StormTest and Python String Formats

S3 Group Document ID: ST-12027

Revision Date: 29 August 2012

Version: 1.1

Product Version: 2.8

Author: Fergus O'Callaghan

Contact us at http://www.s3group.com/tv-technology or stormtest-support@s3group.com

Contents

1 Introduction

Python , like many other programming languages, uses a number of different ways to represent text strings that contain non-ASCII characters. If string formats and Python types are not taken into account when dealing with such text, exceptions can occur.

This document is intended to clarify the issues surrounding this topic, and to give guidelines on how to use StormTest in a way that avoids such exceptions.

Please note that this document is only relevant for Python version 2.x, as used by StormTest. (Python version 3.0 introduced significant changes to the way strings are handled).

1.1 References

1. http://docs.python.org/howto/unicode.html - A description of how Unicode is handled in Python version 2.x
2. http://farmdev.com/talks/unicode/ - A useful presentation covering the whole issue of Python and Unicode

1.2 Revision History

Date	Version	Description
5 Jun 2012	1.0	Initial version
29 Aug 2012	1.1	Updated to indicate difference between OCRSlot and OCRFile with regard to string formats returned. Section 5.1

Table 1 - Revision History

2 Python, Strings and Unicode

Unicode is a character set definition. It simply defines a set of 1,112,064 characters by giving them a "code point". It is basically an attempt to define a universal set of characters, covering all languages. By convention code points are written like this: U+0639.

Python has a specific type for storing a Unicode string – it is called "unicode". This type simply stores these code point values, in either 16-bit or 32-bit units depending on how Python was compiled. Python denotes a Unicode string type like this: u'abc'.

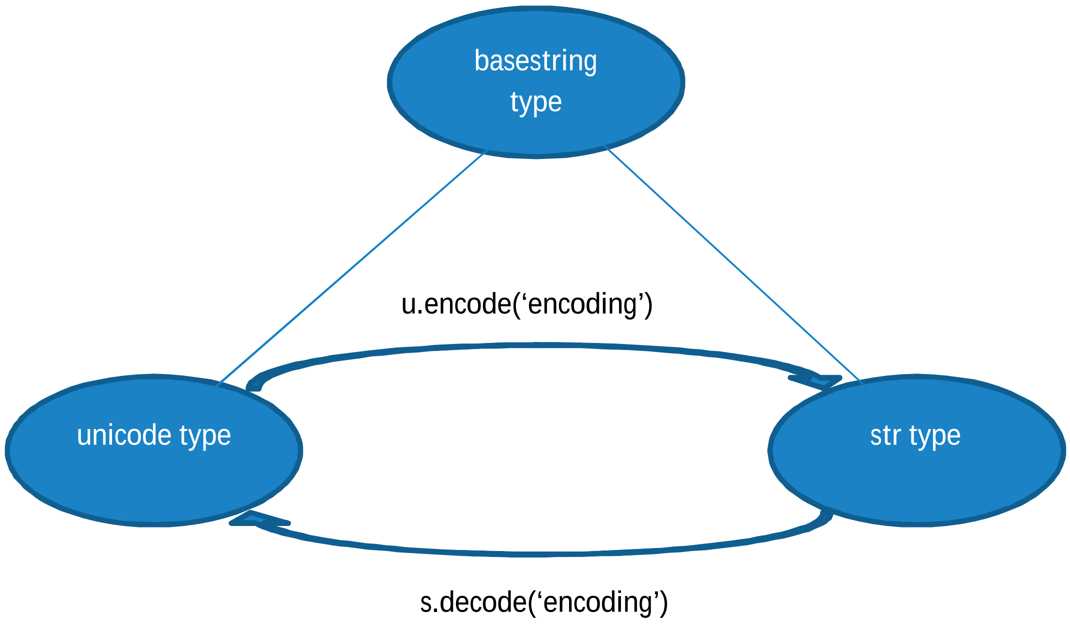
Any string which is to be printed, displayed on a console or stored in a file needs to be *encoded*. (Unicode code points are not encoded). Examples of encoding systems are ASCII, utf-8, utf-16, and iso-8859-1. All of these represent some subset of the Unicode character set by encoding the characters into one or more bytes. Python has a type for storing such encoded strings: it is the "str" type.

Python distinguishes Unicode from encoded strings as follows: it has a two separate classes - "unicode" and "str". (These are both derived from the common ancestor "basestring"). Strings can be converted from the Unicode type to the str type and vice versa using the .encode and .decode methods:

unicodeString.encode('encoding') -> strString[1]

strString.decode('encoding') -> unicodeString

'encoding' in this context represents a python string codec: ascii, utf-8, utf-16, iso-8859-1 are examples of these.



[1] The encode and decode commands have more optional parameters not shown here, for example for handling characters which cannot be encoded/decoded. See reference 1 for more details.

3 Encoding systems

As mentioned above, the python "str" type contains encoded text. The encoding system used can be chosen when creating the string.

Some of the basic characteristics of the various encoding systems:

Encoding	Bits per character	Description
ASCII	7	Only defines 128 characters: mainly English, non-accented characters
ISO-8859-1	8	Also called "latin-1". Defines the same characters as ASCII (in the lower 128 codes), plus a set of Western-European non-ASCII characters.
ISO-8859-2, ISO-8859-3, …. ISO-8859-15	8	Similar to ISO-8859-1, but the non-ASCII characters cover different world regions e.g. ISO-8859-3 is South European, ISO-8859-8 is Latin/Hebrew etc.
UTF-8	Between 8 and 32	Can encode the entire Unicode character set. The first 128 codes correspond one-to-one with ASCII (and with ISO-8859-x)
UTF-16	Between 16 and 32	Uses either 2 or 4 bytes to represent all of the Unicode code points. Incompatible with any of the other encodings mentioned here.

If you try to encode a string (from Unicode) into one of these encoding systems, it will only work for characters which are capable of being encoded. So, for example, if you try to encode "München" into ASCII this will not be possible.

A Python "str" type ultimately just contains a series of bytes. Python has no knowledge of what encoding system was used to create this string. In Python version 2.x, the default string encoding system is ASCII i.e. if Python is not told what encoding system the string uses, it will assume it was ASCII. So if a "str" type is received from elsewhere (e.g. a file, or outputted from an API), Python will try to handle it as ASCII. If it is not ASCII, and contains non-ASCII characters, we are likely to get a python exception.

See the sample interactive python session below. The computer on which the string 'München' was created uses iso-8859-1 as the default encoding. So we can create a Unicode string (python type "Unicode") by calling the decode method, with as parameter the encoding system ISO-8859-1. However, if we try to decode from ASCII, then it fails.

```
>>> iso_8859_1String = 'München'
>>> unicodeString = iso_8859_1String.decode('iso-8859-1')
>>> unicodeString
u'M\x81nchen'
>>>
>>> invalid = iso_8859_1String.decode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError:  'ascii'  codec  can't  decode  byte  0x81  in  position  1:  ordinal  not  in
range(128)
>>>
```

4 Python strings and external files

When you use Python to read text from an external file, then that text will be encoded in one of the encoding systems mentioned above. If the text only contains ASCII characters, then it will make no difference whether the encoding was in ASCII, UTF-8 or any of the ISO-8859-x standards: the bytes representing ASCII characters are all the same for these standards. You can freely print, display or output this text to an API and it should work fine (assuming none of these expect UTF-16, which would be unusual).

However, if the file contains UTF-16, or if it contains non-ASCII characters in one of the other encodings, then two decisions are required:

- how do you want to store this text in your python code, and
- how do you want to output it?

You could choose to *store* the text internally (in your Python code) in Unicode, or in any of the encoding standards mentioned earlier. You can simply use the .decode method on a string read from the file (this converts it to Unicode) and then optionally use the .encode method to encode it to some other string encoding. (You could also do nothing, meaning it stays in the same encoding that the file used, but you will eventually need to know what that encoding was if you want to print or display the text correctly).

The consensus seems to be that the world is moving towards Unicode, so it probably makes sense to store the text in the Python Unicode type.

When it comes to *outputting* the text, you need to know what encoding is expected by the module or device receiving the text. Windows computers in Western Europe typically use ISO-8859-1 as the default encoding. So if you are in Western Europe and want to print text to the display you should probably use this encoding. Chances are, this encoding was also used in the file you read from, so in that case you could keep this encoding throughout.

The diagrams below show two scenarios where text is read from a file, and outputted to a device.

In Figure 1, we read text from a file which uses encoding A. We convert to encoding B for storage in our Python module. The output module requires encoding C, so we convert again at output.

In Figure 2, we store the text in our Python module as Unicode. So on input we convert from encoding A to Unicode, and on output we convert from Unicode to encoding C (required by output).

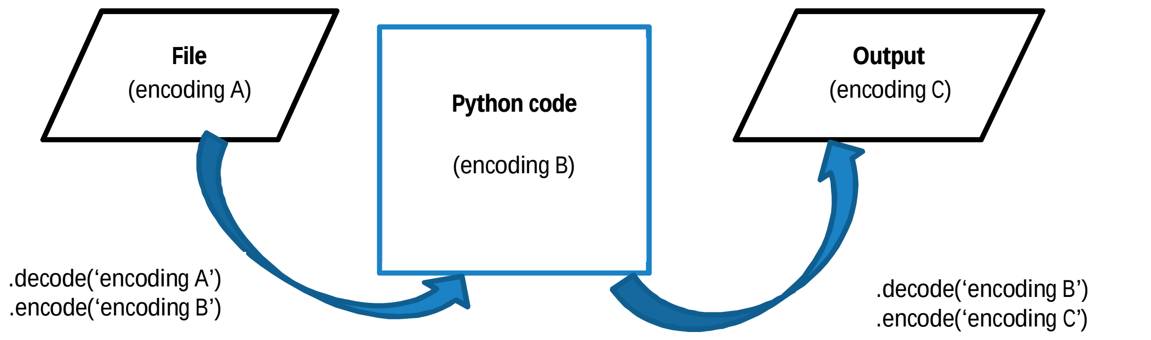


Figure 1: Reading from a file, converting to another encoding for storage, and outputting in a 3rd encoding format

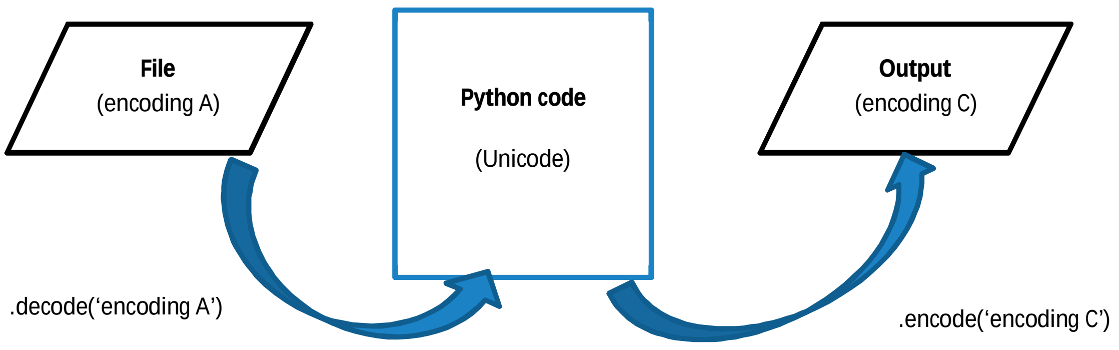


Figure 2: Reading from a file, storing in Unicode, and outputting in a 2nd encoding format

5 The StormTest API

As a general rule, any text inputted to the StormTest API functions should be UTF-8 encoded[2]. If the text only contains ASCII characters, then of course a string of plain ASCII bytes is fine. But if non-ASCII characters are included, UTF-8 encoding should be used.

Let's look at an example. Again an interactive python session will be used. (You can replicate this yourself by just opening a Windows cmd window, and typing "python". If Python is correctly installed, you should see an interactive python session with the prompt >>>).

First we import the StormTest client API, then connect to a server, and then reserve a slot:

```
>>> import stormtest.ClientAPI as st
30/05/2012 11:52:34 INFO     : Completed Init of StormTestClient Version Release, Build
2.7.1.18822
30/05/2012 11:52:34 INFO    : Initial Config Server http://stormtest16:8001/. Machine Name
STENO, User Name fergus
>>> st.ConnectToServer('stormtest16','fergus')
30/05/2012 11:52:49 INFO   : Now using Config Server http://stormtest16:8001
30/05/2012 11:52:49 INFO   : StormTest Client version 2.7.1.18822
30/05/2012 11:52:49 INFO   : Server Version: 2.7.2.19278
True
>>> st.ReserveSlot(14,'default')
30/05/2012 11:53:01 INFO   : The script will run on slot 14
30/05/2012 11:53:01 INFO   : Connecting to video server port 14
………
```

Now, we create a test step name variable, to be used in the BeginTestStep API function. This will contain a non-ASCII character (on windows you can type such characters by holding down the Alt key while typing in a code e.g. Alt+129 gives ü).

```
>>> testStepName = 'München'
```

Because this Windows machine uses iso-8859-1 as default encoding, this is how the string testStepName is encoded. Now we try to pass this into BeginTestStep API:

```
>>> st.BeginTestStep(testStepName)
30/05/2012 11:57:15 ERROR  : Exception in BeginTestStep: Fault: <Fault 0: 'Request from client
does not contain valid XML.'>
……..
```

This fails because the string contains a non-ASCII character and it's not encoded in UTF-8. So we first decode the string from iso-8859-1 to Unicode, and then encode into utf-8:

```
>>> unTestStepName = testStepName.decode('iso-8859-1')
>>> u8TestStepName = unTestStepName.encode('utf-8')
```

Now we try again and it works (the function returns "True").

```
>>> st.BeginTestStep(u8TestStepName)
True
```

One final note: with python you can also concatenate methods like this:

```
>>> st.BeginTestStep(testStepName.decode('iso-8859-1').encode('utf-8'))
True
```

[2] An exception to this is the WriteDebugLine API function: this accepts multiple string input formats, including ASCII, iso-8859-1, utf-8 and the python Unicode type.

5.1 The OCR function

The StormTest OCR (Optical Character Recognition) functions OCRSlot and OCRFile return a string representing the text that was read from the video or file.

For OCRSLot: this string can be contained in two different Python types:

1) If the string only contains ASCII characters, the string is a normal python "str" type
2) If the string contains one or more non-ASCII characters, the string is a python "Unicode" type.

For OCRFile: the returned string is *always* of type "Unicode", as in 2) above.

In case 1) above, the string returned by OCRSlot can be used in the normal way – it is just a string containing plain ASCII bytes. The string can be printed, displayed or used in a StormTest API function as it is.

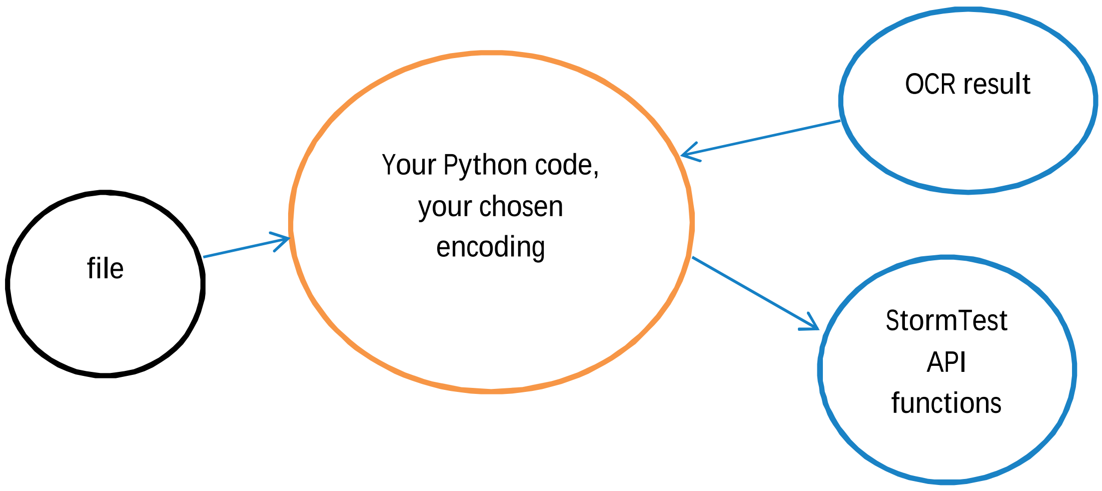
In case 2), you need to be aware that the Unicode type which is returned needs to be converted to a "str" type (with encoding) before it is printed or outputted in any way.

6 Practical solutions for StormTest

There are a number of simple, practical solutions that can be implemented in users' python scripts, to avoid any problems with non-ASCII characters when using StormTest.

The steps to follow are:

1) Choose in which format you want to store text in your python code
2) Convert text returned from OCR into that format (if necessary)
3) Write wrappers to convert from that format into UTF-8 for StormTest API functions



These steps are explained in more detail below.

6.1 Which format to store text in your python code?

If you are located in Western Europe, then there is a good chance that your Windows computer uses ISO-8859-1 as default for non-ASCII characters. This is probably the format that would be used in text files on your computer. So this would be a good choice.

For the rest of the world, you could use Unicode. But you would have to make sure to convert any text read from a file, or received from any APIs, into Unicode. You could also choose one of the other ISO-8859-x encodings.

Finally, if you pass strings into the StormTest APIs quite a lot (e.g. you create test step names and test step comments with strings that might contain non-ASCII characters), you could choose UTF-8 as your encoding. Then these can be passed straight into StormTest API functions.

6.2 Handling OCR returned text

If you expect non-ASCII characters to be returned from OCR calls (e.g. you regularly OCR non-English language menu items, or you set the OCR default language to be something other than English), then you should check the type of the string returned, and convert to your chosen string type if necessary.

Example:

If you choose to store strings in ISO-8859-1, you could do the following

```
ret = OCRSlot([36, 243, 38, 18])
if type(ret[4][0][1]) is unicode:
    ocrText = ret[4][0][1].encode('iso-8859-1')
else:
    ocrText = ret[4][0][1]
```

If you choose to store strings in UTF-8, you could do the following

```
ret = OCRSlot([36, 243, 38, 18])
if type(ret[4][0][1]) is unicode:
    ocrText = ret[4][0][1].encode('utf-8')
else:
    ocrText = ret[4][0][1]
```

If you choose to store strings in Unicode, you could do the following

```
ret = OCRSlot([36, 243, 38, 18])
if type(ret[4][0][1]) is unicode:
    ocrText = ret[4][0][1]
else:
    ocrText = unicode(ret[4][0][1])
```

6.3 Wrapping StormTest API calls

StormTest API functions require text passed into them to be encoded in UTF-8 format. It is possible to write wrappers which convert text to UTF-8 before passing it to the function. (If wrapper functions already exist, this functionality can be added to them).

Note: You *must* know which encoding your python text strings use, to be able to write such a wrapper.

Example:

Say you choose to store your text using ISO-8859-1 encoding (see section 6.1 above). You could write wrappers which convert this to UTF-8 for each StormTest API call. Below is an example of such a wrapper, for the BeginTestStep function. (The wrapper also handles the case where incoming text is Unicode – this should not be necessary but it makes the wrapper more error-tolerant).

```
def RegisterStartTestStep(stepName):
    """
    Description:   Register start of a test step
    Parameters:    step Name
    Returns:       None
    Note:          The incoming text is expected to be in iso-8859-1 encoded format.
                   We try to encode this to utf-8 which is the default format for the
                   StormTest API.
    """
    if type(stepName) is str:
        try:
            # try converting from iso-8859-1
            utf8_stepName = stepName.decode('iso-8859-1').encode('utf-8')
        except:
            #not iso-8859-1 – try using the string as it is
            utf8_stepName = stepName
    elif type(stepName) is unicode:
        # convert Unicode to UTF-8 str type
        utf8_stepName = stepName.encode('utf-8')
    else:
        WriteDebug("Could not determine type of stepName")

    #now call the StormTest API
    try:
        StormTest.BeginTestStep(utf8_stepName)
    except:
        WriteDebug("Exception in BeginTestStep")
```

It is important that the wrapper is written with a particular input encoding in mind. If, for example, you convert text into UTF-8 and feed it into a wrapper function which expects ISO-8859-1 (as above), it will get converted a second time and incorrect characters will result.