Why you should use Python 3 for text processing

Python is a great language for text processing.

Each new version of Python–especially the 3.x series–has enhanced this strength.

String (and byte) objects have grown some handy methods.

Built-in functions have improved or been added.

Refinements and additions have been made to the standard library to cover the most common tasks in text processing.

Why I want to give this talk

A long time ago I wrote a book for Addison Wesley called *Text Processing in Python*; the text of it has always been free at http://gnosis.cx/TPiP/.

Showing my gray hairs, this book was quite up-to-date for Python 2.2. A lot has changed since then! (and a lot still has not changed, nor should it).

Many–but not all–of the nice things in Python 3.3 have been back-ported to 2.7.3 (or were already included in 2.x.

Places to go for sources of amazement and intrigue

http://docs.python.org/3.3/whatsnew/

http://docs.python.org/3.2/whatsnew/

http://docs.python.org/3.1/whatsnew/

http://docs.python.org/3.0/whatsnew/

This talk is an impressionistic review of nice-to-have improvements to text processing that have come to python in the last decade, but with an emphasis on 3.x features.

A little bit of Python version history

Python 3.0	2008-12-03
Python 3.1	2009-06-27
Python 2.7	2010-07-03
Python 3.2	2011-02-20
Python 2.7.3	2012-04-09
Python 3.3	2012-09-29

Cool stuff in `collections`

Improvements to `collections` help with many things, but seem to come up particularly often as nice ways to do text processing tasks: e.g. `namedtuple`; `Counter`; `OrderedDict`; `defaultdict`; `ChainMap`.

Most of what's "new in Python 3.1" has also been back-ported to Python 2.7.x.

Most of this is just cool in general, and not just for text processing.

Cool stuff in `collections`

`namedtuple` is particularly useful for dealing with CSV (a text processing matter) and database rows.

```
import csv
from collections import namedtuple
users = open('users.csv')
headers = users.readline()
UserRecord = namedtuple('UserRec', headers,
                                   rename=True)
for row in csv.reader(users):
    print(UserRecord(*row))

# UserRec(first='John', last='Doe', age='39')
# UserRec(first='Sally', last='Wu', age='52')
# UserRec(first='Ruby', last='Sanchez', age='19')
```

Cool stuff in `collections`

`Counter` is widely useful, but one text processing area that often comes up is histograms.

```
>>> from collections import Counter
>>> c1 = Counter('abaracadabara')
>>> c1.most_common(4)
[('a', 7), ('r', 2), ('b', 2), ('d', 1)]
>>> c1['d'] -= 10       # requires 3.3
>>> c1.most_common()[-2:]
[('c', 1), ('d', -9)]
```

Cool stuff in `collections`

`Counter` also allows some common pseudo-arithmetic operations, and is basically a `defaultdict` to value `0`.

```
>>> c2 = Counter('ramalama bim boom')
>>> (c1 + c2).most_common(4)
[('a', 11), ('b', 4), ('m', 4), ('r', 3)]
>>> +c1    # New in 3.3
Counter({'a': 7, 'b': 2, 'r': 2, 'c': 1})
>>> c1['a'], c1['x']
(7, 0)
```

Cool stuff in `collections`

`ChainMap` is a new Python 3.3 collection that looks interesting. It is a "container of containers" that acts seamlessly. In a sneaky way, it is equivalent to a dynamic inheritance hierarchy and an MRO.

```
>>> d1 = {'a':1, 'b':2, 'c':3, 'd':4}
>>> d2 = {'d':5, 'e':6, 'f':7, 'g':8}
>>> chain = ChainMap(d1, d2)
>>> chain['a'], chain['d'], chain['f']
(1, 4, 7)
>>> child = ChainMap({'a':99, 'x':24}, chain)
>>> child['a'], child['c'], child['x']
(99, 3, 24)
```

Cool syntax improvements for built-in collections

I noticed some of the comprehensions only recently:

```
>>> {i:str(i) for i in range(5)}
{0: '0', 1: '1', 2: '2', 3: '3', 4: '4'}

>>> {str(i) for i in range(5)}
set(['1', '0', '3', '2', '4'])

>>> {'1', '0', '3', '2', '4'}
set(['1', '0', '3', '2', '4'])

>>> (i for i in range(5))  # NOT a tuple...
<generator object <genexpr> at 0x1320eb8>
```

Good stuff about Unicode in Python 3.3

Unicode is hard! However difficult you think it is, it is harder than that to get right.

"Most" of the Unicode you care about is in the BMP (Basic Multilingual Plane). In fact, all of Latin-1 is in range 00 of the BMP... *Most isn't all*!

Internal encoding matters. With fixed-width encoding (i.e. `UTF-32/UCS-4`) you use a lot of memory. With variable-width (`UTF-8`), position indexing is very slow. With `UTF-16/UCS-2` you get the worst of everything: not strictly fixed width (i.e. surrogate pairs) and usually wasted memory.

Good stuff about Unicode in Python 3.3

Variable-width encoding in UTF-16

```
>>> h = 'ク'    # CJK UNIFIED IDEOGRAPH-2008A'
>>> m = 'M'     # ASCII 'M'
>>> h.encode('utf-8'), m.encode('utf-8')
(b'\xf0\xa0\x82\x8a', b'M')
>>> h.encode('utf-16'), m.encode('utf-16')
(b'\xff\xfe@\xd8\x8a\xdc', b'\xff\xfeM\x00')
>>> h.encode('utf-32'), m.encode('utf-32')
(b'\xff\xfe\x00\x00\x8a\x00\x02\x00',
 b'\xff\xfe\x00\x00M\x00\x00\x00')
```

Good stuff about Unicode in Python 3.3 page 3

Internal representation got a lot better with `PEP-393`.

There are lots of details in the PEP about the internal API. What you need to know as a *Python* programmer is that the strings you create will now (usually) be stored in the best choice among `Latin-1`, `UTF-16`, and `UTF-32`.

While micro-benchmarks may do worse with the change, large applications with lots of (usually ASCII) strings in them will probably half their memory usage.

Good stuff about Unicode in Python 3.3 page 4

Python has kept pace with changes in Unicode itself. Thousand of code points have been added, for example. For most developers it won't matter, but better to avoid a strange bug on unusual inputs.

Slightly controversially, 3.3 adds back the explicit unicode literals entirely to aid in porting 2.x software to Python 3.3. Now you are welcome to write `u'Foobar'` everywhere you had written just `'Foobar'` in Python 3 code; more relevantly, your Python 2.x code doesn't require changing the `u'Foobar'` to port.

A nice string method to notice

The following has been in place for years, and many experienced developers (including me) took years to notice it: The string methods `.startswith()` and `.endswith()` will accept a tuple of strings as well as a single string! (but *not* a list or other iterable as an argument)

```
>>> for word in "Mary had a little lamb".split():
...     if word.startswith(('h','l')):
...         print(word, end=';')
...
had;little;lamb;
```

(I have myself written `for prefix in prefixes: …` countless times!)

Module `textwrap`

This has been around for many versions (small features have been added); when I started writing about and in Python, I did *ad hoc* versions of text wrapping many times. Unfortunately, I still see code with similar one-offs in projects, at times.

```
>>> print(textwrap.fill(s, width=35,
      initial_indent="| ", subsequent_indent="| "))
| Lorem ipsum dolor sit amet,
| consectetur adipisicing elit, sed
| do eiusmod tempor incididunt ut
| labore et dolore magna aliqua.
```

Module textwrap

Here is something that I often forget to do; instead I use some awkward workaround. Don't be like me.

```
def myfunc():
    multi_line = """
        Lorem ipsum dolor sit amet,
        consectetur adipisicing elit, sed
        do eiusmod tempor incididunt u
        labore et dolore magna aliqua.
    """
    multi_line = textwrap.dedent(multi_line)
    do_something_with(multi_line)
```

(In docstrings you usually want to preserve line breaks, not overall indentation)

Module `textwrap`

There are a few things new in `textwrap` in 3.x.
`.indent()` is a nice shortcut with some extra power.
The option `tabsize` is new to 3.3.

```
>>> print(textwrap.indent(message, '| ',
                          predicate=lambda l:
                          not l.endswith('wrote:\n')))
```

```
David Mertz, Ph.D. <mertz@gnosis.cx> wrote:
| Lorem ipsum dolor sit amet,
| consectetur adipisicing elit, sed
| do eiusmod tempor incididunt ut
| labore et dolore magna aliqua.
```

Module `html.entities`

Working with HTML is a common task, and one minor thorn is entity definitions. It is easier to deal with now, including the nice `html5` dict in Python 3.3.

```
>>> from html.entities import \
                    entitydefs, html5, codepoint2name
>>> html5['Exists;'], html5['NegativeThinSpace;']
('∃', '\u200b')
>>> entitydefs['Theta'], entitydefs['copy']
('Θ', '©')
>>> codepoint2name[ord('χ')]
'chi'
```

Module unicodedata

I have touched on Unicode already, but in the spirit of `html.entities`, it is worth noticing `unicodedata` also.

```
>>> unicodedata.unidata_version
'6.1.0'
>>> unicodedata.name('ク')
'CJK UNIFIED IDEOGRAPH-2008A'
>>> unicodedata.east_asian_width('ク')
'W'
>>> unicodedata.category('ク')
'Lo'
>>> unicodedata.lookup('GREEK SMALL LETTER CHI')
'χ'
```

Desperately overdue

For many years, a handy function called `quote()` has hidden in `pipes`. While not an ideal location, more importantly it was *undocumented*! In 3.3 it is officially supported in `shlex`. Here's why you need it:

```
>>> from subprocess import call
>>> filename = input("File to display? ")
File to display? non_existent; rm -rf / #
# Uh-oh. This will end badly...
>>> call("cat " + filename, shell=True)
```

I find myself also often writing Python scripts to generate bash scripts; I won't be malicious, but I can certainly overlook escaping requirements.

The format specification mini-language page 1

The `format()` function and `.format()` method of strings are enormously powerful, and enormously confusing. However, they can do more than old-style string-interpolation. Let's try it old-style.

```
>>> costs = (1234.5678, 9900000.1, 83, .02)
>>> for n, item in enumerate(costs):
...     print("Purchase %d:\t$%.2f" % (n+1, item))

Purchase 1:   $1234.57
Purchase 2:   $9900000.10
Purchase 3:   $83.00
Purchase 4:   $0.02
```

The format specification mini-language

We can do better than the last slide using a format specifier. In particular, two things we want in formatted currencies is comma separators in large numbers and right alignment.

```
>>> line = "Purchase {}:\t${:>13,.2f}"
>>> for n, item in enumerate(costs):
...     print(line.format(n+1, item))

Purchase 1:   $     1,234.57
Purchase 2:   $ 9,900,000.10
Purchase 3:   $         83.00
Purchase 4:   $          0.02
```

The format specification mini-language page 3

We compactly described the currency format above. However, I would rather have the dollar sign close to its amount. I think this need be done in two stages.

```
>>> line = "Purchase {}:\t{:>14}"
>>> for n, item in enumerate(costs):
...     amount = "${:,.2f}".format(item)
...     print(line.format(n+1, amount))

Purchase 1:          $1,234.57
Purchase 2:     $9,900,000.10
Purchase 3:            $83.00
Purchase 4:             $0.02
```

Module `email`

The `email` module does a lot. In Python 3.2+ (`email` 5.1), the handling of bytes and unicode is both correct and powerful. Credit for this goes to R. David Murray, partially funded by the PSF.

> *The problem was that emails are typically read and stored in the form of bytes rather than str text, and they may contain multiple encodings within a single email. So, the email package had to be extended to parse and generate email messages in bytes format.*
>
> –"What's new in Python 3.2"

Module `email`

Just for fun, let's play with `email` slightly.

```
>>> import email
>>> msg = email.message_from_file(open('email'))
>>> msg['Subject'], msg.get_content_maintype(), \
     msg.is_multipart(), msg.get_content_subtype()
('Course progress', 'multipart', True, 'signed')
>>> msg.get_payload()
[<email.message.Message object at 0x1010f8810>,
 <email.message.Message object at 0x1010f8990>]
>>> msg.get_payload()[1].get_content_type()
'application/pgp-signature'
```

Module `email`

A bit more exploration of the parts in the email.

```
>>> print(msg.get_payload()[1])
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=signature.asc
Content-Type: application/pgp-signature; name=signature.asc
Content-Description: Message signed with OpenPGP [...]

-----BEGIN PGP SIGNATURE-----
Version: GnuPG/MacGPG2 v2.0.18 (Darwin)

iEYEARECAAYFAlE+dNQACgkQvMseJzFyYtJgzgCcDq4M2dStUQJdZ[...]
-----END PGP SIGNATURE-----
```

Module `email`

Let's examine the main body of the email, before the signature just to wrap up these examples.

```
>>> b = msg.get_payload()[0]
>>> b.keys()
['Content-Disposition', 'Content-Type',
    'Content-Transfer-Encoding']
>>> b.get_content_type(), b['Content-Disposition']
('image/tiff',
 'inline;\n\tfilename=Course-Progress.tiff')
>>> body.get_payload()[:45]
'TU0AKgAAliiAP+BP8AQWDQeEQmFQuGQ2HQ+IRGJROKRWL'
```

Module datetime

Mostly `datetime` has been stable for a long time. But Python 3.x versions have added a few nice touches.

```
>>> from datetime import datetime, timedelta
>>> mytalk = datetime(2013, 3, 16, 12, 30)
>>> str(mytalk), mytalk.timestamp()
('2013-03-16 12:30:00', 1363462200.0)
>>> now = datetime.now()
>>> until_talk = mytalk - now
>>> until_talk.total_seconds()
226188.793891
>>> until_talk
datetime.timedelta(2, 53388, 793891)
```

A few other modules

The module `csv` isn't new, but you should keep it in mind rather than decide that your data files are "so simple you don't need to use the module."

The module `hashlib` is only new if you are as old as I am, but here is something cool in Python 3.2+.

```
>>> hashlib.algorithms_guaranteed
{'sha1','md5','sha384','sha512','sha224','sha256'}
>>> hashlib.algorithms_available
{'SHA1','DSA-SHA','sha','DSA','mdc2','dsaWithSHA','SHA',
 'ecdsa-with-SHA1','SHA384','sha512','SHA512','SHA256',
 'ripemd160','sha1','dsaEncryption','MDC2','md5','md4',
 'RIPEMD160','sha384','SHA224','sha224','MD5','sha256'}
```

Wrap-up / Questions?

There are many things I didn't get to talking about that are somewhat new and/or updated, and somewhat *text processing*-ish. For example, `json`, `xml.etree.ElementTree`, or `decimal`.

Anything special the audience thinks I missed, or hints of your own to share?

Other questions?

Email me at: `<mertz@gnosis.cx>`