More About Unicode in Python 2 and 3

written on Sunday, January 5, 2014

It's becoming increasingly harder to have reasonable discussions about the differences between Python 2 and 3 because one language is dead and the other is actively developed. So when someone starts a discussion about the Unicode support between those two languages it's not an even playing field. So I won't discuss the actual Unicode support of those two languages but the core model of how to deal with text and bytes in both.

I will use this post to show that from the pure design of the language and standard library why Python 2 the better language for dealing with text and bytes.

Since I have to maintain lots of code that deals exactly with the path between Unicode and bytes this regression from 2 to 3 has caused me lots of grief. Especially when I see slides by core Python maintainers about how I should trust them that 3.3 is better than 2.7 makes me more than angry.

The Text Model

The main difference between Python 2 and Python 3 is the basic types that exist to deal with texts and bytes. On Python 3 we have one text type: `str` which holds Unicode data and two byte types `bytes` and `bytearray`.

On the other hand on Python 2 we have two text types: `str` which for all intents and purposes is limited to ASCII + some undefined data

above the 7 bit range, `unicode` which is equivalent to the Python 3 `str` type and one byte type `bytearray` which it inherited from Python 3.

Looking at that you can see that Python 3 removed something: support for non Unicode data text. For that sacrifice it gained a hashable byte type, the `bytes` object. `bytearray` is a mutable type, so it's not suitable for hashing. I very rarely use true binary data as dictionary keys though so it does not show up as big problem. Especially not because in Python 2, you can just put bytes into the `str` type without issues.

The Lost Type

Python 3 essentially removed the byte-string type which in 2.x was called `str`. On the paper there is nothing inherently wrong with it. From a purely theoretical point of view text always in Unicode sounds awesome. And it is. If your whole world is just your interpreter. Unfortunately that's not how it works in the real world where you need to interface with bytes and different encodings on a regular basis and for that, the Python 3 model completely breaks down.

Let me be clear upfront: Python 2's way of dealing with Unicode is error prone and I am all in favour of improving it. My point though is that the one in Python 3 is a step backwards and brought so many more issues that I absolutely hate working with it.

Unicode Errors

Before I go into the details, we need to understand what the differences of the Unicode support in Python 2 and 3 is, and why the decision was made to change it.

Python 2, like many languages before it, was created without support for dealing with strings of different encodings. A string was a string and it contained bytes. It was up to the developer to properly deal with different encodings manually. This actually works remarkably fine for many situations. The Django framework for

many years did not support Unicode at all and used the byte-string interface in Python exclusively.

Python 2 however also gained better and better support for Unicode internally over the years and through this Unicode support it gained support for different encodings to represent that Unicode data.

In Python 2 the way of dealing with strings of a specific encoding was actually remarkably simple when it started out. You took a string you got from somewhere (which was a byte-string) and decoded it from the encoding you got from a side-channel (header data, metadata etc., specification) into an Unicode string. Once it was an Unicode string, it supported the same operations as a regular byte-string but it supported a much larger character range. When you needed to send that string elsewhere for processing you usually encoded it back into an encoding that the other system can deal with and it becomes a byte-string again.

So what were the issues with that? At the core this worked, unfortunately Python 2 needed to provide a nice migration path from the non-Unicode into the Unicode world. This was done by allowing coercion of byte-strings and non byte-strings. When does this happen and how does it work?

Essentially when you have an operation involving a byte-string and a Unicode-string, the byte-string is promoted into a Unicode string by going through an implicit decoding process that uses the "default encoding" which is set to ASCII. Python did provide a way to change this encoding at one point, but nowadays the `site.py` module removes the function to set this encoding after it sets the encoding to `ASCII`. If you start Python with the `-S` flag the `sys.setdefaultencoding` function is still there and you can experiment what happens if you set your Python default encoding to `UTF-8` for instance.

So here are some situations where the default encoding kicks in:

1. Implicit encoding upon string concatenation:

```
>>> "Hello " + u"World"
u'Hello World'
```

Here the string on the left is decoded by using the default system encoding into a Unicode string. If it would contain non-ASCII characters this normally blow up with an `UnicodeDecodeError` because the default encoding is set to ASCII.

2. Implicit encoding through comparison:

```
>>> "Foo" == u"Foo"
True
```

This sounds more evil as it is. Essentially it decodes the left side to Unicode and then compares. In case the left side cannot be decoded it will warn and return False. This is actually surprisingly sane behavior even though it sounds insane at first.

3. Implicit decoding as part of a codec.

This one is an evil one and most likely the source of all confusion about Unicode in Python 2. Confusing enough that Python 3 took the absolutely insanely radical step and removed `.decode()` from Unicode strings and `.encode()` from byte strings and caused me major frustration. In my mind this was an insanely stupid decision but I have been told more than once that my point of view is wrong and it won't be changed back.

The implicit decoding as part of a codec operation looks like this:

```
>>> "foo".encode('utf-8')
'foo'
```

Here the string is obviously a byte-string. We ask it to encode to UTF-8. This by itself makes no sense because the UTF-8 codec encodes from Unicode to UTF-8 bytes. So how does this work? It works because the UTF-8 codec sees that the object is not a Unicode string and first performs a coercion to Unicode through the default codec. Since `"foo"` is ASCII only and the default encoding is ASCII this coercion will succeed and then the resulting `u"foo"` string will be encoded through UTF-8.

Codec System

So you now know that Python 2 has two ways to represent strings: in bytes and in Unicode. The conversion between those two happens by using the Python codec system. However the codec system does not enforce that a conversion always needs to take place between Unicode and bytes or the other way round. A codec can implement a transformation between bytes and bytes and Unicode and Unicode. In fact, the codec system itself can implement a conversion between any Python type. You could have a JSON codec that decodes from a string into a complex Python object if you so desire.

That this might cause issues at one point has been understood from the very start. There is a codec called `'undefined'` which can be set as default encoding in which case any string coercion is disabled:

```
>>> import sys
>>> sys.setdefaultencoding('undefined')

>>> "foo" + u"bar"
Traceback (most recent call last):
    raise UnicodeError("undefined encoding")
UnicodeError: undefined encoding
```

This is implemented as a codec that raises errors for any operation. The sole purpose of that module is to disable the implicit coercion.

So how did Python 3 fix this? Python 3 removed all codecs that don't go from bytes to Unicode or vice versa and removed the now useless `.encode()` method on bytes and `.decode()` method on strings. Unfortunately that turned out to be a terrible decision because there are many, many codecs that are incredibly useful. For instance it's very common to decode with the hex codec in Python 2:

```
>>> "\x00\x01".encode('hex')
'0001'
```

While you might argue that this particular case can also be handled by a module like `binascii`, there is a deeper problem with that which is that the codec module is also separately available. For instance libraries implementing reading from sockets used the codec system to perform partial decoding of zlib streams:

```
>>> import codecs
>>> decoder = codecs.getincrementaldecoder('zlib')('strict')
>>> decoder.decode('x\x9c\xf3H\xcd\xc9\xc9Wp')
```

```
'Hello '
>>> decoder.decode('\xcdK\xceO\xc9\xccK/\x06\x00+\xad\x05\xaf')
'Encodings'
```

This was eventually recognized and Python 3.3 restored those codecs. Now however we're in the land of user confusion again because these codecs don't provide the meta information before the call about what types they can deal with. Because of this you can now trigger errors like this on Python 3:

```
>>> "Hello World".encode('zlib_codec')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
```

(Note that the codec is now called `zlib_codec` instead of `zlib` because Python 3.3 does not have the old aliases set up.)

So given the current state of Python 3.3, what exactly would happen if we would get the `.encode()` method on byte strings back for instance? This is easy to test, even without having to hack the Python interpreter. Let's just settle for a function with the same behavior for the moment:

```
import codecs

def encode(s, name, *args, **kwargs):
    codec = codecs.lookup(name)
    rv, length = codec.encode(s, *args, **kwargs)
    if not isinstance(rv, (str, bytes, bytearray)):
        raise TypeError('Not a string or byte codec')
    return rv
```

Now we can use this as replacement for the `.encode()` method we had on byte strings:

```
>>> b'Hello World'.encode('latin1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'encode'

>>> encode(b'Hello World', 'latin1')
Traceback (most recent call last):
  File "<stdin>", line 4, in encode
TypeError: Can't convert 'bytes' object to str implicitly
```

Oha! Python 3 can already deal with this. And we get a nice error. I would even argue that "Can't convert 'bytes' object to str implicitly" is a lot nicer than "'bytes' object has no attribute 'encode'".

Why do we still not have those encoding methods back? I really don't know and I no longer care either. I have been told multiple times now that my point of view is wrong and I don't understand beginners, or that the "text model" has been changed and my request makes no sense.

Byte-Strings are Gone

Aside from the codec system regression there is also the case that all text operations now are only defined for Unicode strings. In a way this seems to make sense, but it does not really. Previously the interpreter had implementations for operations on byte strings and Unicode strings. This was pretty obvious to the programmer as custom objects had to implement both `__str__` and `__unicode__` if they wanted to be formatted into either. Again, there was implicit coercion going on which confused newcomers, but at least we had the option for both.

Why was this useful? Because for instance if you write low-level protocols you often need to deal with formatting numbers out into byte strings.

Python's own version control system is still not on Python 3 because for years now because the Python team does not bring back string formatting for bytes.

This is getting ridiculous now though, because it turned out that the model chosen for Python 3 just does not work in reality. For instance in Python 3 the developers just "upgraded" some APIs to Unicode only, making them completely useless for real-world situations. For instance you could no longer parse byte only URLs with the standard library, the implicit assumption was that every URL as Unicode (for that matter, you could not handle non-Unicode mails any more either, completely ignoring that binary attachments exist).

This was fixed obviously, but because byte strings are gone now, the URL parsing library ships two implementations now. One for Unicode strings and one for byte objects. Two implementations behind the same function though, just the return value is vastly different now:

```
>>> from urllib.parse import urlparse
>>> urlparse('http://www.google.com/')
ParseResult(scheme='http', netloc='www.google.com',
            path='/', params='', query='', fragment='')
>>> urlparse(b'http://www.google.com/')
ParseResultBytes(scheme=b'http', netloc=b'www.google.com',
                 path=b'/', params=b'', query=b'', fragment=b'')
```

Looks similar? Not at all, because they are made of different types. One is a tuple of strings, the other is more like a tuple of arrays of integers. I have written about this before already and it still pains me. It makes writing code for Python incredibly frustrating now or hugely inefficient because you need to go through multiple encode and decode steps. Aside from that, it's really hard to write fully functional code now. The idea that everything can be Unicode is nice in theory, but totally not applicable for the real world.

Python 3 is riddled with weird workarounds now for situations where you cannot use Unicode strings and for someone like me, who has to deal with those situations a lot, it's ridiculously annoying.

Our Workarounds Break

The Unicode support in 2.x was not perfect, far from it. There was missing APIs and problems left and right, but we as programmers made it work. Unfortunately many of the ways in which we made it work, do not transfer well to Python 3 any more and some of the APIs would have had to have been changed to work well on Python 3.

My favourite example now is the file streams which like before are either text or bytes, but there is no way to reliably figure out which one is which. The trick which I helped to popularize is to read zero bytes from the stream to figure out of which type it is. Unfortunately those workarounds don't work reliably either. For instance passing a

urllib request object to Flask's JSON parse function breaks on Python 3 but works on Python 2 as a result of this:

```
>>> from urllib.request import urlopen
>>> r = urlopen('https://pypi.python.org/pypi/Flask/json')
>>> from flask import json
>>> json.load(r)
Traceback (most recent call last):
  File "decoder.py", line 368, in raw_decode
StopIteration

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: No JSON object could be decoded
```

The Outlook

There are many more problems with Python 3's Unicode support than just those. I started unfollowing Python developers on Twitter because I got so fed up with having to read about how amazing Python 3 is which is in such conflict with my own experiences. Yes, lots of things are cool in Python 3, but the core flow of dealing with Unicode and bytes is not.

(The worst of all of this is that many of the features in Python 3 which are genuinely cool could just as well work on Python 2 as well. Things like `yield from`, `nonlocal`, SNI SSL support etc.)

In light of only about 3% of all Python developers using Python 3 properly and developers proudly declaring on Twitter that "the migration is going as planned" I got so incredibly frustrated that I nearly published an multi page rant about my experience with Python 3 and how we should kill it.

I won't do that now but I do wish Python 3 core developers would become a bit more humble. For 97% of us, Python 2 is our beloved world for years to come and telling us constantly about how amazing Python 3 is not just painful, it's also wrong in light of the many regressions. With people starting to discuss Python 2.8, Stackless

Python preparing a new release with new features and these bad usage numbers, I don't know what failure is, if not that.

This entry was tagged python and thoughts

Contact me via mail, twitter, github or bitbucket.