

# Python and Unicode

Unicode Support in Python

EuroPython Conference 2002  
Charleroi, Belgium

Marc-André Lemburg

EGENIX.COM Software GmbH  
Germany

## Python & Unicode: Overview

1. Introduction to Unicode
2. Python's Path to Unicode
3. Using Unicode in Python
4. The Future



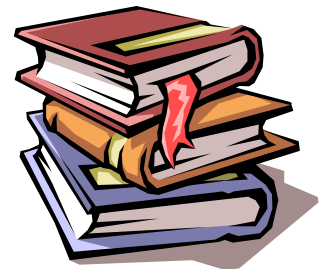
# Python & Unicode: Part 1

1. Introduction to Unicode
2. Python's Path to Unicode
3. Using Unicode in Python
4. The Future



## Introduction to Unicode: The Problem

- Storing scripts: human readable text data
  - Localization (l10n) and Internationalization (i18n) of software and GUIs
  - Basis for national language and script support
  - Common ground for textual information exchange

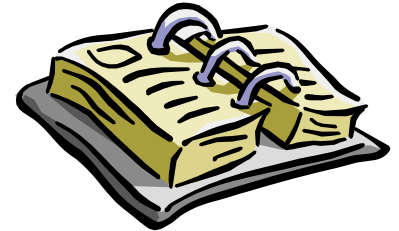


## Introduction to Unicode: First Approximations

- Mappings of bytes to characters: Code Pages (CP)
  - **Problem:** Attaching the encoding information to the data
    - No support in the OS for maintaining per data buffer encoding information
    - Each application/protocol has to implement its own way of dealing with encodings
  - **Problem:** Scripts with many characters
    - e.g. Asian scripts use shift information to address all characters using 8 bits
  - **Problem:** Not available for ancient scripts
    - e.g. Old Italic
  - **Problem:** Incompatible mappings for the same script
    - e.g. Latin-1 and Windows CP-152x

## Introduction to Unicode: The Unicode Consortium Solution

- One encoding for all scripts of the world
- ASCII compatibility (even Latin-1)
- Includes character meta data
  - Case mapping information
  - Numeric conversion
  - Character category information
- Accounts for scripts using different orientations
- Enables sorting and normalization support



Also see the Unicode Consortium web-site at <http://www.unicode.org/>

## Introduction to Unicode: Other Solutions

- ISO 10646:

The ISO way of defining a Universal Character Set

- Code point compatible to Unicode
- Some minor differences in interpretation
- "Closed Source":  
standard documents are only available on a pay-per-page basis
- Independent organization

# Introduction to Unicode: What is a Character ?

- Unicode Terminology



– Graphemes:



This is what users regard as a character.

– Code Points:



U+0301  
Combining  
Accent Acute

This is an Unicode encoding of the string.

– Code Units:



0xCC 0x81  
UTF-8 for U+0301

This is what the implementation stores (UTF-8).



## Introduction to Unicode: Statistics

- Unicode 3.0
  - released: September 1999
  - $17 * 2^{16} - 1 = 1114111 = 0x10FFFF$  code points (17 planes)
  - 49 194 assigned code points
  - No assigned code points outside plane 0, the Basic Multilingual Plane (BMP) which fits into 16 bits
- Unicode 3.1
  - released: May 2001
  - $17 * 2^{16} - 1 = 1114111 = 0x10FFFF$  code points (17 planes)
  - 94 140 assigned code points
  - Assigned code points in plane 1, no longer fits into 16 bits

## Introduction to Unicode: Connecting to the Real World

- Conversions between Unicode and Code Pages (CP)
  - Mapping tables are available at the Unicode web-site
  - Examples:
    - Latin-1 (Western Europe)
    - CP-1250 (Windows Western Europe)
    - KOI8-R (Cyrillic)
- Conversions between Unicode and other encodings
  - Special encoders/decoders ([codecs](#)) are required for each encoding
  - Examples:
    - Shift JIS, EUC-JP (Japanese)
    - Big5, EUC-TW (Chinese)

## Introduction to Unicode: Encoding Issues (Part 1)

- Round-trip safety
  - Unicode .. Encoding .. Unicode
    - UTF-7 (7-bit encoding, for e.g. email)
    - UTF-8 (8-bit encoding, 1-4 bytes per code point)
    - UTF-16 (16-bit encoding, endianness is an issue)
    - UTF-32 (32-bit encoding, memory / disk space intense)
    - These are loss-less encodings !
  - Encoding .. Unicode .. Encoding
    - Most code pages (IBM, Microsoft, etc.)
    - Asian encodings: Chinese, Japanese, Korean, Vietnamese (CJKV)
    - Not necessarily loss-less !

## Introduction to Unicode: Encoding Issues (Part 2)

- Identifying Encodings
  - Byte Order Marks (BOMs)
    - Originally: Marker for little vs. big endian for UTF-16/32
    - Microsoft: uses BOMs as Unicode file magic
  - Auto-Detection:
    - often requires knowledge about the encoded data
    - BOMs + file headers usually go a long way (e.g. for XML-data)
    - Protocols can have encoding meta information (e.g. HTTP Content-Type)

## Introduction to Unicode: Internal Storage Formats (Part 1)

- Unicode Transfer Format 8 (**UTF-8**):
  - 8-bit variable length encoding: 1-4 bytes per code point
  - **Problem**: indexing and slicing
- Universal Character Set 2 (**UCS-2**):
  - 16-bit fixed length encoding: 2 bytes per code point
  - **Problem**: not all code points are representable
- Unicode Transfer Format 16 (**UTF-16**):
  - 16-bit variable length encoding: 1-2 words per code point
  - **Problem**: indexing and slicing

## Introduction to Unicode: Internal Storage Formats (Part 2)

- Universal Character Set 4 (UCS-4):
  - 32-bit fixed length encoding: 4 bytes per code point
  - Requires ISO 10646 standards conformity
  - **Problem:** memory consumption
- Unicode Transfer Format 32 (UTF-32):
  - 32-bit fixed length encoding: 4 bytes per code point
  - Requires Unicode standards conformity
  - **Problem:** memory consumption

For a discussion about UTF-16 vs. UTF-32 see e.g.  
<http://mail.nl.linux.org/linux-utf8/2000-08/msg00025.html>

## Introduction to Unicode: Unicode Implementations

- Java, Windows NT/2000/XP
  - Basis: Unicode 2.x
  - 16-bit code units (UCS-2 / UTF-16)
  - **Problem:** Unicode 3.1 introduces characters which require two code units per code point (UTF-16)
- GNU libc 2.x
  - Basis: ISO 10646
  - 32-bit code units (UCS-4)
- Python 1.6 and later
  - Basis: Unicode 3.0
  - Versions 1.6 – 2.1: 16-bit code units (UCS-2)
  - Version 2.2+: 32-bit code units as configuration option (UCS-4)

## Introduction to Unicode: Comparing Unicode Strings

- **Problem:** There are multiple ways to encode a characters

Example:     $\acute{e} = e + \acute{}$

- **Solution:** Normalization
  - Recode Unicode strings to help finding a common ground for comparisons (Unicode Annex #15)
  - Different forms are available:
    - FORM D: "Canonical Decomposition"
    - **FORM C:** "Canonical Decomposition, followed by Canonical Composition"
    - Other forms for normalization



## Introduction to Unicode: Sorting Unicode Strings

- **Problem:** Sorting order is locale/application specific

Example:

German phone book sorting order: A ... AE ... **Ä** ... AB ... B ...

- **Solution:** Collation Support
  - Recode Unicode strings into **Collation Elements** using a collation table (see Unicode Annex #10)
  - The Collation Elements can then be compared on an lexicographic basis as is done with ASCII

## Introduction to Unicode: Conclusion

- Unicode ...
  - solves real world problems
  - reduces the time / money effort it takes to internationalize software
  - simplifies managing text data
  - is a mature and stable standard
  - is open enough for everyone to adapt

## Python & Unicode: Part 2

1. Introduction to Unicode
2. Python's Path to Unicode
3. Using Unicode in Python
4. The Future



## Python's Path to Unicode: Motivation

- Why Unicode ?
    - All modern programming languages will have to support Unicode (sooner or later)
    - See the "Introduction to Unicode"
  - Possible paths to Unicode support:
    1. Switch to Unicode as basic string type
      - problem:** compatibility
    2. Provide a separate Unicode type and integrate it with the existing string type
      - problem:** integration
- >>> Guido van Rossum chose [Path 2](#).

## Python's Path to Unicode: History

**Background:** In 1999 Hewlett-Packard worked on a project called "e-speak" which was partly written in Python; for the i18n support they needed a Unicode type, so they joined the Python Consortium and contracted CNRI to have it implemented.

**October 1999:** Guido van Rossum subcontracted Fredrik Lundh to write an Unicode aware regular expression engine (SRE) and Marc-André Lemburg for the Unicode integration (deadline March 1st)

**November 1999:** First version of the Unicode integration proposal

**March 2000:** CVS checkin of the Unicode implementation and the SRE engine

**September 2000:** CNRI releases Python 1.6 with Unicode support

## Python's Path to Unicode: Goals of the Implementation

- **Integration:**  
Existing 8-bit strings and Unicode should integrate well with the ultimate goal to use them interchangeably
- **Ease of use:**  
Unicode should be just as easy to use as 8-bit strings
- **Conversions:**  
An extensible codec (encoder / decoder) library should enable built-in conversions between Unicode and other encodings
- **Backward compatibility:**  
Should be maintained if at all possible

## Python's Path to Unicode: When Strings meet Unicode

- Unicode is "more" than an 8-bit string:
  - coercion is always towards Unicode
- **Problem:** 8-bit strings don't carry any encoding information
  - When coercing 8-bit strings to Unicode Python must make an encoding assumption: the **default encoding**
  - Default encoding is a startup run-time parameter
- Question:  
**Which default encoding to choose as default ?**

## Python's Path to Unicode: Default Encoding: UTF-8 ...

- **First approach:**
  - Use **UTF-8** as default encoding
- **Problems:**
  - Variable length encoding (1-4 bytes per code point)
  - **Indexing** can easily fail
  - len(s) not always == number of code points
  - **Slicing** can break the encoding
  - Common encodings like Latin-1 don't map well to UTF-8, e.g. all accented characters require two bytes



## Python's Path to Unicode: ... or let the locale decide ...

- Second approach:
  - Determine the encoding by querying the **current locale**
- Problems:
  - Python code is **not portable**:  
String literal in source code will receive different interpretations depending on the platform
  - Mixing Python code from different origins (locales) will likely fail at run-time
  - Some locales have more than one encoding in common use (e.g. Russia)

## Python's Path to Unicode: ... or let the BDFL decide !

- Final decision by Guido van Rossum:
  - Python's default for the default encoding is ASCII
- **Problems:**
  - Coercion errors are very common for all non-ASCII applications which mix 8-bit strings and Unicode
- **Advantages:**
  - Helps identify the problem areas in programs
  - Encourages: **Explicit is better than implicit !**
  - Works well for ASCII-users

## Python's Path to Unicode: Features of the Implementation

- **Integration:**  
Auto-coercion of 8-bit strings to Unicode based on the default encoding (usually ASCII)
- **Internals:**  
Uses UCS-2 for internal storage, based on Unicode 3.0 (UCS-4 is a configuration option since Python 2.2)
- **Unicode Properties:**  
Provide access to the Unicode property database via string methods
- **Conversions:**  
Provides codecs for most common (Western) encodings; high quality codecs for Eastern encodings are available separately

## Python & Unicode: Part 3

1. Introduction to Unicode
2. Python's Path to Unicode
3. Using Unicode in Python
4. The Future



## Using Unicode in Python: Overview

- Creating Unicode objects in Python
- Converting Unicode to other encodings
- Working with files
- Writing a codec (encoder/decoder)

## Using Unicode in Python: Creating Unicode objects

- Unicode literals:
  - `u"Hello World !"` (note the small `u`)
- Unicode from 8-bit strings:
  - `unicode("Hello World !", "latin-1")`
- Unicode from files:
  - `import codecs`
  - `f = codecs.open("myfile.txt", encoding="latin-1")`
  - `data = f.read()`

## Using Unicode in Python: Encoding Unicode

- Using the Unicode method `.encode(data [,encoding])`:

- `u"ndré Le".encode("utf-8")` (note the small `u`)

`== "ndr\xc3\xa9 Le"`

- `u"ndré Le".encode("latin-1")`

`== "ndr\xe9 Le"`

- `u"ndré Le".encode()` (default encoding)

**UnicodeError**: ASCII encoding error: ordinal not in range(128)

## Using Unicode in Python: Working with Files

- The `codecs` module provides Unicode aware wrappers around file objects:

- `import codecs`

Read the data as UTF-8 and convert it to Unicode on-the-fly:

- `file = codecs.open("myfile.txt", encoding="utf-8")`
- `data = file.read()`

Process the Unicode data (here: using Unicode methods):

- `data = data.upper()`

Write back the Unicode as UTF-16

- `file = codecs.open("myfile.txt", "wb", encoding="utf-16")`
- `file.write(data)`



## Using Unicode in Python: Writing Codecs

- A Latin-1 to UTF-8 recoder written as codec (latin1\_to\_utf8.py):

```
import codecs

# Encoding / decoding functions
def encode(latin1_data):
    return unicode(latin1_data, 'latin-1').encode('utf-8'), len(latin1_data)
def decode(utf8data):
    return unicode(utf8data, 'utf-8').encode('latin-1'), len(utf8data)

# StreamCodecs
class Codec(codecs.Codec):
    def encode(self, latin1_data): return encode(latin1_data)
    def decode(self, utf8data): return decode(utf8data)
class StreamWriter(Codec,codecs.StreamWriter):
    pass
class StreamReader(Codec,codecs.StreamReader):
    pass

# Codec registry entry point
def getregentry():
    return (encode, decode, StreamReader, StreamWriter)
```

## Python & Unicode: Part 4

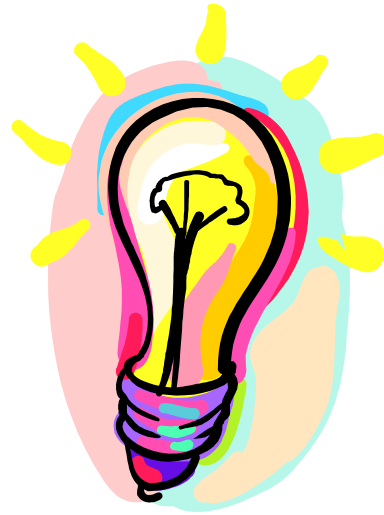
1. Introduction to Unicode
2. Python's Path to Unicode
3. Using Unicode in Python
4. The Future



## The Future: Unicode Support in Python 2.2 and later

- **Internals:**  
Provide support for UCS-4 to fully support Unicode 3.1 and later
- **Unicode Algorithms:**  
Implement the Unicode collation algorithm, the compression algorithm and the normalization algorithms
- **Unicode Helpers:**  
Add helpers which allow indexing Unicode objects based on characters, code points, words and lines
- **Conversions:**  
Add fast codecs for Eastern encodings to the Python core (but as separate download)

## Questions...



Thank you for your time.

## Contact

eGenix.com Software, Skills and Services GmbH

Marc-André Lemburg

Pastor-Löh-Str. 48

D-40764 Langenfeld

Germany

eMail: [mal@egenix.com](mailto:mal@egenix.com)

Phone: +49 211 9304112

Fax: +49 211 3005250

Web: <http://www.egenix.com/>