Python literals

Literals - the data in itself

Now that you have a little knowledge of some of the powerful features offered by the `print()` function, it's time to learn about some new issues, and one important new term - the **literal**.

A literal is data whose values are determined by the literal itself.

As this is a difficult concept to understand, a good example may be helpful.

Take a look at the following set of digits:

```
123
```

Can you guess what value it represents? Of course you can - it's *one hundred twenty three*.

But what about this:

```
c
```

Does it represent any value? Maybe. It can be the symbol of the speed of light, for example. It also can be the constant of integration. Or even the length of a hypotenuse in the sense of a Pythagorean theorem. There are many possibilities.

You cannot choose the right one without some additional knowledge.

And this is the clue: `123` is a literal, and `c` is not.

You use literals **to encode data and to put them into your code**. We're now going to show you some conventions you have to obey when using Python.

Let's start with a simple experiment - take a look at the snippet below.

```
1 print("2")
2 print(2)
```

The first line looks familiar. The second seems to be erroneous due to the visible lack of quotes.

Try to run it.

If everything went okay, you should now see two identical lines.

What happened? What does it mean?

Through this example, you encounter two different types of literals:

- a **string**, which you already know,

- and an **integer** number, something completely new.

The `print()` function presents them in exactly the same way - this example is obvious, as their human-readable representation is also the same. Internally, in the computer's memory, these two values are stored in completely different ways - the string exists as just a string - a series of letters.

The number is converted into machine representation (a set of bits). The `print()` function is able to show them both in a form readable to humans.

We're now going to be spending some time discussing numeric literals and their internal life.

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)
```

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.

Ints vs. floats

The decimal point is essentially important in recognizing floating-point numbers in Python.

Look at these two numbers:

```
4 4.0
```

You may think that they are exactly the same, but Python sees them in a completely different way.

`4` is an **integer** number, whereas `4.0` is a **floating-point** number.

The point is what makes a float.

On the other hand, it's not only points that make a float. You can also use the letter `e`.

When you want to use any numbers that are very large or very small, you can use **scientific notation**.

Take, for example, the speed of light, expressed in *meters per second*. Written directly it would look like this: `300000000`.

To avoid writing out so many zeros, physics textbooks use an abbreviated form, which you have probably already seen: `3 x 10⁸`.

It reads: three times ten to the power of eight.

In Python, the same effect is achieved in a slightly different way - take a look:

```
3E8
```

The letter `E` (you can also use the lower-case letter `e` - it comes from the word **exponent**) is a concise record of the phrase *times ten to the power of*.

Note:

- the **exponent** (the value after the *E*) has to be an integer;
- the **base** (the value in front of the *E*) may be an integer.

Coding floats

Let's see how this convention is used to record numbers that are very small (in the sense of their absolute value, which is close to zero).

A physical constant called *Planck's constant* (and denoted as *h*), according to the textbooks, has the value of: **6.62607×10^{-34}**.

If you would like to use it in a program, you should write it this way:

```
6.62607E-34
```

Note: the fact that you've chosen one of the possible forms of coding float values doesn't mean that Python will present it the same way.

Python may sometimes choose **different notation** than you.

For example, let's say you've decided to use the following float literal:

```
0.000000000000000000001
```

When you run this literal through Python:

```
print(0.000000000000000000001)
```

this is the result:

```
1e-22
```

output

Python always chooses **the more economical form of the number's presentation**, and you should take this into consideration when creating literals.

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.

Coding strings

Now, the next question is: how do you embed an apostrophe into a string placed between apostrophes?

You should already know the answer, or to be precise, two possible answers.

Try to print out a string containing the following message:

```
I'm Monty Python.
```

As you can see, the backslash is a very powerful tool - it can escape not only quotes, but also apostrophes.

We've shown it already, but we want to emphasize this phenomenon once more - **a string can be empty** - it may contain no characters at all.

An empty string still remains a string:

```
''
```

```
""
```

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**.

Challenge: What will be the output of the following snippet of code?

```
print(True > False) print(True < False)
```

Run the code in the Sandbox to check. Can you explain the result?

Key takeaways

1. **Literals** are notations for representing some fixed values in code. Python has various types of literals - for example, a literal can be a number (numeric literals, e.g., `123`), or a string (string literals, e.g., "I am a literal.").

2. The **binary system** is a system of numbers that employs *2* as the base. Therefore, a binary number is made up of 0s and 1s only, e.g., `1010` is *10* in decimal.

Octal and hexadecimal numeration systems, similarly, employ *8* and *16* as their bases respectively. The hexadecimal system uses the decimal numbers and six extra letters.

3. **Integers** (or simply **int**s) are one of the numerical types supported by Python. They are numbers written without a fractional component, e.g., `256`, or `-1` (negative integers).

4. **Floating-point** numbers (or simply **float**s) are another one of the numerical types supported by Python. They are numbers that contain (or are able to contain) a fractional component, e.g., `1.27`.

5. To encode an apostrophe or a quote inside a string you can either use the escape character, e.g., `'I\'m happy.'`, or open and close the string using an opposite set of symbols to the ones you wish to encode, e.g., `"I'm happy."` to encode an apostrophe, and `'He said "Python", not "typhoon"'` to encode a (double) quote.

6. **Boolean values** are the two constant objects `True` and `False` used to represent truth values (in numeric contexts `1` is `True`, while `0` is `False`.

EXTRA

There is one more, special literal that is used in Python: the `None` literal. This literal is a so-called `NoneType` object, and it is used to represent **the absence of a value**. We'll tell you more about it soon.