Module 2: Data types, variables, basic input-output operations, basic operators

2.3.1 Basic operators 2

2.3.2 Arithmetic operators: exponentiation 2

2.3.3 Arithmetic operators: multiplication 2

2.3.4 Arithmetic operators: division 3

2.3.5 Arithmetic operators: integer division 3

2.3.6 Operators: remainder (modulo) 4

2.3.7 Operators: addition 5

2.3.8 The subtraction operator, unary and binary operators 5

2.3.9 List of priorities 5

2.3.1 Basic operators

An **operator** is a symbol of the programming language, which is able to operate on the values.

For example, just as in arithmetic, the + (plus) sign is the operator which is able to **add** two numbers, giving the result of the addition.

Not all Python operators are as obvious as the plus sign, though, so let's go through some of the operators available in Python, and we'll explain which rules govern their use, and how to interpret the operations they perform.

We'll begin with the operators which are associated with the most widely recognizable arithmetic operations:

+, -, *, /, //, %, **

The order of their appearance is not accidental. We'll talk more about it once we've gone through them all.

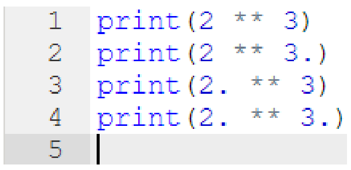
Remember: Data and operators when connected together form **expressions**. The simplest expression is a literal itself.

2.3.2 Arithmetic operators: exponentiation

A ** (double asterisk) sign is an **exponentiation** (power) operator. Its left argument is the **base**, its right, the **exponent**.

Classical mathematics prefers notation with superscripts, just like this: **2^3**. Pure text editors don't accept that, so Python uses ** instead, e.g., 2 ** 3.

Take a look at our examples in the editor window.



```
print(2 ** 3)
print(2 ** 3.)
print(2. ** 3)
print(2. ** 3.)

```

Note: we've surrounded the double asterisks with spaces in our examples. It's not compulsory, but it improves the **readability** of the code.

The examples show a very important feature of virtually all Python **numerical operators**.

Run the code and look carefully at the results it produces. Can you see any regularity here?

Remember: It's possible to formulate the following rules based on this result:

- when **both** ** arguments are integers, the result is an integer, too;
- when **at least one** ** argument is a float, the result is a float, too.

This is an important distinction to remember.

2.3.3 Arithmetic operators: multiplication

An * (asterisk) sign is a **multiplication** operator.

Run the code below and check if our *integer vs. float* rule is still working.

print(2 * 3)

print(2 * 3.)

print(2. * 3)

print(2. * 3.)

2.3.4 Arithmetic operators: division

A / (slash) sign is a **divisional** operator.

The value in front of the slash is a **dividend**, the value behind the slash, a **divisor**.

Run the code below and analyze the results.

print(6 / 3)

print(6 / 3.)

print(6. / 3)

print(6. / 3.)

You should see that there is an exception to the rule.

The result produced by the division operator is always a float, regardless of whether or not the result seems to be a float at first glance: 1 / 2, or if it looks like a pure integer: 2 / 1.

Is this a problem? Yes, it is. It happens sometimes that you really need a division that provides an integer value, not a float.

Fortunately, Python can help you with that.

2.3.5 Arithmetic operators: integer division

A // (double slash) sign is an **integer divisional** operator. It differs from the standard / operator in two details:

- its result lacks the fractional part - it's absent (for integers), or is always equal to zero (for floats); this means that **the results are always rounded**;
- it conforms to the *integer vs. float rule*.

Run the example below and see the results:

print(6 // 3)

print(6 // 3.)

print(6. // 3)

print(6. // 3.)

As you can see, *integer by integer division* gives an **integer result**. All other cases produce floats.

Let's do some more advanced tests.

Look at the following snippet:

print(6 // 4)

print(6. // 4)

Imagine that we used / instead of // - could you predict the results?

Yes, it would be 1.5 in both cases. That's clear.

But what results should we expect with // division?

Run the code and see for yourself.

What we get is two ones - one integer and one float.

The result of integer division is always rounded to the nearest integer value that is less than the real (not rounded) result.

This is very important: **rounding always goes to the lesser integer**.

Look at the code below and try to predict the results once again:

print(-6 // 4)

print(6. // -4)

Note: some of the values are negative. This will obviously affect the result. But how?

The result is two negative twos. The real (not rounded) result is -1.5 in both cases. However, the results are the subjects of rounding. The **rounding goes toward the lesser integer value**, and the lesser integer value is -2, hence: -2 and -2.0.

NOTE

Integer division can also be called **floor division**. You will definitely come across this term in the future.

2.3.6 Operators: remainder (modulo)

The next operator is quite a peculiar one, because it has no equivalent among traditional arithmetic operators.

Its graphical representation in Python is the % (percent) sign, which may look a bit confusing.

Try to think of it as of a slash (division operator) accompanied by two funny little circles.

The result of the operator is a **remainder left after the integer division**.

In other words, it's the value left over after dividing one value by another to produce an integer quotient.

Note: the operator is sometimes called **modulo** in other programming languages.

Take a look at the snippet - try to predict its result and then run it:

print(14 % 4)

As you can see, the result is two. This is why:

- 14 // 4 gives 3 → this is the integer **quotient**;
- 3 * 4 gives 12 → as a result of **quotient and divisor multiplication**;
- 14 - 12 gives 2 → this is the **remainder**.

2.3.7 Operators: addition

The **addition** operator is the + (plus) sign, which is fully in line with mathematical standards.

Again, take a look at the snippet of the program below:

```
print(-4 + 4)

print(-4. + 8)
```

The result should be nothing surprising. Run the code to check it.

2.3.8 The subtraction operator, unary and binary operators

The **subtraction** operator is obviously the - (minus) sign, although you should note that this operator also has another meaning - **it can change the sign of a number**.

This is a great opportunity to present a very important distinction between **unary** and **binary** operators.

In subtracting applications, the **minus operator expects two arguments**: the left (a **minuend** in arithmetical terms) and right (a **subtrahend**).

For this reason, the subtraction operator is considered to be one of the binary operators, just like the addition, multiplication and division operators.

But the minus operator may be used in a different (unary) way - take a look at the last line of the snippet below:

```
print(-4 - 4)

print(4. - 8)

print(-1.1)
```

By the way: there is also a unary + operator. You can use it like this:

```
print(+2)
```

The operator preserves the sign of its only argument - the right one.

Although such a construction is syntactically correct, using it doesn't make much sense, and it would be hard to find a good rationale for doing so.

2.3.9 List of priorities

Since you're new to Python operators, we don't want to present the complete list of operator priorities right now.

Instead, we'll show you a truncated form, and we'll expand it consistently as we introduce new operators.

Look at the table below:

Priority	**Operator**	
1	**	
2	+, - (note: unary operators located next to the right of the power operator bind more strongly)	unary

3	*, /, //, %	
4	+, -	binary

Note: we've enumerated the operators in order **from the highest (1) to the lowest (4) priorities**.

Of course, you're always allowed to use **parentheses**, which can change the natural order of a calculation.

In accordance with the arithmetic rules, **subexpressions in parentheses are always calculated first**.

You can use as many parentheses as you need, and they're often used to **improve the readability** of an expression, even if they don't change the order of the operations.