uniseg-python Documentation

Release 0.7.1

Masaaki Shibata

Apr 15, 2017

Contents

CHAPTER 1

Modules

uniseg.codepoint — Unicode code point

Unicode code point

uniseg.codepoint.**ord**(*c*, *index=None*)

Return the integer value of the Unicode code point *c*

NOTE: Some Unicode code points may be expressed with a couple of other code points ("surrogate pair"). This function treats surrogate pairs as representations of original code points; e.g. `ord(u'\ud842\udf9f')` returns `134047` (`0x20b9f`). `u'\ud842\udf9f'` is a surrogate pair expression which means `u'\U00020b9f'`.

```
>>> ord('a')
97
>>> ord('\u3042')
12354
>>> ord('\U00020b9f')
134047
>>> ord('abc')
Traceback (most recent call last):
  ...
TypeError: need a single Unicode code point as parameter
```

It returns the result of built-in ord() when *c* is a single str object for compatibility:

```
>>> ord('a')
97
```

When *index* argument is specified (to not `None`), this function treats *c* as a Unicode string and returns integer value of code point at `c[index]` (or may be `c[index:index+2]`):

```
>>> ord('hello', 0)
104
>>> ord('hello', 1)
```

```
101
>>> ord('a\U00020b9f', 1)
134047
```

uniseg.codepoint.**unichr**(*cp*)

Return the unicode object represents the code point integer *cp*

```
>>> unichr(0x61) == 'a'
True
```

Notice that some Unicode code points may be expressed with a couple of other code points ("surrogate pair") in narrow-build Python. In those cases, this function will return a unicode object of which length is more than one; e.g. `unichr(0x20b9f)` returns `u'\U00020b9f'` while built-in `unichr()` may raise ValueError.

```
>>> unichr(0x20b9f) == '\U00020b9f'
True
```

uniseg.codepoint.**code_points**(*s*)

Iterate every Unicode code points of the unicode string *s*

```
>>> s = 'hello'
>>> list(code_points(s)) == ['h', 'e', 'l', 'l', 'o']
True
```

The number of iteration may differ from the `len(s)`, because some code points may be represented as a couple of other code points ("surrogate pair") in narrow-build Python.

```
>>> s = 'abc\U00020b9f\u3042'
>>> list(code_points(s)) == ['a', 'b', 'c', '\U00020b9f', '\u3042']
True
```

uniseg.graphemecluster — Grapheme cluster

Unicode grapheme cluster breaking

UAX #29: Unicode Text Segmentation (Unicode 6.2.0) http://www.unicode.org/reports/tr29/tr29-21.html

uniseg.graphemecluster.**grapheme_cluster_break**(*c*, *index=0*)

Return the Grapheme_Cluster_Break property of *c*

c must be a single Unicode code point string.

```
>>> print(grapheme_cluster_break('\x0d'))
CR
>>> print(grapheme_cluster_break('\x0a'))
LF
>>> print(grapheme_cluster_break('a'))
Other
```

If *index* is specified, this function consider *c* as a unicode string and return Grapheme_Cluster_Break property of the code point at c[index].

```
>>> print(grapheme_cluster_break(u'a\x0d', 1))
CR
```

uniseg.graphemecluster.**grapheme_cluster_breakables**(*s*)
Iterate grapheme cluster breaking opportunities for every position of *s*

1 for "break" and 0 for "do not break". The length of iteration will be the same as `len(s)`.

```
>>> list(grapheme_cluster_breakables(u'ABC'))
[1, 1, 1]
>>> list(grapheme_cluster_breakables(u'g'))
[1, 0]
>>> list(grapheme_cluster_breakables(u''))
[]
```

uniseg.graphemecluster.**grapheme_cluster_boundaries**(*s*, *tailor=None*)
Iterate indices of the grapheme cluster boundaries of *s*

This function yields from 0 to the end of the string (== len(s)).

```
>>> list(grapheme_cluster_boundaries('ABC'))
[0, 1, 2, 3]
>>> list(grapheme_cluster_boundaries('g'))
[0, 2]
>>> list(grapheme_cluster_boundaries(''))
[]
```

uniseg.graphemecluster.**grapheme_clusters**(*s*, *tailor=None*)
Iterate every grapheme cluster token of *s*

Grapheme clusters (both legacy and extended):

```
>>> list(grapheme_clusters('g')) == ['g']
True
>>> list(grapheme_clusters('')) == ['']
True
>>> list(grapheme_clusters('')) == ['']
True
```

Extended grapheme clusters:

```
>>> list(grapheme_clusters('')) == ['']
True
>>> list(grapheme_clusters('')) == ['']
True
```

Empty string leads the result of empty sequence:

```
>>> list(grapheme_clusters('')) == []
True
```

You can customize the default breaking behavior by modifying breakable table so as to fit the specific locale in *tailor* function. It receives *s* and its default breaking sequence (iterator) as its arguments and returns the sequence of customized breaking opportunities:

```
>>> def tailor_grapheme_cluster_breakables(s, breakables):
...
...     for i, breakable in enumerate(breakables):
...         # don't break between 'c' and 'h'
...         if s.endswith('c', 0, i) and s.startswith('h', i):
...             yield 0
...         else:
```

```
...             yield breakable
...
>>> s = 'Czech'
>>> list(grapheme_clusters(s)) == ['C', 'z', 'e', 'c', 'h']
True
>>> list(grapheme_clusters(s, tailor_grapheme_cluster_breakables)) == ['C', 'z',
↪'e', 'ch']
True
```

uniseg.wordbreak — Word break

Unicode word breaking

UAX #29: Unicode Text Segmentation (Unicode 6.2.0) http://www.unicode.org/reports/tr29/tr29-21.html

uniseg.wordbreak.**word_break**(*c*, *index=0*)

Return the Word_Break property of *c*

c must be a single Unicode code point string.

```
>>> print(word_break('\x0d'))
CR
>>> print(word_break('\x0b'))
Newline
>>> print(word_break(''))
Katakana
```

If *index* is specified, this function consider *c* as a unicode string and return Word_Break property of the code point at c[index].

```
>>> print(word_break('A', 1))
Katakana
```

uniseg.wordbreak.**word_breakables**(*s*)

Iterate word breaking opportunities for every position of *s*

1 for "break" and 0 for "do not break". The length of iteration will be the same as `len(s)`.

```
>>> list(word_breakables(u'ABC'))
[1, 0, 0]
>>> list(word_breakables(u'Hello, world.'))
[1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1]
>>> list(word_breakables(u'\x01\x01'))
[1, 0, 1]
```

uniseg.wordbreak.**word_boundaries**(*s*, *tailor=None*)

Iterate indices of the word boundaries of *s*

This function yields indices from the first boundary position (> 0) to the end of the string (== len(s)).

uniseg.wordbreak.**words**(*s*, *tailor=None*)

Iterate *user-perceived* words of *s*

These examples bellow is from http://www.unicode.org/reports/tr29/tr29-15.html#Word_Boundaries

```
>>> s = 'The quick ("brown") fox can't jump 32.3 feet, right?'
>>> print('|'.join(words(s)))
```

```
The| |quick| |(|“|brown|”|)| |fox| |can’t| |jump| |32.3| |feet|,| |right|?
>>> list(words(u''))
[]
```

uniseg.sentencebreak — Sentence break

sentence boundaries

UAX #29: Unicode Text Segmentation http://www.unicode.org/reports/tr29/tr29-15.html

uniseg.sentencebreak.**sentence_break**(*c*, *index=0*)

Return Sentence_Break property value of *c*

c must be a single Unicode code point string.

```
>>> print(sentence_break(u'\x0d'))
CR
>>> print(sentence_break(u' '))
Sp
>>> print(sentence_break(u'a'))
Lower
```

If *index* is specified, this function consider *c* as a unicode string and return Sentence_Break property of the code point at c[index].

```
>>> print(sentence_break(u'a\x0d', 1))
CR
```

uniseg.sentencebreak.**sentence_breakables**(*s*)

Iterate sentence breaking opportunities for every position of *s*

1 for “break” and 0 for “do not break”. The length of iteration will be the same as `len(s)`.

```
>>> s = 'He said, “Are you going?” John shook his head.'
>>> list(sentence_breakables(s))
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

uniseg.sentencebreak.**sentence_boundaries**(*s*, *tailor=None*)

Iterate indices of the sentence boundaries of *s*

This function yields from 0 to the end of the string (== len(s)).

```
>>> list(sentence_boundaries(u'ABC'))
[0, 3]
>>> s = 'He said, “Are you going?” John shook his head.'
>>> list(sentence_boundaries(s))
[0, 26, 46]
>>> list(sentence_boundaries(u''))
[]
```

uniseg.sentencebreak.**sentences**(*s*, *tailor=None*)

Iterate every sentence of *s*

```
>>> s = 'He said, “Are you going?” John shook his head.'
>>> list(sentences(s)) == ['He said, “Are you going?” ', 'John shook his head.']
True
```

uniseg.linebreak — Line break

Unicode line breaking algorithm

UAX #14: Unicode Line Breaking Algorithm http://www.unicode.org/reports/tr14/tr14-24.html

uniseg.linebreak.**line_break**(*c*, *index=0*)
Return the Line_Break property of *c*

c must be a single Unicode code point string.

```
>>> print(line_break('\x0d'))
CR
>>> print(line_break(' '))
SP
>>> print(line_break('1'))
NU
```

If *index* is specified, this function consider *c* as a unicode string and return Line_Break property of the code point at c[index].

```
>>> print(line_break(u'a\x0d', 1))
CR
```

uniseg.linebreak.**line_break_breakables**(*s*, *legacy=False*)
Iterate line breaking opportunities for every position of *s*

1 means "break" and 0 means "do not break" BEFORE the postion. The length of iteration will be the same as `len(s)`.

```
>>> list(line_break_breakables('ABC'))
[0, 0, 0]
>>> list(line_break_breakables('Hello, world.'))
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
>>> list(line_break_breakables(u''))
[]
```

uniseg.linebreak.**line_break_boundaries**(*s*, *legacy=False*, *tailor=None*)
Iterate indices of the line breaking boundaries of *s*

This function yields from 0 to the end of the string (== len(s)).

uniseg.linebreak.**line_break_units**(*s*, *legacy=False*, *tailor=None*)
Iterate every line breaking token of *s*

```
>>> s = 'The quick (“brown”) fox can’t jump 32.3 feet, right?'
>>> '|'.join(line_break_units(s)) == 'The |quick |(“brown”) |fox |can’t |jump |32.
↪3 |feet, |right?'
True
>>> list(line_break_units(u''))
[]
```

```
>>> list(line_break_units('αα')) == [u'αα']
True
>>> list(line_break_units(u'αα', True)) == [u'α', u'α']
True
```

uniseg.wrap — Text wrapping

Unicode-aware text wrapping

class uniseg.wrap.**Wrapper**
Text wrapping engine

Usually, you don't need to create an instance of the class directly. Use *wrap()* instead.

wrap (*formatter*, *s*, *cur=0*, *offset=0*, *char_wrap=None*)
Wrap string *s* with *formatter* and invoke its handlers

The optional arguments, *cur* is the starting position of the string in logical length, and *offset* means left-side offset of the wrapping area in logical length — this parameter is only used for calculating tab-stopping positions for now.

If *char_wrap* is set to `True`, the text will be warpped with its grapheme cluster boundaries instead of its line break boundaries. This may be helpful when you don't want the word wrapping feature in your application.

This function returns the total count of wrapped lines.

- *Changed in version 0.7:* The order of the parameters are changed.
- *Changed in version 0.7.1:* It returns the count of lines now.

uniseg.wrap.**wrap** (*formatter*, *s*, *cur=0*, *offset=0*, *char_wrap=None*)
Wrap string *s* with *formatter* using the module's static *Wrapper* instance

See *Wrapper.wrap()* for further details of the parameters.

- *Changed in version 0.7.1:* It returns the count of lines now.

class uniseg.wrap.**Formatter**
The abstruct base class for formatters invoked by a *Wrapper* object

This class is implemented only for convinience sake and does nothing itself. You don't have to design your own formatter as a subclass of it, while it is not deprecated either.

Your formatters should have the methods and properties this class has. They are invoked by a *Wrapper* object to determin *logical widths* of texts and to give you the ways to handle them, such as to render them.

handle_new_line()
The handler method which is invoked when the current line is over and a new line begins

handle_text (*text*, *extents*)
The handler method which is invoked when *text* should be put on the current position with *extents*

reset()
Reset all states of the formatter

tab_width
The logical width of tab forwarding

This property value is used by a *Wrapper* object to determin the actual forwarding extents of tabs in each of the positions.

text_extents (*s*)
Return a list of logical lengths from start of the string to each of characters in *s*

wrap_width
The logical width of text wrapping

Note that returning `None` (which is the default) means *"do not wrap"* while returning `0` means *"wrap as narrowly as possible."*

class `uniseg.wrap.`**`TTFormatter`**(*wrap_width*, *tab_width=8*, *tab_char=u' '*, *ambiguous_as_wide=False*)

A Fixed-width text wrapping formatter

`ambiguous_as_wide`
Treat code points with its East_Easian_Width property is 'A' as those with 'W'; having double width as alpha-numerics

`handle_new_line`()
The handler which is invoked when the current line is over and a new line begins

`handle_text`(*text*, *extents*)
The handler which is invoked when a text should be put on the current position

`lines`()
Iterate every wrapped line strings

`reset`()
Reset all states of the formatter

`tab_char`
Character to fill tab spaces with

`tab_width`
forwarding size of tabs

`text_extents`(*s*)
Return a list of logical lengths from start of the string to each of characters in *s*

`wrap_width`
Wrapping width

`uniseg.wrap.`**`tt_width`**(*s*, *index=0*, *ambiguous_as_wide=False*)

Return logical width of the grapheme cluster at *s[index]* on fixed-width typography

Return value will be `1` (halfwidth) or `2` (fullwidth).

Generally, the width of a grapheme cluster is determined by its leading code point.

```
>>> tt_width('A')
1
>>> tt_width('\u8240')      # U+8240: CJK UNIFIED IDEOGRAPH-8240
2
>>> tt_width('g\u0308')     # U+0308: COMBINING DIAERESIS
1
>>> tt_width('\U00029e3d') # U+29E3D: CJK UNIFIED IDEOGRAPH-29E3D
2
```

If *ambiguous_as_wide* is specified to `True`, some characters such as greek alphabets are treated as they have fullwidth as well as ideographics does.

```
>>> tt_width('\u03b1')      # U+03B1: GREEK SMALL LETTER ALPHA
1
>>> tt_width('\u03b1', ambiguous_as_wide=True)
2
```

`uniseg.wrap.`**`tt_text_extents`**(*s*, *ambiguous_as_wide=False*)

Return a list of logical widths from the start of *s* to each of characters *(not of code points)* on fixed-width typography

```
>>> tt_text_extents('')
[]
>>> tt_text_extents('abc')
[1, 2, 3]
>>> tt_text_extents('\u3042\u3044\u3046')
[2, 4, 6]
>>> import sys
>>> s = '\U00029e3d'   # test a code point out of BMP
>>> actual = tt_text_extents(s)
>>> expect = [2] if sys.maxunicode > 0xffff else [2, 2]
>>> len(s) == len(expect)
True
>>> actual == expect
True
```

The meaning of *ambiguous_as_wide* is the same as that of `tt_width()`.

uniseg.wrap.**tt_wrap**(*s*, *wrap_width*, *tab_width=8*, *tab_char=u' '*, *ambiguous_as_wide=False*, *cur=0*, *offset=0*, *char_wrap=False*)

Wrap *s* with given parameters and return a list of wrapped lines

See `TTFormatter` for *wrap_width*, *tab_width* and *tab_char*, and `tt_wrap()` for *cur*, *offset* and *char_wrap*.

CHAPTER 2

Sample scripts

unibreak.py

```
usage: unibreak.py [-h] [-e ENCODING] [-l] [-m {c,g,l,s,w}] [-o OUTPUT] [file]

positional arguments:
  file                  input text file

optional arguments:
  -h, --help            show this help message and exit
  -e ENCODING, --encoding ENCODING
                        text encoding of the input (UTF-8)
  -l, --legacy          legacy mode (makes sense only with '--mode l')
  -m {c,g,l,s,w}, --mode {c,g,l,s,w}
                        breaking algorithm (w) (c: code points, g: grapheme
                        clusters, s: sentences l: line breaking units, w:
                        words)
  -o OUTPUT, --output OUTPUT
                        leave output to specified file
```

uniwrap.py

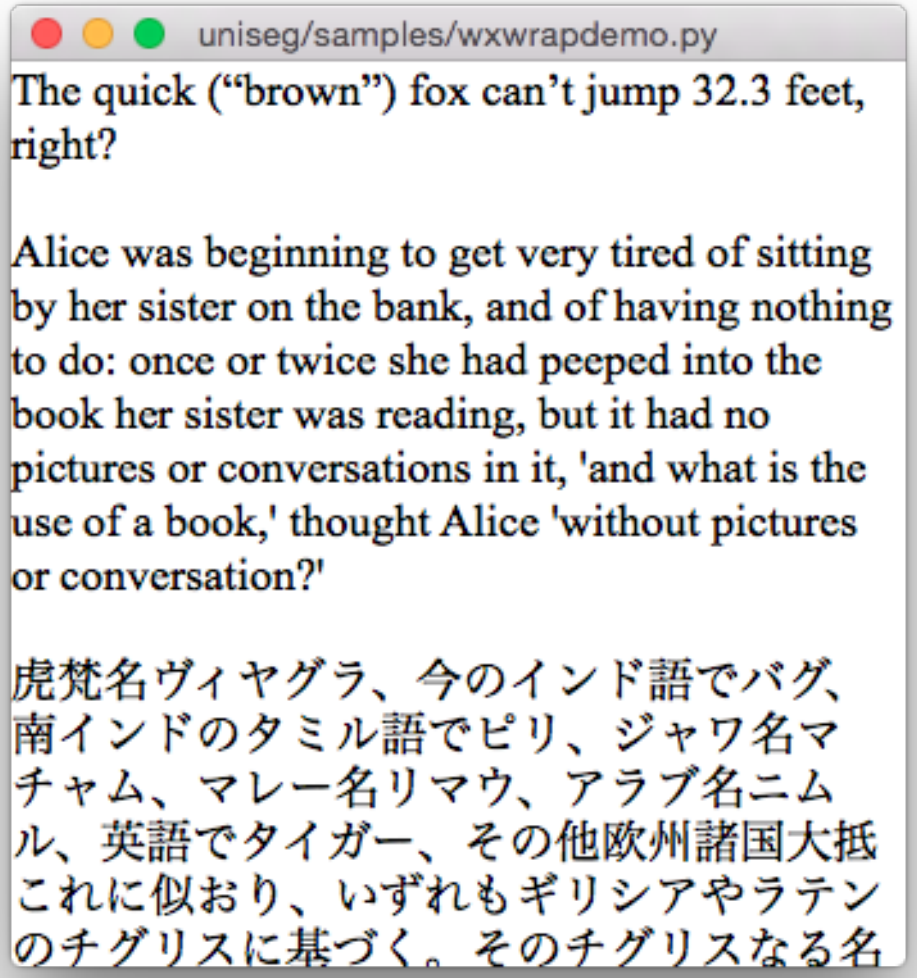
```
usage: uniwrap.py [-h] [-e ENCODING] [-r] [-t TAB_WIDTH] [-l] [-o OUTPUT]
                  [-w WRAP_WIDTH] [-c]
                  [file]

positional arguments:
  file                  input file

optional arguments:
  -h, --help            show this help message and exit
  -e ENCODING, --encoding ENCODING
```

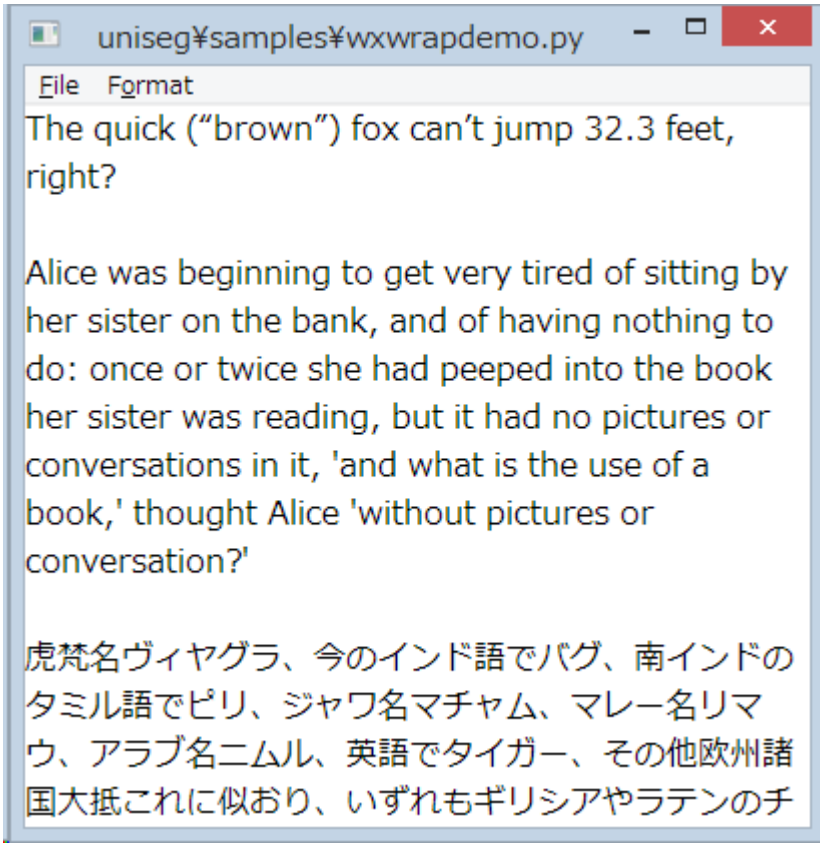
```
                      file encoding (UTF-8)
  -r, --ruler         show ruler
  -t TAB_WIDTH, --tab-width TAB_WIDTH
                      tab width (8)
  -l, --legacy        treat ambiguous-width letters as wide
  -o OUTPUT, --output OUTPUT
                      leave output to specified file
  -w WRAP_WIDTH, --wrap-width WRAP_WIDTH
                      wrap width (60)
  -c, --char-wrap     wrap on grapheme boundaries instead of line break
                      boundaries
```

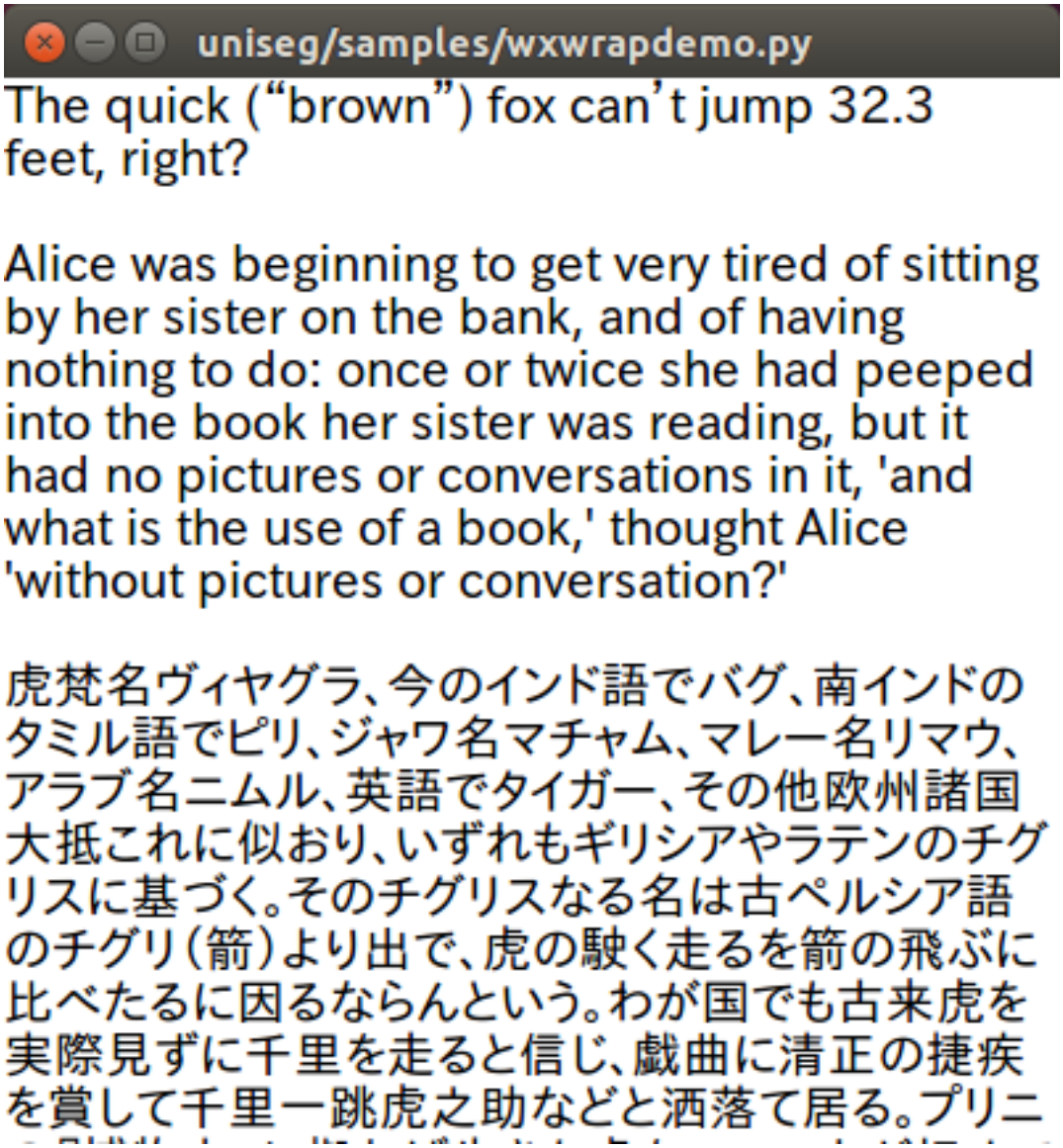
wxwrapdemo.py



On OS X

On Windows


uniseg¥samples¥wxwrapdemo.py
File Format
The quick (“brown”) fox can’t jump 32.3 feet, right?
Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, 'and what is the use of a book,' thought Alice 'without pictures or conversation?'
虎梵名ヴィヤグラ、今のインド語でバグ、南インドのタミル語でピリ、ジャワ名マチャム、マレー名リマウ、アラブ名ニムル、英語でタイガー、その他欧州諸国大抵これに似おり、いずれもギリシアやラテンのチ



On Ubuntu

- genindex
- modindex
- search

CHAPTER 3

Introduction

A Python package to determine Unicode text segmentations.

You can see the full documentation including the package reference on http://uniseg-python.readthedocs.io.

CHAPTER 4

Features

This package provides:

- Functions to get Unicode Character Database (UCD) properties concerned with text segmentations.
- Functions to determine segmentation boundaries of Unicode strings.
- Classes that help implement Unicode-aware text wrapping on both console (monospace) and graphical (monospace / proportional) font environments.

Supporting segmentations are:

code point Code point is *"any value in the Unicode codespace."* It is the basic unit for processing Unicode strings.

grapheme cluster Grapheme cluster approximately represents *"user-perceived character."* They may be made up of single or multiple Unicode code points. e.g. "G" + *acute-accent* is a *user-perceived character*.

word break Word boundaries are familiar segmentation in many common text operations. e.g. Unit for text highlighting, cursor jumping etc. Note that *words* are not determinable only by spaces or punctuations in text in some languages. Such languages like Thai or Japanese require dictionaries to determine appropriate word boundaries. Though the package only provides simple word breaking implementation which is based on the scripts and doesn't use any dictionaries, it also provides ways to customize its default behavior.

sentence break Sentence breaks are also common in text processing but they are more contextual and less formal. The sentence breaking implementation (which is specified in UAX: Unicode Standard Annex) in the package is simple and formal too. But it must be still useful in some usages.

line break Implementing line breaking algorithm is one of the key features of this package. The feature is important in many general text presentations in both CLI and GUI applications.

CHAPTER 5

Requirements

- Python 2.7 / 3.4 / 3.5 / 3.6

CHAPTER 6

Download

Source / binary distributions (PyPI) https://pypi.python.org/pypi/uniseg

All sources and build tools etc. (Bitbucket) https://bitbucket.org/emptypage/uniseg-python

CHAPTER 7

Install

Just type:

```
% pip install uniseg
```

or download the archive and:

```
% python setup.py install
```

CHAPTER 8

Changes

0.7.1 (2015-05-02)

- CHANGE: wrap.Wrapper.wrap(): returns the count of lines now.
- Separate LICENSE from README.txt for the packaging-related reason in some environments.

0.7.0 (2015-02-27)

- CHANGE: Quitted gathering all submodules's members on the top, uniseg module.
- CHANGE: Reform `uniseg.wrap` module and sample scripts.
- Maintained uniseg.wrap module, and sample scripts work again.

0.6.4 (2015-02-10)

- Add `uniseg-dbpath` console command, which just print the path of `ucd.sqlite3`.
- Include sample scripts under the package's subdirectory.

0.6.3 (2015-01-25)

- Python 3.4
- Support modern setuptools, pip and wheel.

0.6.2 (2013-06-09)

- Python 3.3

0.6.1 (2013-06-08)

- Unicode 6.2.0

CHAPTER 9

References

***UAX #14: Unicode Line Breaking Algorithm* (6.2.0)** http://www.unicode.org/reports/tr14/tr14-30.html

***UAX #29 Unicode Text Segmentation* (6.2.0)** http://www.unicode.org/reports/tr29/tr29-21.html

CHAPTER 10

Related / Similar Projects

PyICU - Python extension wrapping the ICU C++ API *PyICU* is a Python extension wrapping International Components for Unicode library (ICU). It also provides text segmentation supports and they just perform richer and faster than those of ours. PyICU is an extension library so it requires ICU dynamic library (binary files) and compiler to build the extension. Our package is written in pure Python; it runs slower but is more portable.

pytextseg - Python module for text segmentation *pytextseg* package focuses very similar goal to ours; it provides Unicode-aware text wrapping features. They designed and uses their original string class (not built-in `unicode` / `str` classes) for the purpose. We use strings as just ordinary built-in `unicode` / `str` objects for text processing in our modules.

CHAPTER 11

License

```
The MIT License (MIT)

Copyright (c) 2013 Masaaki Shibata

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

(This is the output of the sample script, `uniwrap.py` with the option `-w 76`.)

Python Module Index

u

Index