32.7. `tokenize` — Tokenizer for Python source

Source code: Lib/tokenize.py

The `tokenize` module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers," including colorizers for on-screen displays.

To simplify token stream handling, all operator and delimiter tokens and `Ellipsis` are returned using the generic `OP` token type. The exact type can be determined by checking the `exact_type` property on the named tuple returned from `tokenize.tokenize()`.

32.7.1. Tokenizing Input

The primary entry point is a generator:

`tokenize.`**`tokenize`**(*readline*)

The `tokenize()` generator requires one argument, *readline*, which must be a callable object which provides the same interface as the `io.IOBase.readline()` method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *logical* line; continuation lines are included. The 5 tuple is returned as a named tuple with the field names: `type string start end line`.

The returned named tuple has an additional property named `exact_type` that contains the exact operator type for `token.OP` tokens. For all other token types `exact_type` equals the named tuple `type` field.

Changed in version 3.1: Added support for named tuples.

Changed in version 3.3: Added support for `exact_type`.

`tokenize()` determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263**.

All constants from the `token` module are also exported from `tokenize`, as are three additional token type values:

`tokenize.`**COMMENT**
: Token value used to indicate a comment.

`tokenize.`**NL**
: Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical lines.

`tokenize.`**ENCODING**
: Token value that indicates the encoding used to decode the source bytes into text. The first token returned by `tokenize()` will always be an ENCODING token.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

`tokenize.`**untokenize**(*iterable*)
: Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

  The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

  It returns bytes, encoded using the ENCODING token, which is the first token sequence output by `tokenize()`.

`tokenize()` needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

`tokenize.`**detect_encoding**(*readline*)
: The `detect_encoding()` function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the `tokenize()` generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in **PEP 263**. If both a BOM and a cookie are present, but disagree, a SyntaxError will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

If no encoding is specified, then the default of `'utf-8'` will be returned.

Use `open()` to open Python source files: it uses `detect_encoding()` to detect the file encoding.

`tokenize.`**open**(*filename*)

Open a file in read only mode using the encoding detected by `detect_encoding()`.

New in version 3.2.

exception `tokenize.`**TokenError**

Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

or:

```
[1,
 2,
 3
```

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as `ERRORTOKEN`, followed by the tokenization of their contents.

32.7.2. Command-Line Usage

New in version 3.3.

The `tokenize` module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

The following options are accepted:

-h , --help
show this help message and exit

-e , --exact
display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

32.7.3. Examples

Example of a script rewriter that transforms float literals into Decimal objects:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C librar
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across d
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER toker
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
```

```
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

The exact token type names can be displayed using the `-e` option:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
```

```
2,25-2,26:            RPAR           ')'
2,26-2,27:            NEWLINE        '\n'
3,0-3,1:              NL             '\n'
4,0-4,0:              DEDENT         ''
4,0-4,9:              NAME           'say_hello'
4,9-4,10:             LPAR           '('
4,10-4,11:            RPAR           ')'
4,11-4,12:            NEWLINE        '\n'
5,0-5,0:              ENDMARKER      ''
```