Overcoming frustration: Correctly using unicode in python2

In python-2.x, there's two types that deal with text.

1. `str` is for strings of bytes. These are very similar in nature to how strings are handled in C.
2. `unicode` is for strings of unicode *code points*.

> **Note: Just what the dickens is "Unicode"?**
>
> One mistake that people encountering this issue for the first time make is confusing the `unicode` type and the encodings of unicode stored in the `str` type. In python, the `unicode` type stores an abstract sequence of *code points*. Each *code point* represents a *grapheme*. By contrast, byte `str` stores a sequence of bytes which can then be mapped to a sequence of *code points*. Each unicode encoding (*UTF-8*, UTF-7, UTF-16, UTF-32, etc) maps different sequences of bytes to the unicode *code points*.
>
> What does that mean to you as a programmer? When you're dealing with text manipulations (finding the number of characters in a string or cutting a string on word boundaries) you should be dealing with `unicode` strings as they abstract characters in a manner that's appropriate for thinking of them as a sequence of letters that you will see on a page. When dealing with I/O, reading to and from the disk, printing to a terminal, sending something over a network link, etc, you should be dealing with byte `str` as those devices are going to need to deal with concrete implementations of what bytes represent your abstract characters.

In the python2 world many APIs use these two classes interchangably but there are several important APIs where only one or the other will do the right thing. When you give the wrong type of string to an API that wants the other type, you may end up with an exception being raised (`UnicodeDecodeError` or `UnicodeEncodeError`). However, these exceptions aren't always raised because python implicitly converts between types... *sometimes*.

Frustration #1: Inconsistent Errors

Although converting when possible seems like the right thing to do, it's

actually the first source of frustration. A programmer can test out their program with a string like: `The quick brown fox jumped over the lazy dog` and not encounter any issues. But when they release their software into the wild, someone enters the string: `I sat down for coffee at the café` and suddenly an exception is thrown. The reason? The mechanism that converts between the two types is only able to deal with *ASCII* characters. Once you throw non-*ASCII* characters into your strings, you have to start dealing with the conversion manually.

So, if I manually convert everything to either byte `str` or `unicode` strings, will I be okay? The answer is.... *sometimes*.

Frustration #2: Inconsistent APIs

The problem you run into when converting everything to byte `str` or `unicode` strings is that you'll be using someone else's API quite often (this includes the APIs in the python standard library) and find that the API will only accept byte `str` or only accept `unicode` strings. Or worse, that the code will accept either when you're dealing with strings that consist solely of *ASCII* but throw an error when you give it a string that's got non-*ASCII* characters. When you encounter these APIs you first need to identify which type will work better and then you have to convert your values to the correct type for that code. Thus the programmer that wants to proactively fix all unicode errors in their code needs to do two things:

1. You must keep track of what type your sequences of text are. Does `my_sentence` contain `unicode` or `str`? If you don't know that then you're going to be in for a world of hurt.
2. Anytime you call a function you need to evaluate whether that function will do the right thing with `str` or `unicode` values. Sending the wrong value here will lead to a `UnicodeError` being thrown when the string contains non-*ASCII* characters.

> **Note:** There is one mitigating factor here. The python community has been standardizing on using `unicode` in all its APIs. Although there are some APIs that you need to send byte `str` to in order to be safe, (including things as ubiquitous as `print()` as we'll see in the next section), it's getting easier and easier to use `unicode` strings with most APIs.

Frustration #3: Inconsistent treatment of output

Alright, since the python community is moving to using **unicode** strings everywhere, we might as well convert everything to **unicode** strings and use that by default, right? Sounds good most of the time but there's at least one huge caveat to be aware of. Anytime you output text to the terminal or to a file, the text has to be converted into a byte **str**. Python will try to implicitly convert from **unicode** to byte **str**... but it will throw an exception if the bytes are non-*ASCII*:

```
>>> string = unicode(raw_input(), 'utf8')
café
>>> log = open('/var/tmp/debug.log', 'w')
>>> log.write(string)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 3: ordinal
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 3: ordinal
```

Okay, this is simple enough to solve: Just convert to a byte **str** and we're all set:

```
>>> string = unicode(raw_input(), 'utf8')
café
>>> string_for_output = string.encode('utf8', 'replace')
>>> log = open('/var/tmp/debug.log', 'w')
>>> log.write(string_for_output)
>>>
```

So that was simple, right? Well... there's one gotcha that makes things a bit harder to debug sometimes. When you attempt to write non-*ASCII* **unicode** strings to a file-like object you get a traceback everytime. But what happens when you use **print()**? The terminal is a file-like object so it should raise an exception right? The answer to that is.... *sometimes*:

```
$ python
>>> print u'café'
café
```

No exception. Okay, we're fine then?

We are until someone does one of the following:

- Runs the script in a different locale:

  ```
  $ LC_ALL=C python
  >>> # Note: if you're using a good terminal program when running in the C locale
  >>> # The terminal program will prevent you from entering non-ASCII characters
  ```

```
>>> # python will still recognize them if you use the codepoint instead:
>>> print u'caf\xe9'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 3: or
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 3: or
```

- Redirects output to a file:

```
$ cat test.py
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
print u'café'
$ ./test.py  >t
Traceback (most recent call last):
  File "./test.py", line 4, in <module>
    print u'café'
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 3: or
Traceback (most recent call last):
  File "./test.py", line 4, in <module>
    print u'café'
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 3: or
```

Okay, the locale thing is a pain but understandable: the C locale doesn't understand any characters outside of *ASCII* so naturally attempting to display those won't work. Now why does redirecting to a file cause problems? It's because `print()` in python2 is treated specially. Whereas the other file-like objects in python always convert to *ASCII* unless you set them up differently, using `print()` to output to the terminal will use the user's locale to convert before sending the output to the terminal. When `print()` is not outputting to the terminal (being redirected to a file, for instance), `print()` decides that it doesn't know what locale to use for that file and so it tries to convert to *ASCII* instead.

So what does this mean for you, as a programmer? Unless you have the luxury of controlling how your users use your code, you should always, always, always convert to a byte `str` before outputting strings to the terminal or to a file. Python even provides you with a facility to do just this. If you know that every `unicode` string you send to a particular file-like object (for instance, `stdout`) should be converted to a particular encoding you can use a `codecs.StreamWriter` object to convert from a `unicode` string into a byte `str`. In particular, `codecs.getwriter()` will return a `StreamWriter` class that will help you to wrap a file-like object for output. Using our `print()` example:

```
$ cat test.py
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
```

```
import codecs
import sys

UTF8Writer = codecs.getwriter('utf8')
sys.stdout = UTF8Writer(sys.stdout)
print u'café'
$ ./test.py  >t
$ cat t
café
```

Frustrations #4 and #5 – The other shoes

In English, there's a saying "waiting for the other shoe to drop". It means that when one event (usually bad) happens, you come to expect another event (usually worse) to come after. In this case we have two other shoes.

Frustration #4: Now it doesn't take byte strings?!

If you wrap `sys.stdout` using `codecs.getwriter()` and think you are now safe to print any variable without checking its type I am afraid I must inform you that you're not paying enough attention to *Murphy's Law*. The `StreamWriter` that `codecs.getwriter()` provides will take `unicode` strings and transform them into byte `str` before they get to `sys.stdout`. The problem is if you give it something that's already a byte `str` it tries to transform that as well. To do that it tries to turn the byte `str` you give it into `unicode` and then transform that back into a byte `str`... and since it uses the *ASCII* codec to perform those conversions, chances are that it'll blow up when making them:

```
>>> import codecs
>>> import sys
>>> UTF8Writer = codecs.getwriter('utf8')
>>> sys.stdout = UTF8Writer(sys.stdout)
>>> print 'café'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python2.6/codecs.py", line 351, in write
    data, consumed = self.encode(object, self.errors)
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3: ordinal not in r
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python2.6/codecs.py", line 351, in write
    data, consumed = self.encode(object, self.errors)
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3: ordinal not in r
```

To work around this, kitchen provides an alternate version of `codecs.getwriter()` that can deal with both byte `str` and `unicode` strings. Use `kitchen.text.converters.getwriter()` in place of the `codecs` version like this:

```
>>> import sys
>>> from kitchen.text.converters import getwriter
>>> UTF8Writer = getwriter('utf8')
>>> sys.stdout = UTF8Writer(sys.stdout)
>>> print u'café'
café
>>> print 'café'
café
```

Frustration #5: Exceptions

Okay, so we've gotten ourselves this far. We convert everything to **unicode** strings. We're aware that we need to convert back into byte **str** before we write to the terminal. We've worked around the inability of the standard **getwriter()** to deal with both byte **str** and **unicode** strings. Are we all set? Well, there's at least one more gotcha: raising exceptions with a **unicode** message. Take a look:

```
>>> class MyException(Exception):
>>>     pass
>>>
>>> raise MyException(u'Cannot do this')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyException: Cannot do this
>>> raise MyException(u'Cannot do this while at a café')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyException:
>>>
```

No, I didn't truncate that last line; raising exceptions really cannot handle non-*ASCII* characters in a **unicode** string and will output an exception without the message if the message contains them. What happens if we try to use the handy dandy **getwriter()** trick to work around this?

```
>>> import sys
>>> from kitchen.text.converters import getwriter
>>> sys.stderr = getwriter('utf8')(sys.stderr)
>>> raise MyException(u'Cannot do this')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyException: Cannot do this
>>> raise MyException(u'Cannot do this while at a café')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyException>>>
```

Not only did this also fail, it even swallowed the trailing newline that's normally there.... So how to make this work? Transform from **unicode** strings to

byte `str` manually before outputting:

```
>>> from kitchen.text.converters import to_bytes
>>> raise MyException(to_bytes(u'Cannot do this while at a café'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyException: Cannot do this while at a café
>>>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyException: Cannot do this while at a café
```

Warning: If you use `codecs.getwriter()` on `sys.stderr`, you'll find that raising an exception with a byte `str` is broken by the default `StreamWriter` as well. Don't do that or you'll have no way to output non-*ASCII* characters. If you want to use a `StreamWriter` to encode other things on stderr while still having working exceptions, use `kitchen.text.converters.getwriter()`.

Frustration #6: Inconsistent APIs Part deux

Sometimes you do everything right in your code but other people's code fails you. With unicode issues this happens more often than we want. A glaring example of this is when you get values back from a function that aren't consistently **`unicode`** string or byte `str`.

An example from the python standard library is `gettext`. The `gettext` functions are used to help translate messages that you display to users in the users' native languages. Since most languages contain letters outside of the *ASCII* range, the values that are returned contain unicode characters. `gettext` provides you with `ugettext()` and `ungettext()` to return these translations as **`unicode`** strings and `gettext()`, `ngettext()`, `lgettext()`, and `lngettext()` to return them as encoded byte `str`. Unfortunately, even though they're documented to return only one type of string or the other, the implementation has corner cases where the wrong type can be returned.

This means that even if you separate your **`unicode`** string and byte `str` correctly before you pass your strings to a `gettext` function, afterwards, you might have to check that you have the right sort of string type again.

Note: `kitchen.i18n` provides alternate gettext translation objects that return only byte `str` or only **`unicode`** string.

A few solutions

Now that we've identified the issues, can we define a comprehensive strategy for dealing with them?

Convert text at the border

If you get some piece of text from a library, read from a file, etc, turn it into a **`unicode`** string immediately. Since python is moving in the direction of **`unicode`** strings everywhere it's going to be easier to work with **`unicode`** strings within your code.

If your code is heavily involved with using things that are bytes, you can do the opposite and convert all text into byte **`str`** at the border and only convert to **`unicode`** when you need it for passing to another library or performing string operations on it.

In either case, the important thing is to pick a default type for strings and stick with it throughout your code. When you mix the types it becomes much easier to operate on a string with a function that can only use the other type by mistake.

> **Note:** In python3, the abstract unicode type becomes much more prominent. The type named `str` is the equivalent of python2's **`unicode`** and python3's `bytes` type replaces python2's **`str`**. Most APIs deal in the unicode type of string with just some pieces that are low level dealing with bytes. The implicit conversions between bytes and unicode is removed and whenever you want to make the conversion you need to do so explicitly.

When the data needs to be treated as bytes (or unicode) use a naming convention

Sometimes you're converting nearly all of your data to **`unicode`** strings but you have one or two values where you have to keep byte **`str`** around. This is often the case when you need to use the value verbatim with some external resource. For instance, filenames or key values in a database. When you do this, use a naming convention for the data you're working with so you (and others reading your code later) don't get confused about what's being stored in the value.

If you need both a textual string to present to the user and a byte value for an exact match, consider keeping both versions around. You can either use two variables for this or a `dict` whose key is the byte value.

> **Note:** You can use the naming convention used in kitchen as a guide for implementing your own naming convention. It prefixes byte `str` variables of unknown encoding with `b_` and byte `str` of known encoding with the encoding name like: `utf8_`. If the default was to handle `str` and only keep a few `unicode` values, those variables would be prefixed with `u_`.

When outputting data, convert back into bytes

When you go to send your data back outside of your program (to the filesystem, over the network, displaying to the user, etc) turn the data back into a byte `str`. How you do this will depend on the expected output format of the data. For displaying to the user, you can use the user's default encoding using `locale.getpreferredencoding()`. For entering into a file, you're best bet is to pick a single encoding and stick with it.

> **Warning:** When using the encoding that the user has set (for instance, using `locale.getpreferredencoding()`, remember that they may have their encoding set to something that can't display every single unicode character. That means when you convert from `unicode` to a byte `str` you need to decide what should happen if the byte value is not valid in the user's encoding. For purposes of displaying messages to the user, it's usually okay to use the `replace` encoding error handler to replace the invalid characters with a question mark or other symbol meaning the character couldn't be displayed.

You can use `kitchen.text.converters.getwriter()` to do this automatically for `sys.stdout`. When creating exception messages be sure to convert to bytes manually.

When writing unittests, include non-ASCII values and both unicode and str type

Unless you know that a specific portion of your code will only deal with *ASCII*, be sure to include non-*ASCII* values in your unittests. Including a few characters from several different scripts is highly advised as well because some code may have special cased accented roman characters but not know

how to handle characters used in Asian alphabets.

Similarly, unless you know that that portion of your code will only be given **unicode** strings or only byte **str** be sure to try variables of both types in your unittests. When doing this, make sure that the variables are also non-*ASCII* as python's implicit conversion will mask problems with pure *ASCII* data. In many cases, it makes sense to check what happens if byte **str** and **unicode** strings that won't decode in the present locale are given.

Be vigilant about spotting poor APIs

Make sure that the libraries you use return only **unicode** strings or byte **str**. Unittests can help you spot issues here by running many variations of data through your functions and checking that you're still getting the types of string that you expect.

Example: Putting this all together with kitchen

The kitchen library provides a wide array of functions to help you deal with byte **str** and **unicode** strings in your program. Here's a short example that uses many kitchen functions to do its work:

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import locale
import os
import sys
import unicodedata

from kitchen.text.converters import getwriter, to_bytes, to_unicode
from kitchen.i18n import get_translation_object

if __name__ == '__main__':
    # Setup gettext driven translations but use the kitchen functions so
    # we don't have the mismatched bytes-unicode issues.
    translations = get_translation_object('example')
    # We use _() for marking strings that we operate on as unicode
    # This is pretty much everything
    _ = translations.ugettext
    # And b_() for marking strings that we operate on as bytes.
    # This is limited to exceptions
    b_ = translations.lgettext

    # Setup stdout
    encoding = locale.getpreferredencoding()
    Writer = getwriter(encoding)
    sys.stdout = Writer(sys.stdout)

    # Load data.  Format is filename\0description
    # description should be utf-8 but filename can be any legal filename
```

```
# on the filesystem
# Sample datafile.txt:
#   /etc/shells\x00Shells available on caf\xc3\xa9.lan
#   /var/tmp/file\xff\x00File with non-utf8 data in the filename
#
# And to create /var/tmp/file\xff (under bash or zsh) do:
#   echo 'Some data' > /var/tmp/file$'\377'
datafile = open('datafile.txt', 'r')
data = {}
for line in datafile:
    # We're going to keep filename as bytes because we will need the
    # exact bytes to access files on a POSIX operating system.
    # description, we'll immediately transform into unicode type.
    b_filename, description = line.split('\0', 1)

    # to_unicode defaults to decoding output from utf-8 and replacing
    # any problematic bytes with the unicode replacement character
    # We accept mangling of the description here knowing that our file
    # format is supposed to use utf-8 in that field and that the
    # description will only be displayed to the user, not used as
    # a key value.
    description = to_unicode(description, 'utf-8').strip()
    data[b_filename] = description
datafile.close()

# We're going to add a pair of extra fields onto our data to show the
# length of the description and the filesize.  We put those between
# the filename and description because we haven't checked that the
# description is free of NULLs.
datafile = open('newdatafile.txt', 'w')

# Name filename with a b_ prefix to denote byte string of unknown encoding
for b_filename in data:
    # Since we have the byte representation of filename, we can read any
    # filename
    if os.access(b_filename, os.F_OK):
        size = os.path.getsize(b_filename)
    else:
        size = 0
    # Because the description is unicode type,  we know the number of
    # characters corresponds to the length of the normalized unicode
    # string.
    length = len(unicodedata.normalize('NFC', description))

    # Print a summary to the screen
    # Note that we do not let implici type conversion from str to
    # unicode transform b_filename into a unicode string.  That might
    # fail as python would use the ASCII filename.  Instead we use
    # to_unicode() to explictly transform in a way that we know will
    # not traceback.
    print _(u'filename: %s') % to_unicode(b_filename)
    print _(u'file size: %s') % size
    print _(u'desc length: %s') % length
    print _(u'description: %s') % data[b_filename]

    # First combine the unicode portion
    line = u'%s\0%s\0%s' % (size, length, data[b_filename])
    # Since the filenames are bytes, turn everything else to bytes before combining
    # Turning into unicode first would be wrong as the bytes in b_filename
    # might not convert
    b_line = '%s\0%s\n' % (b_filename, to_bytes(line))
```

```
        # Just to demonstrate that getwriter will pass bytes through fine
        print b_('Wrote: %s') % b_line
        datafile.write(b_line)
    datafile.close()

    # And just to show how to properly deal with an exception.
    # Note two things about this:
    # 1) We use the b_() function to translate the string.  This returns a
    #    byte string instead of a unicode string
    # 2) We're using the b_() function returned by kitchen.  If we had
    #    used the one from gettext we would need to convert the message to
    #    a byte str first
    message = u'Demonstrate the proper way to raise exceptions.  Sincerely,  \u3068\u30
    raise Exception(b_(message))
```

See also: `kitchen.text.converters`