24.3. `shlex` — Simple lexical analysis

Source code: Lib/shlex.py

The `shlex` class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The `shlex` module defines the following functions:

`shlex.`**`split`**(*s*, *comments=False*, *posix=True*)

Split the string *s* using shell-like syntax. If *comments* is `False` (the default), the parsing of comments in the given string will be disabled (setting the `commenters` attribute of the `shlex` instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

> **Note:** Since the `split()` function instantiates a `shlex` instance, passing `None` for *s* will read the string to split from standard input.

`shlex.`**`quote`**(*s*)

Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` lets you plug the security hole:

```
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with `split()`:

```
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

New in version 3.3.

The `shlex` module defines the following class:

class `shlex.`**`shlex`**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument defines the operational mode: when *posix* is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the `wordchars` attribute that appear in *punctuation_chars* will be removed from `wordchars`. See Improved Compatibility with Shells for more information.

Changed in version 3.6: The *punctuation_chars* parameter was added.

See also:

Module `configparser`

Parser for configuration files similar to the Windows `.ini` files.

24.3.1. shlex Objects

A `shlex` instance has the following methods:

`shlex.`**`get_token`**()

Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, `eof` is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

`shlex.`**`push_token`**(*str*)

Push the argument onto the token stack.

`shlex.`**`read_token`**()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

`shlex.`**`sourcehook`**(*filename*)

When `shlex` detects a source request (see `source` below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the `close()` method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the `push_source()` and `pop_source()` methods.

`shlex.`**`push_source`**(*newstream*, *newfile=None*)

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the `sourcehook()` method.

`shlex.`**`pop_source`**()

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.**error_leader**(*infile=None*, *lineno=None*)

This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage `shlex` users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of `shlex` subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.**commenters**

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.**wordchars**

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If `punctuation_chars` is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there.

shlex.**whitespace**

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.**escape**

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.**quotes**

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.**escapedquotes**

Characters in `quotes` that will interpret escape characters defined in `escape`. This is only used in POSIX mode, and includes just `'"'` by default.

shlex.**whitespace_split**

If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with `shlex`, getting tokens in a similar way to shell arguments. If this attribute is `True`, `punctuation_chars` will have no effect, and splitting will happen only on whitespaces. When using `punctuation_chars`, which is intended to provide parsing closer to that implemented by shells, it is advisable to leave `whitespace_split` as `False` (the default value).

shlex.**infile**

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.**instream**

The input stream from which this `shlex` instance is reading characters.

shlex.**source**

This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the `close()` method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.**debug**

If this attribute is numeric and `1` or more, a `shlex` instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.**lineno**

Source line number (count of newlines seen so far plus one).

shlex.**token**

The token buffer. It may be useful to examine this when catching exceptions.

shlex.**eof**

Token used to determine end of file. This will be set to the empty string (`''`), in non-POSIX mode, and to `None` in POSIX mode.

shlex.**punctuation_chars**

Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, '>>>' could be returned as a token, even though it may not be recognised as such by shells.

New in version 3.6.

24.3.2. Parsing Rules

When operating in non-POSIX mode, `shlex` will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `"Do"` and `Separate`);
- If `whitespace_split` is `False`, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is `True`, `shlex` will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, `shlex` will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of `escapedquotes` (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of `escapedquotes` (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in `escape`. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a `None` value;
- Quoted empty strings (`''`) are allowed.

24.3.3. Improved Compatibility with Shells

New in version 3.6.

The shlex class provides compatibility with the parsing performed by common Unix shells like bash, dash, and sh. To take advantage of this compatibility, specify the punctuation_chars argument in the constructor. This defaults to False, which preserves pre-3.6 behaviour. However, if it is set to True, then parsing of the characters ();<>|& is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> list(shlex.shlex(text))
['a', '&', '&', 'b', ';', 'c', '&', '&', 'd', '|', '|', 'e', ';', 'f'
"'abc'", ';', '(', 'def', '"ghi"', ')']
>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', "'abc
';', '(', 'def', '"ghi"', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing True as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Note: When punctuation_chars is specified, the wordchars attribute is augmented with the characters ~-./*?=. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. --color=auto). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

For best effect, punctuation_chars should be set in conjunction with posix=True. (Note that posix=False is the default for shlex.)