareroots
- And then comes Multiplication (*) and Division (/) where Multiplication and division have the same precedence level and are performed from left to right
- Suppose we have the expression 8 / 2 * 3. According to the left-to-right rule for multiplication and division:
- Start from the left: 8 / 2 is first evaluated, resulting in 4. Then, multiply by 3: 4 * 3 is next, resulting in 12.

-
- And last is Addition (+) and Subtraction (-): addition and subtraction have the same level of precedence and are performed from left to right.

Let's look at an example to illustrate the order of arithmetic operators: (In Python)

- result= 5 +3 * 2- (8 / 4)** 2

Here, the order of evaluation would be as follows:

Inside the parentheses: (8 / 4) is evaluated first, resulting in 2. Exponentiation: (2) ** 2 is evaluated next, resulting in 4.

Multiplication: 3 * 2 is evaluated, resulting in 6.

Addition and Subtraction: Finally, 5 + 6 - 4 is evaluated, resulting in 7.

Now let’s print result

- print(result)(Showtheoutput)

So, the final value of the result would be 7.

Note that all the operators, except exponentiation(**), follow the left-to-right associativity. It means the evaluation will proceed from left to right while evaluating the expression.

Let's consider an example to illustrate that exponentiation does not follow left-to-right associativity:

(In Python)

- result=2 **3 **2

In this expression, the exponentiation operator (**) is used twice. First, 3 ** 2 is evaluated, resulting in 9.

Then, 2 ** 9 is evaluated, resulting in 512.

This highlights that the operation is performed from right to left when using exponentiation. In mathematics, exponentiation is often written from right to left.

Now, let's delve into more aspects of assignment operators in Python.

In Python, you can also assign values to multiple variables in a single line. For example: (In Python)

- a, b, c = 1, 2, 3

  print(a)

  print(b)

  print(c)

  # Output:

  1

  2

  3

This is a concise way to initialize multiple variables at once. Next, we have is Chained Assignments.

Python supports chained assignments, allowing you to assign the same value to multiple variables in one line. For example:

(In Python)

- x = y = z = 10

  print(x, y, z)

  # Output: 10 10 10

Here, x, y, and z are all assigned the value 10 in a single line.

Assignment operators are also used for swapping the values of two variables without the need for a temporary variable. Let’s see how:

- a = 5

  print(f'old value of a is {a}') # Output: 5

  b = 10

  print(f'old value of b is {b}') # Output: 10 # Swapping values

  a, b = b, a

print(f'new value of a is {a}') # Output: 10

print(f'new value of b is {b}') # Output: 5

In this example, the values of a and b are initially set to 5 and 10, respectively. After the swap, the values are printed, demonstrating the updated values of both variables.

To swap values among three variables without using a temporary variable, you can use a similar approach with multiple assignments. Here's an example:

- # Swapping values among three variables

  a = 5

  print(f'old value of a is {a}')

  b = 10

  print(f'old value of b is {b}')

  c = 15

  print(f'old value of c is {c}')

  # Swapping values a, b, c = b, c, a

  print(f'new value of a is {a}') # Output: 10 print(f'new value of b is {b}') # Output: 15 print(f'new value of c is {c}') # Output: 5

In this example, the values of a, b, and c are initially set to 5, 10, and 15, respectively. The line a, b, c = b, c, a swaps the values among the three variables.

The values on the right side (b, c, a) are assigned to the variables on the left side (a, b, c) in order.

The sequence of assignments occurs from left to right:

a is assigned the value of b which is (10). b is assigned the value of c which is (15). c is assigned the value of a which is (5).

After this line, the values of a, b, and c have effectively been swapped. So, the new values after the swap are:

a is now 10 b is now 15 c is now 5

Assignment operators can also be used with strings to concatenate and assign values. Let’s see an example

- greeting = "Hello"

  print(greeting)

  greeting += " World"

  print(greeting)

  # Output: Hello World

Here, the += operator known as the augmented assignment operator is used for string concatenation. It adds the string " World" to the existing value of the greeting variable. After this line is executed, the greeting variable now contains the string "Hello World."

So the first print prints the original value of the greeting and the second print prints the modified value which is "Hello World".

Assignment operators can also be used within larger expressions to update variables. For example:

x = 5

x = x * 2 + 1 # Equivalent to x *= 2 + 1 print(x) # Output: 11

Initially, the value of x is assigned as 5, and then later the value of x is modified as x * 2 + 1, which means "double the current value of x and then add 1.

At this point, x becomes 11 because 5 * 2 + 1 equals 11.

And x *= 2 + 1 statement is a concise way of expressing the operation x = x * 2 + 1.

Apart from numbers, augmented assignment operators work with other data types, like lists. Let’s see how with an example:

- my_list=[1,2,3]

  my_list += [4, 5] # Equivalent to my_list = my_list + [4, 5] print(my_list)

  # Output: [1, 2, 3, 4, 5]

Here my_list is initially assigned the values [1, 2, 3]. The line my_list += [4, 5] uses the augmented assignment operator += to append the elements [4, 5] to the existing list my_list. And here my_list += [4, 5] is Equivalent to my_list = my_list + [4, 5]

After the operation, my_list contains the extended sequence [1, 2, 3, 4, 5].

Now let’s see how using an assignment operator we can add a new key-value pair to the dictionary.

- my_dict={'a':1,'b': 2}

  my_dict['c'] = 3 # Adding a new key-value pair print(my_dict)

  # Output: {'a': 1, 'b': 2, 'c': 3}

Here, a new key-value pair 'c': 3 is added to the dictionary my_dict. This is done by assigning the value 3 to the key 'c'.

So, the updated dictionary with the new key-value pair is (Show the output)

{'a': 1, 'b': 2, 'c': 3}

Continuing, we'll now delve into more details regarding relational operators in Python.

Python allows you to chain multiple comparison operators together in a single expression. This can make your code more concise. For example:

- x = 5

  y = 10

  z = 15

result = x < y < z print(result)

Output: True

In this example, the expression x < y < z is True because both comparisons x < y and y < z are true.

Let’s see another example where we prompt the user to enter a value and check if the value is greater than or equal to 18 and less than 30.

(In Python)

- user_value=int(input("Enteravalue:"))

  # Check if the value is greater than or equal to 18 and less than 30 is_between_18_and_30 = 18 <= user_value < 30

  # Print the result

  print(f"Is the value between 18 (inclusive) and 30? {is_between_18_and_30}")

Let’s say the user enters the value as 22, so the output will be:

Is the value between 18 and 30? True

This is correct because 22 is indeed greater than or equal to 18 and less than 30.

Relational operators are not limited to numeric types; they can be used with other data types as well. For example:

- string1="apple"string2 = "banana"result=string1<string2print(result)

And the Output will be True.

Because the comparison starts with the first characters of each string. In this case, 'a' from "apple" and 'b' from "banana". Since 'a' comes before 'b' in the alphabet, the result of string1 < string2 is True. The comparison stops at the first differing character.

If the first characters are the same, the second characters are compared, and so on, until a difference is found or one of the strings

is exhausted. If one string is a prefix of the other, the shorter string is considered less than the longer one.

In this specific example, the result is True because "apple" comes before "banana" in lexicographical order.

Let's consider one more example. (In Python)

- word = "python"result5='a'<word<'z'print(result5)

The Output will be True.

It is so because the comparison 'a' < word < 'z' is checking whether the string variable word falls between the strings 'a' and 'z' in lexicographical (dictionary) order.

'a' < word part checks if the string 'a' comes before the string stored in the variable word which is 'python'. In this case, 'a' comes before 'p', so this part of the comparison is True.

word < 'z' part checks whether the string stored in the variable word comes before the string 'z'. In this case, 'p' comes before 'z,' so this part of the comparison is also True.

Since both parts of the chained comparison are True, the overall result of 'a' < word < 'z' is True.

Now let’s check if the two lists are equal or not.

- list1=[1,2, 3]

  list2 = [1, 2, 3]

  # Using the equality operator (==) to check if the lists are equal
  are_lists_equal = list1 == list2

  # Print the result

  print(f"Are the two lists equal? {are_lists_equal}")

In this example, the equality operator (==) is used to compare two lists, list1 and list2. If the elements in both lists are the same and in the same order, the result will be True. If there is any difference, the result will be False.

Since list1 and list2 have the same elements in the same order, the result of list1 == list2 is True.

Now, let's compare two lists.

- a = [1, 2, 3]
- b = [1, 3, 2]
- print(a>b)

In this case, the output will be False. The comparison of lists is also done element-wise. We start by comparing the first elements: 1 in a vs. 1 in b. They are equal, so we move to the next elements.

Then we compare the second elements: 2 in a vs. 3 in b. Since 2 is less than 3, the comparison stops even though the elements are the same. Therefore, the result is False.

Moving forward, let's investigate further aspects of logical operators in Python.

Consider the following example (In Python)

- a = 15
- b = 25
- c = 5

result = (a > b) or ((b < c) and (c > 0)) print(result)

Here the variables a,b, and c are assigned values 15, 25, and 5 respectively.

The expression involves logical operators (“or” and “and”) and comparison operators (>, <). (a > b) is False because 15 is not greater than 25.

(b < c) is also False because 25 is not less than 5.

(c > 0) is True because 5 is greater than 0.

Now, the expression becomes False or (False and True).

The “and” operation is evaluated first, so False and True are False. The final expression is now False or False.

The “or” operation is evaluated, and False or False is False.

So the order of precedence of logical operators in Python, from highest to lowest, is as follows:

firstly, not has the highest precedence. secondly, and is evaluated next and Finally, or is evaluated.

Let’s consider an example demonstrating the order of precedence of logical operators

- x = True

  y = False

  z = True

  result= x or not y and z print(f"Result : {result}")

In this example, the logical expression x or not y and z involves the use of the “or”, “not”, and “and” operators.

So, the expression is evaluated as follows:

First, we evaluate not y:

And thus not False evaluates to True.

So, the expression becomes: x or True and z. Next, we evaluate True and z:

True and True evaluates to True.

So, the expression becomes: x or True. Lastly, we evaluate x or True:

Since the or operator returns True if at least one operand is True, the final result is True. Let’s see another example:

- print(not(FalseorTrue))

In this example, the innermost parentheses (False or True) are evaluated first. After the innermost parentheses are evaluated, the not operator is applied to the result.

Even though not have a higher order of precedence than or, the innermost parentheses are evaluated first because parentheses

have the highest precedence in Python. It ensures that operations within parentheses are performed before other operations.

let's discuss the concept of "interning" in Python, where small integers and some strings are "interned" and stored in a shared memory pool.

Let’s see an example (In Python)

- a = 256 b=256

  print(a is b)

  print(id(a))

  print(id(b))

  Firstly a and b variables are assigned the value 256.

print(a is b) checks if a and b refer to the same memory location. For small integers, Python may "intern" them, meaning they share the same memory space. So, a is b is True.

print(id(a)) and print(id(b)) show the memory location of a and b, and you'll see the same ID because they are sharing the same space.

So Python internally "interns" (reuses the same memory space) for small integers to save memory and improve efficiency. This means that for common small integers, like numbers between -5 and 256, Python typically uses the same memory space for the same value.

And Python may not intern large integers in the same way it does for small integers. Each occurrence of a large integer may have its own unique memory space.

Continuing, we'll delve into more details regarding membership operators as discussed in the previous chapter.

Membership operators are not limited to lists; they can also be used with other sequence types, including strings. This can be particularly useful for checking if a substring exists within a larger string. Let’s see an example:

(In Python)

- sentence = "Python is a powerful programming language."

  print("Python" in sentence) # True

  print("Java" in sentence) # False

print("Java" in sentence) # False

First, a string variable named sentence is defined and assigned a value. It then checks if the substring "Python" is present in the variable sentence. The in operator returns True if the substring is found, and False otherwise. In this case, since "Python" is indeed present in the sentence, so the output will be True.

print("Java" in sentence) checks if the substring "Java" is present in the sentence. Since "Java" is not part of the original sentence, the membership test evaluates to False, and the output will be False.

fruits = ["apple", "banana", "orange", "grape"] print("apple" in fruits) # will be True

In this example, fruit is a list containing four strings: "apple", "banana", "orange", and "grape". The statement "apple" in fruits checks whether the string "apple" is a member of the fruits list. The in operator is a membership operator in Python, and when used with lists, it checks if the specified element is present in the list. If the element is found, it returns True; otherwise, it returns False.

In this specific case, the output of the print("apple" in fruits) statement will be True because "apple" is one of the elements in the fruits list.

print(["banana" in fruits and "grape" in fruits)

will be True as both "banana" and "grape" are in the list

In this, the “and” logical operator is used to check if both conditions are true: "apple" in fruits: Checks if "apple" is present in the fruits list.

"banana" in fruits: Checks if "banana" is present in the fruits list.

If both conditions are true, the print statement will output True. Otherwise, it will output False. In this case, the output will be True because both "apple" and "banana" are elements in the fruits list.

print(["banana" in fruits and "kiwi" in fruits) #False

This line of code checks two conditions using the “and” logical operators:

"banana" in fruits: Checks if the string "banana" is present in the fruits list. In this case, it is true because "banana" is one of the elements in the list.

"kiwi" in fruits: Checks if the string "kiwi" is present in the fruits list. In this case, it is false because "kiwi" is not one of the elements in the list.

The “and” operator requires both conditions to be true for the entire expression to be true. Since the second condition is false, the entire expression evaluates to false.

Therefore, the output of the print statement will be False.