Set 2: Operators

Skill 2.1: Use Python as a calculator

Skill 2.1 Concepts

We have already explored the print function and how it can be used to print literals passed to it by arguments. Now, we are going to see how the print function can be used to display the result of arithmetic operations. Take a look at the code snippet below,

Code	Output
`print(2+2)`	4

The example above illustrates how Python can be used as a calculator. Not a very handy one, and definitely not a pocket one, but a calculator nonetheless. In this lesson we will explore Python's built-in **operators** and apply them to create complex **expressions**.

Skill 2.2: Explore Python's built in operators

Skill 2.2 Concepts

An **operator** is a symbol of a programming language which can 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,

Operator	Action
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Integer division
%	Modulus
**	Exponentiation

Exponentiation

The ** symbols are used for exponentiation. For example, 2^3 = 2**3. To understand how Python interprets mathematical expressions look at the code examples below,

Code	Output

`print(2 ** 3)`
`print(2 ** 3.)`
`print(2. ** 3)`
`print(2. ** 3.)`
	8
	8.0
	8.0
	8.0

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

- 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.

Multiplication

The * symbol is used for multiplication.

Code	Output

`print(2 * 3)`
`print(2 * 3.)`
`print(2. * 3)`
`print(2. * 3.)`
	6
	6.0
	6.0
	6.0

Notice in the above example, the integer versus float rule still applies.

Division

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

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

Code	Output

`print(6 / 3)`
`print(6 / 3.)`
`print(6. / 3)`
`print(6. / 3.)`
	2.0
	2.0
	2.0
	2.0

Notice in the example above, there is an exception to the integer versus float 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.

Integer division (floor division)

A **//** (double slash) sign is an **integer division** 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).
- It conforms to the *integer vs. float rule*.

Code	Output

`print(6 // 3)`
`print(6 // 3.)`
`print(6. // 3)`
`print(6. // 3.)`
	2
	2.0
	2.0
	2.0

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

Let's do some more advanced tests.

Code	Output

`print(6 // 4)`
`print(6. // 4)`
`print(6 // 8)`
`print(6 // 8.)`
	1
	1.0
	0
	0.0

The result of integer division is always floored to the nearest integer - it is not rounded.

Now consider the following example,

Code	Output
`print(-6 // 4)` `print(6. // -4)`	-2 -2.0

The result is two negative twos. The real (not rounded) result is **-1.5** in both cases. However, the results are floored in the negative direction (the lesser integer value), hence: **-2** and **-2.0**. For this reason, integer division can also be called **floor division**.

Skill 2.2 Exercise 1

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.

The result of the modulo 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.

Code	Output

`print(14 % 4)`
`print(14 % 4.)`
`print(14 % 2)`
`print(13 % 2.)`
`print(13 % 0)`
	2
	2.0
	0
	1.0
	ZeroDivisionError

To arrive at the output in the first example above Python did the following steps,

- **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**.

The last example in the output above produced an error because **division by zero doesn't work**.

Do **not** try to:

- perform a division by zero;
- perform an integer division by zero;
- find a remainder of a division by zero.

Skill 2.2 Exercise 2

Addition

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

Code	Output
`print(-4 + 4)` `print(-4. + 8)`	0 4.0

The result above should not be surprising. And just like before, the integer versus float rule applies.

Subtraction

The **subtraction** operator is the **-** (minus) sign, which is fully in line with mathematical standards as well.

Code	Output
`print(99 - 200)` `print(2.0 - 5)`	-101 -3.0

Skill 2.2 Exercise 3

Skill 2.3: Apply operators and their priorities

Skill 2.3 Concepts

So far, we've treated each operator as if it had no connection with the others. Obviously, such an ideal and simple situation is a rarity in real programming. Also, you will very often find more than one operator in one expression, and then things are no longer so simple. Consider the following expression:

```
2 + 3 * 5
```

You probably remember from math class that **multiplications precede additions**. So to evaluate the above example, we must first multiply 3 by 5, then add it to 2, thus getting the result of 17.

The phenomenon that causes some operators to act before others is known as **the hierarchy of priorities**.

Python precisely defines the priorities of all operators, and assumes that operators of a higher priority perform their operations before the operators of a lower priority.

Operators and their bindings

The **binding** of the operator determines the order of computations performed by some operators with equal priority, put side by side in one expression. Most of Python's operators have left-sided binding, which means that the calculation of the expression is conducted from left to right. This example below will show you how it works.

```
print(9 % 6 % 2)
```

There are two possible ways of evaluating the above expression,

- from left to right: first 9 % 6 gives 3, and then 3 % 2 gives 1;
- from right to left: first 6 % 2 gives 0, and then 9 % 0 causes **a fatal error**.

The output below confirms the modulo (%) operator has left-sided binding.

Code	Output
`print(9 % 6 % 2)`	1

Now, consider the example below.

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

The two possible ways of evaluating the expression above are as follows,

- 2 ** 2 → 4
  4 ** 3 → 64
- 2 ** 3 → 8
  2 ** 8 → 256

Code	Output
`print(2 ** 2 ** 3)`	256

The result above shows that **the exponentiation operator uses right-sided binding**.

Skill 2.3 Exercise 1

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 as we introduce new operators.

Below is a summary.

Priority	Operator	Binding	Code Examples	Output
1	**	right	`print(2 ** 2 ** 3)`	256
2	- (unary)	left	`print(-2 ** 2)` `print(-2 * -2)`	-4 4
3	*, /, //, %	left	`print(-12 // 2 ** 2)` `print(-2 * -2 // 2 ** 2)`	1.0 1
4	+, -	left	`print(-2 * -2 // 2 + 2 ** 2)`	6

Skill 2.3 Exercise 2

Skill 2.4: Apply operators with parentheses

Skill 2.4 Concepts

To avoid confusion, 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.

An example of an expression with multiple parentheses is here,

Code	Output
`print((5 * ((25 % 13) + 100) / (2 * 13)) // 2)`	10.0