यूनिकोड 유니코드 אוניקאָד

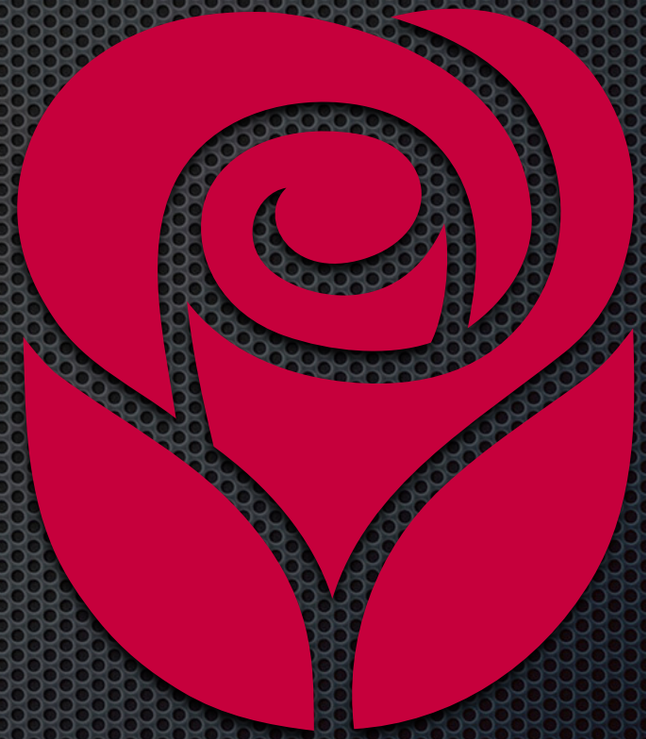
Unicode "Myth Busted"

统一码 統一碼 유니코드 يونيكود

Who We Are

Len Thomas
six@choushi.net

Mike Crute
mcrute@gmail.com
@mcrute



AGINTERACTIVE

Myths

- Why can't everybody just speak English?
- Unicode is not fully covered in UTF-8
- Typographic quotes are not in Unicode

The Basics

Unicode is just text in number clothing

Unicode

Forget everything you know about text

What is Unicode?

- Industry standard for the consistent representation and handling of text
- Latest version of Unicode consists of a repertoire of more than 107,000 characters covering 90 scripts
- Set of 1.2 million code-points ranging from 0 - 10FFFF

Representation in Python

U+0 - U+00FF

```
# Representing the © Symbol in Python
```

```
>>> s = u'\xa9'
>>> s = u'\u00a9'
>>> s = u'\U000000a9'
>>> s = unichr(0xa9)
>>> s = unichr(0x00a9)
>>> s = unichr(0x000000a9)
>>> s = unichr(169)
```

Representation in Python

U+0100 - U+FFFF

```
# Representing the ℞ Symbol in Python
```

```
>>> s = u'\u211e'
>>> s = u'\U0000211e'
>>> s = unichr(0x211e)
>>> s = unichr(0x0000211e)
>>> s = unichr(8478)
```

Representation in Python

U+ > FFFF

```
# Representing the 🍛 Symbol in Python

>>> s = u'\U0001F06B'
>>> s = unichr(0x1F06B)
>>> s = unichr(127083)
```

Font != Code-Point

```
>>> t = 'Â'.decode('utf-8')
>>> repr(t)
u'\u212b'
>>> u = 'Â'.decode('utf-8')
>>> repr(u)
u'\xc5'
>>> t == u
False


>>> wtf
NameError: name 'wtf' is not defined
```

Font != Code-Point

```
>>> from unicodedata import normalize
>>> t = normalize('NFC', t)
>>> u = normalize('NFC', u)
>>> t == u
True
```

windows-1252 hzgb macroman utf-16

Charsets

ascii shift-jis latin-1 utf-8 big5

C Primer

- 3 fundamental data types
- treats memory as an array
- char not capable of holding code-points

Data Type	Size
char	8-bits
int	32-bits
long	64-bits

C Primer

Strings

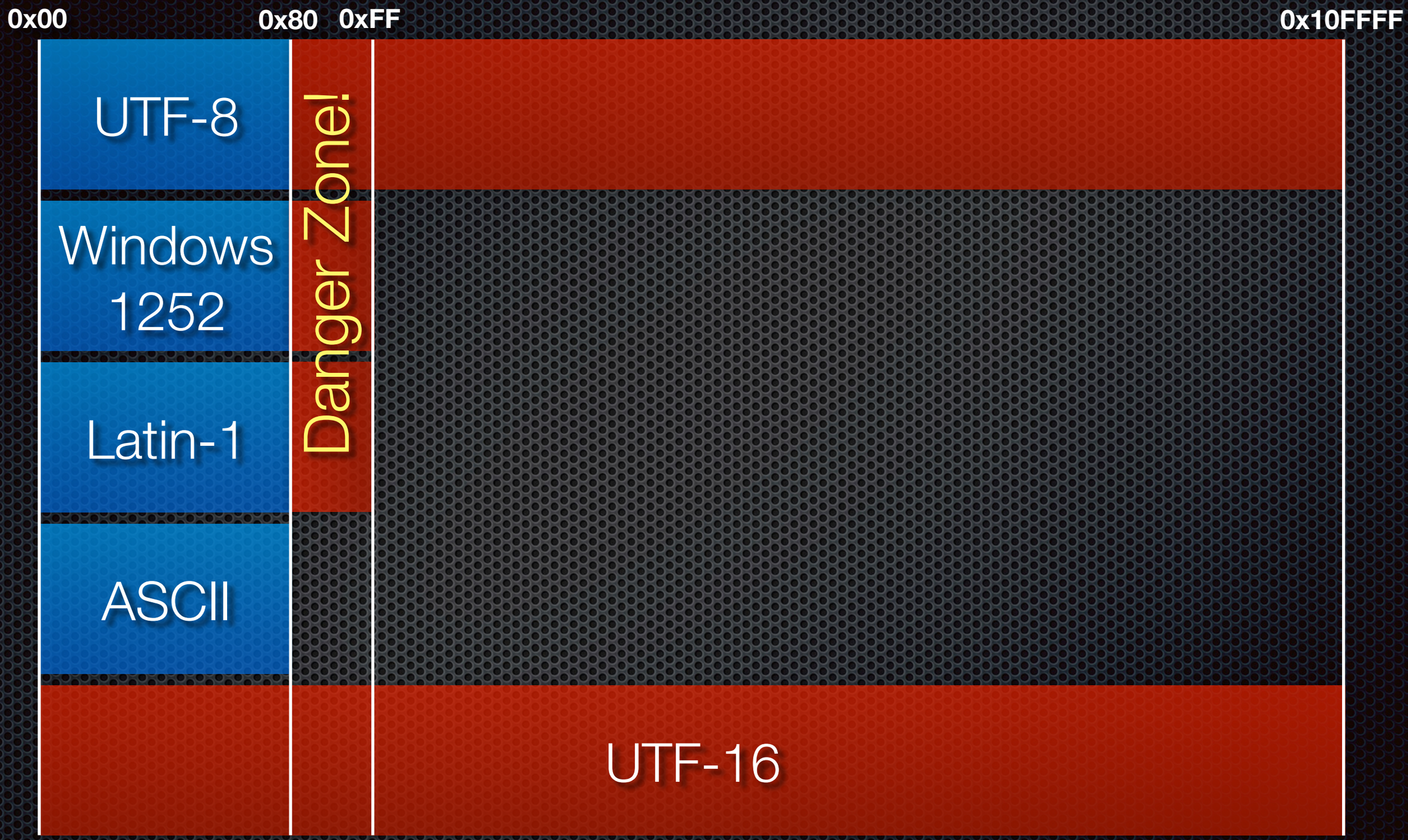
- Null-terminated character array
- Character can't hold code-points
- Much C code relies on this assumption

```
char[10] word = "Hi World!";

['H','i',' ','W','o','r','l','d','!', 0x00]
```

What is a Charset?

- Binary representation of a code-point
- Translation-table of sorts
- Facilitates converting code-points into something storable
- Internally everything is still a byte array


0x00
0x80
0xFF
0x10FFFF
UTF-8
Windows 1252
Latin-1
ASCII
Danger Zone!
UTF-16

Even Latin-1 Can't Save You

```
>>> y = get_data().decode('utf-8')
>>> y
u'\u0645\u0631\u062d\u0628\u0627
\u0627\u0644\u0639\u0627\u0644\u0645'
```

```
>>> print(y)
مرحبا العالم

>>> x = y.encode('utf-8').decode('latin-1')
>>> print(x)
Ù Ø±Ø Ø¨Ø§ Ø§Ù Ø¹Ø§Ù
```

Proper Encoding Can!

```
>>> m = get_data().decode('utf-8')

>>> print(m)
مرحبا العالم

>>> m.encode('latin-1')
UnicodeEncodeError: 'latin-1' codec can't
encode characters in position 0-4: ordinal not
in range(256)
```

What is a Charset?

- Decode: converts bytes to unicode

```
>>> '\xe1\x88\xb4'.decode('utf-8')

u'\u1234'
```

- Encode: converts unicode to bytes

```
>>> u'\u1234'.encode('utf-8')
'\xe1\x88\xb4'
```

UTF-8

- 8-bit Unicode Transformation Format
- UTF-8 encodes each code-point in 1 to 4 octets
- First 128 code-points of the UTF-8 character set use a single octet with the same binary value as in ASCII
- Supported on all modern operating systems

UTF-16

- 16-bit Unicode Transformation Format
- Variable-length character encoding in 2 to 4 octets
- Uses a byte order marker (`\xff\xfe` or `\xfe\xff`)
- Default on Windows operating systems

UTF-8 vs. UTF-16

Round 1

```
>>> u'Got some good \u211e?'.encode('utf-8')
'Got some good \xe2\x84\x9e?'

>>> len(previous_string)
18
```

UTF-8 vs. UTF-16

Round 1

```
>>> u'Got some good \u211e?'.encode('utf-16')
'\xff\xfeG\x00o\x00t\x00 \x00s\x00o\x00m\x00
e\x00 \x00g\x00o\x00o\x00d\x00 \x00\x1e!?\x00'

>> len(previous_string)
34
```

UTF-8 vs. UTF-16

Round 2

```
>>> 'こんにちは、お元気ですか'
'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab
 \xe3\x81\xa1\xe3\x81\xaf\xe3\x80\x81
 \xe3\x81\x8a\xe5\x85\x83\xe6\xb0\x97
 \xe3\x81\xa7\xe3\x81\x99\xe3\x81\x8b'

>>> len(previous_string)
36
```

UTF-8 vs. UTF-16

Round 2

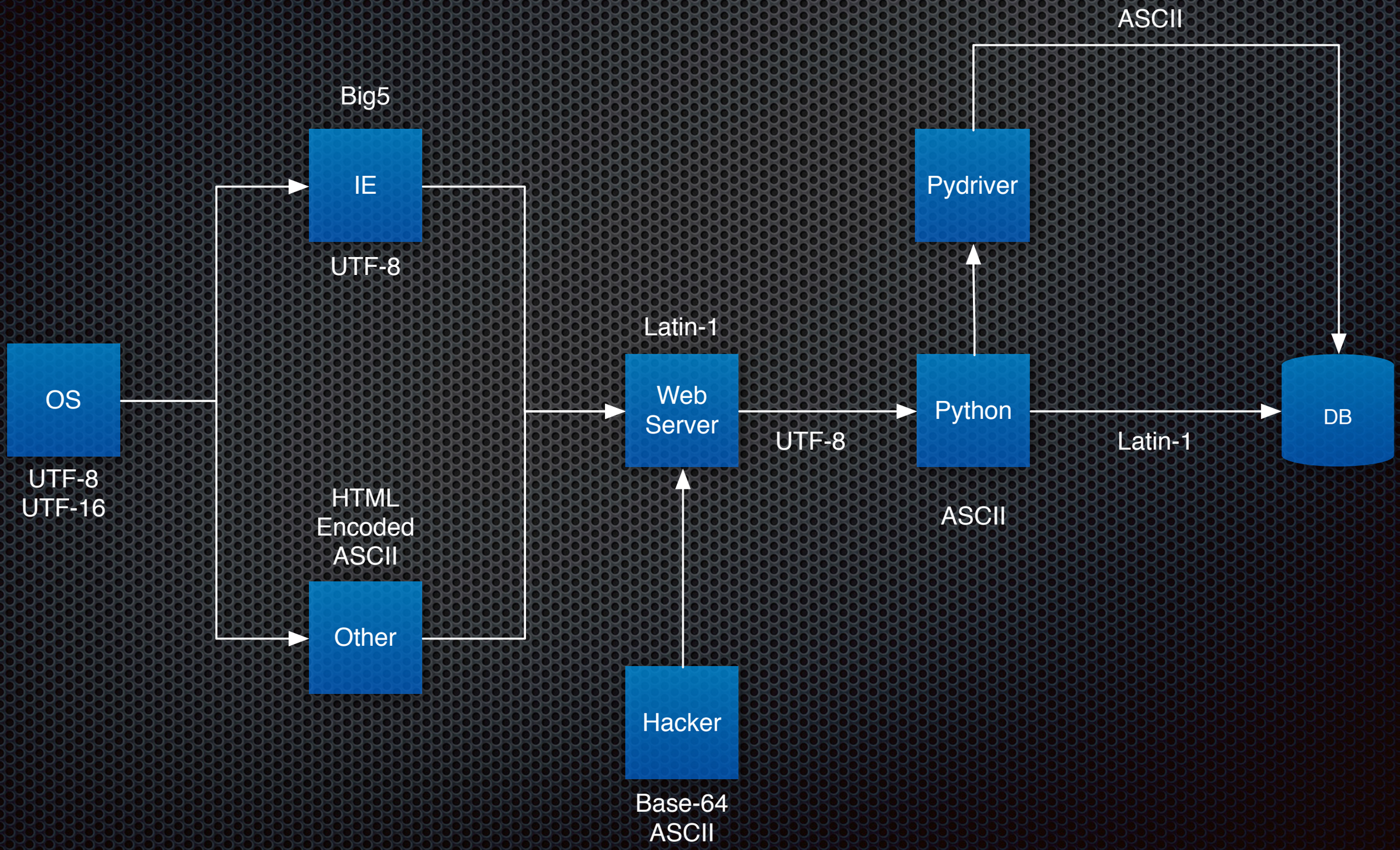
```
>>> a = 'こんにちは、お元気ですか'.decode('utf-8')
>>> a = a.encode('utf-16')
>>> a
'\xff\xfeS0\x930k0a0o0\x010J0CQ\x17lg0Y0K0'

>>> len(a)
26
```

Efficiency

- Latin language (English, Romance Languages)
  - Use UTF-8
- Asian and Russian Languages
  - Use UTF-16
- UTF-32
  - Just don't do it!

The Character Set Waltz



Conversion Advice

1. Accept UTF-8
2. Decode from UTF-8 to Unicode
3. Re-encode UTF-8 to Latin-1
4. Catch encoding exceptions and send back an HTTP 400 (Bad Request)

Conversion Advice

- Get your strs to unicodes
- Don't do `x = "Hello World"`
- Do `x = u"Hello World"`
- Use `unicode(foo)` instead of `str(foo)`
  - Remember to override __unicode__ not __str__, even though unicode(foo) will call foo.__str__

Conversion Advice

- basestring means both str and unicode, use it for isinstance checks
- Use CHARDET
- Set charset in Content-type header
- HTML <meta http-equiv="content-type" />
- XML define an encoding attribute in the XML prolog

Python 2

Awesome! Awesome enough?

Codecs

```
>>> import codecs as c
>>> for line in c.open('foo.txt', 'r', 'utf-8'):
...     print(line)
Line \u123
Line \u185
Line \u111
```

When You Need a Hammer

```
>>> x = u"Hello \u1234\u4568\u7890"
>>> x.encode('ascii', 'strict')
UnicodeEncodeError: ...
>>> x.encode('ascii', 'ignore')
'Hello '
>>> x.encode('ascii', 'replace')
'Hello ???'
>>> x.encode('ascii', 'xmlcharrefreplace')
'Hello &#4660;&#17768;&#30864;'
```

Nifty Tricks

- To convert a string that contains a representation of unicode characters:

```
>>> t = '\\U00020021'
>>> t = t.decode('unicode-escape')
>>> t
u'\U00020021'
```

- To calculate the byte length of a unicode string:

```
len(a.encode('unicode-internal'))
```

Nifty Tricks

- sys module can get default encodings:

```
>>> sys.getdefaultencoding()
'ascii'
>>> sys.getfilesystemencoding()
'utf-8'
```

Python 3

This changes everything, again

unicode Becomes str

- No more unicode
- str is now the old unicode
- Default encoding is UTF-8
- `from io import StringIO`

str Becomes bytes

- bytes function replaces str
  - `bytes(unicode_string, 'utf-8')`
- bytes literals
  - `b"some string"`
- bytes objects immutable but new bytearray
  - most things that take bytes accept bytearray
- `from io import BytesIO`

Other Changes

- basestring has been removed
- Pass open an encoding keyword
- Non-ascii characters allowed in identifiers

Questions?

Thanks

Len Thomas six@choushi.net

Mike Crute mcrute@gmail.com @mcrute

Backup

Internal Representation

```
# if defined(MS_WIN32) && Py_UNICODE_SIZE == 2
#  define HAVE_USABLE_WCHAR_T
#  define PY_UNICODE_TYPE wchar_t
# endif
```

```
# if defined(Py_UNICODE_WIDE)
#  define PY_UNICODE_TYPE Py_UCS4
# endif
...
# if SIZEOF_INT >= 4
typedef unsigned int Py_UCS4;
# elif SIZEOF_LONG >= 4
typedef unsigned long Py_UCS4;
# endif
```

Internal Representation

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t length;
    Py_UNICODE *str;
    long hash;
    PyObject *defenc;
} PyUnicodeObject;
```

Proof For Data Type Size

```c
#include <stdio.h>

int main()
{
    printf("char: %i\n", (int)sizeof(char));
    printf("int:  %i\n", (int)sizeof(int));
    printf("long: %i\n", (int)sizeof(long));
}
```

Chardet Example

```
>>> import chardet
>>> chardet.detect('\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab')
{'confidence': 0.87625, 'encoding': 'utf-8'}

>>> chardet.detect('\xe3\x81\x93\xe3')
{'confidence': 0.505, 'encoding': 'utf-8'}
```

Canonical and Compatible

- Some code-points are equivalent for decomposition
- Canonical equivalence decomposes character to itself plus diacritic marks
- Compatible equivalence decomposes character to its ASCII representation
- NFC, Normal Form C, Canonical De/Re-Composition
- NFD, Normal Form D, Canonical Decomposition