PEP 263 - PEP 0263 -- Defining Python Source Code Encodings

Abstract

This PEP proposes to introduce a syntax to declare the encoding of
a Python source file. The encoding information is then used by the
Python parser to interpret the file using the given encoding. Most
notably this enhances the interpretation of Unicode literals in
the source code and makes it possible to write Unicode literals
using e.g. UTF-8 directly in an Unicode aware editor.

Problem

In Python 2.1, Unicode literals can only be written using the
Latin-1 based encoding "unicode-escape". This makes the
programming environment rather unfriendly to Python users who live
and work in non-Latin-1 locales such as many of the Asian
countries. Programmers can write their 8-bit strings using the
favorite encoding, but are bound to the "unicode-escape" encoding
for Unicode literals.

Proposed Solution

I propose to make the Python source code encoding both visible and changeable on a per-source file basis by using a special comment at the top of the file to declare the encoding.

To make Python aware of this encoding declaration a number of concept changes are necessary with respect to the handling of Python source code data.

Defining the Encoding

Python will default to ASCII as standard encoding if no other encoding hints are given.

To define a source code encoding, a magic comment must be placed into the source files either as first or second line in the file, such as:

```
# coding=<encoding name>
```

or (using formats recognized by popular editors)

```
#!/usr/bin/python
# -*- coding: <encoding name> -*-
```

or

```
        #!/usr/bin/python
        # vim: set fileencoding=<encoding name> :
```

More precisely, the first or second line must match the regular expression "coding[:=]\s*([-\w.]+)". The first group of this expression is then interpreted as encoding name. If the encoding is unknown to Python, an error is raised during compilation. There must not be any Python statement on the line that contains the encoding declaration.

To aid with platforms such as Windows, which add Unicode BOM marks to the beginning of Unicode files, the UTF-8 signature '\xef\xbb\xbf' will be interpreted as 'utf-8' encoding as well (even if no magic encoding comment is given).

If a source file uses both the UTF-8 BOM mark signature and a magic encoding comment, the only allowed encoding for the comment is 'utf-8'. Any other encoding will cause an error.

Examples

These are some examples to clarify the different styles for

defining the source code encoding at the top of a Python source
file:

1. With interpreter binary and using Emacs style file encoding
comment:

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
import os, sys
...

#!/usr/bin/python
# -*- coding: iso-8859-15 -*-
import os, sys
...

#!/usr/bin/python
# -*- coding: ascii -*-
import os, sys
...
```

2. Without interpreter line, using plain text:

```
# This Python file uses the following encoding: utf-8
import os, sys
...
```

3. Text editors might have different ways of defining the file's
encoding, e.g.

```
#!/usr/local/bin/python
# coding: latin-1
import os, sys
...
```

4. Without encoding comment, Python's parser will assume ASCII text:

```
#!/usr/local/bin/python
import os, sys
...
```

5. Encoding comments which don't work:

Missing "coding:" prefix:

```
#!/usr/local/bin/python
# latin-1
import os, sys
...
```

Encoding comment not on line 1 or 2:

```
#!/usr/local/bin/python
#
# -*- coding: latin-1 -*-
import os, sys
...
```

Unsupported encoding:

```
#!/usr/local/bin/python
# -*- coding: utf-42 -*-
import os, sys
...
```

Concepts

The PEP is based on the following concepts which would have to be implemented to enable usage of such a magic comment:

1. The complete Python source file should use a single encoding.
   Embedding of differently encoded data is not allowed and will result in a decoding error during compilation of the Python source code.

   Any encoding which allows processing the first two lines in the way indicated above is allowed as source code encoding, this includes ASCII compatible encodings as well as certain multi-byte encodings such as Shift_JIS. It does not include encodings which use two or more bytes for all characters like e.g. UTF-16. The reason for this is to keep the encoding detection algorithm in the tokenizer simple.

2. Handling of escape sequences should continue to work as it does
now, but with all possible source code encodings, that is
standard string literals (both 8-bit and Unicode) are subject to
escape sequence expansion while raw string literals only expand
a very small subset of escape sequences.

3. Python's tokenizer/compiler combo will need to be updated to
work as follows:

1. read the file

2. decode it into Unicode assuming a fixed per-file encoding

3. convert it into a UTF-8 byte string

4. tokenize the UTF-8 content

5. compile it, creating Unicode objects from the given Unicode data
and creating string objects from the Unicode literal data
by first reencoding the UTF-8 data into 8-bit string data
using the given file encoding

Note that Python identifiers are restricted to the ASCII

subset of the encoding, and thus need no further conversion
after step 4.

Implementation

For backwards-compatibility with existing code which currently
uses non-ASCII in string literals without declaring an encoding,
the implementation will be introduced in two phases:

1. Allow non-ASCII in string literals and comments, by internally
treating a missing encoding declaration as a declaration of
"iso-8859-1". This will cause arbitrary byte strings to
correctly round-trip between step 2 and step 5 of the
processing, and provide compatibility with Python 2.2 for
Unicode literals that contain non-ASCII bytes.

A warning will be issued if non-ASCII bytes are found in the
input, once per improperly encoded input file.

2. Remove the warning, and change the default encoding to "ascii".

The builtin compile() API will be enhanced to accept Unicode as
input. 8-bit string input is subject to the standard

procedure for
encoding detection as described above.

If a Unicode string with a coding declaration is passed to compile(),
a SyntaxError will be raised.

SUZUKI Hisao is working on a patch; see [2] for details. A patch
implementing only phase 1 is available at [1].

Phases

Implementation of steps 1 and 2 above were completed in 2.3,
except for changing the default encoding to "ascii".

The default encoding was set to "ascii" in version 2.5.

Scope

This PEP intends to provide an upgrade path from the current
(more-or-less) undefined source code encoding situation to a more
robust and portable definition.

References

[1] Phase 1 implementation:
http://python.org/sf/526840

[2] Phase 2 implementation:
http://python.org/sf/534304

History

1.10 and above: see CVS history
1.8: Added '.' to the coding RE.
1.7: Added warnings to phase 1 implementation. Replaced the
Latin-1 default encoding with the interpreter's default
encoding. Added tweaks to compile().
1.4 - 1.6: Minor tweaks
1.3: Worked in comments by Martin v. Loewis:
UTF-8 BOM mark detection, Emacs style magic comment,
two phase approach to the implementation

Copyright