Jedi Documentation

Release 0.14.1

Jedi contributors

Aug 08, 2019

Contents

Release v0.14.1. (*Installation*)

Jedi is a static analysis tool for Python that can be used in IDEs/editors. Jedi has a focus on autocompletion and goto functionality. Jedi is fast and is very well tested. It understands Python and stubs on a deep level.

Jedi has support for different goto functions. It's possible to search for usages and list names in a Python file to get information about them.

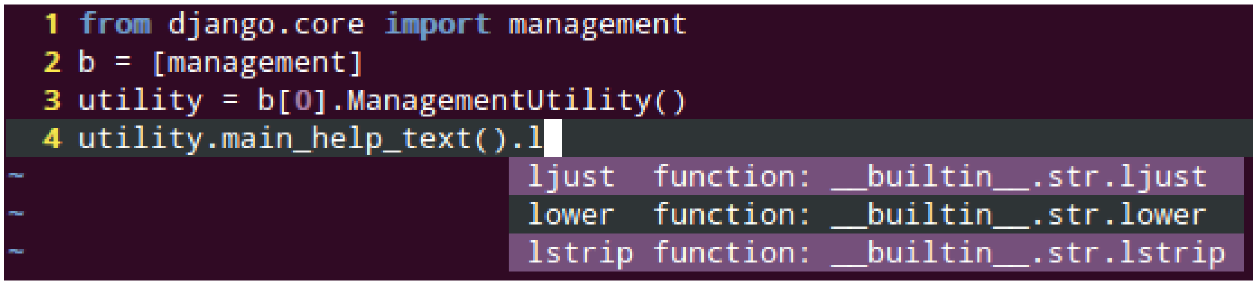
Jedi uses a very simple API to connect with IDE's. There's a reference implementation as a VIM-Plugin, which uses Jedi's autocompletion. We encourage you to use Jedi in your IDEs. Autocompletion in your REPL is also possible, IPython uses it natively and for the CPython REPL you have to install it.

Here's a simple example of the autocompletion feature:

```
>>> import jedi
>>> source = '''
... import json
... json.lo'''
>>> script = jedi.Script(source, 3, len('json.lo'), 'example.py')
>>> script
<Script: 'example.py' ...>
>>> completions = script.completions()
>>> completions
[<Completion: load>, <Completion: loads>]
>>> print(completions[0].complete)
ad
>>> print(completions[0].name)
load
```

As you see Jedi is pretty simple and allows you to concentrate on writing a good text editor, while still having very good IDE features for Python.

Autocompletion can look like this (e.g. VIM plugin):



```
from django.core import management
b = [management]
utility = b[0].ManagementUtility()
utility.main_help_text().l
~                                       ljust   function: __builtin__.str.ljust
~                                       lower   function: __builtin__.str.lower
~                                       lstrip  function: __builtin__.str.lstrip
```

CHAPTER 1

Docs

1.1 End User Usage

If you are a not an IDE Developer, the odds are that you just want to use *Jedi* as a browser plugin or in the shell. Yes that's *also possible*!

Jedi is relatively young and can be used in a variety of Plugins and Software. If your Editor/IDE is not among them, recommend *Jedi* to your IDE developers.

1.1.1 Editor Plugins

Vim:

- jedi-vim
- YouCompleteMe
- deoplete-jedi

Emacs:

- Jedi.el
- elpy
- anaconda-mode

Sublime Text 2/3:

- SublimeJEDI (ST2 & ST3)
- anaconda (only ST3)

SynWrite:

- SynJedi

TextMate:

- Textmate (Not sure if it's actually working)

Kate:

- Kate version 4.13+ supports it natively, you have to enable it, though.

Visual Studio Code:

- Python Extension

Atom:

- autocomplete-python-jedi

GNOME Builder:

- GNOME Builder supports it natively, and is enabled by default.

Gedit:

- gedi

Eric IDE:

- Eric IDE (Available as a plugin)

Web Debugger:

- wdb

and many more!

1.1.2 Tab Completion in the Python Shell

Starting with Ipython *6.0.0* Jedi is a dependency of IPython. Autocompletion in IPython is therefore possible without additional configuration.

There are two different options how you can use Jedi autocompletion in your Python interpreter. One with your custom `$HOME/.pythonrc.py` file and one that uses `PYTHONSTARTUP`.

Using `PYTHONSTARTUP`

To use Jedi completion in Python interpreter, add the following in your shell setup (e.g., `.bashrc`). This works only on Linux/Mac, because readline is not available on Windows. If you still want Jedi autocompletion in your REPL, just use IPython instead:

```
export PYTHONSTARTUP="$(python -m jedi repl)"
```

Then you will be able to use Jedi completer in your Python interpreter:

```
$ python
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.path.join('a', 'b').split().in<TAB>            # doctest: +SKIP
..dex   ..sert
```

Using a custom `$HOME/.pythonrc.py`

`jedi.utils.setup_readline`(*namespace_module=<module '__main__' from '/home/docs/checkouts/readthedocs.org/user_builds/jedi/envs/latest/bin/sphinx-build'>*)

Install Jedi completer to `readline`.

This function setups `readline` to use Jedi in Python interactive shell. If you want to use a custom `PYTHONSTARTUP` file (typically `$HOME/.pythonrc.py`), you can add this piece of code:

```
try:
    from jedi.utils import setup_readline
    setup_readline()
except ImportError:
    # Fallback to the stdlib readline completer if it is installed.
    # Taken from http://docs.python.org/2/library/rlcompleter.html
    print("Jedi is not installed, falling back to readline")
    try:
        import readline
        import rlcompleter
        readline.parse_and_bind("tab: complete")
    except ImportError:
        print("Readline is not installed either. No tab completion is enabled.")
```

This will fallback to the readline completer if Jedi is not installed. The readline completer will only complete names in the global namespace, so for example:

```
ran<TAB>
```

will complete to `range`

with both Jedi and readline, but:

```
range(10).cou<TAB>
```

will show complete to `range(10).count` only with Jedi.

You'll also need to add `export PYTHONSTARTUP=$HOME/.pythonrc.py` to your shell profile (usually `.bash_profile` or `.profile` if you use bash).

1.2 Installation and Configuration

You can either include *Jedi* as a submodule in your text editor plugin (like jedi-vim does by default), or you can install it systemwide.

Note: This just installs the *Jedi* library, not the *editor plugins*. For information about how to make it work with your editor, refer to the corresponding documentation.

1.2.1 The normal way

Most people use Jedi with a *editor plugins*. Typically you install Jedi by installing an editor plugin. No necessary steps are needed. Just take a look at the instructions for the plugin.

1.2.2 With pip

On any system you can install *Jedi* directly from the Python package index using pip:

```
sudo pip install jedi
```

If you want to install the current development version (master branch):

```
sudo pip install -e git://github.com/davidhalter/jedi.git#egg=jedi
```

1.2.3 System-wide installation via a package manager

Arch Linux

You can install *Jedi* directly from official Arch Linux packages:

- python-jedi (Python 3)
- python2-jedi (Python 2)

The specified Python version just refers to the *runtime environment* for *Jedi*. Use the Python 2 version if you're running vim (or whatever editor you use) under Python 2. Otherwise, use the Python 3 version. But whatever version you choose, both are able to complete both Python 2 and 3 *code*.

(There is also a packaged version of the vim plugin available: vim-jedi at Arch Linux.)

Debian

Debian packages are available in the unstable repository.

Others

We are in the discussion of adding *Jedi* to the Fedora repositories.

1.2.4 Manual installation from GitHub

If you prefer not to use an automated package installer, you can clone the source from GitHub and install it manually. To install it, run these commands:

```
git clone --recurse-submodules https://github.com/davidhalter/jedi
cd jedi
sudo python setup.py install
```

1.2.5 Inclusion as a submodule

If you use an editor plugin like jedi-vim, you can simply include *Jedi* as a git submodule of the plugin directory. Vim plugin managers like Vundle or Pathogen make it very easy to keep submodules up to date.

1.3 Features and Caveats

Jedi obviously supports autocompletion. It's also possible to get it working in (*your REPL (IPython, etc.)*).

Static analysis is also possible by using the command `jedi.names`.

Jedi would in theory support refactoring, but we have never publicized it, because it's not production ready. If you're interested in helping out here, let me know. With the latest parser changes, it should be very easy to actually make it work.

1.3.1 General Features

- Python 2.7 and 3.4+ support
- Ignores syntax errors and wrong indentation
- Can deal with complex module / function / class structures
- Great Virtualenv support
- Can infer function arguments from sphinx, epydoc and basic numpydoc docstrings, and PEP0484-style type hints (*type hinting*)
- Stub files

1.3.2 Supported Python Features

Jedi supports many of the widely used Python features:

- builtins
- returns, yields, yield from
- tuple assignments / array indexing / dictionary indexing / star unpacking
- with-statement / exception handling
- `*args` / `**kwargs`
- decorators / lambdas / closures
- generators / iterators
- some descriptors: property / staticmethod / classmethod
- some magic methods: `__call__`, `__iter__`, `__next__`, `__get__`, `__getitem__`, `__init__`
- `list.append()`, `set.add()`, `list.extend()`, etc.
- (nested) list comprehensions / ternary expressions
- relative imports
- `getattr()` / `__getattr__` / `__getattribute__`
- function annotations
- class decorators (py3k feature, are being ignored too, until I find a use case, that doesn't work with *Jedi*)
- simple/usual `sys.path` modifications
- `isinstance` checks for if/while/assert
- namespace packages (includes `pkgutil`, `pkg_resources` and PEP420 namespaces)

- Django / Flask / Buildout support

1.3.3 Not Supported

Not yet implemented:

- manipulations of instances outside the instance variables without using methods

Will probably never be implemented:

- metaclasses (how could an auto-completion ever support this)
- `setattr()`, `__import__()`
- writing to some dicts: `globals()`, `locals()`, `object.__dict__`
- evaluating `if` / `while` / `del`

1.3.4 Caveats

Slow Performance

Importing `numpy` can be quite slow sometimes, as well as loading the builtins the first time. If you want to speed things up, you could write import hooks in *Jedi*, which preload stuff. However, once loaded, this is not a problem anymore. The same is true for huge modules like `PySide`, `wx`, etc.

Security

Security is an important issue for *Jedi*. Therefore no Python code is executed. As long as you write pure Python, everything is evaluated statically. But: If you use builtin modules (`c_builtin`) there is no other option than to execute those modules. However: Execute isn't that critical (as e.g. in pythoncomplete, which used to execute *every* import!), because it means one import and no more. So basically the only dangerous thing is using the import itself. If your `c_builtin` uses some strange initializations, it might be dangerous. But if it does you're screwed anyways, because eventually you're going to execute your code, which executes the import.

1.3.5 Recipes

Here are some tips on how to use *Jedi* efficiently.

Type Hinting

If *Jedi* cannot detect the type of a function argument correctly (due to the dynamic nature of Python), you can help it by hinting the type using one of the following docstring/annotation syntax styles:

PEP-0484 style

https://www.python.org/dev/peps/pep-0484/

function annotations

```
def myfunction(node: ProgramNode, foo: str) -> None:
    """Do something with a ``node``.

    """
    node.| # complete here
```

assignment, for-loop and with-statement type hints (all Python versions). Note that the type hints must be on the same line as the statement

```
x = foo()  # type: int
x, y = 2, 3  # type: typing.Optional[int], typing.Union[int, str] # typing module is␣
↪mostly supported
for key, value in foo.items():  # type: str, Employee  # note that Employee must be␣
↪in scope
    pass
with foo() as f:  # type: int
    print(f + 3)
```

Most of the features in PEP-0484 are supported including the typing module (for Python < 3.5 you have to do `pip install typing` to use these), and forward references.

You can also use stub files.

Sphinx style

http://www.sphinx-doc.org/en/stable/domains.html#info-field-lists

```
def myfunction(node, foo):
    """Do something with a ``node``.

    :type node: ProgramNode
    :param str foo: foo parameter description

    """
    node.| # complete here
```

Epydoc

http://epydoc.sourceforge.net/manual-fields.html

```
def myfunction(node):
    """Do something with a ``node``.

    @type node: ProgramNode

    """
    node.| # complete here
```

Numpydoc

https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

In order to support the numpydoc format, you need to install the numpydoc package.

```
def foo(var1, var2, long_var_name='hi'):
    r"""A one-line summary that does not use variable names or the
    function name.

    ...

    Parameters
    ----------
    var1 : array_like
        Array_like means all those objects -- lists, nested lists,
        etc. -- that can be converted to an array. We can also
        refer to variables like `var1`.
```

(continues on next page)

(continued from previous page)

```
    var2 : int
        The type above can either refer to an actual Python type
        (e.g. ``int``), or describe the type of the variable in more
        detail, e.g. ``(N,) ndarray`` or ``array_like``.
    long_variable_name : {'hi', 'ho'}, optional
        Choices in brackets, default first when optional.

    ...

    """
    var2.| # complete here
```

1.3.6 A little history

The Star Wars Jedi are awesome. My Jedi software tries to imitate a little bit of the precognition the Jedi have. There's even an awesome scene of Monty Python Jedis :-).

But actually the name hasn't so much to do with Star Wars. It's part of my second name.

After I explained Guido van Rossum, how some parts of my auto-completion work, he said (we drank a beer or two):

> *"Oh, that worries me. . . "*

When it's finished, I hope he'll like it :-)

I actually started Jedi, because there were no good solutions available for VIM. Most auto-completions just didn't work well. The only good solution was PyCharm. But I like my good old VIM. Rope was never really intended to be an auto-completion (and also I really hate project folders for my Python scripts). It's more of a refactoring suite. So I decided to do my own version of a completion, which would execute non-dangerous code. But I soon realized, that this wouldn't work. So I built an extremely recursive thing which understands many of Python's key features.

By the way, I really tried to program it as understandable as possible. But I think understanding it might need quite some time, because of its recursive nature.

1.4 API Overview

Note: This documentation is for Plugin developers, who want to improve their editors/IDE autocompletion

If you want to use *Jedi*, you first need to `import jedi`. You then have direct access to the `Script`. You can then call the functions documented here. These functions return *API classes*.

1.4.1 Deprecations

The deprecation process is as follows:

1. A deprecation is announced in the next major/minor release.
2. We wait either at least a year & at least two minor releases until we remove the deprecated functionality.

1.4.2 API Documentation

The API consists of a few different parts:

- The main starting points for completions/goto: `Script` and `Interpreter`

- Helpful functions: `names()`, `preload_module()` and `set_debug_function()`
- *API Result Classes*
- *Python Versions/Virtualenv Support* with functions like `find_system_environments()` and `find_virtualenvs()`

Static Analysis Interface

Jedi is a static analysis tool for Python that can be used in IDEs/editors. Jedi has a focus on autocompletion and goto functionality. Jedi is fast and is very well tested. It understands Python and stubs on a deep level.

Jedi has support for different goto functions. It's possible to search for usages and list names in a Python file to get information about them.

Jedi uses a very simple API to connect with IDE's. There's a reference implementation as a VIM-Plugin, which uses Jedi's autocompletion. We encourage you to use Jedi in your IDEs. Autocompletion in your REPL is also possible, IPython uses it natively and for the CPython REPL you have to install it.

Here's a simple example of the autocompletion feature:

```
>>> import jedi
>>> source = '''
... import json
... json.lo'''
>>> script = jedi.Script(source, 3, len('json.lo'), 'example.py')
>>> script
<Script: 'example.py' ...>
>>> completions = script.completions()
>>> completions
[<Completion: load>, <Completion: loads>]
>>> print(completions[0].complete)
ad
>>> print(completions[0].name)
load
```

As you see Jedi is pretty simple and allows you to concentrate on writing a good text editor, while still having very good IDE features for Python.

class jedi.**Script**(*source=None, line=None, column=None, path=None, encoding='utf-8', sys_path=None, environment=None, _project=None*)

A Script is the base for completions, goto or whatever you want to do with *Jedi*.

You can either use the `source` parameter or `path` to read a file. Usually you're going to want to use both of them (in an editor).

The script might be analyzed in a different `sys.path` than *Jedi*:

- if *sys_path* parameter is not `None`, it will be used as `sys.path` for the script;
- if *sys_path* parameter is `None` and `VIRTUAL_ENV` environment variable is defined, `sys.path` for the specified environment will be guessed (see `jedi.evaluate.sys_path.get_venv_path()`) and used for the script;
- otherwise `sys.path` will match that of *Jedi*.

Parameters

- **source** (*str*) – The source code of the current file, separated by newlines.
- **line** (*int*) – The line to perform actions on (starting with 1).

- **column** (*int*) – The column of the cursor (starting with 0).
- **path** (*str or None*) – The path of the file in the file system, or `''` if it hasn't been saved yet.
- **encoding** (*str*) – The encoding of `source`, if it is not a `unicode` object (default `'utf-8'`).
- **sys_path** (*list*) – `sys.path` to use during analysis of the script
- **environment** (*Environment*) – TODO

completions()

Return `classes.Completion` objects. Those objects contain information about the completions, more than just names.

Returns Completion objects, sorted by name and __ comes last.

Return type list of `classes.Completion`

goto_definitions(***kwargs*)

Return the definitions of a the path under the cursor. goto function! This follows complicated paths and returns the end, not the first definition. The big difference between *goto_assignments()* and *goto_definitions()* is that *goto_assignments()* doesn't follow imports and statements. Multiple objects may be returned, because Python itself is a dynamic language, which means depending on an option you can have two different versions of a function.

Parameters

- **only_stubs** – Only return stubs for this goto call.
- **prefer_stubs** – Prefer stubs to Python objects for this type inference call.

Return type list of `classes.Definition`

goto_assignments(*follow_imports=False*, *follow_builtin_imports=False*, ***kwargs*)

Return the first definition found, while optionally following imports. Multiple objects may be returned, because Python itself is a dynamic language, which means depending on an option you can have two different versions of a function.

Note: It is deprecated to use follow_imports and follow_builtin_imports as positional arguments. Will be a keyword argument in 0.16.0.

Parameters

- **follow_imports** – The goto call will follow imports.
- **follow_builtin_imports** – If follow_imports is True will decide if it follow builtin imports.
- **only_stubs** – Only return stubs for this goto call.
- **prefer_stubs** – Prefer stubs to Python objects for this goto call.

Return type list of `classes.Definition`

usages(*additional_module_paths=()*, ***kwargs*)

Return `classes.Definition` objects, which contain all names that point to the definition of the name under the cursor. This is very useful for refactoring (renaming), or to show all usages of a variable.

Parameters

- **additional_module_paths** – Deprecated, never ever worked.
- **include_builtins** – Default True, checks if a usage is a builtin (e.g. `sys`) and in that case does not return it.

Return type list of `classes.Definition`

call_signatures()

Return the function object of the call you're currently in.

E.g. if the cursor is here:

```
abs(# <-- cursor is here
```

This would return the `abs` function. On the other hand:

```
abs()# <-- cursor is here
```

This would return an empty list..

Return type list of `classes.CallSignature`

class jedi.**Interpreter**(*source*, *namespaces*, ***kwds*)

Jedi API for Python REPLs.

In addition to completion of simple attribute access, Jedi supports code completion based on static code analysis. Jedi can complete attributes of object which is not initialized yet.

```
>>> from os.path import join
>>> namespace = locals()
>>> script = Interpreter('join("").up', [namespace])
>>> print(script.completions()[0].name)
upper
```

Parse *source* and mixin interpreted Python objects from *namespaces*.

Parameters

- **source** (*str*) – Code to parse.
- **namespaces** (*list of dict*) – a list of namespace dictionaries such as the one returned by `locals()`.

Other optional arguments are same as the ones for *Script*. If *line* and *column* are None, they are assumed be at the end of *source*.

jedi.**names**(*source=None*, *path=None*, *encoding='utf-8'*, *all_scopes=False*, *definitions=True*, *references=False*, *environment=None*)

Returns a list of *Definition* objects, containing name parts. This means you can call `Definition.goto_assignments()` and get the reference of a name. The parameters are the same as in *Script*, except or the following ones:

Parameters

- **all_scopes** – If True lists the names of all scopes instead of only the module namespace.
- **definitions** – If True lists the names that have been defined by a class, function or a statement (`a = b` returns `a`).
- **references** – If True lists all the names that are not listed by `definitions=True`. E.g. `a = b` returns `b`.

`jedi.`**`preload_module`**`(`**modules*`)`

Preloading modules tells Jedi to load a module now, instead of lazy parsing of modules. Usful for IDEs, to control which modules to load on startup.

Parameters **`modules`** – different module names, list of string.

`jedi.`**`set_debug_function`**`(`*func_cb=<function print_to_stdout>*, *warnings=True*, *notices=True*, *speed=True*`)`

Define a callback debug function to get all the debug messages.

If you don't specify any arguments, debug messages will be printed to stdout.

Parameters **`func_cb`** – The callback function for debug messages, with n params.

Environments

Environments are a way to activate different Python versions or Virtualenvs for static analysis. The Python binary in that environment is going to be executed.

`jedi.`**`find_system_environments`**`()`

Ignores virtualenvs and returns the Python versions that were installed on your system. This might return nothing, if you're running Python e.g. from a portable version.

The environments are sorted from latest to oldest Python version.

Yields `Environment`

`jedi.`**`find_virtualenvs`**`(`*paths=None*, ***kwargs*`)`

Parameters

- **`paths`** – A list of paths in your file system to be scanned for Virtualenvs. It will search in these paths and potentially execute the Python binaries. Also the VIRTUAL_ENV variable will be checked if it contains a valid Virtualenv.
- **`safe`** – Default True. In case this is False, it will allow this function to execute potential *python* environments. An attacker might be able to drop an executable in a path this function is searching by default. If the executable has not been installed by root, it will not be executed.

Yields `Environment`

`jedi.`**`get_system_environment`**`(`*version*`)`

Return the first Python environment found for a string of the form 'X.Y' where X and Y are the major and minor versions of Python.

Raises *`InvalidPythonEnvironment`*

Returns `Environment`

`jedi.`**`create_environment`**`(`*path*, *safe=True*`)`

Make it possible to manually create an Environment object by specifying a Virtualenv path or an executable path.

Raises *`InvalidPythonEnvironment`*

Returns `Environment`

`jedi.`**`get_default_environment`**`()`

Tries to return an active Virtualenv. If there is no VIRTUAL_ENV variable set it will return the latest Python version installed on the system. This makes it possible to use as many new Python features as possible when using autocompletion and other functionality.

Returns `Environment`

exception `jedi.`**`InvalidPythonEnvironment`**

If you see this exception, the Python executable or Virtualenv you have been trying to use is probably not a correct Python version.

class `jedi.api.environment.`**`Environment`**(*executable*)

This class is supposed to be created by internal Jedi architecture. You should not create it directly. Please use create_environment or the other functions instead. It is then returned by that function.

`get_sys_path`(**args*, ***kwargs*)

The sys path for this environment. Does not include potential modifications like `sys.path.append`.

Returns list of str

1.4.3 Examples

Completions:

```
>>> import jedi
>>> source = '''import json; json.l'''
>>> script = jedi.Script(source, 1, 19, '')
>>> script
<jedi.api.Script object at 0x2121b10>
>>> completions = script.completions()
>>> completions
[<Completion: load>, <Completion: loads>]
>>> completions[1]
<Completion: loads>
>>> completions[1].complete
'oads'
>>> completions[1].name
'loads'
```

Definitions / Goto:

```
>>> import jedi
>>> source = '''def my_func():
...     print 'called'
...
... alias = my_func
... my_list = [1, None, alias]
... inception = my_list[2]
...
... inception()'''
>>> script = jedi.Script(source, 8, 1, '')
>>>
>>> script.goto_assignments()
[<Definition inception=my_list[2]>]
>>>
>>> script.goto_definitions()
[<Definition def my_func>]
```

Related names:

```
>>> import jedi
>>> source = '''x = 3
... if 1 == 2:
```

(continues on next page)

(continued from previous page)

```
...     x = 4
... else:
...     del x'''
>>> script = jedi.Script(source, 5, 8, '')
>>> rns = script.related_names()
>>> rns
[<RelatedName x@3,4>, <RelatedName x@1,0>]
>>> rns[0].start_pos
(3, 4)
>>> rns[0].is_keyword
False
>>> rns[0].text
'x'
```

1.5 API Return Classes

The *jedi.api.classes* module contains the return classes of the API. These classes are the much bigger part of the whole API, because they contain the interesting information about completion and goto operations.

`jedi.api.classes.`**`defined_names`**(*evaluator*, *context*)
List sub-definitions (e.g., methods in class).

Return type list of Definition

class `jedi.api.classes.`**`BaseDefinition`**(*evaluator*, *name*)

`module_path`
Shows the file path of a module. e.g. `/usr/lib/python2.7/os.py`

`name`
Name of variable/function/class/module.

For example, for `x = None` it returns `'x'`.

Return type str or None

`type`
The type of the definition.

Here is an example of the value of this attribute. Let's consider the following source. As what is in `variable` is unambiguous to Jedi, *jedi.Script.goto_definitions()* should return a list of definition for `sys`, `f`, `C` and `x`.

```
>>> from jedi._compatibility import no_unicode_pprint
>>> from jedi import Script
>>> source = '''
... import keyword
...
... class C:
...     pass
...
... class D:
...     pass
...
... x = D()
...
```

(continues on next page)

(continued from previous page)

```
... def f():
...     pass
...
... for variable in [keyword, f, C, x]:
...     variable'''
```

```
>>> script = Script(source)
>>> defs = script.goto_definitions()
```

Before showing what is in `defs`, let's sort it by *`line`* so that it is easy to relate the result to the source code.

```
>>> defs = sorted(defs, key=lambda d: d.line)
>>> no_unicode_pprint(defs)  # doctest: +NORMALIZE_WHITESPACE
[<Definition full_name='keyword', description='module keyword'>,
 <Definition full_name='__main__.C', description='class C'>,
 <Definition full_name='__main__.D', description='instance D'>,
 <Definition full_name='__main__.f', description='def f'>]
```

Finally, here is what you can get from *`type`*:

```
>>> defs = [str(d.type) for d in defs]  # It's unicode and in Py2 has u␣
↪before it.
>>> defs[0]
'module'
>>> defs[1]
'class'
>>> defs[2]
'instance'
>>> defs[3]
'function'
```

Valid values for are `module`, `class`, `instance`, `function`, `param`, `path` and `keyword`.

`module_name`
The module name.

```
>>> from jedi import Script
>>> source = 'import json'
>>> script = Script(source, path='example.py')
>>> d = script.goto_definitions()[0]
>>> print(d.module_name)  # doctest: +ELLIPSIS
json
```

`in_builtin_module``()`
Whether this is a builtin module.

`line`
The line where the definition occurs (starting with 1).

`column`
The column where the definition occurs (starting with 0).

`docstring``(`*raw=False*, *fast=True*`)`
Return a document string for this completion object.

Example:

```
>>> from jedi import Script
>>> source = '''\
... def f(a, b=1):
...     "Document for function f."
... '''
>>> script = Script(source, 1, len('def f'), 'example.py')
>>> doc = script.goto_definitions()[0].docstring()
>>> print(doc)
f(a, b=1)
<BLANKLINE>
Document for function f.
```

Notice that useful extra information is added to the actual docstring. For function, it is call signature. If you need actual docstring, use `raw=True` instead.

```
>>> print(script.goto_definitions()[0].docstring(raw=True))
Document for function f.
```

Parameters **`fast`** – Don't follow imports that are only one level deep like `import foo`, but follow `from foo import bar`. This makes sense for speed reasons. Completing *import a* is slow if you use the `foo.docstring(fast=False)` on every object, because it parses all libraries starting with `a`.

`description`
A textual description of the object.

`full_name`
Dot-separated path of this object.

It is in the form of `<module>[.<submodule>[...]][.<object>]`. It is useful when you want to look up Python manual of the object at hand.

Example:

```
>>> from jedi import Script
>>> source = '''
... import os
... os.path.join'''
>>> script = Script(source, 3, len('os.path.join'), 'example.py')
>>> print(script.goto_definitions()[0].full_name)
os.path.join
```

Notice that it returns `'os.path.join'` instead of (for example) `'posixpath.join'`. This is not correct, since the modules name would be `<module 'posixpath' ...>``. However most users find the latter more practical.

`is_stub`()

`goto_assignments`(***kwargs*)

`infer`(***kwargs*)

`params`
Deprecated! Will raise a warning soon. Use get_signatures()[…].params.

Raises an `AttributeError` if the definition is not callable. Otherwise returns a list of *Definition* that represents the params.

`parent`()

get_line_code (*before=0, after=0*)

Returns the line of code where this object was defined.

Parameters

- **before** – Add n lines before the current line to the output.
- **after** – Add n lines after the current line to the output.

Return str Returns the line(s) of code or an empty string if it's a builtin.

get_signatures ()

execute ()

class jedi.api.classes.**Completion** (*evaluator*, *name*, *stack*, *like_name_length*)

Completion objects are returned from `api.Script.completions()`. They provide additional information about a completion.

complete

Return the rest of the word, e.g. completing `isinstance`:

```
isinstan# <-- Cursor is here
```

would return the string 'ce'. It also adds additional stuff, depending on your *settings.py*.

Assuming the following function definition:

```
def foo(param=0):
    pass
```

completing `foo(par` would give a `Completion` which *complete* would be *am=*

name_with_symbols

Similar to `name`, but like `name` returns also the symbols, for example assuming the following function definition:

```
def foo(param=0):
    pass
```

completing `foo(` would give a `Completion` which `name_with_symbols` would be "param=".

docstring (*raw=False*, *fast=True*)

Return a document string for this completion object.

Example:

```
>>> from jedi import Script
>>> source = '''\
... def f(a, b=1):
...     "Document for function f."
... '''
>>> script = Script(source, 1, len('def f'), 'example.py')
>>> doc = script.goto_definitions()[0].docstring()
>>> print(doc)
f(a, b=1)
<BLANKLINE>
Document for function f.
```

Notice that useful extra information is added to the actual docstring. For function, it is call signature. If you need actual docstring, use `raw=True` instead.

```
>>> print(script.goto_definitions()[0].docstring(raw=True))
Document for function f.
```

Parameters **fast** – Don't follow imports that are only one level deep like `import foo`, but follow `from foo import bar`. This makes sense for speed reasons. Completing *import a* is slow if you use the `foo.docstring(fast=False)` on every object, because it parses all libraries starting with `a`.

description
Provide a description of the completion object.

follow_definition(**args*, ***kwargs*)
Deprecated!

Return the original definitions. I strongly recommend not using it for your completions, because it might slow down *Jedi*. If you want to read only a few objects (<=20), it might be useful, especially to get the original docstrings. The basic problem of this function is that it follows all results. This means with 1000 completions (e.g. numpy), it's just PITA-slow.

class `jedi.api.classes.`**Definition**(*evaluator*, *definition*)
Definition objects are returned from `api.Script.goto_assignments()` or `api.Script.goto_definitions()`.

description
A description of the `Definition` object, which is heavily used in testing. e.g. for `isinstance` it returns `def isinstance`.

Example:

```
>>> from jedi._compatibility import no_unicode_pprint
>>> from jedi import Script
>>> source = '''
... def f():
...     pass
...
... class C:
...     pass
...
... variable = f if random.choice([0,1]) else C'''
>>> script = Script(source, column=3)  # line is maximum by default
>>> defs = script.goto_definitions()
>>> defs = sorted(defs, key=lambda d: d.line)
>>> no_unicode_pprint(defs)  # doctest: +NORMALIZE_WHITESPACE
[<Definition full_name='__main__.f', description='def f'>,
 <Definition full_name='__main__.C', description='class C'>]
>>> str(defs[0].description)  # strip literals in python2
'def f'
>>> str(defs[1].description)
'class C'
```

desc_with_module
In addition to the definition, also return the module.

Warning: Don't use this function yet, its behaviour may change. If you really need it, talk to me.

defined_names(**args*, ***kwargs*)
List sub-definitions (e.g., methods in class).

Return type list of Definition

is_definition()
Returns True, if defined as a name in a statement, function or class. Returns False, if it's a reference to such a definition.

class jedi.api.classes.**Signature**(*evaluator*, *signature*)
Signature objects is the return value of *Script.function_definition*. It knows what functions you are currently in. e.g. *isinstance(* would return the *isinstance* function. without *(* it would return nothing.

params

Return list of ParamDefinition

to_string()

class jedi.api.classes.**CallSignature**(*evaluator*, *signature*, *call_details*)
CallSignature objects is the return value of *Script.call_signatures*. It knows what functions you are currently in. e.g. *isinstance(* would return the *isinstance* function with its params. Without *(* it would return nothing.

index
The Param index of the current call. Returns None if the index cannot be found in the curent call.

bracket_start
The line/column of the bracket that is responsible for the last function call.

class jedi.api.classes.**ParamDefinition**(*evaluator*, *definition*)

infer_default()

Return list of Definition

infer_annotation(***kwargs*)

Return list of Definition

Parameters **execute_annotation** – If False, the values are not executed and you get classes instead of instances.

to_string()

kind
Returns an enum instance. Returns the same values as the builtin `inspect.Parameter.kind`.

No support for Python < 3.4 anymore.

1.6 Settings

This module contains variables with global *Jedi* settings. To change the behavior of *Jedi*, change the variables defined in `jedi.settings`.

Plugins should expose an interface so that the user can adjust the configuration.

Example usage:

```
from jedi import settings
settings.case_insensitive_completion = True
```

1.6.1 Completion output

`jedi.settings.`**`case_insensitive_completion = True`**

The completion is by default case insensitive.

`jedi.settings.`**`add_bracket_after_function = False`**

Adds an opening bracket after a function, because that's normal behaviour. Removed it again, because in VIM that is not very practical.

`jedi.settings.`**`no_completion_duplicates = True`**

If set, completions with the same name don't appear in the output anymore, but are in the *same_name_completions* attribute.

1.6.2 Filesystem cache

`jedi.settings.`**`cache_directory = '/home/docs/.cache/jedi'`**

The path where the cache is stored.

On Linux, this defaults to `~/.cache/jedi/`, on OS X to `~/Library/Caches/Jedi/` and on Windows to `%APPDATA%\Jedi\Jedi\`. On Linux, if environment variable `$XDG_CACHE_HOME` is set, `$XDG_CACHE_HOME/jedi` is used instead of the default one.

`jedi.settings.`**`use_filesystem_cache = True`**

Use filesystem cache to save once parsed files with pickle.

1.6.3 Parser

`jedi.settings.`**`fast_parser = True`**

Use the fast parser. This means that reparsing is only being done if something has been changed e.g. to a function. If this happens, only the function is being reparsed.

1.6.4 Dynamic stuff

`jedi.settings.`**`dynamic_array_additions = True`**

check for *append*, etc. on arrays: [], {}, () as well as list/set calls.

`jedi.settings.`**`dynamic_params = True`**

A dynamic param completion, finds the callees of the function, which define the params of a function.

`jedi.settings.`**`dynamic_params_for_other_modules = True`**

Do the same for other modules.

`jedi.settings.`**`additional_dynamic_modules = []`**

Additional modules in which *Jedi* checks if statements are to be found. This is practical for IDEs, that want to administrate their modules themselves.

`jedi.settings.`**`auto_import_modules = ['gi']`**

Modules that are not analyzed but imported, although they contain Python code. This improves autocompletion for libraries that use `setattr` or `globals()` modifications a lot.

1.6.5 Caching

`jedi.settings.`**`call_signatures_validity = 3.0`**

Finding function calls might be slow (0.1-0.5s). This is not acceptible for normal writing. Therefore cache it for a short time.

1.7 Jedi Development

Note: This documentation is for Jedi developers who want to improve Jedi itself, but have no idea how Jedi works. If you want to use Jedi for your IDE, look at the plugin api. It is also important to note that it's a pretty old version and some things might not apply anymore.

1.7.1 Introduction

This page tries to address the fundamental demand for documentation of the *Jedi* internals. Understanding a dynamic language is a complex task. Especially because type inference in Python can be a very recursive task. Therefore *Jedi* couldn't get rid of complexity. I know that **simple is better than complex**, but unfortunately it sometimes requires complex solutions to understand complex systems.

Since most of the Jedi internals have been written by me (David Halter), this introduction will be written mostly by me, because no one else understands to the same level how Jedi works. Actually this is also the reason for exactly this part of the documentation. To make multiple people able to edit the Jedi core.

In five chapters I'm trying to describe the internals of *Jedi*:

- *The Jedi Core*
- *Core Extensions*
- *Imports & Modules*
- *Caching & Recursions*
- *Helper modules*

Note: Testing is not documented here, you'll find that right here.

1.7.2 The Jedi Core

The core of Jedi consists of three parts:

- *Parser*
- *Python code evaluation*
- *API*

Most people are probably interested in *code evaluation*, because that's where all the magic happens. I need to introduce the *parser* first, because `jedi.evaluate` uses it extensively.

Parser

Jedi used to have it's internal parser, however this is now a separate project and is called parso.

The parser creates a syntax tree that *Jedi* analyses and tries to understand. The grammar that this parsers uses is very similar to the official Python grammar files.

Evaluation of python code (evaluate/__init__.py)

Evaluation of Python code in *Jedi* is based on three assumptions:

- The code uses as least side effects as possible. Jedi understands certain list/tuple/set modifications, but there's no guarantee that Jedi detects everything (list.append in different modules for example).
- No magic is being used:
  - metaclasses
  - `setattr()` / `__import__()`
  - writing to `globals()`, `locals()`, `object.__dict__`
- The programmer is not a total dick, e.g. like this :-)

The actual algorithm is based on a principle called lazy evaluation. That said, the typical entry point for static analysis is calling `eval_expr_stmt`. There's separate logic for autocompletion in the API, the evaluator is all about evaluating an expression.

TODO this paragraph is not what jedi does anymore, it's similar, but not the same.

Now you need to understand what follows after `eval_expr_stmt`. Let's make an example:

```
import datetime
datetime.date.toda# <-- cursor here
```

First of all, this module doesn't care about completion. It really just cares about `datetime.date`. At the end of the procedure `eval_expr_stmt` will return the `date` class.

To *visualize* this (simplified):

- `Evaluator.eval_expr_stmt` doesn't do much, because there's no assignment.
- `Context.eval_node` cares for resolving the dotted path
- `Evaluator.find_types` searches for global definitions of datetime, which it finds in the definition of an import, by scanning the syntax tree.
- Using the import logic, the datetime module is found.
- Now `find_types` is called again by `eval_node` to find `date` inside the datetime module.

Now what would happen if we wanted `datetime.date.foo.bar`? Two more calls to `find_types`. However the second call would be ignored, because the first one would return nothing (there's no foo attribute in `date`).

What if the import would contain another `ExprStmt` like this:

```
from foo import bar
Date = bar.baz
```

Well… You get it. Just another `eval_expr_stmt` recursion. It's really easy. Python can obviously get way more complicated then this. To understand tuple assignments, list comprehensions and everything else, a lot more code had to be written.

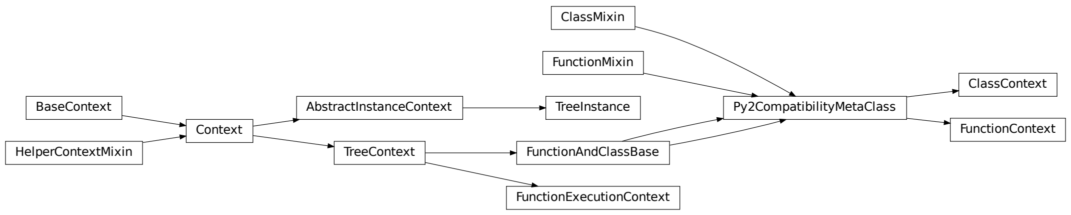
Jedi has been tested very well, so you can just start modifying code. It's best to write your own test first for your "new" feature. Don't be scared of breaking stuff. As long as the tests pass, you're most likely to be fine.

I need to mention now that lazy evaluation is really good because it only *evaluates* what needs to be *evaluated*. All the statements and modules that are not used are just being ignored.

Evaluation Contexts (evaluate/base_context.py)

Contexts are the "values" that Python would return. However Contexts are at the same time also the "contexts" that a user is currently sitting in.

A ContextSet is typically used to specify the return of a function or any other static analysis operation. In jedi there are always multiple returns and not just one.



Name resolution (evaluate/finder.py)

Searching for names with given scope and name. This is very central in Jedi and Python. The name resolution is quite complicated with descripter, `__getattribute__`, `__getattr__`, `global`, etc.

If you want to understand name resolution, please read the first few chapters in http://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/.

Flow checks

Flow checks are not really mature. There's only a check for `isinstance`. It would check whether a flow has the form of `if isinstance(a, type_or_tuple)`. Unfortunately every other thing is being ignored (e.g. a == '' would be easy to check for -> a is a string). There's big potential in these checks.

API (api/__init__.py and api/classes.py)

The API has been designed to be as easy to use as possible. The API documentation can be found here. The API itself contains little code that needs to be mentioned here. Generally I'm trying to be conservative with the API. I'd rather not add new API features if they are not necessary, because it's much harder to deprecate stuff than to add it later.

1.7.3 Core Extensions

Core Extensions is a summary of the following topics:

- *Iterables & Dynamic Arrays*
- *Dynamic Parameters*
- *Docstrings*
- *Refactoring*

These topics are very important to understand what Jedi additionally does, but they could be removed from Jedi and Jedi would still work. But slower and without some features.

Iterables & Dynamic Arrays (evaluate/context/iterable.py)

To understand Python on a deeper level, *Jedi* needs to understand some of the dynamic features of Python like lists that are filled after creation:

Contains all classes and functions to deal with lists, dicts, generators and iterators in general.

Array modifications

If the content of an array (`set`/`list`) is requested somewhere, the current module will be checked for appearances of `arr.append`, `arr.insert`, etc. If the `arr` name points to an actual array, the content will be added

This can be really cpu intensive, as you can imagine. Because *Jedi* has to follow **every** `append` and check wheter it's the right array. However this works pretty good, because in *slow* cases, the recursion detector and other settings will stop this process.

It is important to note that:

1. Array modfications work only in the current module.
2. Jedi only checks Array additions; `list.pop`, etc are ignored.

Parameter completion (evaluate/dynamic.py)

One of the really important features of *Jedi* is to have an option to understand code like this:

```
def foo(bar):
    bar. # completion here
foo(1)
```

There's no doubt wheter bar is an `int` or not, but if there's also a call like `foo('str')`, what would happen? Well, we'll just show both. Because that's what a human would expect.

It works as follows:

- *Jedi* sees a param
- search for function calls named `foo`
- execute these calls and check the input.

Docstrings (evaluate/docstrings.py)

Docstrings are another source of information for functions and classes. *`jedi.evaluate.dynamic`* tries to find all executions of functions, while the docstring parsing is much easier. There are three different types of docstrings that *Jedi* understands:

- Sphinx
- Epydoc
- Numpydoc

For example, the sphinx annotation `:type foo: str` clearly states that the type of `foo` is `str`.

As an addition to parameter searching, this module also provides return annotations.

Refactoring (evaluate/refactoring.py)

THIS is not in active development, please check https://github.com/davidhalter/jedi/issues/667 first before editing.

Introduce some basic refactoring functions to *Jedi*. This module is still in a very early development stage and needs much testing and improvement.

> **Warning:** I won't do too much here, but if anyone wants to step in, please do. Refactoring is none of my priorities

It uses the *Jedi* API and supports currently the following functions (sometimes bug-prone):

- rename
- extract variable
- inline variable

1.7.4 Imports & Modules

- Modules
- *Builtin Modules*
- *Imports*

Compiled Modules (evaluate/compiled.py)

Imports (evaluate/imports.py)

`jedi.evaluate.imports` is here to resolve import statements and return the modules/classes/functions/whatever, which they stand for. However there's not any actual importing done. This module is about finding modules in the filesystem. This can be quite tricky sometimes, because Python imports are not always that simple.

This module uses imp for python up to 3.2 and importlib for python 3.3 on; the correct implementation is delegated to _compatibility.

This module also supports import autocompletion, which means to complete statements like `from datetim` (cursor at the end would return `datetime`).

1.7.5 Caching & Recursions

- *Caching*
- *Recursions*

Caching (cache.py)

This caching is very important for speed and memory optimizations. There's nothing really spectacular, just some decorators. The following cache types are available:

- `time_cache` can be used to cache something for just a limited time span, which can be useful if there's user interaction and the user cannot react faster than a certain time.

This module is one of the reasons why *Jedi* is not thread-safe. As you can see there are global variables, which are holding the cache information. Some of these variables are being cleaned after every API usage.

Recursions (recursion.py)

Recursions are the recipe of *Jedi* to conquer Python code. However, someone must stop recursions going mad. Some settings are here to make *Jedi* stop at the right time. You can read more about them *here*.

Next to `jedi.evaluate.cache` this module also makes *Jedi* not thread-safe. Why? `execution_recursion_decorator` uses class variables to count the function calls.

Settings

Recursion settings are important if you don't want extremly recursive python code to go absolutely crazy.

The default values are based on experiments while completing the *Jedi* library itself (inception!). But I don't think there's any other Python library that uses recursion in a similarly extreme way. Completion should also be fast and therefore the quality might not always be maximal.

`jedi.evaluate.recursion.`**`recursion_limit = 15`**
: Like `sys.getrecursionlimit()`, just for *Jedi*.

`jedi.evaluate.recursion.`**`total_function_execution_limit = 200`**
: This is a hard limit of how many non-builtin functions can be executed.

`jedi.evaluate.recursion.`**`per_function_execution_limit = 6`**
: The maximal amount of times a specific function may be executed.

`jedi.evaluate.recursion.`**`per_function_recursion_limit = 2`**
: A function may not be executed more than this number of times recursively.

1.7.6 Helper Modules

Most other modules are not really central to how Jedi works. They all contain relevant code, but you if you understand the modules above, you pretty much understand Jedi.

Python 2/3 compatibility (_compatibility.py)

To ensure compatibility from Python `2.7 - 3.x`, a module has been created. Clearly there is huge need to use conforming syntax.

1.8 Jedi Testing

The test suite depends on `tox` and `pytest`:

```
pip install tox pytest
```

To run the tests for all supported Python versions:

```
tox
```

If you want to test only a specific Python version (e.g. Python 2.7), it's as easy as:

```
tox -e py27
```

Tests are also run automatically on Travis CI.

You want to add a test for *Jedi*? Great! We love that. Normally you should write your tests as *Blackbox Tests*. Most tests would fit right in there.

For specific API testing we're using simple unit tests, with a focus on a simple and readable testing structure.

1.8.1 Blackbox Tests (run.py)

Jedi is mostly being tested by what I would call "Blackbox Tests". These tests are just testing the interface and do input/output testing. This makes a lot of sense for *Jedi*. Jedi supports so many different code structures, that it is just stupid to write 200'000 unittests in the manner of `regression.py`. Also, it is impossible to do doctests/unittests on most of the internal data structures. That's why *Jedi* uses mostly these kind of tests.

There are different kind of tests:

- completions / goto_definitions `#?`
- goto_assignments: `#!`
- usages: `#<`

How to run tests?

Jedi uses pytest to run unit and integration tests. To run tests, simply run `pytest`. You can also use tox to run tests for multiple Python versions.

Integration test cases are located in `test/completion` directory and each test case is indicated by either the comment `#?` (completions / definitions), `#!` (assignments), or `#<` (usages). There is also support for third party libraries. In a normal test run they are not being executed, you have to provide a `--thirdparty` option.

In addition to standard *-k* and *-m* options in pytest, you can use the *-T* (*–test-files*) option to specify integration test cases to run. It takes the format of `FILE_NAME[:LINE[,LINE[,...]]]` where `FILE_NAME` is a file in `test/completion` and `LINE` is a line number of the test comment. Here is some recipes:

Run tests only in `basic.py` and `imports.py`:

```
pytest test/test_integration.py -T basic.py -T imports.py
```

Run test at line 4, 6, and 8 in `basic.py`:

```
pytest test/test_integration.py -T basic.py:4,6,8
```

See `pytest --help` for more information.

If you want to debug a test, just use the `--pdb` option.

Alternate Test Runner

If you don't like the output of `pytest`, there's an alternate test runner that you can start by running `./run.py`. The above example could be run by:

```
./run.py basic 4 6 8 50-80
```

The advantage of this runner is simplicity and more customized error reports. Using both runners will help you to have a quicker overview of what's happening.

Auto-Completion

Uses comments to specify a test in the next line. The comment says which results are expected. The comment always begins with *#?*. The last row symbolizes the cursor.

For example:

```
#? ['real']
a = 3; a.rea
```

Because it follows `a.rea` and a is an `int`, which has a `real` property.

Goto Definitions

Definition tests use the same symbols like completion tests. This is possible because the completion tests are defined with a list:

```
#? int()
ab = 3; ab
```

Goto Assignments

Tests look like this:

```
abc = 1
#! ['abc=1']
abc
```

Additionally it is possible to specify the column by adding a number, which describes the position of the test (otherwise it's just the end of line):

```
#! 2 ['abc=1']
abc
```

Usages

Tests look like this:

```
abc = 1
#< abc@1,0 abc@3,0
abc
```

1.8.2 Refactoring Tests (refactor.py)

Refactoring tests work a little bit similar to Black Box tests. But the idea is here to compare two versions of code. **Note: Refactoring is currently not in active development (and was never stable), the tests are therefore not really valuable - just ignore them.**

CHAPTER 2

Resources

- Source Code on Github
- Travis Testing
- Python Package Index

Python Module Index

Index