shlex — Parse Shell-style Syntaxes

Purpose: Lexical analysis of shell-style syntaxes.

The shlex module implements a class for parsing simple shell-like syntaxes. It can be used for writing a domain-specific language, or for parsing quoted strings (a task that is more complex than it seems on the surface).

Parsing Quoted Strings

A common problem when working with input text is to identify a sequence of quoted words as a single entity. Splitting the text on quotes does not always work as expected, especially if there are nested levels of quotes. Take the following text as an example.

```
This string has embedded "double quotes" and
'single quotes' in it, and even "a 'nested example'".
```

A naive approach would be to construct a regular expression to find the parts of the text outside the quotes to separate them from the text inside the quotes, or vice versa. That would be unnecessarily complex and prone to errors resulting from edge cases like apostrophes or even typos. A better solution is to use a true parser, such as the one provided by the shlex module. Here is a simple example that prints the tokens identified in the input file using the shlex class.

```
# shlex_example.py

import shlex
import sys

if len(sys.argv) != 2:
    print('Please specify one filename on the command line.')
    sys.exit(1)

filename = sys.argv[1]
with open(filename, 'r') as f:
    body = f.read()
print('ORIGINAL: {!r}'.format(body))
print()

print('TOKENS:')
lexer = shlex.shlex(body)
for token in lexer:
    print('{!r}'.format(token))
```

When run on data with embedded quotes, the parser produces the list of expected tokens.

```
$ python3 shlex_example.py quotes.txt

ORIGINAL: 'This string has embedded "double quotes" and\n\'singl
e quotes\' in it, and even "a \'nested example\'".\n'

TOKENS:
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
','
'and'
'even'
'"a \'nested example\'"'
'.'
```

Isolated quotes such as apostrophes are also handled. Consider this input file.

```
This string has an embedded apostrophe, doesn't it?
```

The token with the embedded apostrophe is no problem.

```
$ python3 shlex_example.py apostrophe.txt

ORIGINAL: "This string has an embedded apostrophe, doesn't it?"

TOKENS:
'This'
'string'
'has'
'an'
'embedded'
'apostrophe'
','
"doesn't"
'it'
'?'
```

Making Safe Strings for Shells

The `quote()` function performs the inverse operation, escaping existing quotes and adding missing quotes for strings to make them safe to use in shell commands.

```
# shlex_quote.py

import shlex

examples = [
    "Embedded'SingleQuote",
    'Embedded"DoubleQuote',
    'Embedded Space',
    '~SpecialCharacter',
    r'Back\slash',
]

for s in examples:
    print('ORIGINAL : {}'.format(s))
    print('QUOTED   : {}'.format(shlex.quote(s)))
    print()
```

It is still usually safer to use a list of arguments when using `subprocess.Popen`, but in situations where that is not possible `quote()` provides some protection by ensuring that special characters and white space are quoted properly.

```
$ python3 shlex_quote.py

ORIGINAL : Embedded'SingleQuote
QUOTED   : 'Embedded'"'"'SingleQuote'

ORIGINAL : Embedded"DoubleQuote
QUOTED   : 'Embedded"DoubleQuote'

ORIGINAL : Embedded Space
QUOTED   : 'Embedded Space'

ORIGINAL : ~SpecialCharacter
QUOTED   : '~SpecialCharacter'

ORIGINAL : Back\slash
QUOTED   : 'Back\slash'
```

Embedded Comments

Since the parser is intended to be used with command languages, it needs to handle comments. By default, any text following a # is considered part of a comment and ignored. Due to the nature of the parser, only single-character comment prefixes are supported. The set of comment characters used can be configured through the `commenters` property.

```
$ python3 shlex_example.py comments.txt

ORIGINAL: 'This line is recognized.\n# But this line is ignored.
\nAnd this line is processed.'

TOKENS:
'This'
'line'
'is'
'recognized'
'.'
'And'
'this'
'line'
'is'
'processed'
'.'
```

Splitting Strings into Tokens

To split an existing string into component tokens, the convenience function `split()` is a simple wrapper around the parser.

```
# shlex_split.py
```

```
import shlex

text = """This text has "quoted parts" inside it."""
print('ORIGINAL: {!r}'.format(text))
print()

print('TOKENS:')
print(shlex.split(text))
```

The result is a list.

```
$ python3 shlex_split.py

ORIGINAL: 'This text has "quoted parts" inside it.'

TOKENS:
['This', 'text', 'has', 'quoted parts', 'inside', 'it.']
```

Including Other Sources of Tokens

The `shlex` class includes several configuration properties that control its behavior. The `source` property enables a feature for code (or configuration) re-use by allowing one token stream to include another. This is similar to the Bourne shell `source` operator, hence the name.

```
# shlex_source.py
```

```
import shlex

text = "This text says to source quotes.txt before continuing."
print('ORIGINAL: {!r}'.format(text))
print()

lexer = shlex.shlex(text)
lexer.wordchars += '.'
lexer.source = 'source'

print('TOKENS:')
for token in lexer:
    print('{!r}'.format(token))
```

The string “source quotes.txt” in the original text receives special handling. Since the source property of the lexer is set to "source", when the keyword is encountered, the filename appearing on the next line is automatically included. In order to cause the filename to appear as a single token, the . character needs to be added to the list of characters that are included in words (otherwise “quotes.txt” becomes three tokens, “quotes”, “.”, “txt”). This what the output looks like.

```
$ python3 shlex_source.py

ORIGINAL: 'This text says to source quotes.txt before
continuing.'

TOKENS:
'This'
'text'
'says'
'to'
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
','
'and'
'even'
'"a \'nested example\'"'
'.'
'before'
'continuing.'
```

The source feature uses a method called sourcehook() to load the additional input source, so a subclass of shlex can provide an alternate implementation that loads data from locations other than files.

Controlling the Parser

An earlier example demonstrated changing the wordchars value to control which characters are included in words. It is also possible to set the quotes character to use additional or alternative quotes. Each quote must be a single character, so it is not possible to have different open and close quotes (no parsing on parentheses, for example).

```
# shlex_table.py

import shlex

text = """|Col 1||Col 2||Col 3|"""
print('ORIGINAL: {!r}'.format(text))
print()

lexer = shlex.shlex(text)
lexer.quotes = '|'

print('TOKENS:')
for token in lexer:
    print('{!r}'.format(token))
```

In this example, each table cell is wrapped in vertical bars.

```
$ python3 shlex_table.py

ORIGINAL: '|Col 1||Col 2||Col 3|'

TOKENS:
'|Col 1|'
'|Col 2|'
'|Col 3|'
```

It is also possible to control the whitespace characters used to split words.

```
# shlex_whitespace.py

import shlex
import sys

if len(sys.argv) != 2:
    print('Please specify one filename on the command line.')
    sys.exit(1)

filename = sys.argv[1]
with open(filename, 'r') as f:
    body = f.read()
print('ORIGINAL: {!r}'.format(body))
print()

print('TOKENS:')
lexer = shlex.shlex(body)
lexer.whitespace += '.,'
for token in lexer:
    print('{!r}'.format(token))
```

If the example in `shlex_example.py` is modified to include period and comma, the results change.

```
$ python3 shlex_whitespace.py quotes.txt

ORIGINAL: 'This string has embedded "double quotes" and\n\'singl
e quotes\' in it, and even "a \'nested example\'".\n'

TOKENS:
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
'and'
'even'
'"a \'nested example\'"'
```

Error Handling

When the parser encounters the end of its input before all quoted strings are closed, it raises `ValueError`. When that happens, it is useful to examine some of the properties maintained by the parser as it processes the input. For example, `infile` refers to the name of the file being processed (which might be different from the original file, if one file sources another). The `lineno` reports the line when the error is discovered. The `lineno` is typically the end of the file, which may be far away from the first quote. The `token` attribute contains the buffer of text not already included in a valid token. The `error_leader()` method produces a message prefix in a style similar to Unix compilers, which enables editors such as `emacs` to parse the error and take the user directly to the invalid line.

```
# shlex_errors.py

import shlex

text = """This line is ok.
This line has an "unfinished quote.
This line is ok, too.
"""

print('ORIGINAL: {!r}'.format(text))
print()

lexer = shlex.shlex(text)

print('TOKENS:')
try:
    for token in lexer:
```

```
        print('{!r}'.format(token))
except ValueError as err:
    first_line_of_error = lexer.token.splitlines()[0]
    print('ERROR: {} {}'.format(lexer.error_leader(), err))
    print('following {!r}'.format(first_line_of_error))
```

The example produces this output.

```
$ python3 shlex_errors.py

ORIGINAL: 'This line is ok.\nThis line has an "unfinished quote.
\nThis line is ok, too.\n'

TOKENS:
'This'
'line'
'is'
'ok'
'.'
'This'
'line'
'has'
'an'
ERROR: "None", line 4:  No closing quotation
following '"unfinished quote.'
```

POSIX vs. Non-POSIX Parsing

The default behavior for the parser is to use a backwards-compatible style that is not POSIX-compliant. For POSIX behavior, set the `posix` argument when constructing the parser.

```
# shlex_posix.py

import shlex

examples = [
    'Do"Not"Separate',
    '"Do"Separate',
    'Escaped \e Character not in quotes',
    'Escaped "\e" Character in double quotes',
    "Escaped '\e' Character in single quotes",
    r"Escaped '\'' \"\'\" single quote",
    r'Escaped "\"" \'\"\' double quote',
    "\"'Strip extra layer of quotes'\"",
]

for s in examples:
    print('ORIGINAL : {!r}'.format(s))
    print('non-POSIX: ', end='')

    non_posix_lexer = shlex.shlex(s, posix=False)
    try:
        print('{!r}'.format(list(non_posix_lexer)))
    except ValueError as err:
        print('error({})'.format(err))

    print('POSIX    : ', end='')
    posix_lexer = shlex.shlex(s, posix=True)
    try:
        print('{!r}'.format(list(posix_lexer)))
    except ValueError as err:
        print('error({})'.format(err))

    print()
```

Here are a few examples of the differences in parsing behavior.

```
$ python3 shlex_posix.py
```

```
ORIGINAL : 'Do"Not"Separate'
non-POSIX: ['Do"Not"Separate']
POSIX    : ['DoNotSeparate']

ORIGINAL : '"Do"Separate'
non-POSIX: ['"Do"', 'Separate']
POSIX    : ['DoSeparate']

ORIGINAL : 'Escaped \\e Character not in quotes'
non-POSIX: ['Escaped', '\\', 'e', 'Character', 'not', 'in',
'quotes']
POSIX    : ['Escaped', 'e', 'Character', 'not', 'in', 'quotes']

ORIGINAL : 'Escaped "\\e" Character in double quotes'
non-POSIX: ['Escaped', '"\\e"', 'Character', 'in', 'double',
'quotes']
POSIX    : ['Escaped', '\\e', 'Character', 'in', 'double',
'quotes']

ORIGINAL : "Escaped '\\e' Character in single quotes"
non-POSIX: ['Escaped', "'\\e'", 'Character', 'in', 'single',
'quotes']
POSIX    : ['Escaped', '\\e', 'Character', 'in', 'single',
'quotes']

ORIGINAL : 'Escaped \'\\\'\' \\"\\\'\\" single quote'
non-POSIX: error(No closing quotation)
POSIX    : ['Escaped', '\\ \\"\\"', 'single', 'quote']

ORIGINAL : 'Escaped "\\"" \\\'\\"\\\' double quote'
non-POSIX: error(No closing quotation)
POSIX    : ['Escaped', '"', '\'"\'', 'double', 'quote']

ORIGINAL : '"\'Strip extra layer of quotes\'"'
non-POSIX: ['"\'Strip extra layer of quotes\'"']
POSIX    : ["'Strip extra layer of quotes'"]
```

See also

- Standard library documentation for shlex
- `cmd` – Tools for building interactive command interpreters.
- `argparse` – Command line option parsing.
- `subprocess` – Run commands after parsing the command line.

cmd — Line-oriented Command Processors

configparser — Work with Configuration Files

This page was last updated 2016-12-18.

Navigation
cmd — Line-oriented Command Processors
configparser — Work with Configuration Files

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?