

PYTHON EXPRESSIONS

1

CONTENTS



5.2.1 Arithmetic Operators

5.2.2 Assignment operator

5.2.3 Comparison Operators

5.2.4 Logical Operators

5.2.5 Bitwise operators

5.2.6 Membership Operators.

5.2.7 Identity Operators

5.2.8 Precedence and associativity of operators

5.2.9 Mixed-Mode Arithmetic

ARITHMETIC OPERATORS

5.2.1 Arithmetic Operators

The various arithmetic operators in Python are tabulated in Table 5.1. The remainder operator (%) requires both operands to be integers, and the second operand is non-zero. Similarly, the division operator (/) requires that the second operand be non-zero. The floor division operator (//) returns the floor value of the quotient of a division operation. See the example below:

```
>>> 7.0//2
3.0
>>> 7//2
3
```

% (MODULO) GIVES YOU THE REMAINDER

// (FLOOR DIVISION) GIVES YOU THE QUOTIENT

- Now let's see some examples:
- $7 / 5$ quotient = 1, remainder = 2
- $7 // 5 = 1$ # floor division shows how many times 5 goes into the 7 without the remainder
- $7 \% 5 = 2$ # modulus gives the remainder only from the division operation
- $5 / 7$ quotient = 0, remainder equals = 5 # divide 5 by 7 and you get a remainder(modulus) of 5 with a quotient (floor division) of 0
- $5 // 7 = 0$ # 7 goes into 5 exactly 0 times
- $5 \% 7 = 5$ # the remainder of dividing 5 by 7 is 5


 Floor division is also called integer division.

Table 5.1: Arithmetic operators

Operation	Operator
Negation	-
Addition	+
Subtraction	--
Multiplication	*
Division	/
Floor division	//
Remainder	%
Exponentiation	**

Table 5.2: Comparison operators

Operation	Operator
Equal to	==
Not equal to	!=
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=

ASSIGNMENT OPERATOR

5.2.2 Assignment operator

'=' is the assignment operator. The **assignment statement** creates new variables and gives them values that can be used in subsequent arithmetic expressions. See the example:

```
>>> a=10
>>> b=5
>>> a+b
15
```

You can also assign different values to multiple variables. See below

```
>>> a,b,c=1,2.5,"ram"  
>>> a  
1  
>>> b  
2.5  
>>> c  
'ram'
```

Python supports the following six additional assignment operators (called *compound assignment* operators): `+=`, `-=`, `*=`, `/=`, `//=` and `%=`. These are described below:

Expression	Equivalent to
<code>a+=b</code>	<code>a=a+b</code>
<code>a-=b</code>	<code>a=a-b</code>
<code>a*=b</code>	<code>a=a*b</code>
<code>a/=b</code>	<code>a=a/b</code>
<code>a//=b</code>	<code>a=a//b</code>
<code>a%=b</code>	<code>a=a%b</code>

COMPARISON OPERATOR

5.2.3 Comparison Operators

Table 5.2 shows the various comparison operators. Comparison operators are also called relational operators. The result of a comparison is either **True** or **False**. **==** and **!=** are also known as *equality operators*.

The use of comparison operators is illustrated below:

```
>>> i,j,k=3,4,7
>>> i>j
False
>>> (j+k)>(i+5)
True
```

Comparison operators support chaining. For example, $x < y \leq z$ is equivalent to $x < y$ and $y \leq z$.

LOGICAL OPERATORS

Table 5.3: Truth tables for logical OR and logical AND operators

a	b	a or b	a and b
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Table 5.4: Truth table for logical NOT

a	not a
False	True
True	False

In the context of logical operators, Python interprets all non-zero values as **True** and zero as **False**. See examples below:

```
>>> 7 and 1
1
>>> -2 or 0
-2
>>> -100 and 0
0
```

MEMBERSHIP OPERATORS

5.2.6 Membership Operators

These operators test for the membership of a data item in a sequence, such as a string. Two membership operators are used in Python.

- **in** – Evaluates to **True** if it finds the item in the specified sequence and **False** otherwise.
- **not in** – Evaluates to **True** if it does not find the item in the specified sequence and **False** otherwise.

See the examples below:

```
>>> 'A' in 'ASCII'
True
>>> 'a' in 'ASCII'
False
>>> 'a' not in 'ASCII'
True
```

IDENTITY OPEARTORS

5.2.7 Identity Operators

is and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located in the same part of the memory. **x is y** evaluates to **true** if and only if **x** and **y** are the same object. **x is not y** yields the inverse truth value.

13

NUMBER CONVERSION

DECIMAL TO BINARY

- **Decimal**

- Decimal number is a number expressed in the base 10 numeral system. Decimal number's digits have 10 symbols: 0,1,2,3,4,5,6,7,8,9. Each digit of a decimal number counts a power of 10.

- Decimal number example:

$$653_{10} = 6 \times 10^2 + 5 \times 10^1 + 3 \times 10^0$$

- **Binary**

- Binary number is a number expressed in the base 2 numeral system. Binary number's digits have 2 symbols: zero (0) and one (1). Each digit of a binary number counts a power of 2.

- Binary number example:

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13_{10}$$

HOW TO CONVERT DECIMAL TO BINARY

Conversion steps:

- Divide the number by 2.
- Get the integer quotient for the next iteration.
- Get the remainder for the binary digit.
- Repeat the steps until the quotient is equal to 0.

Example #1

Convert 13_{10} to binary:

Division by 2	Quotient	Remainder	Bit #
13/2	6	1	0
6/2	3	0	1
3/2	1	1	2
1/2	0	1	3

So $13_{10} = 1101_2$

Example #2

Convert 174_{10} to binary:

Division by 2	Quotient	Remainder	Bit #
174/2	87	0	0
87/2	43	1	1
43/2	21	1	2
21/2	10	1	3
10/2	5	0	4
5/2	2	1	5
2/2	1	0	6
1/2	0	1	7

So $174_{10} = 10101110_2$

BITWISE OPERATOR (DIVIDED INTO 3 CATEGORIES)

5.2.5 Bitwise operators

Bitwise operators take the binary representation of the operands and work on their bits, one bit at a time. The bits of the operand(s) are compared starting with the rightmost bit - the least significant bit, then moving towards the left and ending with the leftmost (most significant) bit. The result of the comparison will depend on the compared bits and the operation being performed. These bitwise operators can be divided into three general categories as discussed below:

ONE'S COMPLEMENT OPERATOR

5.2.5.1 One's complement operator

One's complement is denoted by the symbol \sim . It operates by changing all zeroes to ones and ones to zeroes in the binary representation of the operand. The operand must be an integer-type quantity.

Example 5.1. This example illustrates the one's complement operator. See below:

```
>>> ~98
-99
>>> ~102
-103
```

Let us understand the results obtained. Take 102. Its binary is 01100110. Flipping the bits, yields $10011001 = -103$. This is shown below:

$$\begin{array}{rcl} 102 & = & 0110 \ 0110 \\ & & \hline \sim 102 & = & 1001 \ 1001 \\ & & = -103 \end{array}$$


LOGICAL BITWISE OPERATOR

5.2.5.2 Logical bitwise operators

There are three logical bitwise operators: bitwise and (&), bitwise exclusive or (^), and bitwise or (|). Each of these operators require two integer-type operands. The operations are performed on each pair of corresponding bits of the operands based on the following rules:

- A **bitwise and** expression will return 1 if both the operand bits are 1. Otherwise, it will return 0.
- A **bitwise or** expression will return 1 if at least one of the operand bits is 1. Otherwise, it will return 0.
- A **bitwise exclusive or** expression will return 1 if the bits are not alike (one bit is 0 and the other is 1). Otherwise, it will return 0.

XOR



A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

These results are summarized in Table 5.5. In this table, b_1 and b_2 represent the corresponding bits within the first and second operands, respectively.

Table 5.5: Logical bitwise operators

b_1	b_2	$b_1 \& b_2$	$b_1 \mid b_2$	$b_1 \wedge b_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```
>>> a=20
>>> b=108
>>> a&b
4
>>> a|b
124
>>> a^b
120
```

The following justifies these results.

$$\begin{array}{rcl} a & = & 0001 \ 0100 \\ b & = & 0110 \ 1100 \\ \hline a \ \& \ b & = & 0000 \ 0100 \\ & = & 4 \end{array}$$

$$\begin{array}{rcl}
 a & = & 0001 \ 0100 \\
 b & = & 0110 \ 1100 \\
 \hline
 a \mid b & = & 0111 \ 1100 \\
 & = & 124
 \end{array}$$

$$\begin{array}{rcl}
 a & = & 0001 \ 0100 \\
 b & = & 0110 \ 1100 \\
 \hline
 a \wedge b & = & 0111 \ 1000 \\
 & = & 120
 \end{array}$$

BITWISE SHIFT OPERATOR

5.2.5.3 Bitwise shift operators

The two bitwise shift operators are shift left (\ll) and shift right (\gg). The expression $x \ll n$ shifts each bit of the binary representation of x to the left, n times. Each time we shift the bits left, the vacant bit position at the right end is filled with a zero.

The expression $x \gg n$ shifts each bit of the binary representation of x to the right, n times. Each time we shift the bits right, the vacant bit position at the left end is filled with a zero.

Example 5.3. This example illustrates the bitwise shift operators as shown below:

```
>>> 120>>2
30
>>> 10<<3
80
```

Let us now see the operations in detail.

$$\begin{array}{r} 120 = 0111 \ 1000 \\ \hline \text{Right shifting once} - 0011 \ 1100 \\ \text{Right shifting twice} - 0001 \ 1110 \\ \hline 120>>2 = 30 \end{array}$$

Let us now see the operations in detail.

$$120 = 0111 \ 1000$$

Right shifting once – 0011 1100

Right shifting twice – 0001 1110

$$120 >> 2 = 30$$

$$10 = 0000 \ 1010$$

Left shifting once – 0001 0100

Left shifting twice – 0010 1000

Left shifting thrice – 0101 0000

$$10 << 3 = 80$$

Order of Operations:

- When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**.
- Python follows the same precedence rules for its mathematical operators that mathematics does.
- The acronym **PEMDAS** is a useful way to remember the order of operations:

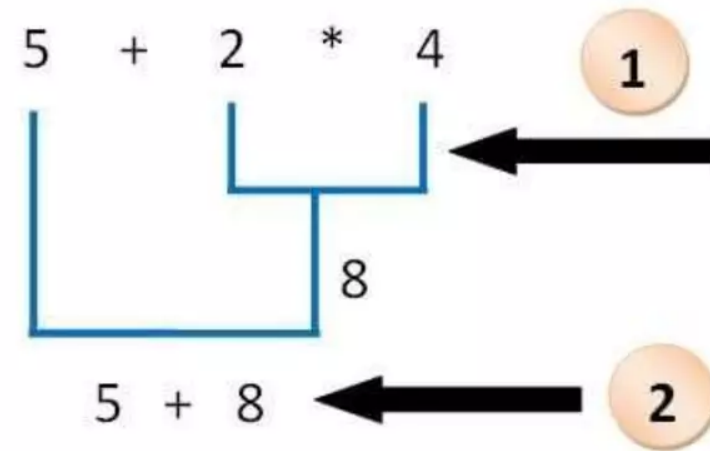
- 1. Parentheses** have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2*(3-1)$ is 4, and $(1+1)**(5-2)$ is 8. You can also use parentheses to make an expression easier to read, as in $(\text{minute}*100)/60$, even though it doesn't change the result.
- 2. Exponentiation** has the next highest precedence, so $2**1+1$ is 3 and not 4, and $3*1**3$ is 3 and not 27.
- 3. Multiplication and Division** have the same precedence, which is higher than **Addition** and **Subtraction**, which also have the same precedence.

- So $2*3-1$ yields 5 rather than 4, and $2/3-1$ is -1, not 1 (remember that in integer division, $2/3=0$).
- Operators with the same precedence are evaluated from left to right.
- So in the expression $minute*100/60$, the multiplication happens first, yielding $5900/60$, which in turn yields 98.
- If the operations had been evaluated from right to left, the result would have been $59*1$, which is 59, which is wrong.
- Similarly, in evaluating $17-4-3$,
 - $17-4$ is evaluated first.
 - If in doubt, use parentheses.

Operator Precedence

Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+X, -X, ~X	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
		Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR

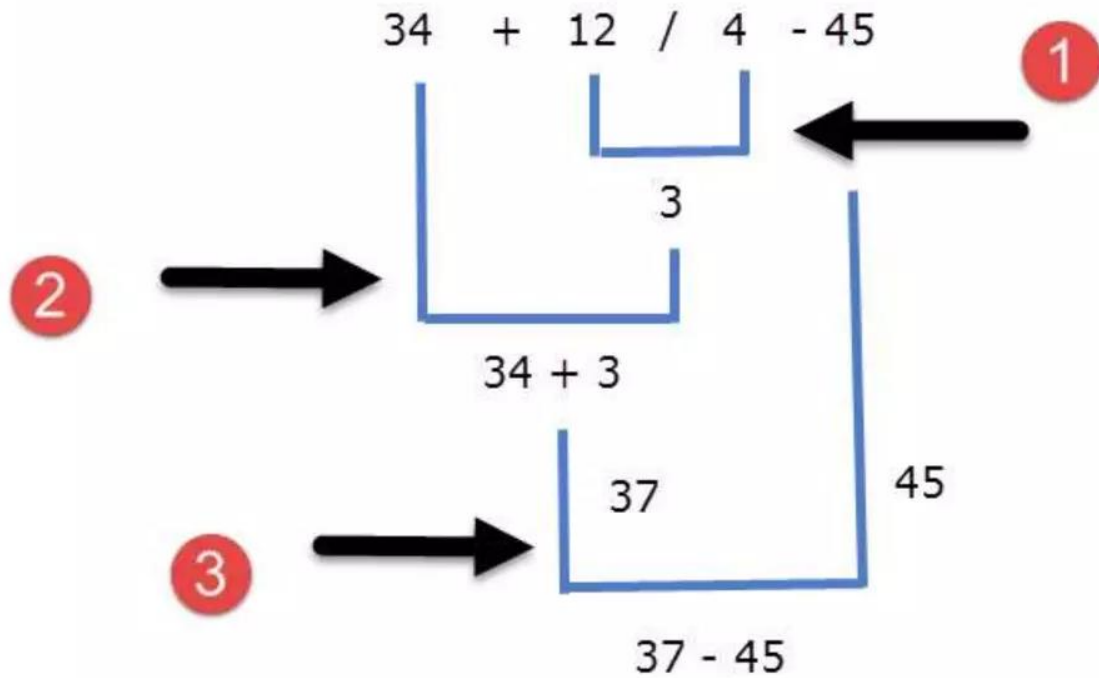
Example 1: $5+2*4$



Ans : 13

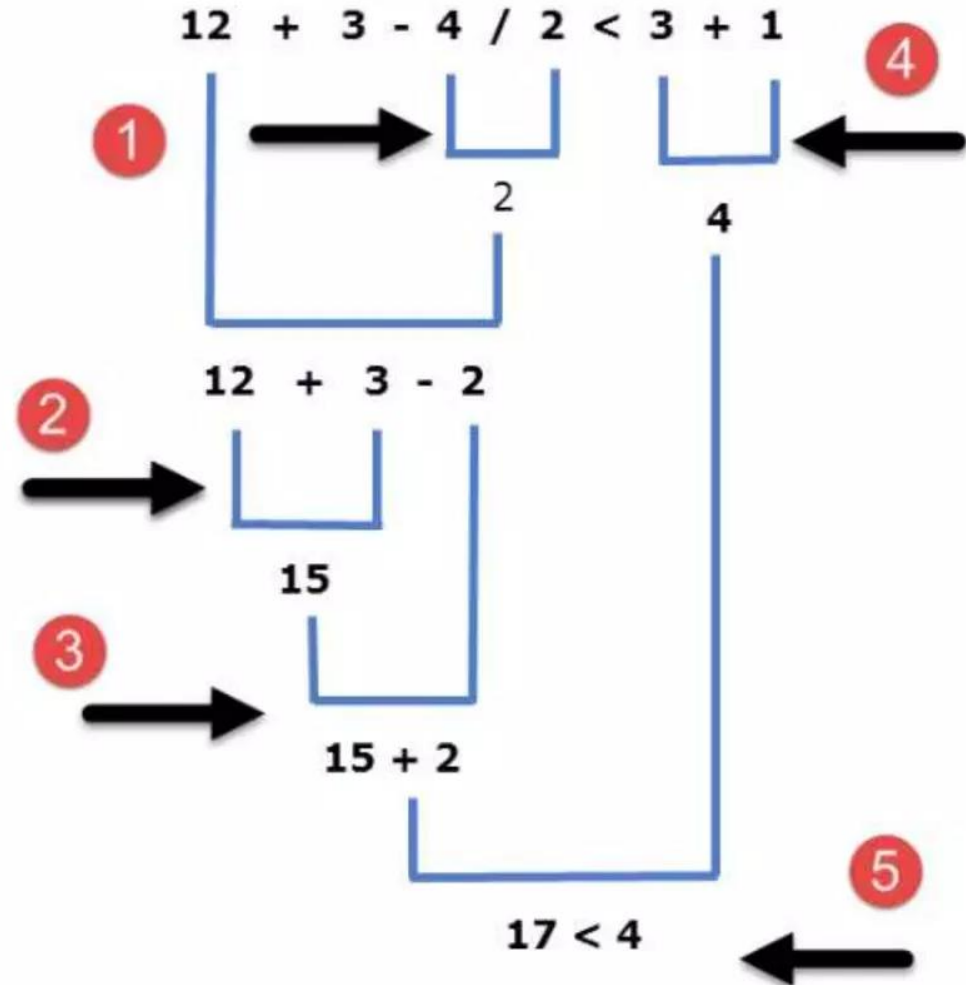
Operator Precedence

Example 2: $34 + 12 / 4 - 45$



Ans : -8

Example 3: $12 + 3 - 4 / 2 < 3 + 1$



Ans: 0

PRECEDENCE AND ASSOOCIATIVITY

- If an expression has multiple operators with the same precedence, the tie is resolved using associativity rules.
- Associativity is of two types— Left to Right ($L \rightarrow R$) and Right to Left ($R \rightarrow L$).
- The third column of Table 5.6 lists the associativity of the operators. $L \rightarrow R$ means that when there are two operators with the same precedence, the operator that comes first on a left-to-right scan of the expression will be given priority.
- Similarly with $R \rightarrow L$, the right operator has higher priority.

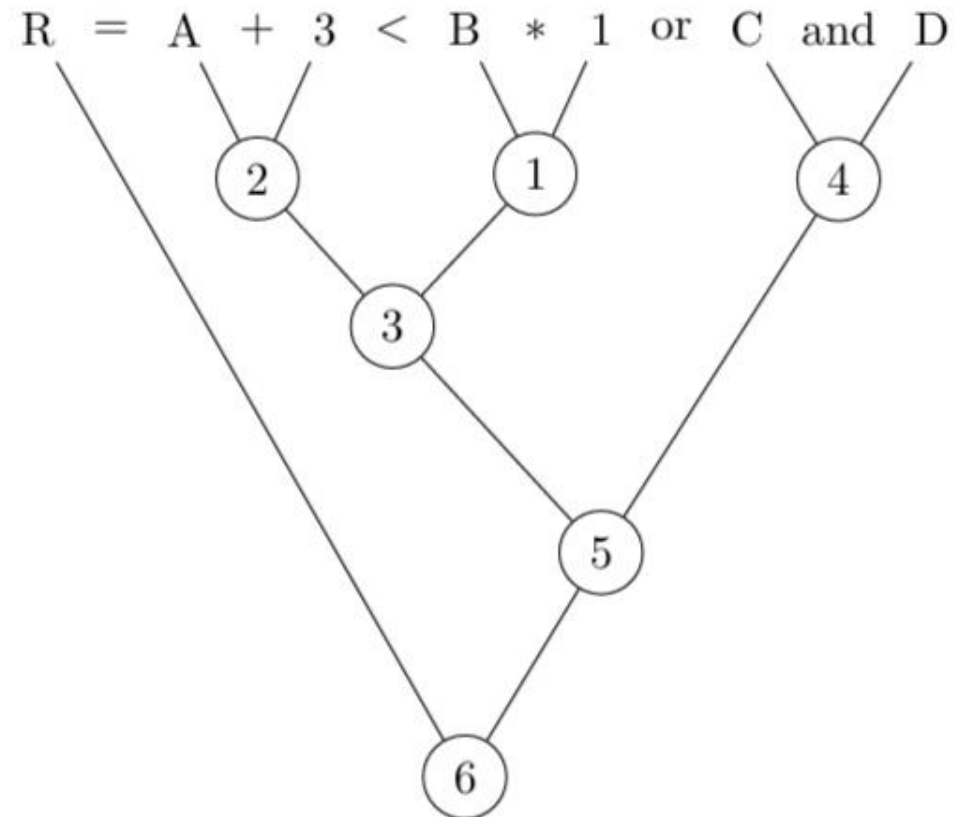
Precedence group	Operators	Associativity
Parenthesis	()	L → R
Exponentiation	**	R → L
Unary plus, Unary minus, One's complement	+, -, ~	R → L
Multiplication, Division, Floor division, Modulus	*, /, //, %	L → R
Addition, Subtraction	+, -	L → R
Bitwise shift operators	<<, >>	L → R
Bitwise AND	&	L → R
Bitwise XOR	^	L → R
Bitwise OR		L → R
Comparisons, Identity and Membership operators	==, !=, <, <=, >=, >, is, is not, in, not in	L → R
Logical NOT	not	R → L
Logical AND	and	L → R
Logical OR	or	L → R
Assignment operators	=, +=, -=, *=, /=, //=, %=	R → L

Example 5.4. Consider the assignment statement

$$R = A + 3 < B * 1 \text{ or } C \text{ and } D$$

Let the values of the variables be $A = 1$, $B = 5$, $C = -1$, and $D = \text{True}$. Figure 5.1 shows the structure of the evaluation. The numbers shown in the circle denote the order in which the various operators are applied. The final result is 1, which is assigned to R .

Consider the expression $a - b + c$. Since $+$ and $-$ are of the same precedence, so look for associativity. The associativity is $L \rightarrow R$. Thus $-$ will be evaluated first as it comes to the left.



Order	Operation	Resultant
1	$B * 1$	5
2	$A + 3$	4
3	$A + 3 < B * 1$	1
4	$C \&\& D$	1
5	$A + 3 < B * 1 C \&\& D$	1
6	$R = A + 3 < B * 1 C \&\& D$	1

Figure 5.1: Evaluation of an arithmetic expression

Table 5.7: More examples of expression evaluations

<u>Expression</u>	<u>Evaluation</u>	<u>Value</u>
<code>3 + 4 * 2</code>	<code>3 + 8</code>	11
<code>(3 + 4) * 2</code>	<code>7 * 2</code>	14
<code>2 ** 3 ** 2</code>	<code>2 ** 9</code>	512
<code>(2 ** 3) ** 2</code>	<code>8 ** 2</code>	64
<code>-3 ** 2</code>	<code>-(3 ** 2)</code>	-9
<code>-(3) ** 2</code>	<code>(-3) ** 2</code>	9
<code>not True and False or True</code>	<code>(False and False) or True</code>	True

MIXED-MODE ARITHMETIC

- Performing calculations involving operands of different data types is called mixed-mode arithmetic.
- Consider computing the area of a circle having 3 unit radius:

```
>>> 3.14 * 3 ** 2  
28.26
```

- Python supports mixed-mode arithmetic through **type coercion**, wherein the resultant of an expression will have the most general data type among all operand data types involved.
- The operand of a less general type will be temporarily and automatically converted to the more general type before the operation is performed.
- The various conversion rules are summarized in Table 5.9.

Table 5.9: Type coercion rules

Operands type	Result type
int and int	int
float and float	float
int and float	float