Internationalizing Python

Martin von Löwis

Humboldt-Universität zu Berlin

Abstract

When writing software for people speaking different languages and coming from different cultures, a large amount of work is spent on adapting the software to those languages. Internationalization helps to reduce that work. This paper discusses approaches to internationalization in Python and presents a library to support some of them.

1 Internationalization and Nationalization

When software developers first started customizing their software to different cultural regions, the only possible solution was to make a copy of the software and adopt it to the cultures and languages. While this is the most flexible answer to the questions, it is also the one that involves most work. Initially, the work might be acceptable. However, as the different versions continue to evolve, additional efforts are necessary synchronizing them. The process of adopting the software to a certain language and culture is called nationalization, because it is often done on a per-country basis.

In order to simplify the nationalization process, it is desirable to separate the language-dependent parts of the software from the language-independent ones. If done right, both parts can evolve reasonably independent from each other. For example, the language-dependent part could be translated to additional languages (like Greek or Armenian) without changing the language-independent part. Vice versa, the algorithmic part could be extended with additional features, and only those new features would need reflection in the different translations.

The techniques used to allow simple nationalization and separation of language-dependent and language-independent parts are collectively called internationalization. Because this is a long word and because there are 18 letters between 'i' and 'n', this is often abbreviated as i18n.

Today, different aspects of internationalization are distinguished:

- Different character sets are used for different languages, and for the same language on different operating environments.
- Text displayed by the program needs to be translated.
- Cultures use different notations for common data, like calendar dates and postal addresses.
- When entering text, different typing systems are used.
- When displaying text, fonts, writing directions and other typographic aspects are relevant.
- The documentation for the software must be available in the native language of the user.
- The programming environment itself might adapt to the native language of the programmer.

First I will discuss them, and then present solutions.

1.1 Character Sets

The biggest cultural difference between people is that they speak different languages. Usually, computer users use written communication when interacting with the computer, that is, they type text on a keyboard and read text from the display. In order to allow machine processing of text, character codes were invented. In a character set, each character is assigned a number, which is called the code point.

Most character sets use one byte to present a single character, allowing for a maximum of 256 characters. Since there are much more characters used, different character sets are necessary. Since most languages use less than 256 characters, single-byte character sets are possible. Common examples for these character sets are:

- the American Standard Code for Information Interchange (ASCII), supporting English,
- the ISO 8859 series of character sets, covering most European languages and other Latin-based writing systems, as well as Arabic and Hebrew and

- vendor specific character sets (IBM code pages, Apple Macintosh character sets).

Because the computer industry is traditionally English-centered, displaying English text in addition to the native text is an additional requirement . Most of these character sets achieve ASCII compatibility by copying the ASCII characters. This is possible because ASCII uses only code points up to 128.

Writing systems that do not use letters, like Chinese and Japanese, often include more than 256 characters. Two different solutions are used: a multi-byte character set (MBCS) uses a different number of bytes to represent a single character; the wide character set (WCS) uses a fixed number of bytes for all characters. One advantage of a MBCS is that ASCII compatibility is possible: the ASCII characters are always represented as a single byte; multiple bytes are only used for the native characters. Designing a MBCS is not straight-forward. One requirement is that a byte sequence has at most one interpretation; the encoding should not be ambiguous. Another requirement is recoverability: If a byte gets altered, or if the beginning of a text is lost, the following text should still be understandable. Common MBCS systems are:

- Shift-JIS, used for Japanese and
- BIG FIVE, for Chinese.

Using a MBCS in a program is not easy. For example, in C programming, it is not possible anymore to get the nth character by using C array operations. Instead, one has to parse the string from the beginning until n-1 characters have been processed. Likewise, the length of the string in characters is not related to the amount of bytes used for the string.

A wide character set does not have the limitations. Two kinds of wide characters are currently in use: 2-byte characters and 4-byte characters. In both cases, the traditional C string operation can be implemented again. The only difference is that, when indexing, the index needs to be multiplied with the number of bytes per character in order to get the byte position. There is a clear disadvantage to WCS systems: the encoding is not guaranteed to be null-byte free. Both for the single-byte character sets, and the MBCS systems, the byte 0 could be avoided, simply by avoiding the code point 0. In a WCS system, many non-zero code points are encoded using a null byte. This breaks software that relies on null terminated strings, like the UNIX system call interface.

In order to avoid null bytes, some of the wide character sets avoid most of the 256 possible bytes. They only use 94 of them, which allows to define encodings that are even 7-bit-clean. Examples of such character sets are:

- the Korean standard KS C 5601 (in the EUC-KR encoding),
- the Japanese standard JIS X 208, and
- the Chinese standard GB 2312.

Those character sets include some European scripts as well, usually Latin, Greek, and Cyrillic. Still, in a mixed-language text, additional character sets might be necessary. ISO 2022 defines a mechanism for switching between character sets using escape sequence. If this mechanism is used, those character sets again become MBCS because of the escape sequences.

It seems desirable to have a wide character set that covers all languages that could ever be found in a text document. The Unicode consortium has defined the character set Unicode [Uni96] which covers most of the current languages. It is related to the international standard ISO 10646, which copied a large part of the Unicode code points. A detailed discussion of Unicode is given later on.

1.2 Message Output

One of the early concerns of internationalization is the number of strings spread throughout a program text. Some of those strings are in a natural language, whereas others have technical uses (like http protocol commands). Only the former ones are interesting for i18n; the existence of the latter ones makes it difficult to develop automatic translation tools.

Strings are commonly used for the following purposes:

- Display messages to the user, like error messages and help messages.
- In GUI environments, construct menus and dialogs using those strings.

For the latter problem, some GUI environments already provide solution. On Microsoft Windows, menus and dialogs are stored in separate files, and the application logic operates on a menu as a whole. Nationalization can be achieved by translating the menu, and then having the program logic load the translated menu instead.

This approach is not easily applicable to message texts, because the message texts are often tied into the program logic. For example, if the developer introduces new features, those features probably can produce new error messages and require an update to the help messages. The developer is then used to placing the message directly into the program source, which makes i18n support difficult.

The solution, of course, is to separate the message text from the place where it is displayed. The separated messages are called message catalog or string table. The program logic then needs to access a specific string in the message catalog, which can be thought of as a Python dictionary. Two different key types are in use:

- integer keys: give a unique number to each string and
- string keys: use the original string itself as a key.

One problem with both approaches is that the key might be invalid. In that case, fall-back mechanisms are required. In the case of the integer key, a separate string must be kept around, usually the English text. In the string key case, the key itself can be used.

The disadvantage of using integer keys is that the program becomes less readable and more difficult to maintain. The disadvantage of string keys is that the implementation of the message catalog is less efficient, and that there is a danger of an English language string that needs different translations when used in different places.

One special aspect of message catalogs relates to parametrized messages. Parameters in messages are often achieved in a printf fashion, i.e. by adding %s in appropriate places. The problem arises when the message parameters are inserted in different order in different languages.

1.3 Locales

Among the first standardized interfaces to nationalization issues are the ANSI C locales. This interface is a set of C functions that allows program-independent customization of certain aspects of the program. Those aspects are classified into categories:

- LC_COLLATE defines the collation (i.e. sorting) order for character strings. For example, accented Latin characters are sorted different in different languages.
- LC_CTYPE defines the behavior of the character type functions. For example, the question whether 'ö' is a letter depends on the locale.
- LC_MONETARY defines the symbol used for the currency, the character separating currency units and other aspects of currency display.
- LC_TIME defines the formatting of date and time. For example, the month names are different in different languages.

Not all of those categories are equally useful. When displaying a sorted list of strings, the sorting should follow the collating rules of the language, which means that the LC_COLLATE category should be used. On the other hand, using the local currency symbol is not always desirable. A famous spreadsheet program used to have a bug where a currency amount of '42.00 DM' would get converted to '$ 42.00' when switching between the German and the English versions. Of course, exchange rates would need to be taken into account.

Other regional settings have to be taken into account, like the formatting of the address (number of digits and letters in the ZIP code, presence of state/province). These are not supported by ANSI locales.

1.4 Character Input

Text is usually entered using the keyboard. However, no keyboard accommodates keys for all available characters. Several approaches are used to allow input of additional characters:

- With modifier keys (e.g. shift), the keys are essentially re-labeled.
- Dead keys are an approach to enter diacritic marks. For example, the French accents can be entered by first typing the accent sign, followed by the accented letter. When adding the cedilla as a possible dead key to the Linux system, Linus Torvalds jokingly said: "Only a dead cedilla is a good cedilla".
- Compose keys serve essentially the same purpose as dead keys. However, the key is only dead if pressed together with a modifier, the compose key.

These simple systems only help when there is a small number of characters to type. For the East Asian languages, different typing systems where designed. Some are based on phonetic input; some are modeled after the hand-writing technique. In order to simplify the typing, the system often tries to guess the character based on a few key strokes. If the guess is wrong, the user can select alternatives in an additional window.

Because of the complex operations involved, the typing systems are called input methods.

1.5 Character Display

Text display is usually based on fonts. A font contains a graphical representation for each character. This means that a font is closely related to the character set. Fonts for the same character set differ in size and style. When using multiple fonts, the glyphs should be of the same size and similar style. In order to simplify the font selection procedure, fonts can often be grouped into families or sets.

When displaying text for a variety of languages, the writing order becomes an important issue. For historical reasons, most displays prefer horizontal text over vertical text. Some languages adopted to this requirement, so languages that would traditionally displayed vertically (e.g. Chinese) are presented horizontally on a computer. Even for horizontal text, there are still languages that differ in the writing direction. Whereas European languages write from left to right, Arabic and Hebrew are typed from right to left. When mixing both writing directions, a complex algorithm called BIDI (bi-directional typing) has to be applied.

For several scripts, the presentation of a single character changes if it is displayed next to another one. In the Arabic script, a character looks different in the middle of a word than in the beginning or the end.

If a text cannot be displayed because of limited terminal capabilities, several work-arounds are possible. The Russian KOI-8R character set features a special case for 7-bit display: When stripping the most significant bit, the text is converted into a transcripted ASCII version. The usual approach to non-displayable characters is to display some standard glyph.

1.6 Documentation

Documentation (either on-line or printed) is usually not so tied into the program logic as the other aspects of internationalization. Still, translating the documentation is likely the largest part of localizing a software system for some country. Two problems are usually related to multi-language documentation: accuracy and space requirements. Accurate documentation can be obtained by using version control: if the English documentation changes in some place, the translations should change as well. The space problems are usually handled at shipping installation time: The user is asked to select which languages the documentation should provide. For online documentation, support for multiple languages is desirable even if the user chooses to install only a single language.

1.7 I18N and the Interpreter Proper

The issues mentioned above are primarily interesting for the comfort of the end user, the one who will ultimately use the software being internationalized. An interesting question is whether the programming tools itself should speak the language of their users, who happen to be programmers.

It is not so clear that localizing the programming tools is a good thing. Most computer experts today have some knowledge of the English language, in many cases as part of their training. Also, computer literature often restricts itself to a simple language in order to avoid ambiguities - which might just be introduced with translations.

Nevertheless, the following items need consideration:

- Localization of the keywords of the programming language,
- Localization of identifiers, and
- Localization of the documentation.

2 Supporting Multiple Character Sets

In order to support multiple character sets in Python, flexible conversion routines need to be supported. Rather than defining new mechanisms, the author decided to built the Python support upon Unicode. Conversion of a character set will then take place to and from Unicode. Unless a character set goes beyond Unicode capabilities, this conversion is loss-less.

2.1 Unicode

Unicode is a character set that covers the following languages and scripts: Latin (Basic Latin, extensions for European languages, IPA extensions, diacritical marks), Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayam, Thai, Lao, Tibetan, Georgian, Hangul, Hiragana, Katakana, Bopomofo, Kanbun, CJK, and various mathematical and technical symbols [Uni96]. The current version is 2.0.

Related to Unicode is the international standard 10646, adopted in 1993 [ISO93]. This standard supports an (abstract) character set that is organized in groups, planes, rows and cells. As each of these hierarchy levels is defined to support 256 variants, the character set contains a total of 2L**32 cells (or code points). A plane then contains 65,536 code points. Unicode is defined as the plane 0, which is also called the Basic Multilingual Plane (BMP).

There are currently no characters defined outside the BMP. When defining Unicode, some of the glyphs used in Chinese, Japanese and Korean had to be dropped because they would have exceeded the limit of 2**16 characters. Instead, characters identifying the same concept were assigned the same code point, which is called Han unification. Since other scripts, like ancient scripts or various African and Central Asian scripts are not covered at the moment, it is anticipated that there will be character allocations outside the BMP at some day.

As a result, a BMP code point can be represented in 2 bytes, which is called a rune. The natural representation of a rune is an encoding called UCS-2. It is defined to be big-endian, although implementations are free to use the native endianness for the type wchar_t. Because ISO-8859-1 is located in row 0 of the BMP, conversion to ISO-8859-1 is very simple for UCS-2. The same holds for the encoding UCS-4, which uses four bytes per character, and thus supports full ISO 10646.

Unfortunately, the UCS encodings are not null-byte free, leading to problems with legacy software. Thus, alternative encodings, called transfer forms, have been defined. UTF-8 [Dav94] is a multi-byte encoding which uses a prefix byte to indicate the number of following bytes. Each following byte contributes six bits to the rune. As a result, a BMP rune will use between one and three bytes. As a special feature, plain ASCII is left as-is, because the prefix bytes and the follow-ups are above cell 128. Unfortunately, this gives problems with legacy systems that require 7-bit encodings, such as UseNet News. The UTF-7 encoding [GD94] solves this problem by introducing an escape byte (+), followed by a sequence of base-64 codes and a unescape byte (-). This coding still leaves the ASCII letters unmodified, but is more difficult to process than UTF-8.

In order to allow a two-byte encoding to support more than the BMP, the planes 0xD8 to 0xDF have been reserved. Two consecutive runes of these planes are combined, each one contributing ten bits. These 20 bits allow to support 16 additional planes, 1 to 16. The encoding that takes this conversion into account is called UTF-16.

2.2 The wstring Module

The author has designed a Python module *wstring* providing an object similar to the built-in *string* type, but consisting of Unicode characters instead. Similar to the string module, the wstring module bases on a *wstrop* module. Where available, this module uses the native *wchar_t* type of the C library. Otherwise, it defines *wchar_t* as *unsigned long*, giving 32 bit characters on those systems. Thus, the library allows to represent the full ISO 10646 character set.

A *wstring* object behaves similar to strings, i.e. indexing, slicing, and concatenation are available. More work needs to be done to support other string functions, like *split*. Unlike string objects, wstring objects are not directly recognized by the parser. Instead, they can be created using operations of the wstring module. Initially, the following operations are available*:* *from_ucs2, from_ucs2, from_utf7, from_utf8, from_utf16,* and *decode*.

Of those decoding functions, UTF-8 offers the greatest flexibility. The reader should recall that plain ASCII is left unmodified by UTF-8, so Python source code is identical in UTF-8 and ASCII. The only exception is within strings and comments, where characters above 128 are allowed. Now suppose the developer is using a Unicode text editor to write a Python program. He is free to place comments and strings in whatever language Unicode supports. If he saves the source text as UTF-8, the resulting file is valid Python input. The only thing that is missing is the creation of wstring objects from the Unicode UTF-8 strings. Because UTF-8 is anticipated as the typical encoding produced by the editor, the function *L* is a synonym to *from_utf8*. So the developer could write

```
from wstring import L
#...
s=L("<put Unicode string here>")
```

The only remaining inconvenience is that Unicode strings have to be wrapped with a constructor. Otherwise, they behave just like ordinary strings. Future Python versions might simplify this support even more.

In order to convert a wstring object into some encoding, each wstring object supports the operations *ucs2, ucs4, utf7, utf8, utf16*, and *encode*. The reader might wonder what the decode and encode operations offer. The wstring module contains an extensible mechanism for converting between character sets. Two conversion mechanisms are possble: dictionary-based and function-based. In the dictionary-based approach, a converter can register a dictionary of how to map a character set to Unicode and back. This approach is best used for small character sets. A built-in converter is available for ISO-8859-1, converters for the other ISO-8859 sets are available with the distribution. In order to convert between ISO-8859-1 and ISO-8859-2, the following code could be used:

```
def L1toL2(s):
  import wstring,iso8859
  return wstring.decode("LATIN1",s).
     encode("LATIN2")
```

LATIN1 is registered as aliases for the official character set name, ISO_8859-1:1987, and LATIN2 likewise. The wstring module supports the following operations for installing converters:

- install_encoding_map expects a character set name and two dictionaries.
- install_alias expects the alias name and the official name.
- install_encoding_function expects a character set name and two conversion functions.

The conversion functions are primarily meant for more complex conversion operations, such as converting between JIS X 0208 and Unicode. It allows the integration of existing converters, which only have to be wrapped with the appropriate Python functions. Each converter receives as an argument the string to convert and possible flags. It either returns the converted string or raises an exception. The following flags are defined so far:

- SKIP_INVALID: unconvertible characters should be ignored,
- UTF7_QUOTE_OPTIONALS: If UTF-7 is used for e-mail headers, the number of available characters is even more restricted than in the message bodies. ASCII characters that are not allowed here will be quoted just as non-ASCII characters if this flag is given.
- UCS_SWITCH_BYTEORDER: When encoding or decoding UCS strings, they should be considered little endian.
- UCS2_DO_UTF16: Characters outside the BMP should be encoded using UTF-16

If the conversion fails, either because the input string is ill-formed or because the target character set is not powerful enough, the exception *wstring.ConvertError* is raised, unless the SKIP_INVALID flag was given. The flags can be specified as an optional parameter to *decode* and *encode*.

3 The intl Module

In order to export the functionality provided by the C libraries, the intl module was designed to support the various locale-related aspects. Currently, it wraps two libraries: the C locales, and the message catalogs.

3.1 Locales in Python

The locale support in the intl module just wraps around the C *setlocale* and *localeconv* functions. *setlocale* needs to be called in the beginning of a programming to indicate that the program is locale-aware. A typical call would be

```
intl.setlocale(intl.LC_ALL,"")
```

The first parameter is used to request support for all categories. The second parameter indicates the locale requested (like "English"); an empty string request the system default. On Unix, the system default is obtained from the LANG environment variable. The predefined "C" locale re-activates the inital settings.

The setlocale function returns the name of the activated locale, or None if the operation failed. Omitting the second parameter requests the current locale setting rather than setting it. Modifying individual categories is possible as well.

Once locale support is activated, some Python functions start behaving different. For example, the function *time.strftime* will use the local date format. In addition, the *localeconv* function will return information about the current locale in a dictionary. The keys of this dictionary are named as the corresponding fields returned from the localeconv C function.

Unfortunately, it is not possible to modify the string comparison operator to work locale-dependent, because to many programs would break then. Instead, a new function *intl.strcmp* will be provided which does locale-aware string-comparison.

3.2 Message Catalogs

As mentioned earlier, there are different approaches to message catalogs. The initial intl implementation is based on the Uniforum message catalogs, which are implemented by Solaris as well as GNU gettext [Dre95]. The catgets interface (X/Open message catalogs) has the disadvantage of using numbers to indicate message translations instead of the messages self. Also, the Win32 FormatMessage interface has been initially rejected for the same reason. A unification of these interfaces on the Python level, or alternative implementations similar to anydbm/dumbdbm are currently being investigated.

Using Uniforum message catalogs, the messages of an application are divided into different domains, and each domain has its own catalog. For example, the C library usually has a different catalog than the application program. When retrieving a message, either the domain has to be passed to *intl.dgettext*, or the domain is initially set using *intl.bindtextdomain*, and only the requested message must be passed to *intl.gettext*.

Here are several ways of producing the message catalogs. This is the one suggested by the GNU gettext documentation. First, all translatable message strings in the source code must be marked. In order to disturb readability as little as possible, the wrapper function around each string should be called _ (underscore). This use of the underscore usually does not interfere with its meaning in the interactive mode. A gettextized module would then begin with

```
import intl
_=intl.gettext
```

The marking of translatable strings must be done manually. However, the gettext mode for GNU Emacs simplifies this procedure. In the end, a possible line could look like

```
  l = Tkinter.Label(parent, text =
_("Hello, world"))
```

Once all strings are marked, the program xgettext can be used to extract the marked strings and produce a message catalog. A line in this catalog would then look like

```
#: t.py:4
msgid "Hello, world"
msgstr ""
```

A translator could then insert the correct translation, e.g. translating the above message to "Hallo, Welt". The GNU emacs mode simplifies locating the source code so that the translator can check the program context were the message is used. GNU gettext also supports updating of translated message catalogs by copying the old translations to the new version of the catalog.

Once the catalog is finished, it needs to be compiled and installed. The details of this procedure depend on the operating system. Finally, the program will use the messages according to the LC_MESSAGE locale category.

When discussing message catalogs for use with C, an issue is always the parametrization of message when the order of the parameters depends. In Python, this issue can be solved using dictionary arguments to the % operator:

```
  param={'name': 'John', 'number': 4}
  l=Tkinter.Label(p, text =
        ("There are %(number)d people
called %(name)s") % param)
```

Another issue is the translation of messages of wide strings. The simplest solution is to introduce a L_ function, which first looks up the UTF-8 string in the catalog, and then converts the returned UTF-8 translation into a wide character string.

4 Nationalizing Graphical User Interfaces

Using the techniques presented above, it is pretty easy to write software that allows nationalization to Western European languages, because it involves only one character set, ISO-8859-1. This character set is supported by both MS Windows and X11, and conversion to Macintosh character sets is well-understood.

However, it does not help if one can translate messages to, say, Armenian, if the windowing is not capable of receiving Armenian input and producing Armenian output. This chapter describes the state of the art as seen by the author.

4.1 Tkinter

A growing number of graphical Python applications is written using Tk [Ous95]. Although John Ousterhout repeatedly mentioned that i18n is an issue, no 'official' solution is available so far. Instead, some patches are floating around.

For X11, internationalization was first introduced in release 5, providing concepts called input methods and font sets. An input method is a set of procedures that allow the user to enter arbitrary characters. With X11R6, some of the concepts were revised and streamlined. An X11 application must explicitly request input-method-based operation, e.g. by calling the appropriate functions to convert keystrokes to characters. The input produces either multi-byte or wide characters. In order to output text, X11R5 introduced font sets, which could be selected using the X logical font names. A font set would group a number of similar fonts, which could use different character sets, thus allowing multi-language output. In X11R6, this approach was extended by output methods. Output methods allow more complex rendering of output, e.g. by taking writing directions into account. The author is still investigating how output methods can be used in Tk. In addition to the input and output methods, X11R6 also contains a set of conversion routines for various character sets and encodings. In particular, the conversion between the Japanese JIS X 0208 and Unicode could be exported to the *wstring* module.

On MS Windows, things are much simpler, at least on NT. Unicode support is built into NT, which means that there are functions to create 'wide character windows'. Whenever a user types a key in such a window, this key is delivered as a Unicode character to the application. NT also supports various input methods. Rendering of text is simple as well: There is a Unicode variant of the function to draw a string. The only obstacle is that there are only a few Unicode fonts. Unfortunately, this approach fails for Windows 95, as this system does not implement much of the Win32 Unicode API functions.

Once Tk is capable of displaying Unicode text, another issue is the programming interface to access text properties. The current proposal is to use UTF-8 strings, as this nicely fits into the current Tcl mechanisms. As the final step, Tkinter could be extended to silently convert text resources to UTF-8 if the value is a wide string.

4.2 PythonWin

PythonWin is based on the Microsoft Foundation Classes[HKT96], which themselves support either single-byte, multi-byte, or wide characters. The current PythonWin release enables the multi-byte character support, where a single character might be represented by multiple bytes in a Python string. It seems possible to introduce wide characters in PythonWin, although this would probably mean that all strings passed to PythonWin should be wide strings. This possibility has not yet been investigated further. The limitations of Windows 95 apply: a full Unicode PythonWin implementation would not run on Windows 95.

5 Documentation

When internationalizing Python, two documentation issues arise: the Python documentation itself, and documentation for Python applications. At the moment, the online Python documentation mainly consists of a library reference and a tutorial. The author is not aware of efforts to translate this documentation into various languages, although this would be certainly desirable. There is, however, secondary literature on Python in different languages. At the moment, there are books available in English and German ([Web97]), and plans for books in other languages.

For application documentation, HTML is more and more used. This is especially handy as Grail [CNRI97] could be used to display the documentation inside a Tkinter application. Recent extensions to HTTP support requests to ask for HTML pages in a specific language. Currently, Grail does not support these protocol elements, although future versions probably will. Also, it has been proposed to allow Unicode encodings for in HTML. With the modules presented here, it would be possible to store such HTML files. In order to display them, the proposed Tk extension would need to be implemented.

6 Translating Source Code

As mentioned before, the programmer itself is in a situation similar to the end user: She might not speak English as a native language. Since Python is so tightly bound to English, a programmer might have problems getting started with Python. Certainly, documentation in various languages is an important issue as explained before.

Sometimes, the programmer wants to go beyond offline documentation when using her native language during programming. In Python, nationalization is interesting for keywords, identifiers, exceptions, comments, and doc strings. This section is intended to stimulate discussion about the topic, as no implemented solution is known to the author.

Certainly, the developer is free to choose whatever identifiers she wants. However, the choice is restricted to ASCII characters at the moment. If this restriction was lifted to allow arbitrary character sets, the interpreter would need to know what a letter is in that character set, so that it could detect the identifiers. Technically, it seems possible to allow arbitrary bytes above 128 in identifiers as well. However, such an extension of the language might render Python source code unreadably in some editors.

The same restrictions hold for the keywords themselves. In addition, experiences of Microsoft with Visual Basic suggest that Python keywords should not be considered for localization. One reason is that any Python interpreter would need to support all localized keyword sets. For example, consider a program was written with keywords translated to German:

```
Klasse Tier: #...
  def Laufe(selber,strecke):
    für i im bereich(1,strecke):
      selber.schritt()
```

Today's Python interpreters would reject this source code because they don't recognize the syntax. Furthermore, there is a requirement that keywords are not allowed as identifiers. This restriction is difficult to control as new localizations will introduce new keywords.

Comments and doc strings don't share these limitations: arbitrary bytes are allowed for both of them, unless these bytes indicate the end-of-comment (line break) or end-of-string (quote sign). Since the encodings presented before follow these restrictions, arbitrary native text can be put into these documentations.

Still, using a native language other than English in these places has draw-backs: For comments, multiple copies of the same text in different languages hurt readability rather than improving it. For doc strings, only one strings is allowed in the first place. As a result, a reader not speaking the language of the programmer will suffer from not being able to read the documentation.

The simplest approach is to take the localized documentation off-line as discussed above. Advanced solutions could suppress documentation in languages foreign to a reader.

For comments, a special mark-up could be used to indicate the language of the comment to the editor, which would use the mark-up to suppress undesired comments.

Doc strings could be translated using the same techniques as strings displayed to the end-user. The source code would copy only one version of the string, which is then used as a key to a data base of translations stored externally. A browser could make the lookup automatically, following the programmer settings and displaying the original string if no translation is available. It would be desirable if such a browser could automatically know the character set and encoding of the doc string before displaying it. This could be achieved by convention (key in ASCII, value in a per-language encoding), or by additional mark-up.

Finally, exceptions need special consideration. If caught, their textual representation is totally irrelevant. If not caught, they usually indicate a flaw in the program, so they are interesting mainly to the programmer. Still, users of Python programs will be confronted with trace back print outs sooner or later as well.

Class exceptions are printed using their string representation. The developer can choose any localization strategy for these exceptions. String exceptions are more difficult to localize: In order to catch them, a reference to the exact string being raised is necessary, so there should be no translation process involved. In addition, the exception parameters are printed using their string representation. Since the trace-back is meaningless without the source code in many cases, these limitations seem acceptable.

7 Summary and Availability

Various aspects of internationalization in the Python context have been discussed. For character set issues, a wstring module was represented. Locale issues can be processed using the intl module. Both are available from http://www.informatik.hu-berlin.de/~loewis/python/i18n.html.

As a major obstacle, the lack of GUI libraries that support wide characters has been identified. Future work should concentrate on extending the available libraries.

8 Literature

[Uni96] The Unicode Consortium. The Unicode Standard, Version 2.0. Addison-Wesley, 1996

[ISO93] ISO/IEC 10646:1993. Information Technology – Universal Multi-Octet Character Set (UCS). ISO, 1993

[Dav94] M. Davis. UCS Transformation Format 8 (UTF-8). Document ISO/IEC JTC1/SC2/WG2 N 1036, 1994

[GD94] D. Goldsmith, M. Davis. UTF-7 – A Mail-Safe Transformation Format of Unicode. RFC 1642, 1994

[Dre95] U. Drepper. GNU gettext. Free Software Foundation, 1995

[Ous95] J. K. Ousterhout. Tcl und Tk. Addison-Wesley, 1995

[HKT96] F. Heinemann, G. Krüger, N. Turianski. Einführung in Visual C++ 4. Addison-Wesley, 1996

[CNRI97] CNRI. Grail 0.3 Home Page. http://grail.cnri.reston.va.us/grail/, CNRI, 1997

[Web97] Python.org Webmaster. Python Documentation. http://www.python.org/doc/#books, 1997

[Lut97] Mark Lutz. What's New? http://shell.rmi.net/~lutz/#S1