asttokens Documentation

Release 1.0.0

Grist Labs

Oct 30, 2022

Contents

The `asttokens` module annotates Python abstract syntax trees (ASTs) with the positions of tokens and text in the source code that generated them. This is helpful for tools that make source code transformations.

CHAPTER 1

User Guide

The `asttokens` module annotates Python abstract syntax trees (ASTs) with the positions of tokens and text in the source code that generated them.

It makes it possible for tools that work with logical AST nodes to find the particular text that resulted in those nodes, for example for automated refactoring or highlighting.

1.1 Installation

asttokens is available on PyPI: https://pypi.python.org/pypi/asttokens/:

```
pip install asttokens
```

The code is on GitHub: https://github.com/gristlabs/asttokens.

The API Reference is here: http://asttokens.readthedocs.io/en/latest/api-index.html.

1.2 Usage

ASTTokens works with both Python2 and Python3.

ASTTokens can annotate both trees built by ast, AND those built by astroid.

Here's an example:

```
import asttokens, ast
source = "Robot('blue').walk(steps=10*n)"
atok = asttokens.ASTTokens(source, parse=True)
```

Once the tree has been marked, nodes get `.first_token`, `.last_token` attributes, and the `ASTTokens` object offers helpful methods:

```
attr_node = next(n for n in ast.walk(atok.tree) if isinstance(n, ast.Attribute))
print(atok.get_text(attr_node))
start, end = attr_node.last_token.startpos, attr_node.last_token.endpos
print(atok.text[:start] + 'RUN' + atok.text[end:])
```

Which produces this output:

```
Robot('blue').walk
Robot('blue').RUN(steps=10*n)
```

The `ASTTokens` object also offers methods to walk and search the list of tokens that make up the code (or a particular AST node), which is more useful and powerful than dealing with the text directly.

1.3 Contribute

To contribute:

1. Fork this repository, and clone your fork.
2. Install the package with test dependencies (ideally in a virtualenv) with:

   ```
   pip install -e '.[test]'
   ```

3. Run tests in your current interpreter with the command `pytest` or `python -m pytest`.
4. Run tests across all supported interpreters with the `tox` command. You will need to have the interpreters installed separately. We recommend `pyenv` for that. Use `tox -p auto` to run the tests in parallel.
5. By default certain tests which take a very long time to run are skipped, but they are run on travis CI. To run them locally, set the environment variable `ASTTOKENS_SLOW_TESTS`. For example run `ASTTOKENS_SLOW_TESTS=1 tox` to run the full suite of tests.

CHAPTER 2

API Reference

This module enhances the Python AST tree with token and source code information, sufficent to detect the source text of each AST node. This is helpful for tools that make source code transformations.

2.1 ASTTokens

class asttokens.**ASTTokens**(*source_text*, *parse=False*, *tree=None*, *filename='<unknown>'*, *tokens=None*)

ASTTokens maintains the text of Python code in several forms: as a string, as line numbers, and as tokens, and is used to mark and access token and position information.

`source_text` must be a unicode or UTF8-encoded string. If you pass in UTF8 bytes, remember that all offsets you'll get are to the unicode text, which is available as the `.text` property.

If `parse` is set, the `source_text` will be parsed with `ast.parse()`, and the resulting tree marked with token info and made available as the `.tree` property.

If `tree` is given, it will be marked and made available as the `.tree` property. In addition to the trees produced by the `ast` module, ASTTokens will also mark trees produced using `astroid` library <https://www.astroid.org>.

If only `source_text` is given, you may use `.mark_tokens(tree)` to mark the nodes of an AST tree created separately.

filename
The filename that was parsed

find_token(*start_token*, *tok_type*, *tok_str=None*, *reverse=False*)
Looks for the first token, starting at start_token, that matches tok_type and, if given, the token string. Searches backwards if reverse is True. Returns ENDMARKER token if not found (you can check it with *token.ISEOF(t.type)*.

get_text_positions(*node*, *padded*)
Returns two `(lineno, col_offset)` tuples for the start and end of the given node. If the positions can't be determined, or the nodes don't correspond to any particular text, returns `(1, 0)` for both.

`padded` corresponds to the `padded` argument to `ast.get_source_segment()`. This means that if `padded` is True, the start position will be adjusted to include leading whitespace if `node` is a multiline statement.

get_token(*lineno*, *col_offset*)
Returns the token containing the given (lineno, col_offset) position, or the preceeding token if the position is between tokens.

get_token_from_offset(*offset*)
Returns the token containing the given character offset (0-based position in source text), or the preceeding token if the position is between tokens.

get_token_from_utf8(*lineno*, *col_offset*)
Same as get_token(), but interprets col_offset as a UTF8 offset, which is what *ast* uses.

get_tokens(*node*, *include_extra=False*)
Yields all tokens making up the given node. If include_extra is True, includes non-coding tokens such as tokenize.NL and .COMMENT.

mark_tokens(*root_node*)
Given the root of the AST or Astroid tree produced from source_text, visits all nodes marking them with token and position information by adding `.first_token` and `.last_token``attributes. This is done automatically in the constructor when ``parse` or `tree` arguments are set, but may be used manually with a separate AST or Astroid tree.

next_token(*tok*, *include_extra=False*)
Returns the next token after the given one. If include_extra is True, includes non-coding tokens from the tokenize module, such as NL and COMMENT.

prev_token(*tok*, *include_extra=False*)
Returns the previous token before the given one. If include_extra is True, includes non-coding tokens from the tokenize module, such as NL and COMMENT.

text
The source code passed into the constructor.

token_range(*first_token*, *last_token*, *include_extra=False*)
Yields all tokens in order from first_token through and including last_token. If include_extra is True, includes non-coding tokens such as tokenize.NL and .COMMENT.

tokens
The list of tokens corresponding to the source code from the constructor.

tree
The root of the AST tree passed into the constructor or parsed from the source code.

2.2 LineNumbers

class asttokens.**LineNumbers**(*text*)
Class to convert between character offsets in a text string, and pairs (line, column) of 1-based line and 0-based column numbers, as used by tokens and AST nodes.

This class expects unicode for input and stores positions in unicode. But it supports translating to and from utf8 offsets, which are used by ast parsing.

from_utf8_col(*line*, *utf8_column*)
Given a 1-based line number and 0-based utf8 column, returns a 0-based unicode column.

line_to_offset(*line*, *column*)
Converts 1-based line number and 0-based column to 0-based character offset into text.

offset_to_line(*offset*)
Converts 0-based character offset to pair (line, col) of 1-based line and 0-based column numbers.

2.3 util

class asttokens.util.**Token**
TokenInfo is an 8-tuple containing the same 5 fields as the tokens produced by the tokenize module, and 3 additional ones useful for this module:

- [0] .type Token type (see token.py)
- [1] .string Token (a string)
- [2] .start Starting (row, column) indices of the token (a 2-tuple of ints)
- [3] .end Ending (row, column) indices of the token (a 2-tuple of ints)
- [4] .line Original line (string)
- [5] .index Index of the token in the list of tokens that it belongs to.
- [6] .startpos Starting character offset into the input text.
- [7] .endpos Ending character offset into the input text.

asttokens.util.**token_repr**(*tok_type*, *string*)
Returns a human-friendly representation of a token with the given type and string.

asttokens.util.**visit_tree**(*node*, *previsit*, *postvisit*)
Scans the tree under the node depth-first using an explicit stack. It avoids implicit recursion via the function call stack to avoid hitting 'maximum recursion depth exceeded' error.

It calls `previsit()` and `postvisit()` as follows:

- **previsit(node, par_value) - should return (par_value, value)** `par_value` is as returned from `previsit()` of the parent.
- **postvisit(node, par_value, value) - should return value** `par_value` is as returned from `previsit()` of the parent, and `value` is as returned from `previsit()` of this node itself. The return `value` is ignored except the one for the root node, which is returned from the overall `visit_tree()` call.

For the initial node, `par_value` is None. `postvisit` may be None.

asttokens.util.**walk**(*node*)
Recursively yield all descendant nodes in the tree starting at `node` (including `node` itself), using depth-first pre-order traversal (yieling parents before their children).

This is similar to `ast.walk()`, but with a different order, and it works for both `ast` and `astroid` trees. Also, as `iter_children()`, it skips singleton nodes generated by `ast`.

asttokens.util.**replace**(*text*, *replacements*)
Replaces multiple slices of text with new values. This is a convenience method for making code modifications of ranges e.g. as identified by `ASTTokens.get_text_range(node)`. Replacements is an iterable of `(start, end, new_text)` tuples.

For example, `replace("this is a test", [(0, 4, "X"), (8, 9, "THE")])` produces `"X is THE test"`.

CHAPTER 3

License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index