1.3 JSON – our new friend

Do you sometimes feel that coping with IT issues is a continuous struggle with acronyms? Well, you're not alone. We share this opinion. An old anecdote says that computer technology development is in fact based on TLA. What is TLA? It's simple – it's a Three-Letter Acronym. Close your eyes, strain your mind and try to recall five acronyms commonly used in the IT world.

We bet four of them will be TLA.

And – look! – TLA is a TLA too. What a coincidence!

Now it's time to break this pattern. The acronym we'll present to you now is four letters long. What a nice change!

Let us explain its meaning:

- Java…
- …Script
- Object
- Notation

As you can see, there's a little riddle inside the name. It may be a bit disturbing, too, and some sudden questions may have appeared in your mind just now, e.g., "Java? Do you want me to learn Java? Or even JavaScript?"

No, we don't. Fortunately, the notation we want to tell you about, although created with JavaScript environments in mind, works perfectly without JavaScript. In fact, it can be used by virtually all modern programming environments thanks to the standardized libraries of functions and unified services. It doesn't matter what programming language has been used to implement a certain solution – JSON is a kind of universal bridge able to move data between seemingly incompatible parties.

JSON is the answer to quite a basic need – the need to transfer data that is the content of an object or set of objects. The mechanism which solves it should be portable and platform independent.

How can we meet such a requirement?

The problem we need to struggle with is **how to represent an object** (understood as a set of data of different types, including other objects) or even a single value in a way that can survive network transfers and inter-platform conversions.

JSON solves the problem using two simple tricks:

- it uses **UTF-8 coded text** – this means that no machine/platform-dependent formats are used; it also means that the data JSON carries is readable (poorly, but always readable) and comprehensible by humans;
- it uses a simple and not very expanded **format** (we can call it syntax, or even grammar) to represent mutual dependencies and relations between different parts of objects, and is able to transfer not only the values of objects' properties, but also their names.

In JSON, it can be an unnamed value like a number, a string, a Boolean or… nothing , although this is not what we like most about JSON. JSON is able to carry far more complex data, collected and aggregated in larger compounds.

If you want to transfer not only raw data but also all the names bound to it (like the way in which objects hold their properties), JSON offers a syntax which looks like a close relative of Python's dictionary, which is, in fact, a set of `key:value` pairs. Making such an assumption leads us to the following question – can we use Python's

syntax to code and decode network messages in REST?

Yes, we can, but it won't be JSON. If you want your data to be widely understood (not only by Python counterparts), you should use JSON.

Let's start with a very simple example. We want to build a message encapsulating an object containing just one property named prop along with its real (floating-point) value of `2.78`.

This is how JSON comes to this:

```
{ "prop": 2.78 }
```

Simple? Absolutely!

Now we're ready to go deeper. Let us show you how JSON represents values of common types.

Note: our presentation is rather comprehensive. If you need more details, refer to the original JSON documentation available here: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

If you want to encode an integer value (e.g., 123) you will specify this in the following way:

```
123
```

Looks familiar, doesn't it? Don't get carried away just yet. JSON's integer literals use **some different tricks** to those of Python. In general, you may expect no surprises when you use regular decimal integers, but keep in mind that **JSON knows nothing about numbers written using radices different to 10**, so literals like these:

- 0x10
- 0o10
- 0b10

won't be recognized in the JSON environment. Feel free to use a minus sign to put negative numbers inside the JSON text. **Don't use a plus sign** to show that a number is positive. Moreover, **don't use leading zeros**. That's an order.

Real numbers in JSON are the same as in Python, including utilization of scientific notation.

Here's an example:

- 3.141592653589
- 3.0857E16
- −1.6021766208E−19

JSON **strings** may look familiar, but there is one important difference – **you must not use apostrophes** to delimit the text. The only delimiter allowed is a quote, like here:

```
"Python"
```

This means that you can't just insert a quote inside the string – you have to use our old friend backslash (\) followed by a quote instead.

Nothing exciting – but we're used to that:

```
"\"Monty Python's\""
```

Just like in Python, there are some more digraphs and moregraphs (we're joking, don't let us confuse you) starting with \ in JSON. We've collected all of them here:

digraph	effective meaning
\\	\
\/	/
\b	backspace
\f	form feed
\n	line feed
\	carriage return
\t	tabulation
\uxxxx \Uxxxx	UNICODE codepoint

(**r** carriage return)

Note that in Python you don't have to precede / with \.

The \u (or \U) digraph starts a hexadecimal UNICODE code point and must contain exactly **four hexadecimal digits** (it doesn't matter if the letters are upper- or lower-case).

Don't forget that JSON strings **cannot be split over multiple lines** – each string must fit entirely on one line (of course, there may be more than one string on the same line).

Boolean values are represented (like in Python) by two specific identifiers (the literal name tokens): **true** and **false**:

```
true
false
```

Note: you have to **preserve the case of the literals**, as this is the only acceptable form.

There is one more literal name token in JSON, whose meaning is similar to the one known in Python as None – it may be used to represent no value, or a value without any meaning.

It is called `null` in JSON:

```
null
```

In JSON, all the above values may be combined (or packed) in two ways:

- inside **arrays** (which are a very close relative to Python lists);
- inside **objects** (which resemble Python dictionaries more than objects)

It should be noted that both ways can recursively incorporate any of the two, e.g., a list may contain an object which contains an object which contains a list and so on.

Any JSON object property may contain (or carry) an array. The syntax JSON uses to encode arrays is very similar to the one used by Python to describe lists. For example, it uses square brackets (or just brackets) to delimit array content and uses commas to separate an array's elements – just like here:

```
[1, 2, 3]
```

An empty array is denoted simply as a pair of brackets – just like in Python:

```
[ ]
```

Contrary to an array, a JSON object is a set of property specifications surrounded by a pair of braces (curly brackets) – just like here:

```
{ "prop": 2.78 }
```

The property specification is a `name:value` pair with a colon as a separator where the name **must be enclosed in quotes**.

It's worth mentioning that braces are commonly used in all C-derived programming languages and play a role similar to the one known from Python nesting – they mark the boundaries of data definitions or blocks of code. No wonder, then, that they appear in JSON, as JavaScript derives from the C-language too. The similarity to Python dictionary syntax is unintentional.

In this approach, a JSON object is a **set of property specifications separated by commas**.

One important (and very surprising) thing should be stated here. There are no restrictions on property names. No, not at all. These names don't identify variables, so they don't have to be unique. They don't have to start with a letter. They can even contain a colon, which may seem a little weird at first glance.

This doesn't mean that you have to use weird property names. We suggest you don't do that at all. It only means that the property name's semantic isn't a part of the JSON standard. In other words, JSON is semantically blind when it comes to property names. It's none of its business how you name your properties.

If you want to express the fact that a particular **object is empty**, you need to leave the braces and ensure that there is no content between them – just like here:

```
{ }
```

When there is more than one property in an object, you can specify all of them in **any order** using commas to separate the items from each other. As JSON ignores white spaces (including tabs) which aren't a part of strings, you can format (or unformat) the text in any way.

For example, both these JSON objects are the same:

```
{
x: 123,
y: -1
}
```

```
{x:123, y:-1}
```

The first is easier to read (for humans), while the second is cheaper to transmit (it occupies fewer bytes).

Of course, you may incorporate different types of data inside one object:

```
    { me: "Python",
    pi: 3.141592653589,
    parsec: 3.0857E16,
    electron: -1.6021766208E-19
    friend: "JSON",
```

```
    off: true,
    on: false,
    set: null }
```