Python talking about Python

...

Elyézer Rezende - elyezermr@gmail.com

Who am I?

- Brazilian
- Pythonista
- Podcaster (http://castalio.info/ in pt_BR)
- Open Source advocate (https://github.com/elyezer)
- Vim user (https://github.com/elyezer/.vim)
- Local Python event organizer (http://www.inatel.br/pythonday/ in pt_BR)
- Senior Quality Engineer at Red Hat on Satellite 6 Team
- Some (little) presence on Twitter (https://twitter.com/elyezer)

Agenda

- Context
- Abstract Syntax Tree
- ast module
- inspect module
- Building a Real Application
- Q&A

Context

Context

- Robottelo Automation Framework
  - Leverages Python packages like Requests, Paramiko and Selenium WebDriver to test Satellite 6 API, CLI and UI interfaces respectively.
  - Other useful tools FauxFactory (http://fauxfactory.readthedocs.org/) and NailGun (http://nailgun.readthedocs.org/).
- Testimony
  - Read and parse tests docstrings in order to provide reports of tests.
  - Uses ast module in order to provide its functionality.

Abstract Syntax Tree

Abstract Syntax Tree

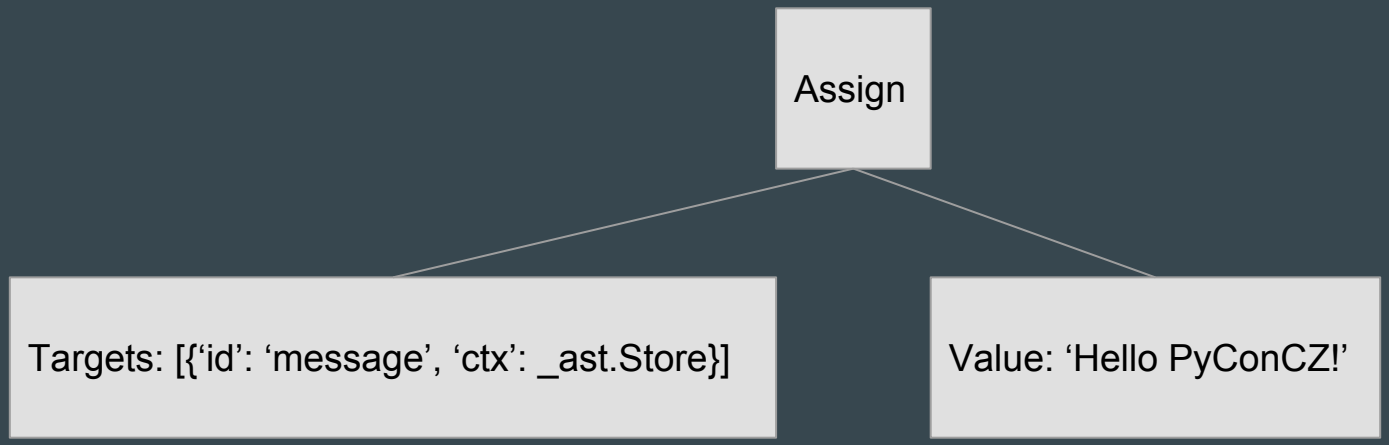
- It’s a tree representation of the abstract syntactic structure of source code written in a programming language, like Python :).
- Each node of the tree denotes a construct occurring in the source code.
- The syntax is "abstract" since it does not represent every detail appearing in the real syntax.
- Data structures used widely used in compilers during semantic analysis. In order to check correct usage of elements of the language and program.

Sample AST in Python

Code:

```
message = 'Hello PyCon CZ!'
```

AST:



ast module

ast module

The ast module helps Python applications to process trees of the Python abstract syntax grammar.

Helpers:

- *ast.parse(source, filename='<unknown>', mode='exec')*: Parse the source into an AST node. Equivalent to *compile(source, filename, mode, ast.PyCF_ONLY_AST)*.
- *ast.get_docstring(node, clean=True)*: Return the docstring of the given node (which must be a *FunctionDef*, *ClassDef* or *Module* node), or *None* if it has no docstring. If clean is true, clean up the docstring's indentation with *inspect.cleandoc()*.

ast module

Helpers (continuation):

- *ast.iter_child_nodes(node)*: Yield all direct child nodes of node, that is, all fields that are nodes and all items of fields that are lists of nodes.
- *ast.walk(node)*: Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

inspect module

inspect module

The inspect module provides several useful functions to help get information about live objects such as:

- modules
- classes
- methods
- functions
- tracebacks
- frame objects
- code objects.

Inspecting Types and Members

The *inspect.getmembers(object[, predicate])* function retrieves the members of an *object* such as a class or module.

The inspect module provides some functions whose names begin with “is”, like *inspect.ismodule*, *inspect.isclass*, *inspect.ismethod*, etc. These functions are suited for using as *predicate* and, if it is supplied, *inspect.getmembers* will include only members which the predicate returns a true value.

Depending on the members returned by the inspect.getmember some attributes can be expected. For example, a method object have __doc__, __name__, __qualname__, __func__ and __self__ attributes.

Retrieving information about Source Code

The *inspect* module can also retrieve information about the Source Code, like:

- Getting an object's docstring: *inspect.getdoc(object)*
- Getting the file which an object was defined: *inspect.getfile(object)*
- Trying to guess the module which an object was defined: *inspect.getmodule (object)*
- Getting the source code of an object: *inspect.getsource(object)*

Introspecting callables

```
>>> import inspect
>>> def foo(a, *args, b:int, **kwargs):
...     pass

>>> signature = inspect.signature(foo)
>>> signature.parameters['b'].annotation
<class 'int'>

>>> for name, parameter in signature.parameters.items():
...     print(parameter.kind, ':', name, '=', parameter.default)
POSITIONAL_OR_KEYWORD : a = <class 'inspect._empty'>
VAR_POSITIONAL : args = <class 'inspect._empty'>
KEYWORD_ONLY : b = <class 'inspect._empty'>
VAR_KEYWORD : kwargs = <class 'inspect._empty'>
```

POSITIONAL_ONLY: currently unsupported by Python function declaration syntax, but exemplified by existing functions implemented in C.

Inspecting a generator

A generator can be in one of the following four states and the current state can be determined by the use of *inspect.getgeneratorstate(...)* function.

- 'GEN_CREATED': Waiting to start execution.
- 'GEN_RUNNING': Currently being executed by the interpreter. *Visible in multithreaded application or if the generator call *getgeneratorstate* on itself.
- 'GEN_SUSPENDED': Currently suspended at a *yield* expression.
- 'GEN_CLOSED': Execution has completed.

Inspecting a coroutine

A coroutine (objects created by *async def* functions) can be in one of the following four states and the current state can be determined by the use of *inspect.getcoroutinestate(coroutine)* function.

- 'CORO_CREATED': Waiting to start execution.
- 'CORO_RUNNING': Currently being executed by the interpreter. *Visible in multithreaded application or if the generator call *getcoroutinestate* on itself.
- 'CORO_SUSPENDED': Currently suspended at an *await* expression.
- 'CORO_CLOSED': Execution has completed.

Building a Real Application

Application definition

- Given a python module, the app should be able to read the source code and generate a report in Markdown format, which will be printed to the stdout.
- For every class, its name will be a Markdown title and its docstring will be its description.
- For every method, its name will be a Markdown subtitle and its docstring will be its description.
- The application should be able to receive the python module path as command line argument.

```python
import unittest


class FeatureTestCase(unittest.TestCase):
  """Tests for a Feature."""

  def test_positive_action_1(self):
    """Test if Action 1 works properly."""

  def test_negative_action_1(self):
    """Test if Action 1 fails as expected."""

  def test_positive_action_2(self):
    """Test if Action 2 works properly.

    More description of Action 2 when it works.
    """

  def test_negative_action_1(self):
    """Test if Action 2 fails as expected."""
```

Sample python module

```
1  import argparse
2  import ast
3  import io
4
5
6  def title(message, underline='='):
7      """Return a string formated as a Markdown title.
8
9      underline argument will be printed on the line below the message.
10     """
11     return '{}\n{}\n\n'.format(message, underline * len(message))
12
```

Imports and helper function

```
35  def main():
36      parser = argparse.ArgumentParser()
37      parser.add_argument(
38          'test_path',
39          help='path to the test module',
40          type=argparse.FileType()
41      )
42      args = parser.parse_args()
43      print(generate_report(args.test_path), end='')
44
45  if __name__ == "__main__":
46      main()
```

Building the command line interface and the main function

```
14  def generate_report(file_handler):
15      """Generate a report about a test module in Markdown format."""
16      tree = ast.parse(file_handler.read())
17      with io.StringIO() as output:
18          nodes = [
19              node for node in ast.walk(tree)
20              if isinstance(node, (ast.ClassDef, ast.FunctionDef))
21          ]
22          for node in nodes:
23              separator = '='
24              if isinstance(node, ast.FunctionDef):
25                  separator = '-'
26              output.write(title(node.name, separator))
27              docstring = ast.get_docstring(node)
28              if docstring:
29                  output.write('{}\n\n'.format(docstring))
30              else:
31                  output.write('No docstring provided.\n')
32          return output.getvalue()
```

Generate the report

FeatureTestCase

Tests for a Feature.

test_positive_action_1

Test if Action 1 works properly.

test_negative_action_1

Test if Action 1 fails as expected.

test_positive_action_2

Test if Action 2 works properly.

More description of Action 2 when it works.

test_negative_action_1

Test if Action 2 fails as expected.

Parsed output

References

References

- Abstract Syntax Tree: https://en.wikipedia.org/wiki/Abstract_syntax_tree
- ast module: https://docs.python.org/3.4/library/ast.html
- inspect module: https://docs.python.org/3.4/library/inspect.html
- Keyword-Only Arguments: https://www.python.org/dev/peps/pep-3102/
- Fluent Python: http://shop.oreilly.com/product/0636920032519.do

Be Fluent on Python

Fluent Python

Clear, Concise, and Effective Programming

By Luciano Ramalho

Publisher: O'Reilly Media
Final Release Date: July 2015
Pages: 770

4.9

Read **25** Reviews | Write a Review

Thank You

elyezermr@gmail.com

https://twitter.com/elyezer

https://github.com/elyezer

https://github.com/elyezer/python-talking-about-python