PYTHON'S UNICODE INTERNALS

Benjamin Peterson <benjamin@python.org>

“Modern programs must handle Unicode —Python has excellent support for Unicode, and will keep getting better.” - GvR

PURPOSES

- Explain the history Python's Unicode support.
- Examine in depth the current Unicode implementation.

SOME HISTORY

MORE INNOCENT TIMES

UNICODE IN PYTHON 2

GENESIS - PEP 100 (PYTHON 2.0)

- unicode type
- `codecs` module
- str <-> unicode coercion model
- Simple (4.5 Kloc)

DATA FORMAT

ARRAY OF `UNSIGNED SHORT` CODEUNITS (UTF-16)

UNICODE AS AN OPTIONAL FEATURE

```
$ python -S
Python 2.7.3+ (2.7:f6e74759d740, Jan  1 2013, 23:06:34)
[GCC 4.5.4] on linux2
>>> unicode
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'unicode' is not defined
>>> type(u"Hello, World!")
<type 'str'>
```

PEP 261 - SUPPORT FOR 'WIDE' UNICODE CHARACTERS

`--ENABLE-UNICODE=(UCS2|UCS4)`

'THIS PEP REPRESENTS THE LEAST-EFFORT SOLUTION.'

UNICODE IN PYTHON 3

A BRAVE NEW WORLD

KEY PYTHON 3 UNICODE CHANGES

- `str` is now a Unicode type
- The old `str` type becomes `bytes`
- `bytes` and `str` are not implicitly coercible
- Identifiers are Unicode

NEW STRESSES ON AN OLD SYSTEM

WHAT'S A PATHNAME?

UNIX: BYTESTRINGS

WINDOWS: UTF-16

OSX: UTF-16 (WITH WEIRD NORMALIZATION)

PYTHON: `STR`

UNIX IS PROBLEMATIC

```
>>> filename = b"myfile-\xeX\xe.txt"
>>> open(filename, "w").write("hi")
>>> filename.decode("utf-8")
Traceback (most recent call last):
  File "", line 1, in
UnicodeDecodeError: 'utf-8' failed
>>> filename.decode("utf-16")
Traceback (most recent call last):
  File "", line 1, in
UnicodeDecodeError: 'utf16' failed
```

PEP 383 (PYTHON 3.1)

NON-DECODABLE BYTES IN SYSTEM CHARACTER INTERFACES

PRESERVING UNDECODABLE BYTES

On decode (e.g. `os.listdir`), map undecodable bytes to private use codepoints. (U+E000 to U+F8FF)

On encode (e.g. `os.stat`), map private use characters back to bytes.

ROUND-TRIPPING BYTESTRING FILENAMES

```
>>> filename = b"myfile-\xef\xef.txt"
>>> new = os.fsdecode(b"myfile-\xeX\xe.txt")
>>> new
'myfile-\udceX\udce.txt'
>>> os.fsencode(new)
b"myfile-\xeX\xe.txt"
```

UTF-16 VS. UTF-32 REPRESENTATION

NARROW BUILD

```
>>> char = u"\U0001F07F"
>>> len(char)
2
>>> unicodedata.category(char[0])
'Cs' # Surrogate
```

WIDE BUILD

```
>>> char = u"\U0001F07F"
>>> len(char)
1
>>> unicodedata.category(char[0])
'So' # Symbol
```

UTF-16 WASTES MEMORY

UTF-32 WASTES **TONS** OF MEMORY

PEP 393 (PYTHON 3.3)

FLEXIBLE STRING REPRESENTATION

IDEA

USE SMALLEST REPRESENTATION POSSIBLE FOR A GIVEN STRING

PEP 393 DATA REPRESENTATION

Maximum codepoint	Data size	ASCII flag	Example
127	1	1	Hello, World!
255	1	0	Schlüssel
65535	2	0	لعربية
1114111	4	0	☃??

INVARIANT: SMALLEST REPRESENTATION IS ALWAYS USED.

E.G. ASCII STRINGS ALWAYS USE ONE BYTE STRINGS

PEP 393 IMPLICATIONS

EVERYTHING IS A CODEPOINT!

- Narrow vs wide builds abolished.
- `len(s)` gives length in codepoints.
- Indexing a string always gives a valid codepoint.

PERFORMANCE CHARACTERISTICS ARE COMPLICATED

ASCII CAN BE BLAZING FAST

(AND A LOT OF THINGS ARE ASCII.)

OTHER CASES ARE LESS CLEAR

COMPLEXITY

LINES IN CORE UNICODE IMPLEMENTATION

- 3.2: 15,000
- hg tip: 20,000

COMPLEXITY

```
#define PyUnicode_GET_SIZE(op)
  \
    (assert(PyUnicode_Check(op)),
  \
     (((PyASCIIObject *)(op))->wstr) ?
  \
      PyUnicode_WSTR_LENGTH(op) :
  \
      ((void)PyUnicode_AsUnicode((PyObject *)(op)),
  \
       assert(((PyASCIIObject *)(op))->wstr),
  \
       PyUnicode_WSTR_LENGTH(op)))
```

C-API

OLD C-API

```
Py_ssize_t count_ascii(PyObject *string) {
    Py_UNICODE *data = PyUnicode_AS_UNICODE(string)
;
    Py_ssize_t i, count = 0;
    for (i = 0; i < PyUnicode_GET_SIZE(string); i++
) {
        if (data[i] <= 127)
            count++;
    }
    return count;
}
```

NEW C-API

```
Py_ssize_t count_ascii(PyObject *string) {
    if (PyUnicode_READY(string) < 0)
        return -1;
    int kind = PyUnicode_KIND(string);
    void *data = PyUnicode_DATA(string);
    Py_ssize_t i, count = 0, len = PyUnicode_GET_LENGTH(strin
g);
    for (i = 0; i < len; i++) {
        if (PyUnicode_READ(kind, data, i) <= 127)
            count++;
    }
    return count;
}
```

OLD C-API MUST CONTINUE TO WORK.

RESULT: *LEGACY STRINGS*

FUTURE WORK

- More performance improvements
- Unicode spec compliance
- More Unicode algorithms
- `re` module could use some work

LESSONS

- Global configuration options are bad.
- It's okay to start simple; evolution is possible.
- It's much easier to preserve compatibility for Python code than C-API clients.
- Optimize for the common case.

QUESTIONS?

Further contact: benjamin@python.org