

Re: Prothon should not borrow Python strings!

Source: <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2004-05/3012.html>

From: Paul Prescod (paul_at_prescod.net)

Date: 05/24/04

Date: Mon, 24 May 2004 14:20:27 -0700

To: Mark Hahn <mark@prothon.org>

Mark Hahn wrote:

> *"Paul Prescod" <paul@prescod.net> wrote*
>
>
>...
>
> *Is there any dynamic language that already does this right for us to steal*
> *from or is this new territory? I know for sure that I don't want to steal*
> *Java's streams. I remember hating them with a passion.*

I don't consider myself an expert: there are just some big mistakes that I can recognize. But I'll give you as much guidance as I can.

Start here:

<http://www.joelonsoftware.com/articles/Unicode.html>

Summary:

""It does not make sense to have a string without knowing what encoding it uses. You can no longer stick your head in the sand and pretend that "plain" text is ASCII.

There Ain't No Such Thing As Plain Text.

If you have a string, in memory, in a file, or in an email message, you have to know what encoding it is in or you cannot interpret it or display it to users correctly.""

One thing I should have told you is that it is just as important to get your internal APIs right as your syntax. If you embed the "ASCII assumption" into your APIs you will have a huge legacy of third party modules that expect all characters to be <255 and you'll be stuck in the same cul de sac as Python.

I would define macros like

Re: Prothon should not borrow Python strings!

comp.lang.python: Re: Prothon should not borrow Python strings!

```
#define PROTHON_CHAR int
```

and functions like

```
Prothon_String_As_UTF8
```

```
Prothon_String_As_ASCII // raises error if there are high characters
```

Obviously I can't think through the whole API. Look at Python, JavaScript and JNI, I guess.

<http://java.sun.com/docs/books/jni/html/objtypes.html#4001>

The gist is that extensions should not poke into the character string data structure expecting the data to be a "char *" of ASCII bytes. Rather it should ask you to decode the data into a new buffer. Maybe you could do some tricky buffer reuse if the encoding they ask for happens to be the same as your internal structure (look at the Java "isCopy" stuff). But if you promise users the ability to directly fiddle with the internal data then you may have to break that promise one day.

To get from a Prothon string to a C string requires encoding because `_there ain't no such thing as a plain string_`. If the C programmer doesn't tell you how they want the data encoded, how will you know?

If you get the APIs right, it will be much easier to handle everything else later.

Choosing an internal encoding is actually pretty tricky because there are space versus time tradeoffs and you need to make some guesses about how often particular characters are likely to be useful to your users.

==

On the question of types: there are two models that seem to work okay in practice. Python's split between byte strings and Unicode strings is actually not bad except that the default string literal is a BYTE string (for historical reasons) rather than a character string.

```
>>> a = "a \u1234"
>>> b = u"ab\u1234"
>>> a
'a \u1234'
>>> b
u'ab\u1234'
>>> len(a)
8
>>> len(b)
3
```

Here's what Javascript does (i.e. better):

```
<script>
str = "a \u1234"
alert(str.length) // 3
</script>
```

====

By the way, if you have the courage to distance yourself from every other language under the sun, I would propose that you throw an exception on unknown escape sequences. It is very easy in Python to accidentally used an escape sequence that is incorrect as above. Plus, it is near impossible to add new escape sequences to Python because they may break some code somewhere. I don't understand why this case is special enough to break the usual Python commitment to "not guess" what programmers mean in the face of ambiguity. This is another one of those things you have to get right at the beginning because it is tough to change later! Also, I totally hate how character numbers are not delimited. It should be `\u{1}` or `\u{1234}` or `\u{12345}`. I find Python totally weird:

```
>>> u"\1"
u'\x01'
>>> u"\12"
u'\n'
>>> u"\123"
u'S'
>>> u"\1234"
u'S4'
>>> u"\u1234"
u'\u1234'
>>> u"\u123"
UnicodeDecodeError: 'unicodeescape' codec can't decode bytes in position
0-4: end of string in escape sequence
```

=====

So anyhow, the Python model is that there is a distinction between character strings (which Python calls "unicode strings") and byte strings (called 8-bit strings). If you want to decode data you are reading from a file, you can just:

```
file("filename").read().decode("ascii")
```

or

```
file("filename").read().decode("utf-8")
```

Here's an illustration of a clean split between character strings and byte strings:

comp.lang.python: Re: Prothon should not borrow Python strings!

```
>>> file("filename").read()
<bytestring ['a', 'b', 'c'...]>
>>> file("filename").read().decode("ascii")
"abc"
```

Now the Javascript model, which also seems to work, is a little bit different. There is only one string type, but each character can take values up to 2^{16} (more on this number later).

<http://www.mozilla.org/js/language/es4/formal/notation.html#string>

If you read binary data in JavaScript, the implementations seem to just map each byte to a corresponding Unicode code point (another way of saying that is that they default to the latin-1 encoding). This should work in most browsers:

```
<SCRIPT language = "Javascript">
datafile = "http://www.python.org/pics/pythonHi.gif"
```

```
    httpconn = new XMLHttpRequest();
    httpconn.open("GET",datafile,false);
    httpconn.send(null);
    alert(httpconn.responseText);
</SCRIPT>
<BODY></BODY>
</HTML>
```

(ignore the reference to "Xml" above. For some reason Microsoft decided to conflate XML and HTTP in their APIs. In this case we are doing nothing with XML whatsoever)

I was going to write that Javascript also has a function that allows you to explicitly decode. That would be logical. You could imagine that you could do as many levels of decoding as you like:

```
objXml.decode("utf-8").decode("latin-1").decode("utf-8").decode("koi8-r")
```

This model is a little bit "simpler" in that there is only one string object and the programmer just keeps straight in their head whether it has been decoded already (or how many times it has been decoded, if for some strange reason it were double or triple-encoded).

But it turns out that I can't find a Javascript Unicode decoding function through Google. More evidence that Javascript is brain-dead I suppose.

Anyhow, that describes two models: one where byte (0-255) and character (0- 2^{16} or 2^{32}) strings are strictly separated and one where byte strings are just treated as a subset of character strings. What you absolutely do not want is to leave character handling totally in the domain of the application programmer as C and early and versions of

Re: Prothon should not borrow Python strings!

Python did.

On to character ranges. Strictly speaking, the Unicode cap is 2^{20} characters. You'll notice that this is just beyond 2^{16} , which is a much more convenient (and space efficient) number. There are three basic ways of dealing with this situation.

1. You can use two bytes per character and simply ignore the issue. "Those characters are not available. Deal with it!" That isn't as crazy as it sounds because the high characters are not in common use yet.

2. You could directly use 3 (or more likely 4) bytes per character. "Memory is cheap. Deal with it!"

3. You could do tricks where you sort of page switch from two-byte to four-byte mode using "surrogates".[1] This is actually not that far from "1" if you leave the manipulation of the surrogates entirely in application code. I believe this is the strategy used by Java[2] and Javascript.[3]

[1] <http://www.i18nguy.com/surrogates.html>

[2] "The methods that only accept a char value cannot support supplementary characters. They treat char values from the surrogate ranges as undefined characters."

<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Character.html>

"Characters are single Unicode 16-bit code points. We write them enclosed in single quotes ' and '. There are exactly 65536 characters: '«u0000»', '«u0001»', ..., 'A', 'B', 'C', ..., '«uFFFF»' (see also notation for non-ASCII characters). Unicode surrogates are considered to be pairs of characters for the purpose of this specification."

[3] <http://www.mozilla.org/js/language/js20-2000-07/formal/notation.html>

From a correctness point of view, 4-byte chars is obviously Unicode-correct. From a performance point of view, most language designers people have chosen to sweep the issue under the table and hope that 16 bits per char continue to be enough "most of the time" and that those who care about more will explicitly write their own code to deal with high characters.

Paul Prescod