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

2.2 Python Literals

2.2.1 Integers

You may already know a little about how computers perform calculations on numbers. Perhaps you've heard of the **binary system**, and know that it's the system computers use for storing numbers, and that they can perform any operation upon them.

We won't explore the intricacies of positional numeral systems here, but we'll say that the numbers handled by modern computers are of two types:

- **integers**, that is, those which are devoid of the fractional part;
- and **floating-point** numbers (or simply **floats**), that contain (or are able to contain) the fractional part.

This definition is not entirely accurate, but quite sufficient for now. The distinction is very important, and the boundary between these two types of numbers is very strict. Both of these kinds of numbers differ significantly in how they're stored in a computer memory and in the range of acceptable values.

The characteristic of the numeric value which determines its kind, range, and application, is called the **type**.

If you encode a literal and place it inside Python code, the form of the literal determines the representation (type) Python will use to **store it in the memory**.

For now, let's leave the floating-point numbers aside (we'll come back to them soon) and consider the question of how Python recognizes integers.

The process is almost like how you would write them with a pencil on paper - it's simply a string of digits that make up the number. But there's a reservation - you must not interject any characters that are not digits inside the number.

Take, for example, the number eleven million one hundred and eleven thousand one hundred and eleven. If you took a pencil in your hand right now, you would write the number like this: 11,111,111, or like this: 11.111.111, or even like this: 11 111 111.

It's clear that this provision makes it easier to read, especially when the number consists of many digits. However, Python doesn't accept things like these. It's **prohibited**. What Python does allow, though, is the use of **underscores** in numeric literals.*

Therefore, you can write this number either like this: 11111111, or like that: 11_111_111.

NOTE *Python 3.6 has introduced underscores in numeric literals, allowing for placing single underscores between digits and after base specifiers for improved readability. This feature is not available in older versions of Python.

And how do we code negative numbers in Python? As usual - by adding a **minus**. You can write: -11111111, or -11_111_111.

Positive numbers do not need to be preceded by the plus sign, but it's permissible, if you wish to do it. The following lines describe the same number: +11111111 and 11111111.

Integers: octal and hexadecimal numbers

There are two additional conventions in Python that are unknown to the world of mathematics. The first allows us to use numbers in an **octal** representation.

If an integer number is preceded by an 0O or 0o prefix (zero-o), it will be treated as an octal value. This means that the number must contain digits taken from the [0..7] range only.

0o123 is an **octal** number with a (decimal) value equal to 83.

The print() function does the conversion automatically. Try this:

print(0o123)

The second convention allows us to use **hexadecimal** numbers. Such numbers should be preceded by the prefix 0x or 0X (zero-x).

0x123 is a **hexadecimal** number with a (decimal) value equal to 291. The print() function can manage these values too. Try this:

print(0x123)

2.2.2 Floats

Now it's time to talk about another type, which is designed to represent and to store the numbers that (as a mathematician would say) have a **non-empty decimal fraction**.

They are the numbers that have (or may have) a fractional part after the decimal point, and although such a definition is very poor, it's certainly sufficient for what we wish to discuss.

Whenever we use a term like *two and a half* or *minus zero point four*, we think of numbers which the computer considers **floating-point** numbers:

2.5

-0.4

Note: *two and a half* looks normal when you write it in a program, although if your native language prefers to use a comma instead of a point in the number, you should ensure that your **number doesn't contain any commas** at all.

Python will not accept that, or (in very rare but possible cases) may misunderstand your intentions, as the comma itself has its own reserved meaning in Python.

If you want to use just a value of two and a half, you should write it as shown above. Note once again - there is a point between *2* and *5* - not a comma.

As you can probably imagine, the value of **zero point four** could be written in Python as:

0.4
But don't forget this simple rule - you can omit zero when it is the only digit in front of or after the decimal point.

In essence, you can write the value 0.4 as:

.4
For example: the value of 4.0 could be written as:

4.
This will change neither its type nor its value.

2.2.3 Strings

Strings are used when you need to process text (like names of all kinds, addresses, novels, etc.), not numbers.

You already know a bit about them, e.g., that **strings need quotes** the way floats need points.

This is a very typical string: "I am a string."

However, there is a catch. The catch is how to encode a quote inside a string which is already delimited by quotes.

Let's assume that we want to print a very simple message saying:

I like "Monty Python"

How do we do it without generating an error? There are two possible solutions.

The first is based on the concept we already know of the **escape character**, which you should remember is played by the **backslash**. The backslash can escape quotes too. A quote preceded by a backslash changes its meaning - it's not a delimiter, but just a quote. This will work as intended:

```
print("I like \"Monty Python\"")
```

Note: there are two escaped quotes inside the string - can you see them both?

The second solution may be a bit surprising. Python can use **an apostrophe instead of a quote**. Either of these characters may delimit strings, but you must be **consistent**.

If you open a string with a quote, you have to close it with a quote.

If you start a string with an apostrophe, you have to end it with an apostrophe.

This example will work too:

```
print('I like "Monty Python"')
```

Note: you don't need to do any escaping here.

2.2.4 Boolean values

To conclude with Python's literals, there are two additional ones.

They're not as obvious as any of the previous ones, as they're used to represent a very abstract value - **truthfulness**.

Each time you ask Python if one number is greater than another, the question results in the creation of some specific data - a **Boolean** value.

The name comes from George Boole (1815-1864), the author of the fundamental work, *The Laws of Thought*, which contains the definition of **Boolean algebra** - a part of algebra which makes use of only two distinct values: True and False, denoted as 1 and 0.

A programmer writes a program, and the program asks questions. Python executes the program, and provides the answers. The program must be able to react according to the received answers.

Fortunately, computers know only two kinds of answers:

- Yes, this is true;

- No, this is false.

You'll never get a response like: *I don't know* or *Probably yes, but I don't know for sure*.

Python, then, is a **binary** reptile.

These two Boolean values have strict denotations in Python:

True

False

You cannot change anything - you have to take these symbols as they are, including **case-sensitivity**.