String in action

Comparing strings

Python's strings **can be compared using the same set of operators** which are in use in relation to numbers.

Take a look at these operators - they can all compare strings, too:

- `==`
- `!=`
- `>`
- `>=`
- `<`
- `<=`

There is one "but" - the results of such comparisons may sometimes be a bit surprising. Don't forget that Python is not aware (it cannot be in any way) of subtle linguistic issues - it just **compares code point values**, character by character.

The results you get from such an operation are sometimes astonishing. Let's start with the simplest cases.

Two strings are equal when they consist of the same characters in the same order. By the same fashion, two strings are not equal when they don't consist of the same characters in the same order.

Both comparisons give `True` as a result:

```
'alpha' == 'alpha'
'alpha' != 'Alpha'
```

The final relation between strings is determined by **comparing the first different character in both strings** (keep ASCII/UNICODE code points in mind at all times.)

When you compare two strings of different lengths and the shorter one is identical to the longer one's beginning, the **longer string is considered greater**.

Just like here:

```
'alpha' < 'alphabet'
```

The relation is `True`.

String comparison is always case-sensitive (**upper-case letters are taken as lesser than lower-case**).

The expression is `True`:

```
'beta' > 'Beta'
```

Even **if a string contains digits only, it's still not a number**. It's interpreted as-is, like any other regular string, and its (potential) numerical aspect is not taken into consideration in any way.

Look at the examples:

```
'10' == '010'
'10' > '010'
'10' > '8'
'20' < '8'
'20' < '80'
```

They produce the following results:

```
False
True
False
True
True
```

Comparing strings against numbers is generally a bad idea.

The only comparisons you can perform with impunity are these symbolized by the `==` and `!=` operators. The former always gives `False`, while the latter always produces `True`.

Using any of the remaining comparison operators will raise a `TypeError` exception.

Let's check it:

```
'10' == 10
'10' != 10
'10' == 1
'10' != 1
'10' > 10
```

The results in this case are:

```
False
True
False
True
TypeError exception
```

Run all the examples, and carry out more experiments.

Sorting

Comparing is closely related to sorting (or rather, sorting is in fact a very sophisticated case of comparing).

This is a good opportunity to show you two possible ways to **sort lists containing strings**. Such an operation is very common in the real world - any time you see a list of names, goods, titles, or cities, you expect them to be sorted.

Let's assume that you want to sort the following list:

```
greek = ['omega', 'alpha', 'pi', 'gamma']
```

In general, Python offers two different ways to sort lists.

The first is implemented as **a function named** `sorted()`.

The function takes one argument (a list) and **returns a new list**, filled with the sorted argument's elements. (Note: this description is a bit simplified compared to the actual implementation - we'll discuss it later.)

The original list remains untouched.

Look at the code in the editor, and run it.

```
# Demonstrating the sorted() function
firstGreek = ['omega', 'alpha', 'pi', 'gamma']
firstGreek2 = sorted(firstGreek)

print(firstGreek)
print(firstGreek2)

print()

# Demonstrating the sort() method
secondGreek = ['omega', 'alpha', 'pi', 'gamma']
print(secondGreek)

secondGreek.sort()
print(secondGreek)
```

The snippet produces the following output:

```
['omega', 'alpha', 'pi', 'gamma']
```

```
['alpha', 'gamma', 'omega', 'pi']
```

The second method affects the list itself - **no new list is created**. Ordering is performed in situ by the method named `sort()`.

The output hasn't changed:

```
['omega', 'alpha', 'pi', 'gamma']
```

```
['alpha', 'gamma', 'omega', 'pi']
```

If you need an order other than non-descending, you have to convince the function/method to change its default behaviors. We'll discuss it soon.

Strings vs. numbers

There are two additional issues that should be discussed here: **how to convert a number (an integer or a float) into a string, and vice versa**. It may be necessary to perform such a transformation. Moreover, it's a routine way to process input/output data.

The number-string conversion is simple, as it is always possible. It's done by a function named `str()`.

Just like here:

```
itg = 13
flt = 1.3
si = str(itg)
sf = str(flt)

print(si + ' ' + sf)
```

The code outputs:

```
13 1.3
```

The reverse transformation (string-number) is possible when and only when the string represents a valid number. If the condition is not met, expect a `ValueError` exception.

Use the `int()` function if you want to get an integer, and `float()` if you need a floating-point value.

Just like here:

```
si = '13'
sf = '1.3'
itg = int(si)
flt = float(sf)

print(itg + flt)
```

This is what you'll see in the console:

```
14.3
```

In the next section, we're going to show you some simple programs that process strings.