



Formatting Output

004_Python

“Old Style” String Formatting (% Operator)

Strings in Python have a unique built-in operation that can be accessed with the “%” (percentage sign) operator.

This lets you do simple positional formatting very easily.

(If you’ve ever worked with a print f-style function in C, you’ll recognize how this works instantly).

The “%” (*percentage sign*) operator is used to format a set of variables enclosed in a “**tuple**” (**a fixed size list**), together with a format string, which contains normal text together with “*argument specifiers*”, special symbols like “%s” (percentage S) and “%d” (percentage D) .

“Old Style” String Formatting (% Operator)

Let's say you have a variable called "name" with your name in it,
and you would then like to print out a greeting to that user.

```
# This prints out "Hello, John!"  
name = "John"  
print("Hello, %s!" % name)
```

Hello, John!

I'm using the %s format specifier here, to tell Python where to place the string; "name".

There are other format specifiers available that let you control the output format. Other than the "%s" for strings, we have "%d" for decimal integers (Not floating point)

Using two or more arguments

To use two or more argument specifiers, use a tuple (parentheses):

```
# This prints out "John is 23 yesrs old"  
name = "John"  
age = 23  
print("%s is %d years old." % (name, age))
```

Type

Escape	Description	Example
%d	Decimal integers (not floating point)	"%d" % 45==45
%i	Same as %d	"%i" % 45==45
%o	Octal number	"%o" %10 ==12
%X or %x	Hexadecimal	"%x" %10 == a
%e or %E	Exponential notation	"%e" %100 ==1.000000e+01
%f or %F	Floating point real number	"%f" % 10.10==10.100000
%c	Character format	"%c" %65 ='A'
%s	String format	"%s there" %'Hi' == "Hi there"

% (String Formatting(modulo) Operator)

Formats the string according to the specified format.

The general syntax for a format placeholder is

`%[flags][width][.precision]type`

Conversion Flags

`'0'`

The conversion will be zero padded for numeric values.

`'-'`

The converted value is left adjusted (overrides the '0' conversion if both are given).

`' '`

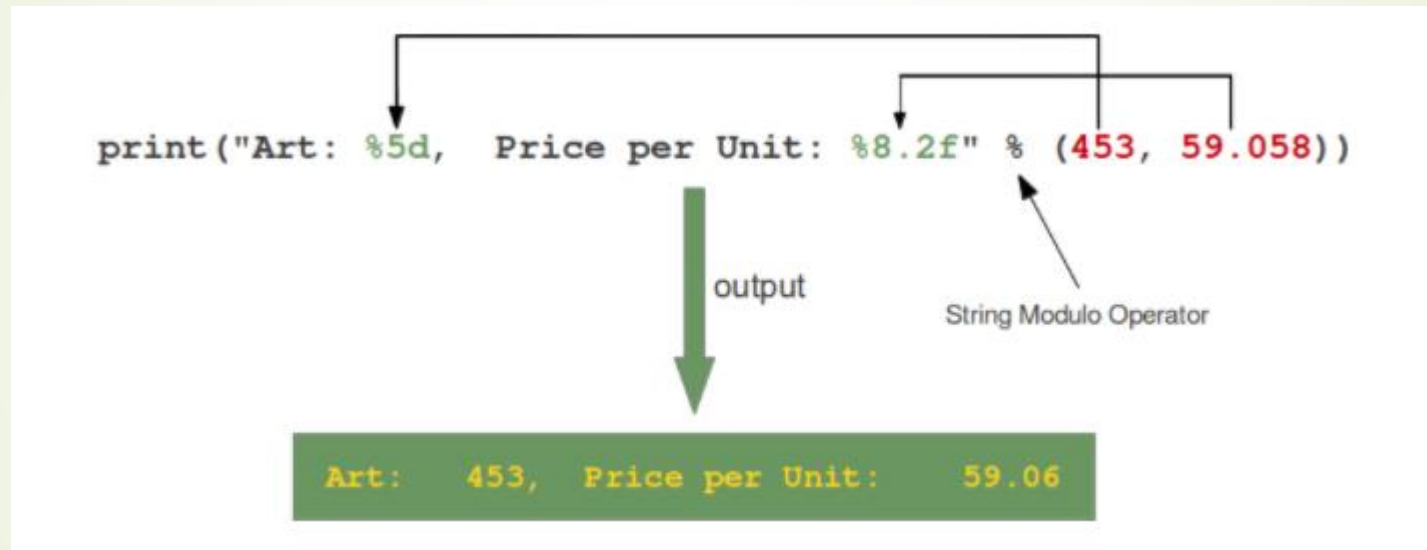
(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

`'+'`

A sign character ('+' or '-') will precede the conversion (overrides a "space" flag).

% (String Formatting(modulo) Operator)

The following diagram depicts how the string modulo operator works:



The format string contains placeholders. There are two of those in our example: "%5d" and "%8.2f".

%(String Formatting(modulo)Operator)

On the left side of the "*string modulo operator*" there is the so-called "*format string*"

On the right side is a "**tuple**" with the content, which is interpolated in the format string.

The values can be literals, variables or arbitrary arithmetic expressions.

```
print("Art: %5d, Price per Unit: %8.2f" % (453, 59.058))
```

The diagram illustrates the components of the string formatting operation in the provided code snippet. A horizontal line with a downward-pointing bracket spans the text "Art: %5d, Price per Unit: %8.2f", which is labeled "Format String" below. To the right of this, a green arrow points upwards to the percent sign "%", which is labeled "String Modulo Operator" below. Further to the right, another horizontal line with a downward-pointing bracket spans the text "(453, 59.058)", which is labeled "Tuple with values" below.

Basic formatting

Simple positional formatting is probably the most common use-case. Use it if the order of your arguments is not likely to change and you only have very few elements you want to concatenate.

Since the elements are not represented by something as descriptive as a name this simple style should only be used to format a relatively small number of elements.

Old

```
'%s %s' % ('one', 'two')
```

New

```
'{} {}'.format('one', 'two')
```

Output

```
o n e   t w o
```

Old

```
'%d %d' % (1, 2)
```

New

```
'{} {}'.format(1, 2)
```

Output

```
1   2
```

Index formatting

- ▶ Index formatting is not possible in the old-style formatting

New

```
'{1} {0}'.format('one', 'two')
```

Output

```
t w o   o n e
```

Padding and aligning strings

By default, values are formatted to take up only as many characters as needed to represent the content.

It is however also possible to define that a value should be padded to a specific length.

Unfortunately, the default alignment differs between old and new style formatting.

The old-style defaults to right aligned while for new style it's left.

Align right:

Old `'%10s' % ('test',)`

New `'{:>10}'.format('test')`

Output `test`

Align left:

Old `'%-10s' % ('test',)`

New `'{:10}'.format('test')`

Output `test`

Padding and aligning Strings

Again, new style formatting surpasses the old variant by providing more control over how values are padded and aligned.

You can choose the padding character:

This operation is not available with old-style formatting.

New `'{: _<10}'.format('test')`

Output `t e s t _ _ _ _ _`

And also center align values:

This operation is not available with old-style formatting.

New `'{: ^10}'.format('test')`

Output `t e s t`

Padding and aligning Strings

When using centre alignment where the length of the string leads to an uneven split of the padding characters the extra character will be placed on the right side:

This operation is not available with old-style formatting.

New

```
'{: ^6}'.format('zip')
```

Output

```
z i p
```

Truncating long strings

Inverse to padding, it is also possible to truncate overly long values to a specific number of characters.

The number behind a ' .' in the format specifies the precision of the output.

For strings that means that the output is truncated to the specified length.

In our example this would be 5 characters.

Old

```
'%.5s' % ('xylophone',)
```

New

```
'{: .5}'.format('xylophone')
```

Output

```
x y l o p
```

Combining truncating and padding

It is also possible to combine truncating and padding:

Old

```
'%-10.5s' % ('xylophone',)
```

New

```
'{:10.5}'.format('xylophone')
```

Output

x	y	l	o	p					
---	---	---	---	---	--	--	--	--	--

Formatting Numbers

It is also possible to format numbers.

Integers:

Old `'%d' % (42,)`

New `'{:d}'.format(42)`

Output `4 2`

Floats:

Old `'%f' % (3.141592653589793,)`

New `'{:f}'.format(3.141592653589793)`

Output `3 . 1 4 1 5 9 3`

Specifying Width for numbers

Similar to strings numbers can also be set to a specific width.

Old

```
'%4d' % (42,)
```

New

```
'{:4d}'.format(42)
```

Output

```
  4 2
```

Specifying precision

Again, like truncating strings the precision for floating point numbers limits the number of positions after the decimal point. For a floating point, the padding value represents the length of the complete output.

In the example on the right, we want our output to have at least 6 characters with 2 after the decimal point.

Old

```
'%06.2f' % (3.141592653589793,)
```

New

```
'{:06.2f}'.format(3.141592653589793)
```

Output

```
0 0 3 . 1 4
```

Specifying precision for Integers

For integer values providing a precision, doesn't make much sense and is actually forbidden in the new style (it will result in a ValueError)

```
print("Employee Number : |%10.2d|" %(101))  
Employee Number : |      101|
```

```
print("Employee Number : |{0:10.2d}|".format(101))  
File "c:\Users\Angela\Desktop\PythonScripts\Classwork\format_method.py", line 19, in <module>  
    print("Employee Number : |{0:10.2d}|".format(101))  
ValueError: Precision not allowed in integer format specifier
```

Signed numbers

By default, only negative numbers are prefixed with a sign. This can be changed of course.

Old

```
'%+d' % (42,)
```

New

```
'{:+d}'.format(42)
```

Output

```
+ 4 2
```

Signed numbers

Use a “- ” minus character to indicate that negative numbers should be prefixed with a minus symbol

and a leading space should be used for positive ones.

Old `'% d' % ((- 23),)`

New `'{: d}'.format((- 23))`

Output `- 23`

Old `'% d' % (42,)`

New `'{: d}'.format(42)`

Output `42`

Signed numbers

New style formatting is also able to control the position of the sign symbol relative to the padding.

This operation is not available with old-style formatting.

New `'{:5d}'.format((- 23))`

Output `- 23`

New `'{:=5d}'.format(23)`

Output `+ 23`



Which Style to choose?

In Python 3, the “new style” string formatting is to be preferred over % - style formatting.

While “old style” formatting has been de-emphasized, it has not been deprecated. It is still supported in the latest versions of Python.



Operators

What is an Operator?

An operator is a symbol that performs an operation on one or more operands. An operand is a variable or a value on which we perform the operation.



Operator

Python Operator falls into 7 categories:

Operator		Operator	
Arithmetic Operator	(+, -, *, /, ...)	Membership Operator	(in, not in)
Relational Operator	(<, >, ==, ...)	Identity Operator	(is, is not)
Assignment Operator	(=, +=, ...)	Bitwise Operator	(&, , ~, ^)
Logical Operator	(and, or, not, ...)		

Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Note:

/ operator always performs floating point arithmetic. Hence it will always return float value.

But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If at least one argument is float type then result is float type.

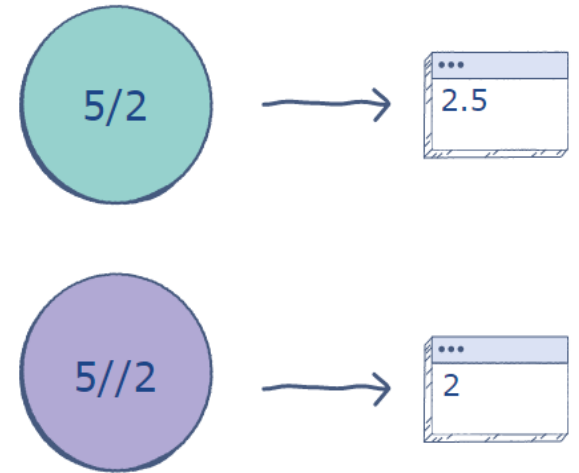
Arithmetic Operation	Description	Pseudocode Example	Python Example	Output
Addition	Adds two values together.	5 + 2	5 + 2	7
Subtraction	Subtracts the second value from the first value.	3 - 5	3 - 5	-2
Multiplication	Multiplies two values together.	4 * 3	4 * 3	12
Exponentiation	Calculates the power of a number.	7 ** 2	7 ** 2	49
Real Division	More commonly simply referred to as just " division ", it divides the first value by the second value.	15 / 2	15 / 2	7.5
Integer Division	Also known as " whole number division ", where the first value is divided by the second value but only shows only the digits to the left of the decimal point (the integers).	16 DIV 3	16 // 3	5
Modulus	This finds the remainder from an integer division. After an integer division has been performed (16 // 3) = 5 a multiplication is carried out by multiplying that answer by the second value (5 * 3) = 15. The difference between these two values is the remainder (16 - 15 = 1).	16 MOD 3	16 % 3	1

Types of division operators

In Python, there are two types of division operators:

`“ / ”` Divides the number on its left by the number on its right and **returns a floating point value**.

`“ // ”` Divides the number on its left by the number on its right, rounds down the answer, and **returns a whole number**. But even if one of the operand is a float, then the result is float.



Example - Division

" / " Always returns float value

```
print(4/2)
print(4//2)

print(11/2)
print(11//2)
```

With floor divide **a // b**:

If **a** is divisible by **b** the result is the integer value of (a / b)

If **a** is not divisible by **b** the result is the integer value below (a / b)

Example – Floor Division

```
a = 10
b = 3

print(a//b)  # 3

a = -10
b = -3

print(a//b)  # 3

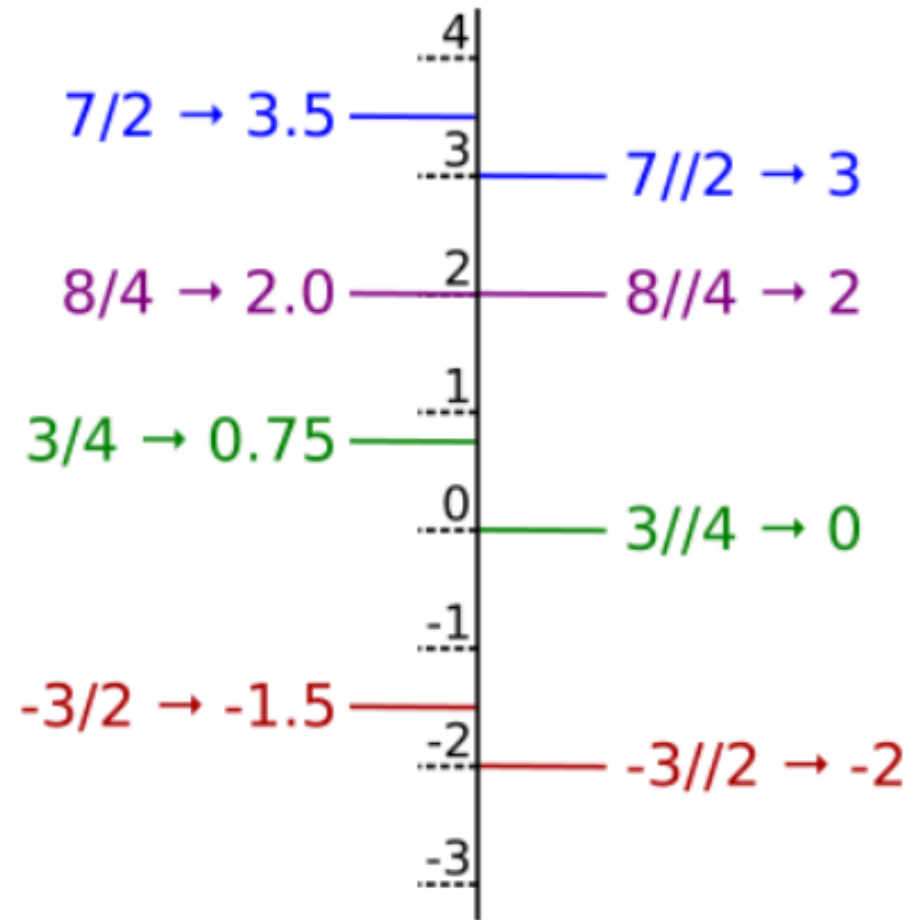
a = 10
b = -3
print(a//b)  # -4

a = -10
b = 3
print(a//b)  # -4
```

The example on the left uses the floor division operators with positive , negative integers and float values

```
print(10//3)
#3
print(10.0//3)
#3.0
print(10.3//3)
#3.0
```

Example



Comparison/Relational operators

Comparison operators are used to compare values.

It returns either True or False according to the condition.

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	<code>x > y</code>
<	Less than - True if left operand is less than the right	<code>x < y</code>
==	Equal to - True if both operands are equal	<code>x == y</code>
!=	Not equal to - True if operands are not equal	<code>x != y</code>
>=	Greater than or equal to - True if left operand is greater than or equal to the right	<code>x >= y</code>
<=	Less than or equal to - True if left operand is less than or equal to the right	<code>x <= y</code>


```
x = 10
```

```
y = 12
```

```
# Output: x > y is False
```

```
print('x > y is',x>y)
```

```
# Output: x < y is True
```

```
print('x < y is',x<y)
```

```
# Output: x == y is False
```

```
print('x == y is',x==y)
```

```
# Output: x != y is True
```

```
print('x != y is',x!=y)
```

```
# Output: x >= y is False
```

```
print('x >= y is',x>=y)
```

```
# Output: x <= y is True
```

```
print('x <= y is',x<=y)
```

Output

```
x > y is False
```

```
x < y is True
```

```
x == y is False
```

```
x != y is True
```

```
x >= y is False
```

```
x <= y is True
```

Comparison operators - Example

Comparing strings

Comparison operators can be used not just on numbers, you can also compare strings.

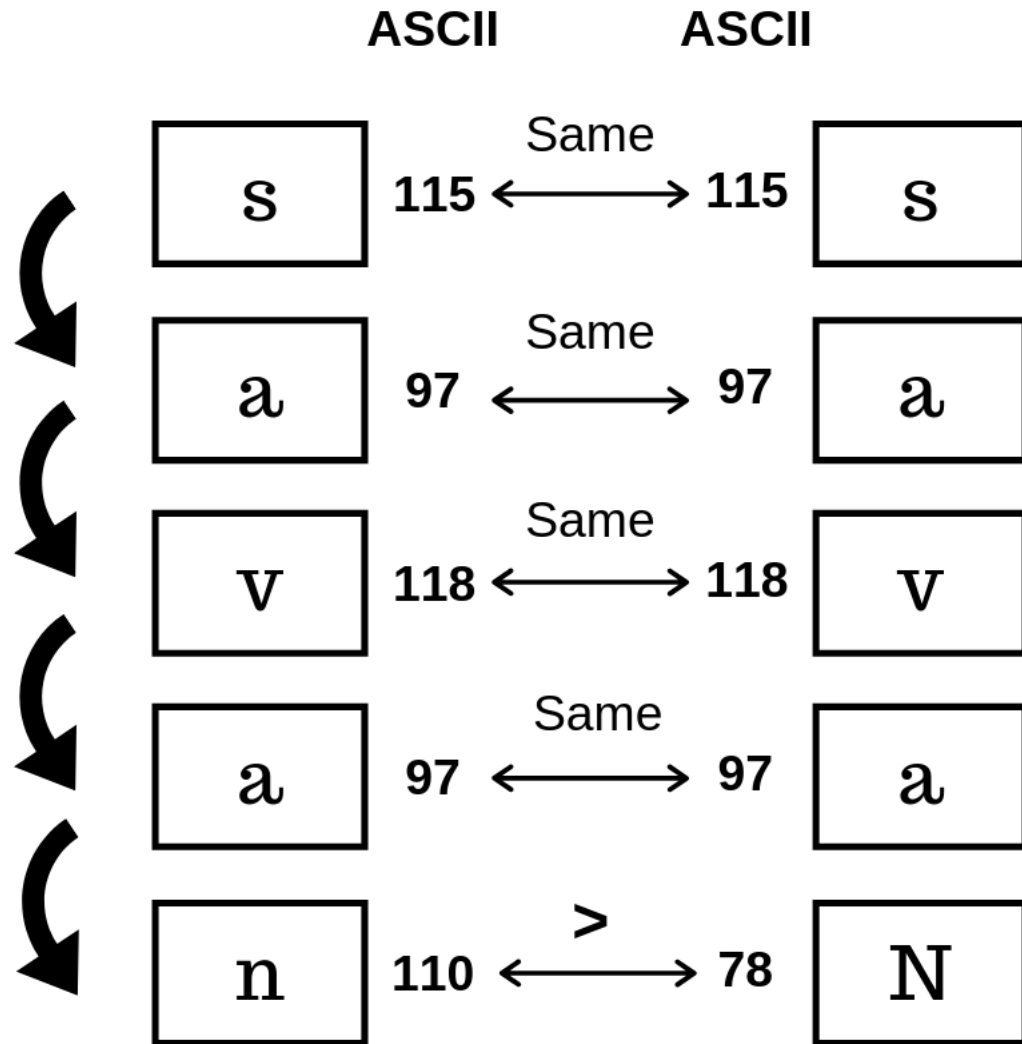
```
print('Savan' > 'savan')
```

The above snippet will return us False.

Its because the ASCII value of “s” is 115 and ASCII value of “S” is 83.

When you compare strings, the ASCII values of the first characters are first compared. If the ASCII values are different, a boolean result is returned based on the comparison. If the ASCII values are same, the next characters are compared and so on.

```
print( 'savan' > 'savaN' )
```



True

Comparing strings

what happens when you run the following code?

```
print('savan' > 'savaN')
```

The starting 4 characters of both the strings are same. Only the last character is different. In this case, python will compare the ASCII values of all the characters until the last. Since the ASCII value of n(110) is greater than the ASCII value of N(78), Python will return us True. The image on the side shows the pictorial representation of the process.

Equality operator (==)

Equality operator is used to check the equality of the values of its operands.

Not to be confused with the equal to (=) operator.

The = operator is used to assign a value to a variable. When you say `x = 5`, you are assigning the value 5 to variable x.

The == operator just checks if the values of both left and right operands are equal or not, and returns a Boolean value.

```
x = 5
print(x == 5) # Returns True
print(x == 4) # Returns False

# Equality with strings

print('Savan' == 'Savan') # Returns True
print('Savan' == 'savan') # Returns False
```

Comparison Operators - Example

```
>>> "angela" > 45
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    "angela" > 45
TypeError: '>' not supported between instances of 'str' and 'int'
>>> "angela" == 45
False
```

```
>>> 10<20
True
>>> 10<20<30
True
>>> 10<20>30
False
>>> |
```

```
>>> 10=='10'
False
>>> 1==True
True
>>> 0==False
True
>>> 1>True
False
>>> 1>=True
True
>>> |
```

Logical operators

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Logical operators are the **and, or, not** operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Examples

```
x = True
y = False

print('x and y is',x and y)

print('x or y is',x or y)

print('not x is',not x)
```

Output

```
x and y is False
x or y is True
not x is False
```

```
x = 2
y = -2

# and
print(x > 0 and y < 0)           # True

# or
print(x > 0 or y < 0)           # True

# not
print(not(x > 0 and y < 0))     # False
```

Logical operators for Non-Boolean types

0 means False

Non-zero means True

Empty string is always treated as False

x and y:

If x is evaluates to false return x otherwise return y

Eg:

10 and 20 → 20

0 and 20 → 0

If first argument is zero then result is zero otherwise result is y

Logical operators for Non-Boolean types

x or y:

If x evaluates to True then result is x otherwise result is y

10 or 20 → 10

0 or 20 → 20

not x:

If x is evalutates to False then result is True otherwise False

not 10 → False

not 0 → True



Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

They are applicable for int and bool type.

Bitwise Operators

There are 6 bitwise operators in Python. The below table provides short details about them.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR
>>	Bitwise right shift
<<	Bitwise left shift

Basic logic Truth Tables

Basic logic truth tables

AND Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NOT Truth Table

A	B
0	1
1	0

Bitwise AND Operator

Python bitwise and operator returns 1 if both the bits are 1, otherwise 0.

```
>>> 10 & 7  
2  
>>> |
```

A = 10 => 1010 (Binary)

B = 7 => 111 (Binary)

A & B = 1010

&

0111

= 0010

= 2 (Decimal)

Bitwise OR Operator

Python bitwise or operator returns 1 if any of the bits is 1. If both the bits are 0, then it returns 0.

```
>>> 10 | 7
15
>>> |
```

A = 10 => 1010 (Binary)

B = 7 => 111 (Binary)

A | B = 1010

|

0111

= 1111

= 15 (Decimal)

Bitwise XOR Operator

Python bitwise XOR operator returns 1 if one of the bits is 0 and the other bit is 1. If both the bits are 0 or 1, then it returns 0.

```
>>> 10 ^ 7
13
>>> |
```

A = 10 => 1010 (Binary)

B = 7 => 111 (Binary)

A ^ B = 1010

^

0111

= 1101

= 13 (Decimal)

4. Bitwise Complement Operator

- Python Ones' complement of a number 'A' is equal to

$-(A+1)$

```
>>> ~10
-11
>>> ~~10
10
>>>
```

A = 10 => 1010 (Binary)

**~A = ~1010
= -(1010+1)
= -(1011)
= -11 (Decimal)**

Bitwise complement Operator

A = 0000....1010

~A = 1111....0101

1st bit(MSB) - 1 - negative number
hence rest of the 31 bits are in 2's complement

2's complement = 111.....0101

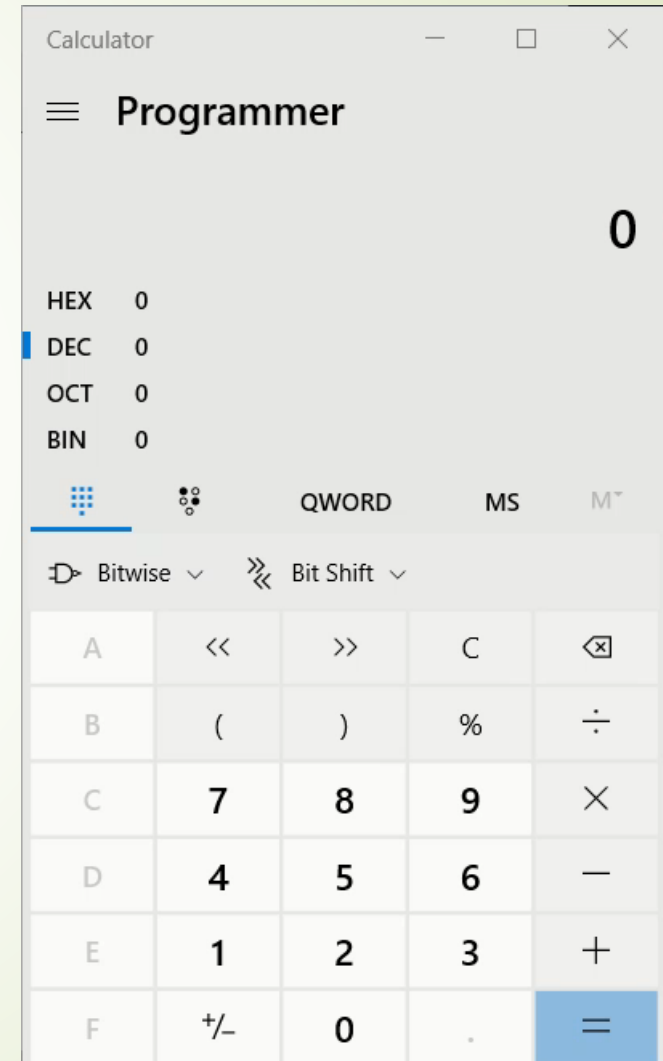
1's complement = 0001010

add 1	+	1

000.....1011		

Which is number 11

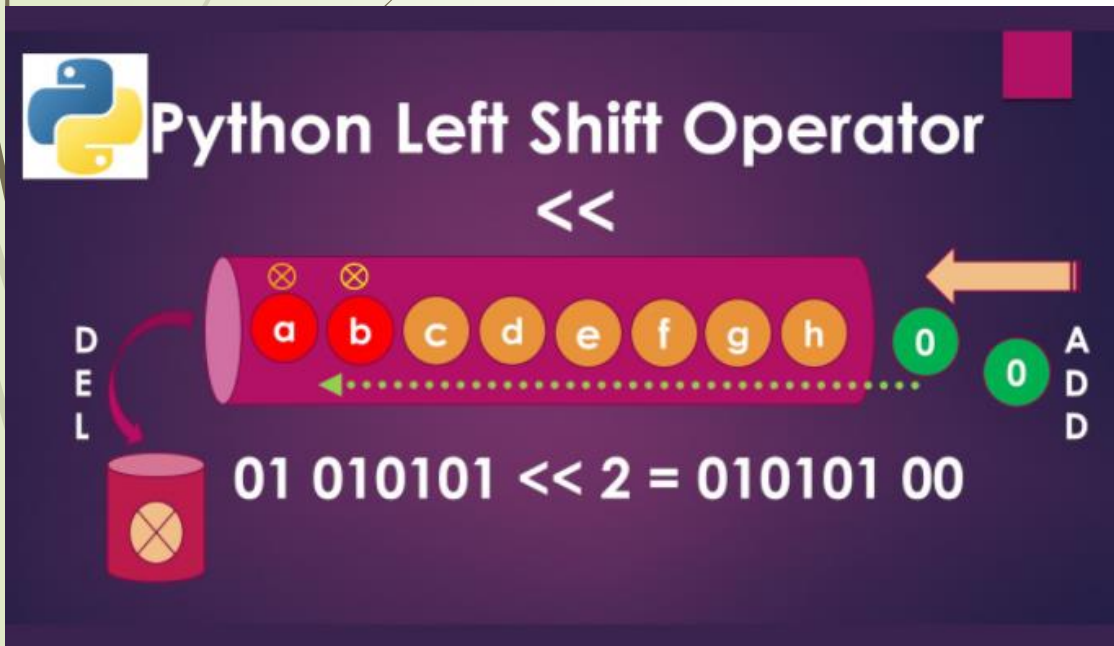
Because the MSB is 1, the result is -11



Bitwise Left Shift Operator

Python bitwise left shift operator shifts the left operand bits towards the left side for the given number of times in the right operand.

In simple terms, the binary number is appended with 0s at the end.



```
>>> 10 << 2
40
>>>
```

A = 10 => 1010 (Binary)

**A << 2 = 1010 << 2
= 101000
= 40 (Decimal)**

1111 1111 1111 1111 1111 1111 1111 0110 - left shift this by 2 digits
11 1111 1111 1111 1111 1111 1111 011000

MSB = 1 - Which means the above is a negative number

Negative numbers will be stored as 2's complement in the memory

1 1111 1111 1111 1111 1111 1111 011000 - 2's complement

step 1: Find the complement of the above

```
0 0000 0000 0000 0000 0000 0000 100111
+
-----
0 0000 0000 0000 0000 0000 0000 101000
```

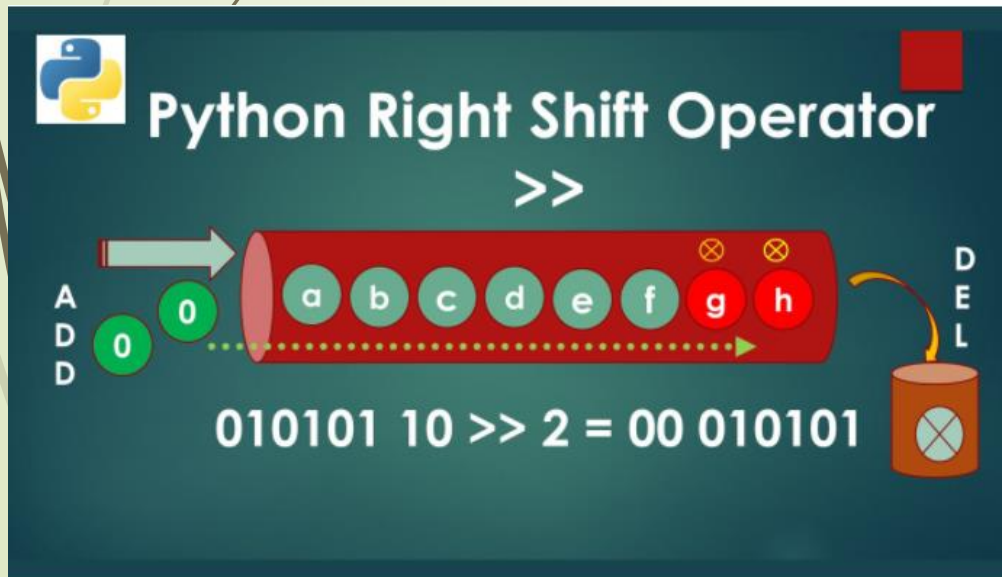
|543210

$2^5 + 2^3 = 32 + 8 = 40$

6. Bitwise Right Shift Operator

Python right shift operator is exactly the opposite of the left shift operator.

- Then left side operand bits are moved towards the right side for the given number of times. In simple terms, the right-side bits are removed.
- After shifting the empty cells we have to fill with sign bit.(0 for +ve and 1 for -ve)



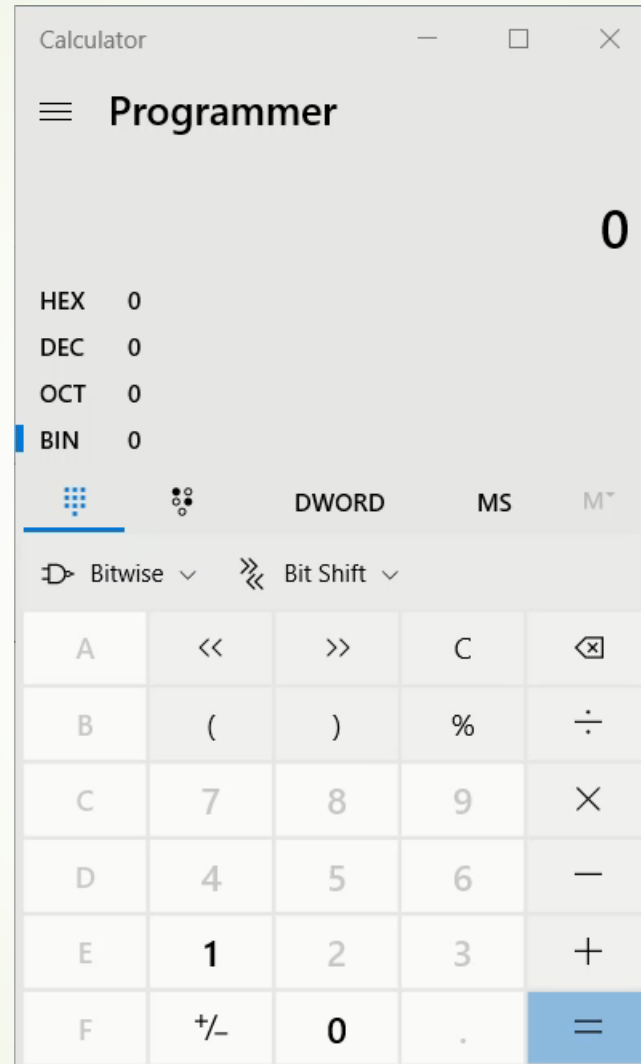
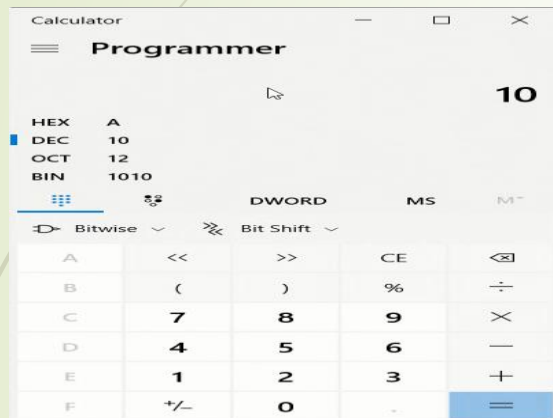
```
>>> 10 >> 2  
2  
>>>
```

A = 10 => 1010 (Binary)

A>>2 = 1010>>2

= 10

= 2 (Decimal)



```
>>> 10 >> 2  
2  
>>> -10 >> 2  
-3  
>>> |
```

Using bitwise operators for Boolean types

We can apply bitwise operators for boolean types also

- 🌀 **`print(True & False) → False`**
- 🌀 **`print(True | False) → True`**
- 🌀 **`print(True ^ False) → True`**
- 🌀 **`print(~True) → -2`**
- 🌀 **`print(True<<2) → 4`**
- 🌀 **`print(True>>2) → 0`**





Assignment operators

Assignment operators are used in Python to assign values to variables.

`a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left.

There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to

$$a = a + 5$$



Operator	Example	Equivalent to
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 5</code>	<code>x = x + 5</code>
-=	<code>x -= 5</code>	<code>x = x - 5</code>
*=	<code>x *= 5</code>	<code>x = x * 5</code>
/=	<code>x /= 5</code>	<code>x = x / 5</code>
%=	<code>x %= 5</code>	<code>x = x % 5</code>
//=	<code>x //= 5</code>	<code>x = x // 5</code>
**=	<code>x **= 5</code>	<code>x = x ** 5</code>
&=	<code>x &= 5</code>	<code>x = x & 5</code>
=	<code>x = 5</code>	<code>x = x 5</code>
^=	<code>x ^= 5</code>	<code>x = x ^ 5</code>
>>=	<code>x >>= 5</code>	<code>x = x >> 5</code>

Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

Identity operators

is and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.



Identity Operators

Operator	Meaning
is	True if the operands are identical (refer to the same object)
is not	True if the operands are not identical (do not refer to the same object)

Identity Operators

Here, we see that `x1` and `y1` are integers of the same values, so they are equal as well as identical. Same is the case with `x2` and `y2` (strings).

But `x3` and `y3` are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

# Output: False
print(x1 is not y1)

# Output: True
print(x2 is y2)

# Output: False
print(x3 is y3)
```

Output

```
False
True
False
```

Difference Between “==” and “IS” Operator

Identity operator (“is” and “is not”) is used to compare the object's memory location. When an object is created in memory a unique memory address is allocated to that object.


‘==’ compares if both the object values are identical or not.

‘is’ compares if both the object belongs to the same memory location.

```
>>> Name = 'karthick'
>>> Name1 = 'Mano'
>>> Name2 = 'karthick'
>>>
>>> type(Name), type(Name1), type(Name2)
(<class 'str'>, <class 'str'>, <class 'str'>)
>>>
>>> Name == Name1
False
>>> Name1 == Name2
False
>>> Name == Name2
True
>>> █
```

Difference Between “==” and “IS” Operator

```
>>> # id() function is used to check the identity of an object.  
...  
>>> id(Name)  
140517987689648  
>>> id(Name1)  
140517987697976  
>>> id(Name2)  
140517987689648
```



When I compare Name and Name1 or Name2 using the identity operator what it does at the backend is it simply runs “id(Name) == id(Name2)”. Since id(Name) and id(Name2) both share the same memory location, it returns True.

```
>>> id(Name)  
140517987689648  
>>> id(Name1)  
140517987697976  
>>> id(Name2)  
140517987689648  
>>>  
>>> Name is Name1  
False  
>>> Name is Name2  
True
```

Membership operators

in and **not in** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Membership Operators

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive).

Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

```
x = 'Hello world'
y = {1:'a',2:'b'}

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)

# Output: True
print(1 in y)

# Output: False
print('a' in y)
```

Output

```
True
True
True
False
```

What is Python Ternary Operator?

Ternary operators in Python are conditional expressions. These are operators that test a condition and based on that, evaluate a value.

This operator, if used properly, can reduce code size and enhance readability.

```
[on_true] if [expression] else [on_false]
```

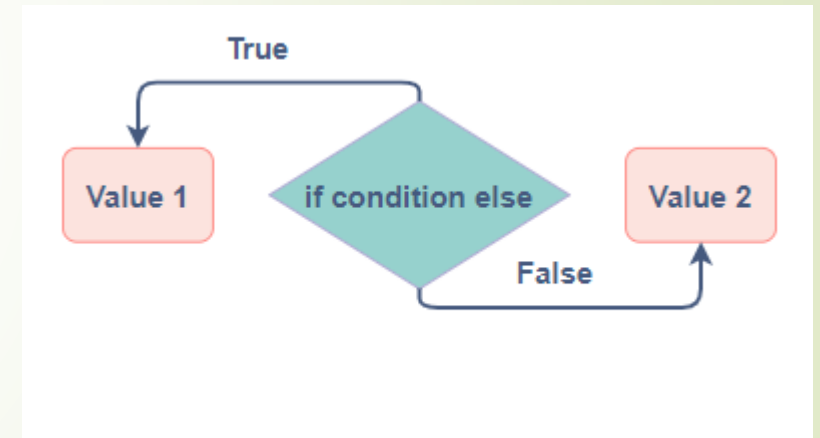

The three operands

The three operands in a ternary operator include:

condition: A boolean expression that evaluates to either true or false.

true_val: A value to be assigned if the expression is evaluated to true.

false_val: A value to be assigned if the expression is evaluated to false.



Ternary Operator

1. Python if-else code

Let's write code to compare two integers.

```
>>> a,b=2,3
>>> if a>b:
        print("a")
else:
        print("b")
```

Output

b

2. Equivalent code with Ternary operator

So, let's try doing the same with ternary operators:

```
>>> a,b=2,3
>>> print("a" if a>b else "b")
```

Output

b

Done in one line. Python first evaluates the condition. If true, it evaluates the first expression; otherwise, it evaluates the second.



Precedence and Associativity of Operators



Precedence of Python Operators

- The combination of values, variables, operators, and function calls is termed as an expression. The Python interpreter can evaluate a valid expression.

- For example:

```
>>> 5 - 7  
-2
```

- Here 5 - 7 is an expression. There can be more than one operator in an expression.
- To evaluate these types of expressions there is a rule of precedence in Python. It guides the order in which these operations are carried out.

Precedence of Python Operators

- For example, multiplication has higher precedence than subtraction.

```
>>> 10 - 4 * 2  
2
```

- But we can change this order using parentheses () as it has higher precedence than multiplication.

- # Parentheses() has higher precedence

```
>>> (10 - 4) * 2  
12
```

Operator Precedence Table

Operators	Meaning
<code>()</code>	Parentheses
<code>**</code>	Exponent
<code>+X</code> , <code>-X</code> , <code>~X</code>	Unary plus, Unary minus, Bitwise NOT
<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>	Multiplication, Division, Floor division, Modulus
<code>+</code> , <code>-</code>	Addition, Subtraction
<code><<</code> , <code>>></code>	Bitwise shift operators

<code>&</code>	Bitwise AND
<code>^</code>	Bitwise XOR
<code> </code>	Bitwise OR
<code>==</code> , <code>!=</code> , <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>is</code> , <code>is not</code> , <code>in</code> , <code>not in</code>	Comparisons, Identity, Membership operators
<code>not</code>	Logical NOT
<code>and</code>	Logical AND
<code>or</code>	Logical OR

Python Operators Precedence Rule – PEMDAS

- You might have heard about the **BODMAS** rule in your school's mathematics class.
- Python also uses a similar type of rule known as PEMDAS.
 - P – Parentheses
 - E – Exponentiation
 - M – Multiplication
 - D – Division
 - A – Addition
 - S – Subtraction
- The precedence of operators is listed from High to low.
- To remember the abbreviations, we have a funny mnemonic:
“**P**lease **E**xcuse **M**y **D**ear **A**unt **S**ally”.

Operator precedence

addition operator ← lower precedence higher precedence → **multiplication operator**

10 + 20 * 30

multiply will happen first
as * has higher precedence

10 + 60

Now addition will happen
as + has lower precedence



Using parenthesis

There is nothing wrong with making liberal use of parentheses, even when they aren't necessary to change the order of evaluation. In fact, it is considered good practice, because it can make the code more readable, and it relieves the reader of having to recall operator precedence from memory. Consider the following:

```
(a < 10) and (b > 30)
```

Here the parentheses are fully unnecessary, as the comparison operators have higher precedence than `and` and would have been performed first anyhow. But some might consider the intent of the parenthesized version more immediately obvious than this version without parentheses:

```
a < 10 and b > 30
```



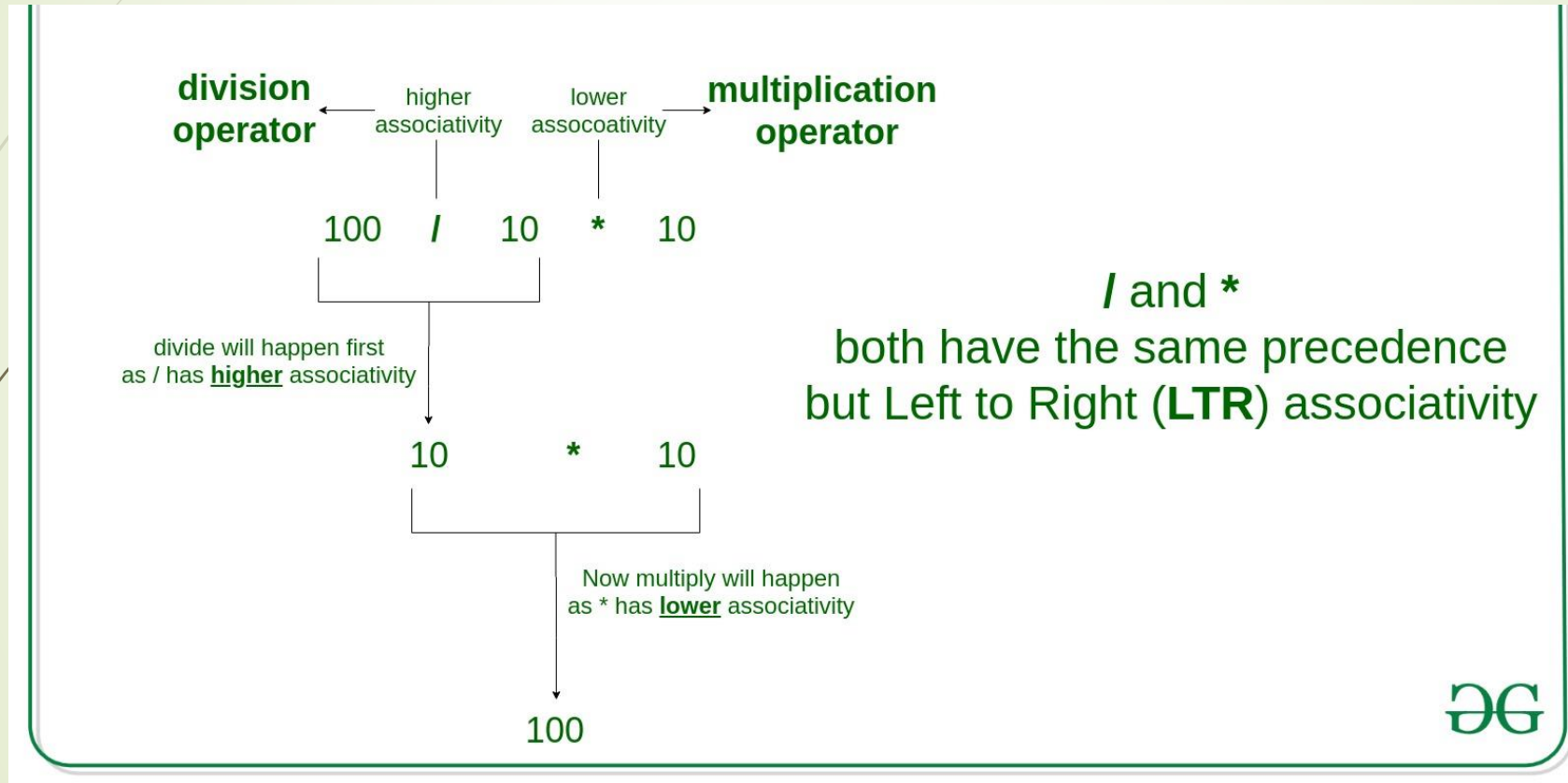
Associativity of Python Operators

We can see in the precedence table that more than one operator exists in the same group. These operators have the same precedence.

When two operators have the same precedence, associativity helps to determine the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operators of the same precedence. Almost all the operators have left-to-right associativity.

Operator Associativity



Operator	Description	Associativity	is, is not	Identity	left-to-right
			in, not in	Membership operators	
()	Parentheses	left-to-right	&	Bitwise AND	left-to-right
**	Exponent	right-to-left	^	Bitwise exclusive OR	left-to-right
* / %	Multiplication/division/modulus	left-to-right		Bitwise inclusive OR	left-to-right
+ -	Addition/subtraction	left-to-right	not	Logical NOT	right-to-left
<< >>	Bitwise shift left, Bitwise shift right	left-to-right	and	Logical AND	left-to-right
< <=	Relational less than/less than or equal to	left-to-right	or	Logical OR	left-to-right
> >=	Relational greater than/greater than or equal to		=	Assignment	right-to-left
== !=	Relational is equal to/is not equal to	left-to-right	+= -=	Addition/subtraction assignment	
			*= /=	Multiplication/division assignment	
			%= &=	Modulus/bitwise AND assignment	
			^= =	Bitwise exclusive/inclusive OR assignment	
			<<= >>=	Bitwise shift left/right assignment	

Associativity of Python Operators

For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, the left one is evaluated first.

```
# Left-right associativity
# Output: 3
print(5 * 2 // 3)

# Shows left-right associativity
# Output: 0
print(5 * (2 // 3))
```

Output

```
3
0
```

Note: Exponent operator `**` has right-to-left associativity in Python.

```
# Shows the right-left associativity of **
# Output: 512, Since 2**(3**2) = 2**9
print(2 ** 3 ** 2)

# If 2 needs to be exponentated first, need to use ()
# Output: 64
print((2 ** 3) ** 2)
```

We can see that `2 ** 3 ** 2` is equivalent to `2 ** (3 ** 2)`.

Non associative operators

Some operators like assignment operators and comparison operators do not have associativity in Python. There are separate rules for sequences of this kind of operator and cannot be expressed as associativity.

For example, $x < y < z$ neither means $(x < y) < z$ nor $x < (y < z)$. $x < y < z$ is equivalent to $(x < y)$ and $(y < z)$ and this is evaluated from left-to-right.