IIT Madras
BSc Degree

Lesson-1.3

Arithmetic Expressions

Precedence

Let us start looking at arithmetic expressions that involve multiple operators:

```
>>> 4 // 2 - 1
1
```

We can interpret this statement in two ways:

- `(4 // 2) - 1 = 2 - 1 = 1`
- `4 // (2 - 1) = 4 // 1 = 4`

Clearly, we see that the interpreter is following the first way. When an expression has different operators, the interpreter has to make a decision about the way the expression is to be parenthesized, i.e., which operator takes **precedence** over the others. From the above example, we see that the floor division operator (`//`) has greater precedence than the subtraction operator (`-`).

In general, the following table describes the precedence rules for operators. Those with higher precedence come at the top of the table. Operators in a given cell have the same precedence. For example, `+` and `-` have same precedence.

Operators	Operation
`**`	exponentiation
`+x, -x`	unary +, unary - (positive sign), (negative sign)
`*, /, //, %`	multiplication, division, floor division, modulus
`+, -`	addition, subtraction

high

low

Let us take another example:

```
>>> 3 ** 2 * 4 - 4
32
```

Going by the precedence rules, we apply the parenthesis in the following sequence:

1. `(3 ** 2) * 4 - 4`
2. `((3 ** 2) * 4) - 4`

This is evaluated as: `((3 ** 2) * 4) - 4 = (9 * 4) - 4 = 36 - 4 = 32`

Order

Consider the following example:

```
>>> 3 - 2 + 1
2
```

We can interpret this statement in two ways:

- `(3 - 2) + 1 = 1 + 1 = 2`
- `3 - (2 + 1) = 3 - 3 = 0`.

The interpreter is following the first way. Does this mean that subtraction has greater precedence than addition? No, we just saw that they have the same precedence! We have to be careful here. Python evaluates expressions from **left to right**. There are two exceptions to this rule, the `**` and `=` operator, both of which are evaluated from right to left. We shall return to this in a while.

Now for another example. Consider the following expression:

```
>>> 4 - 3 - 1
0
```

The two ways of doing this are:

- `(4 - 3) - 1 = 1 - 1 = 0`
- `4 - (3 - 1) = 4 - 2 = 2`

The first way is the one followed by the interpreter. Going back to the evaluation order followed by Python, we see that this expression is evaluated from left to right.

Let us take another example:

```
>>> 8 % 4 % 2
0
```

Run the following code in the interpreter. Which of the following parenthesizations matches the expression given above? This is left as an exercise for you to try out.

```
>>> (8 % 4) % 2
>>> 8 % (4 % 2)
```

Finally, `**` is a unique operator in this regard:

```
>>> 2 ** 3 ** 0
2
```

The two ways of doing this are:

- `(2 ** 3) ** 0`
- `2 ** (3 ** 0)`

The interpreter is following the second way, i.e., the statement is being executed from right to left. This kind of execution happens only in the case of the exponentiation operator and the assignment operator.

Boolean expressions

The simplest example of an expression that results in a boolean value is given below:

```
>>> 1 > 0
True
>>> type(1 > 0)
<class 'bool'>
```

The following expression conveys the fact that `3.14` lies between 3 and 4:

```
>>> 3 < 3.14 and 4 > 3.14
True
```

This can also be written in the following manner:

```
>>> 3 < 3.14 < 4
True
```

Let us add some boolean literals into the mix:

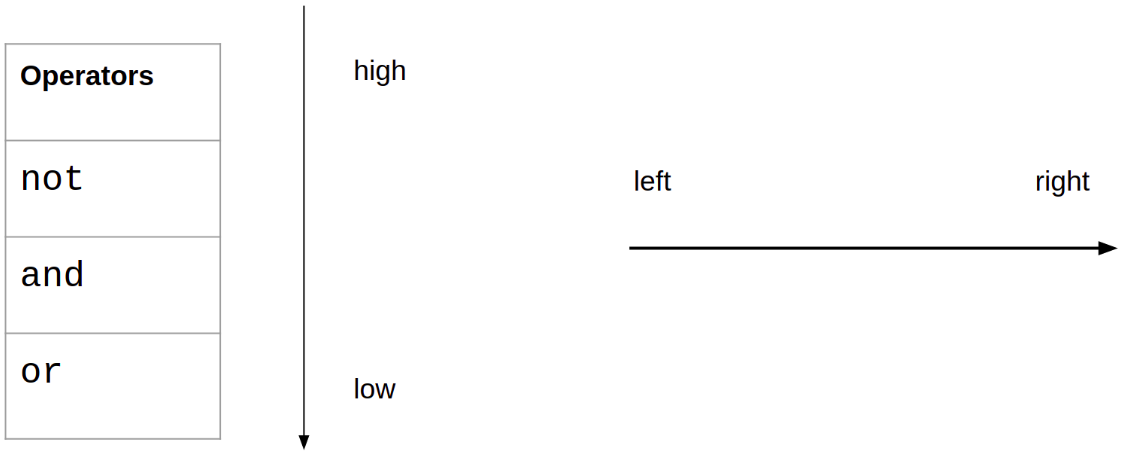
```
>>> 10 > 20 or True
True
```

As an exercise, try the following:

```
>>> False or False or False or False or True
```

Precedence and Order

Similar to arithmetic operators, logical operators also have precedence. Boolean expressions are also going to evaluated from left to right:



To see this rule in action, consider the following example:

```
>>> not True and False
False
```

There are two different parenthesizations:

- `not(True) and False = False and False = False`
- `not(True and False) = not(False) = True`

Clearly, the interpreter is following the first parenthesization. This is in accordance with the precedence rule for logical operators. The evaluation order is from left to right. But we will return to this in more detail in the section on short circuit evaluation. Another example, this time with `and` and `or`:

```
>>> True or False and False
True
```

There are two different parenthesizations:

- `(True or False) and False = True and False = False`
- `True or (False and False) = True or False = True`

According to the precedence rules, `and` has greater precedence than `or`. So, the second way is the one followed by Python.

Beware of `float`!

Execute the following expression in the interpreter:

```
>>> 10.00000000000000000000001 > 10
False
```

This seems surprising! `10.00000000000000000000001 > 10` is a perfectly valid mathematical statement that evaluates to `True`. The reason this returns `False` in Python has to do with the way floating point numbers are represented. Python, and programming languages in general, do not support arbitrary precision for representing real numbers. When the number cannot be represented exactly, an approximate value is returned. As a result of this behaviour, we should be careful when using `float` values in expressions that involve comparisons. Another example:

```
>>> 0.1 ** 100 == 0.0
False
>>> 0.1 ** 1000 == 0.0
True
```

The above expression presents a typical case of approximation when dealing with `float`. The number `0.1 ** 1000` is extremely small. So, the interpreter is going to represent that as 0. One more example follows:

```
>>> 0.1 * 3 == 0.3
False
```

Let us see what is happening here by starting with the expression to the left of the `==` operator:

```
>>> 0.1 * 3
0.30000000000000004
```

Note: The following explanation can be skipped.

The problem is with the way `0.1` is represented in binary - it has a non-terminating, recurring sequence of bits after the decimal point. As the computer uses a finite number of bits to represent data, this sequence will be truncated at some stage. This results in an approximate representation of `0.1`. For a more detailed explanation, refer to this resource.

Short Circuit Evaluation

Now, we come to an important feature in Python. Execute the following expression in the interpreter:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Division by zero is not allowed, and the interpreter promptly hits back with an error message. This is not surprising. But what is surprising is the following statement:

```
>>> True or (1 / 0)
True
```

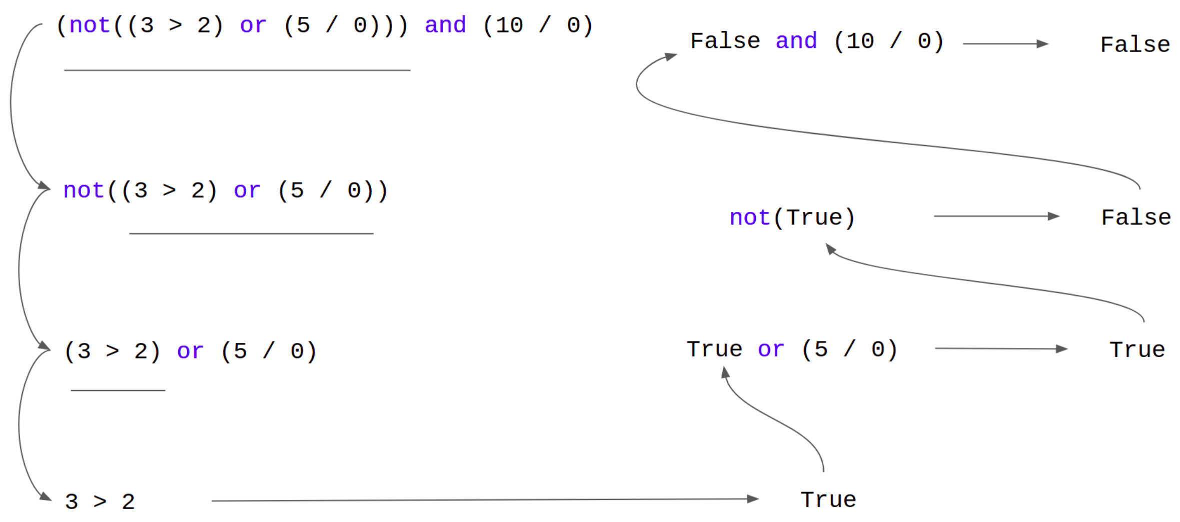
No error message! How do we explain this behaviour?

The expression is evaluated from left to right. The operator is `or`. Since the operand on the left is `True`, the whole expression will evaluate to `True` irrespective of the operand on the right. So, the interpreter skips evaluating the operand on the right. This behaviour is called short circuit evaluation.

Consider a more complex example:

```
>>> (not((3 > 2) or (5 / 0))) and (10 / 0)
False
```

Let us break this down using the diagram given below. The arrows on the left give us an idea of the expression that has to be evaluated first. If we keep following the arrows, the last expression in this image on the bottom-left is the first to be evaluated. By following the arrows on the right, we can see that the two offending expressions - `5 / 0` and `10 / 0` - are never evaluated.


(not((3 > 2) or (5 / 0))) and (10 / 0)
False and (10 / 0)
False
not((3 > 2) or (5 / 0))
not(True)
False
(3 > 2) or (5 / 0)
True or (5 / 0)
True
3 > 2
True