Jedi Documentation

Release 0.18.0

Jedi contributors

Dec 25, 2020

Contents

Github Repository

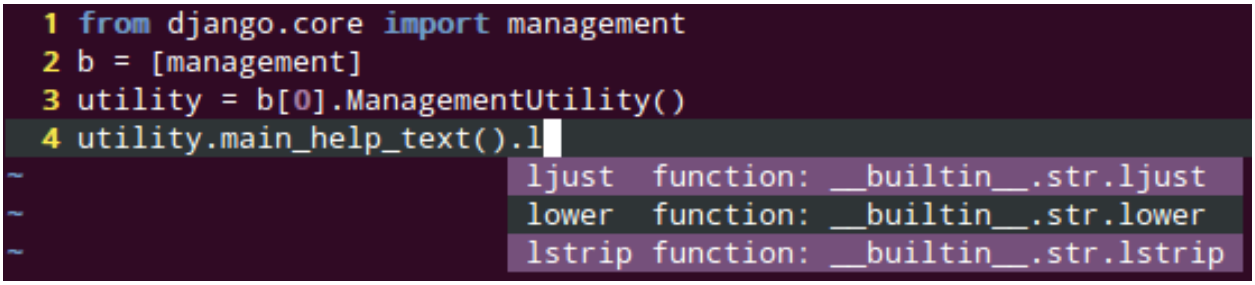
Jedi is a static analysis tool for Python that is typically used in IDEs/editors plugins. Jedi has a focus on autocompletion and goto functionality. Other features include refactoring, code search and finding references.

Jedi has a simple API to work with. There is a reference implementation as a VIM-Plugin. Autocompletion in your REPL is also possible, IPython uses it natively and for the CPython REPL you can install it. Jedi is well tested and bugs should be rare.

Here's a simple example of the autocompletion feature:

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

Autocompletion can for example look like this in jedi-vim:



CHAPTER 1

Docs

1.1 Using Jedi

Jedi is can be used with a variety of *plugins*, *language servers* <*language-servers*> and other software. It is also possible to use Jedi in the *Python shell or with IPython*.

Below you can also find a list of *recipes for type hinting*.

1.1.1 Language Servers

- jedi-language-server
- python-language-server
- anakin-language-server

1.1.2 Editor Plugins

Vim

- jedi-vim
- YouCompleteMe
- deoplete-jedi

Visual Studio Code

- Python Extension

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.

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

xonsh shell

Jedi is a preinstalled extension in xonsh shell. Run the following command to enable:

```
xontrib load jedi
```

and many more!

1.1.3 Tab Completion in the Python Shell

Jedi is a dependency of IPython. Autocompletion in IPython is therefore possible without additional configuration.

Here is an example video how REPL completion can look like in a different shell.

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 3.9.2+ (default, Jul 20 2020, 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/v0.18.0/bin/sphinx-build'>*, *fuzzy=False*)

This function sets up `readline` to use Jedi in a 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
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
```

(continues on next page)

(continued from previous page)

```
        readline.parse_and_bind("tab: complete")
    except ImportError:
        print("Readline is not installed either. No tab completion is enabled.")
else:
    setup_readline()
```

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 Jedi the following code:

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

will complete to `range(10).count`, this does not work with the default cPython `readline` completer.

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

1.1.4 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 docstring/annotation styles below. **Only gradual typing will always work**, all the docstring solutions are glorified hacks and more complicated cases will probably not work.

Official Gradual Typing (Recommended)

You can read a lot about Python's gradual typing system in the corresponding PEPs like:

- PEP 484 as an introduction
- PEP 526 for variable annotations
- PEP 589 for `TypeDict`
- There are probably more :)

Below you can find a few examples how you can use this feature.

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:

```
import typing
x: int = foo()
y: typing.Optional[int] = 3

key: str
value: Employee
for key, value in foo.items():
    pass

f: Union[int, float]
with foo() as f:
    print(f + 3)
```

PEP-0484 should be supported in its entirety. Feel free to open issues if that is not the case. 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.

```

(continues on next page)

(continued from previous page)

```
    ...

    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`.
    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
```

1.2 Features and Limitations

Jedi's main API calls and features are:

- Autocompletion: `Script.complete()`; It's also possible to get it working in *your REPL (IPython, etc.)*
- Goto/Type Inference: `Script.goto()` and `Script.infer()`
- Static Analysis: `Script.get_names()` and `Script.get_syntax_errors()`
- Refactorings: `Script.rename()`, `Script.inline()`, `Script.extract_variable()` and `Script.extract_function()`
- Code Search: `Script.search()` and `Project.search()`

1.2.1 Basic Features

- Python 3.6+ support
- Ignores syntax errors and wrong indentation
- Can deal with complex module / function / class structures
- Great `virtualenv`/`venv` support
- Works great with Python's *type hinting*,
- Understands stub files
- Can infer function arguments for sphinx, epydoc and basic numpydoc docstrings
- Is overall a very solid piece of software that has been refined for a long time. Bug reports are very welcome and are usually fixed within a few weeks.

1.2.2 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
- descriptors: property / staticmethod / classmethod / custom descriptors
- 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
- simple/typical `sys.path` modifications
- `isinstance` checks for if/while/assert
- namespace packages (includes `pkgutil`, `pkg_resources` and PEP420 namespaces)
- Django / Flask / Buildout support
- Understands Pytest fixtures

1.2.3 Limitations

In general Jedi's limit are quite high, but for very big projects or very complex code, sometimes Jedi intentionally stops type inference, to avoid hanging for a long time.

Additionally there are some Python patterns Jedi does not support. This is intentional and below should be a complete list:

- Arbitrary metaclasses: Some metaclasses like enums and dataclasses are reimplemented in Jedi to make them work. Most of the time stubs are good enough to get type inference working, even when metaclasses are involved.
- `setattr()`, `__import__()`
- Writing to some dicts: `globals()`, `locals()`, `object.__dict__`
- Manipulations of instances outside the instance variables without using methods

Performance Issues

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 preload libriaries in Jedi, with *`preload_module()`*. However, once loaded, this should not be a problem anymore. The same is true for huge modules like `PySide`, `wx`, `tensorflow`, `pandas`, etc.

Jedi does not have a very good cache layer. This is probably the biggest and only architectural issue in Jedi. Unfortunately it is not easy to change that. Dave Halter is thinking about rewriting Jedi in Rust, but it has taken Jedi more than 8 years to reach version 1.0, a rewrite will probably also take years.

1.2.4 Security

For `Script`

Security is an important topic for Jedi. By default, no code is executed within Jedi. As long as you write pure Python, everything is inferred statically. If you enable `load_unsafe_extensions=True` for your *`Project`* and you use builtin modules (`c_builtin`) Jedi will execute those modules. If you don't trust a code base, please do not enable that option. It might lead to arbitrary code execution.

For `Interpreter`

If you want security for *`Interpreter`*, `do not` use it. Jedi does execute properties and in general is not very careful to avoid code execution. This is intentional: Most people trust the code bases they have imported, because at that point a malicious code base would have had code execution already.

1.3 API Overview

Note: This documentation is mostly for Plugin developers, who want to improve their editors/IDE with Jedi.

The API consists of a few different parts:

- The main starting points for complete/goto: *`Script`* and *`Interpreter`*. If you work with Jedi you want to understand these classes first.
- *API Result Classes*
- *Python Versions/Virtualenv Support* with functions like *`find_system_environments()`* and *`find_virtualenvs()`*
- A way to work with different *Folders / Projects*
- Helpful functions: *`preload_module()`* and *`set_debug_function()`*

The methods that you are most likely going to use to work with Jedi are the following ones:

`Script.complete`	Completes objects under the cursor.
`Script.goto`	Goes to the name that defined the object under the cursor.
`Script.infer`	Return the definitions of under the cursor.
`Script.help`	Used to display a help window to users.
`Script.get_signatures`	Return the function object of the call under the cursor.

Continued on next page

Table 1 – continued from previous page

Script.get_references	Lists all references of a variable in a project.
Script.get_context	Returns the scope context under the cursor.
Script.get_names	Returns names defined in the current file.
Script.get_syntax_errors	Lists all syntax errors in the current file.
Script.rename	Renames all references of the variable under the cursor.
Script.inline	Inlines a variable under the cursor.
Script.extract_variable	Moves an expression to a new statemenet.
Script.extract_function	Moves an expression to a new function.
Script.search	Searches a name in the current file.
Script.complete_search	Like *Script.search()*, but completes that string.
Project.search	Searches a name in the whole project.
Project.complete_search	Like *Script.search()*, but completes that string.

1.3.1 Script

class jedi.**Script**(*code=None*, *, *path=None*, *environment=None*, *project=None*)

A Script is the base for completions, goto or whatever you want to do with Jedi. The counter part of this class is *Interpreter*, which works with actual dictionaries and can work with a REPL. This class should be used when a user edits code in an editor.

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

The Script's `sys.path` is very customizable:

- If *project* is provided with a `sys_path`, that is going to be used.
- If *environment* is provided, its `sys.path` will be used (see *Environment.get_sys_path*);
- Otherwise `sys.path` will match that of the default environment of Jedi, which typically matches the sys path that was used at the time when Jedi was imported.

Most methods have a `line` and a `column` parameter. Lines in Jedi are always 1-based and columns are always zero based. To avoid repetition they are not always documented. You can omit both line and column. Jedi will then just do whatever action you are calling at the end of the file. If you provide only the line, just will complete at the end of that line.

> **Warning:** By default *jedi.settings.fast_parser* is enabled, which means that parso reuses modules (i.e. they are not immutable). With this setting Jedi is **not thread safe** and it is also not safe to use multiple *Script* instances and its definitions at the same time.
>
> If you are a normal plugin developer this should not be an issue. It is an issue for people that do more complex stuff with Jedi.
>
> This is purely a performance optimization and works pretty well for all typical usages, however consider to turn the setting off if it causes you problems. See also this discussion.

Parameters

- **code** (*str*) – The source code of the current file, separated by newlines.
- **path** (*str or pathlib.Path or None*) – The path of the file in the file system, or `''` if it hasn't been saved yet.
- **environment** (Environment) – Provide a predefined *Environment* to work with a specific Python version or virtualenv.

- **project** (Project) – Provide a *Project* to make sure finding references works well, because the right folder is searched. There are also ways to modify the sys path and other things.

complete(*line=None*, *column=None*, *, *fuzzy=False*)

Completes objects under the cursor.

Those objects contain information about the completions, more than just names.

Parameters **fuzzy** – Default False. Will return fuzzy completions, which means that e.g. `ooa` will match `foobar`.

Returns Completion objects, sorted by name. Normal names appear before "private" names that start with _ and those appear before magic methods and name mangled names that start with __.

Return type list of *Completion*

infer(*line=None*, *column=None*, *, *only_stubs=False*, *prefer_stubs=False*)

Return the definitions of under the cursor. It is basically a wrapper around Jedi's type inference.

This method follows complicated paths and returns the end, not the first definition. The big difference between *goto()* and *infer()* is that *goto()* doesn't follow imports and statements. Multiple objects may be returned, because depending on an option you can have two different versions of a function.

Parameters

- **only_stubs** – Only return stubs for this method.
- **prefer_stubs** – Prefer stubs to Python objects for this method.

Return type list of *Name*

goto(*line=None*, *column=None*, *, *follow_imports=False*, *follow_builtin_imports=False*, *only_stubs=False*, *prefer_stubs=False*)

Goes to the name that defined the object under the cursor. Optionally you can follow imports. Multiple objects may be returned, depending on an if you can have two different versions of a function.

Parameters

- **follow_imports** – The method will follow imports.
- **follow_builtin_imports** – If `follow_imports` is True will try to look up names in builtins (i.e. compiled or extension modules).
- **only_stubs** – Only return stubs for this method.
- **prefer_stubs** – Prefer stubs to Python objects for this method.

Return type list of *Name*

search(*string*, *, *all_scopes=False*)

Searches a name in the current file. For a description of how the search string should look like, please have a look at *Project.search()*.

Parameters **all_scopes** (*bool*) – Default False; searches not only for definitions on the top level of a module level, but also in functions and classes.

Yields *Name*

complete_search(*string*, **kwargs*)

Like *Script.search()*, but completes that string. If you want to have all possible definitions in a file you can also provide an empty string.

Parameters

- **all_scopes** (*bool*) – Default False; searches not only for definitions on the top level of a module level, but also in functions and classes.
- **fuzzy** – Default False. Will return fuzzy completions, which means that e.g. `ooa` will match `foobar`.

Yields *Completion*

help(*line=None*, *column=None*)

Used to display a help window to users. Uses *Script.goto()* and returns additional definitions for keywords and operators.

Typically you will want to display *BaseName.docstring()* to the user for all the returned definitions.

The additional definitions are `Name(...).type == 'keyword'`. These definitions do not have a lot of value apart from their docstring attribute, which contains the output of Python's *help()* function.

Return type list of *Name*

get_references(*line=None*, *column=None*, ***kwargs*)

Lists all references of a variable in a project. Since this can be quite hard to do for Jedi, if it is too complicated, Jedi will stop searching.

Parameters

- **include_builtins** – Default `True`. If `False`, checks if a reference is a builtin (e.g. `sys`) and in that case does not return it.
- **scope** – Default `'project'`. If `'file'`, include references in the current module only.

Return type list of *Name*

get_signatures(*line=None*, *column=None*)

Return the function object of the call under the cursor.

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 *Signature*

get_context(*line=None*, *column=None*)

Returns the scope context under the cursor. This basically means the function, class or module where the cursor is at.

Return type *Name*

get_names(***kwargs*)

Returns names defined in the current file.

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`.

Return type list of *Name*

get_syntax_errors()
Lists all syntax errors in the current file.

Return type list of *SyntaxError*

rename(*line=None*, *column=None*, *, *new_name*)
Renames all references of the variable under the cursor.

Parameters **new_name** – The variable under the cursor will be renamed to this string.

Raises *RefactoringError*

Return type *Refactoring*

extract_variable(*line*, *column*, *, *new_name*, *until_line=None*, *until_column=None*)
Moves an expression to a new statemenet.

For example if you have the cursor on `foo` and provide a `new_name` called `bar`:

```
foo = 3.1
x = int(foo + 1)
```

the code above will become:

```
foo = 3.1
bar = foo + 1
x = int(bar)
```

Parameters

- **new_name** – The expression under the cursor will be renamed to this string.
- **until_line** (*int*) – The the selection range ends at this line, when omitted, Jedi will be clever and try to define the range itself.
- **until_column** (*int*) – The the selection range ends at this column, when omitted, Jedi will be clever and try to define the range itself.

Raises *RefactoringError*

Return type *Refactoring*

extract_function(*line*, *column*, *, *new_name*, *until_line=None*, *until_column=None*)
Moves an expression to a new function.

For example if you have the cursor on `foo` and provide a `new_name` called `bar`:

```
global_var = 3

def x():
    foo = 3.1
    x = int(foo + 1 + global_var)
```

the code above will become:

```
global_var = 3

def bar(foo):
    return int(foo + 1 + global_var)

def x():
    foo = 3.1
    x = bar(foo)
```

Parameters

- **new_name** – The expression under the cursor will be replaced with a function with this name.
- **until_line** (*int*) – The the selection range ends at this line, when omitted, Jedi will be clever and try to define the range itself.
- **until_column** (*int*) – The the selection range ends at this column, when omitted, Jedi will be clever and try to define the range itself.

Raises *RefactoringError*

Return type *Refactoring*

inline(*line=None*, *column=None*)

Inlines a variable under the cursor. This is basically the opposite of extracting a variable. For example with the cursor on bar:

```
foo = 3.1
bar = foo + 1
x = int(bar)
```

the code above will become:

```
foo = 3.1
x = int(foo + 1)
```

Raises *RefactoringError*

Return type *Refactoring*

1.3.2 Interpreter

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

Jedi's API for Python REPLs.

Implements all of the methods that are present in *Script* as well.

In addition to completions that normal REPL completion does like `str.upper`, Jedi also supports code completion based on static code analysis. For example Jedi will complete `str().upper`.

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

All keyword arguments are same as the arguments for *Script*.

Parameters

- **code** (*str*) – Code to parse.
- **namespaces** (*typing.List[dict]*) – A list of namespace dictionaries such as the one returned by `globals()` and `locals()`.

1.3.3 Projects

Projects are a way to handle Python projects within Jedi. For simpler plugins you might not want to deal with projects, but if you want to give the user more flexibility to define sys paths and Python interpreters for a project, *Project* is the perfect way to allow for that.

Projects can be saved to disk and loaded again, to allow project definitions to be used across repositories.

jedi.**get_default_project**(*path=None*)

If a project is not defined by the user, Jedi tries to define a project by itself as well as possible. Jedi traverses folders until it finds one of the following:

1. A `.jedi/config.json`
2. One of the following files: `setup.py`, `.git`, `.hg`, `requirements.txt` and `MANIFEST.in`.

class jedi.**Project**(*path*, ***kwargs*)

Projects are a simple way to manage Python folders and define how Jedi does import resolution. It is mostly used as a parameter to *Script*. Additionally there are functions to search a whole project.

Parameters

- **path** – The base path for this project.
- **environment_path** – The Python executable path, typically the path of a virtual environment.
- **load_unsafe_extensions** – Default False, Loads extensions that are not in the sys path and in the local directories. With this option enabled, this is potentially unsafe if you clone a git repository and analyze it's code, because those compiled extensions will be important and therefore have execution privileges.
- **sys_path** – list of str. You can override the sys path if you want. By default the `sys.path.` is generated by the environment (virtualenvs, etc).
- **added_sys_path** – list of str. Adds these paths at the end of the sys path.
- **smart_sys_path** – If this is enabled (default), adds paths from local directories. Otherwise you will have to rely on your packages being properly configured on the `sys.path`.

classmethod load(*path*)

Loads a project from a specific path. You should not provide the path to `.jedi/project.json`, but rather the path to the project folder.

Parameters **path** – The path of the directory you want to use as a project.

save()

Saves the project configuration in the project in `.jedi/project.json`.

path

The base path for this project.

sys_path

The sys path provided to this project. This can be None and in that case will be auto generated.

smart_sys_path
If the sys path is going to be calculated in a smart way, where additional paths are added.

load_unsafe_extensions
Wheter the project loads unsafe extensions.

search(*string*, *, *all_scopes=False*)
Searches a name in the whole project. If the project is very big, at some point Jedi will stop searching. However it's also very much recommended to not exhaust the generator. Just display the first ten results to the user.

There are currently three different search patterns:

- `foo` to search for a definition foo in any file or a file called `foo.py` or `foo.pyi`.
- `foo.bar` to search for the `foo` and then an attribute `bar` in it.
- `class foo.bar.Bar` or `def foo.bar.baz` to search for a specific API type.

Parameters **all_scopes** (*bool*) – Default False; searches not only for definitions on the top level of a module level, but also in functions and classes.

Yields *Name*

complete_search(*string*, ***kwargs*)
Like *Script.search()*, but completes that string. An empty string lists all definitions in a project, so be careful with that.

Parameters **all_scopes** (*bool*) – Default False; searches not only for definitions on the top level of a module level, but also in functions and classes.

Yields *Completion*

1.3.4 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**(*, *env_vars=None*)
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*, *, *safe=True*, *use_environment_vars=True*)

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.
- **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.
- **use_environment_vars** – Default True. If True, the VIRTUAL_ENV variable will be checked if it contains a valid VirtualEnv. CONDA_PREFIX will be checked to see if it contains a valid conda environment.

Yields *Environment*

jedi.**get_system_environment**(*version*, *, *env_vars=None*)
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*, *env_vars=None*)
Make it possible to manually create an Environment object by specifying a Virtualenv path or an executable path and optional environment variables.

Raises *InvalidPythonEnvironment*

Returns *Environment*

jedi.**get_default_environment**()
Tries to return an active Virtualenv or conda environment. If there is no VIRTUAL_ENV variable or no CONDA_PREFIX variable set 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*, *env_vars=None*)
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()
The sys path for this environment. Does not include potential modifications from e.g. appending to `sys.path`.

Returns list of str

1.3.5 Helper Functions

jedi.**preload_module**(**modules*)
Preloading modules tells Jedi to load a module now, instead of lazy parsing of modules. This can be useful 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.

1.3.6 Errors

exception jedi.**InternalError**
This error might happen a subprocess is crashing. The reason for this is usually broken C code in third party

libraries. This is not a very common thing and it is safe to use Jedi again. However using the same calls might result in the same error again.

exception jedi.**RefactoringError**

Refactorings can fail for various reasons. So if you work with refactorings like *Script.rename()*, *Script.inline()*, *Script.extract_variable()* and *Script.extract_function()*, make sure to catch these. The descriptions in the errors are ususally valuable for end users.

A typical `RefactoringError` would tell the user that inlining is not possible if no name is under the cursor.

1.3.7 Examples

Completions

```
>>> import jedi
>>> code = '''import json; json.l'''
>>> script = jedi.Script(code, path='example.py')
>>> script
<Script: 'example.py' <SameEnvironment: 3.9.0 in /usr>>
>>> completions = script.complete(1, 19)
>>> completions
[<Completion: load>, <Completion: loads>]
>>> completions[1]
<Completion: loads>
>>> completions[1].complete
'oads'
>>> completions[1].name
'loads'
```

Type Inference / Goto

```
>>> import jedi
>>> code = '''\
... def my_func():
...     print 'called'
...
... alias = my_func
... my_list = [1, None, alias]
... inception = my_list[2]
...
... inception()'''
>>> script = jedi.Script(code)
>>>
>>> script.goto(8, 1)
[<Name full_name='__main__.inception', description='inception = my_list[2]'>]
>>>
>>> script.infer(8, 1)
[<Name full_name='__main__.my_func', description='def my_func'>]
```

References

```
>>> import jedi
>>> code = '''\
... x = 3
... if 1 == 2:
...     x = 4
... else:
...     del x'''
>>> script = jedi.Script(code)
>>> rns = script.get_references(5, 8)
>>> rns
[<Name full_name='__main__.x', description='x = 3'>,
 <Name full_name='__main__.x', description='x = 4'>,
 <Name full_name='__main__.x', description='del x'>]
>>> rns[1].line
3
>>> rns[1].column
4
```

1.3.8 Deprecations

The deprecation process is as follows:

1. A deprecation is announced in any release.
2. The next major release removes the deprecated functionality.

1.4 API Return Classes

1.4.1 Abstract Base Class

class jedi.api.classes.**BaseName**(*inference_state*, *name*)

Bases: object

The base class for all definitions, completions and signatures.

module_path

Shows the file path of a module. e.g. `/usr/lib/python3.9/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.infer()* should return a list of definition for `sys`, `f`, `C` and `x`.

```
>>> from jedi import Script
>>> source = '''
... import keyword
...
```

(continues on next page)

(continued from previous page)

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

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

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)
>>> print(defs)  # doctest: +NORMALIZE_WHITESPACE
[<Name full_name='keyword', description='module keyword'>,
 <Name full_name='__main__.C', description='class C'>,
 <Name full_name='__main__.D', description='instance D'>,
 <Name full_name='__main__.f', description='def f'>]
```

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

```
>>> defs = [d.type for d in defs]
>>> defs[0]
'module'
>>> defs[1]
'class'
>>> defs[2]
'instance'
>>> defs[3]
'function'
```

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

module_name

The module name, a bit similar to what `__name__` is in a random Python module.

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

in_builtin_module()

Returns True, if 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).

get_definition_start_position()
The (row, column) of the start of the definition range. Rows start with 1, columns start with 0.

Return type Optional[Tuple[int, int]]

get_definition_end_position()
The (row, column) of the end of the definition range. Rows start with 1, columns start with 0.

Return type Optional[Tuple[int, int]]

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, path='example.py')
>>> doc = script.infer(1, len('def f'))[0].docstring()
>>> print(doc)
f(a, b=1)
<BLANKLINE>
Document for function f.
```

Notice that useful extra information is added to the actual docstring, e.g. function signatures are prepended to their docstrings. If you need the actual docstring, use `raw=True` instead.

```
>>> print(script.infer(1, len('def f'))[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 description of the *Name* object, which is heavily used in testing. e.g. for `isinstance` it returns `def isinstance`.

Example:

```
>>> from jedi import Script
>>> source = '''
... def f():
...     pass
...
... class C:
...     pass
...
... variable = f if random.choice([0,1]) else C'''
>>> script = Script(source)  # line is maximum by default
>>> defs = script.infer(column=3)
>>> defs = sorted(defs, key=lambda d: d.line)
```

(continues on next page)

(continued from previous page)

```
>>> print(defs)  # doctest: +NORMALIZE_WHITESPACE
[<Name full_name='__main__.f', description='def f'>,
 <Name full_name='__main__.C', description='class C'>]
>>> str(defs[0].description)
'def f'
>>> str(defs[1].description)
'class C'
```

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, path='example.py')
>>> print(script.infer(3, len('os.path.join'))[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()
Returns True if the current name is defined in a stub file.

is_side_effect()
Checks if a name is defined as `self.foo = 3`. In case of self, this function would return False, for foo it would return True.

goto(*, *follow_imports=False*, *follow_builtin_imports=False*, *only_stubs=False*, *prefer_stubs=False*)
Like *Script.goto()* (also supports the same params), but does it for the current name. This is typically useful if you are using something like *Script.get_names()*.

Parameters

- **follow_imports** – The goto call will follow imports.
- **follow_builtin_imports** – If follow_imports is True will try to look up names in builtins (i.e. compiled or extension modules).
- **only_stubs** – Only return stubs for this goto call.
- **prefer_stubs** – Prefer stubs to Python objects for this goto call.

Return type list of *Name*

infer(*, *only_stubs=False*, *prefer_stubs=False*)
Like *Script.infer()*, it can be useful to understand which type the current name has.

Return the actual 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 very, very slow.

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 *Name*

parent()
Returns the parent scope of this identifier.

Return type *Name*

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()
Returns all potential signatures for a function or a class. Multiple signatures are typical if you use Python stubs with `@overload`.

Return type list of *BaseSignature*

execute()
Uses type inference to "execute" this identifier and returns the executed objects.

Return type list of *Name*

get_type_hint()
Returns type hints like `Iterable[int]` or `Union[int, str]`.

This method might be quite slow, especially for functions. The problem is finding executions for those functions to return something like `Callable[[int, str], str]`.

Return type str

1.4.2 Name

class jedi.api.classes.**Name**(*inference_state*, *definition*)
Bases: *jedi.api.classes.BaseName*

Name objects are returned from many different APIs including *Script.goto()* or *Script.infer()*.

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

Return type list of *Name*

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.

1.4.3 Completion

class jedi.api.classes.**Completion**(*inference_state*, *name*, *stack*, *like_name_length*, *is_fuzzy*, *cached_name=None*)
Bases: *jedi.api.classes.BaseName*

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

complete
Only works with non-fuzzy completions. Returns None if fuzzy completions are used.

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*)
Documented under *`BaseName.docstring()`*.

type
Documented under *`BaseName.type()`*.

get_completion_prefix_length()
Returns the length of the prefix being completed. For example, completing `isinstance`:

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

would return 8, because len('isinstan') == 8.

Assuming the following function definition:

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

completing `foo(par` would return 3.

1.4.4 BaseSignature

class jedi.api.classes.**BaseSignature**(*inference_state*, *signature*)
Bases: *`jedi.api.classes.Name`*

These signatures are returned by *`BaseName.get_signatures()`* calls.

params
Returns definitions for all parameters that a signature defines. This includes stuff like `*args` and `**kwargs`.

Return type list of *`ParamName`*

to_string()
Returns a text representation of the signature. This could for example look like `foo(bar, baz: int, **kwargs)`.

Return type str

1.4.5 Signature

class jedi.api.classes.**Signature**(*inference_state*, *signature*, *call_details*)
Bases: *jedi.api.classes.BaseSignature*

A full signature object is the return value of *Script.get_signatures()*.

index
Returns the param index of the current cursor position. Returns None if the index cannot be found in the curent call.

Return type int

bracket_start
Returns a line/column tuple of the bracket that is responsible for the last function call. The first line is 1 and the first column 0.

Return type int, int

1.4.6 ParamName

class jedi.api.classes.**ParamName**(*inference_state*, *definition*)
Bases: *jedi.api.classes.Name*

infer_default()
Returns default values like the `1` of `def foo(x=1):`.

Return type list of *Name*

infer_annotation(***kwargs*)

Parameters **execute_annotation** – Default True; If False, values are not executed and classes are returned instead of instances.

Return type list of *Name*

to_string()
Returns a simple representation of a param, like `f: Callable[..., Any]`.

Return type str

kind
Returns an enum instance of `inspect`'s `Parameter` enum.

Return type `inspect.Parameter.kind`

1.4.7 Refactoring

class jedi.api.refactoring.**Refactoring**(*inference_state*, *file_to_node_changes*, *renames=()*)
Bases: `object`

get_renames() → Iterable[Tuple[pathlib.Path, pathlib.Path]]
Files can be renamed in a refactoring.

apply()
Applies the whole refactoring to the files, which includes renames.

class jedi.api.errors.**SyntaxError**(*parso_error*)
Bases: object

Syntax errors are generated by *Script.get_syntax_errors()*.

line
The line where the error starts (starting with 1).

column
The column where the error starts (starting with 0).

until_line
The line where the error ends (starting with 1).

until_column
The column where the error ends (starting with 0).

1.5 Installation and Configuration

Warning: Most people will want to install Jedi as a submodule/vendored and not through pip/system wide. The reason for this is that it makes sense that the plugin that uses Jedi has always access to it. Otherwise Jedi will not work properly when virtualenvs are activated. So please read the documentation of your editor/IDE plugin to install Jedi.

For plugin developers, Jedi works best if it is always available. Vendoring is a pretty good option for that.

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.

1.5.1 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.

1.5.2 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
```

1.5.3 System-wide installation via a package manager

Arch Linux

You can install Jedi directly from official Arch Linux packages:

- python-jedi

(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.

1.5.4 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
```

1.5.5 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.

1.6 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**
: Completions are by default case insensitive.

jedi.settings.**add_bracket_after_function = False**
: Adds an opening bracket after a function for completions.

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 `%LOCALAPPDATA%\Jedi\Jedi\`. On Linux, if the environment variable `$XDG_CACHE_HOME` is set, `$XDG_CACHE_HOME/jedi` is used instead of the default one.

1.6.3 Parser

`jedi.settings.`**`fast_parser = True`**
Uses Parso's diff parser. If it is enabled, this might cause issues, please read the warning on *Script*. This feature makes it possible to only parse the parts again that have changed, while reusing the rest of the syntax tree.

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.`**`auto_import_modules = ['gi']`**
Modules that will not be analyzed but imported, if 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.

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

- *The Jedi Core*
- *Core Extensions*
- *Imports & Modules*
- *Stubs & Annotations*
- *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 type inference*
- *API*

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

Parser

Jedi used to have its 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 parser uses is very similar to the official Python grammar files.

Type inference of python code (inference/__init__.py)

Type inference 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 I call lazy type inference. That said, the typical entry point for static analysis is calling `infer_expr_stmt`. There's separate logic for autocompletion in the API, the inference_state is all about inferring 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 `infer_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 `infer_expr_stmt` will return the `date` class.

To *visualize* this (simplified):

- `InferenceState.infer_expr_stmt` doesn't do much, because there's no assignment.
- `Context.infer_node` cares for resolving the dotted path
- `InferenceState.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 `infer_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 `infer_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.

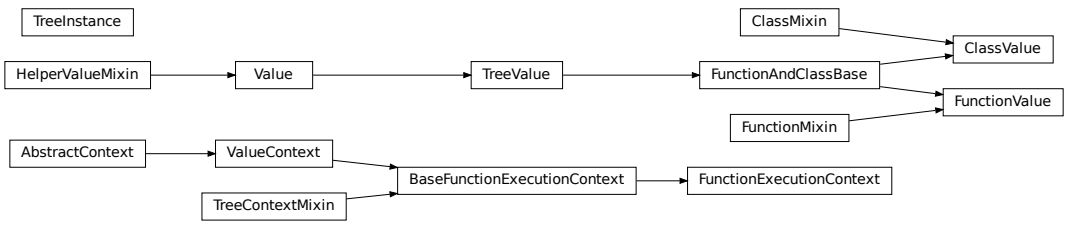
Jedi has been tested very well, so you can just start modifying code. It's best to write your own test first for your "new" feature. Don't be scared of breaking stuff. As long as the tests pass, you're most likely to be fine.

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

Inference Values (inference/base_value.py)

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

A ValueSet 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 (inference/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 (inference/value/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.

Parameter completion (inference/dynamic_params.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 (inference/docstrings.py)

Docstrings are another source of information for functions and classes. *jedi.inference.dynamic_params* 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 (api/refactoring.py)

1.7.4 Imports & Modules

- Modules
- *Builtin Modules*
- *Imports*

Compiled Modules (inference/compiled.py)

Imports (inference/imports.py)

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

Stubs & Annotations (inference/gradual)

It is unfortunately not well documented how stubs and annotations work in Jedi. If somebody needs an introduction, please let me know.

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 the internal `jedi.inference.cache` this module also makes Jedi not thread-safe, because `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.inference.recursion.`**`recursion_limit = 15`**
Like `sys.getrecursionlimit()`, just for Jedi.

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

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

`jedi.inference.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.

1.8 Jedi Testing

The test suite depends on `pytest`:

```
pip install pytest
```

If you want to test only a specific Python version (e.g. Python 3.8), it is as easy as:

```
python3.8 -m pytest
```

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 Integration Tests (run.py)

1.8.2 Refactoring Tests (refactor.py)

1.9 History & Acknowledgements

1.9.1 Acknowledgements

- Dave Halter for creating and maintaining Jedi & Parso.
- Takafumi Arakaki (@tkf) for creating a solid test environment and a lot of other things.
- Danilo Bargen (@dbrgn) for general housekeeping and being a good friend :).
- Guido van Rossum (@gvanrossum) for creating the parser generator pgen2 (originally used in lib2to3).
- Thanks to all the *contributors*.

1.9.2 A Little Bit of History

Written by Dave.

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 has not much to do with Star Wars. It's part of my second name Jedidjah.

I actually started Jedi back in 2012, because there were no good solutions available for VIM. Most auto-completion solutions just did not work well. The only good solution was PyCharm. But I liked my good old VIM very much. There was also a solution called Rope that did not work at all for me. So I decided to write my own version of a completion engine.

The first idea was to execute non-dangerous code. But I soon realized, that this would not work. So I started to build a static analysis tool. The biggest problem that I had at the time was that I did not know a thing about parsers.I did not did not even know the word static analysis. It turns out they are the foundation of a good static analysis tool. I of course did not know that and tried to write my own poor version of a parser that I ended up throwing away two years later.

Because of my lack of knowledge, everything after 2012 and before 2020 was basically refactoring. I rewrote the core parts of Jedi probably like 5-10 times. The last big rewrite (that I did twice) was the inclusion of gradual typing and stubs.

I learned during that time that it is crucial to have a good understanding of your problem. Otherwise you just end up doing it again. I only wrote features in the beginning and in the end. Everything else was bugfixing and refactoring. However now I am really happy with the result. It works well, bugfixes can be quick and is pretty much feature complete.

I will leave you with a small annectote that happend in 2012, if I remember correctly. After I explained Guido van Rossum, how some parts of my auto-completion work, he said:

"Oh, that worries me…"

Now that it is finished, I hope he likes it :-).

1.9.3 Main Authors

- David Halter (@davidhalter) <davidhalter88@gmail.com>
- Takafumi Arakaki (@tkf) <aka.tkf@gmail.com>

1.9.4 Code Contributors

- Danilo Bargen (@dbrgn) <mail@dbrgn.ch>
- Laurens Van Houtven (@lvh) <_@lvh.cc>
- Aldo Stracquadanio (@Astrac) <aldo.strac@gmail.com>
- Jean-Louis Fuchs (@ganwell) <ganwell@fangorn.ch>
- tek (@tek)
- Yasha Borevich (@jjay) <j.borevich@gmail.com>
- Aaron Griffin <aaronmgriffin@gmail.com>
- andviro (@andviro)
- Mike Gilbert (@floppym) <floppym@gentoo.org>
- Aaron Meurer (@asmeurer) <asmeurer@gmail.com>
- Lubos Trilety <ltrilety@redhat.com>
- Akinori Hattori (@hattya) <hattya@gmail.com>
- srusskih (@srusskih)
- Steven Silvester (@blink1073)
- Colin Duquesnoy (@ColinDuquesnoy) <colin.duquesnoy@gmail.com>
- Jorgen Schaefer (@jorgenschaefer) <contact@jorgenschaefer.de>
- Fredrik Bergroth (@fbergroth)
- Mathias Fußenegger (@mfussenegger)
- Syohei Yoshida (@syohex) <syohex@gmail.com>
- ppalucky (@ppalucky)
- immerrr (@immerrr) immerrr@gmail.com
- Albertas Agejevas (@alga)
- Savor d'Isavano (@KenetJervet) <newelevenken@163.com>
- Phillip Berndt (@phillipberndt) <phillip.berndt@gmail.com>
- Ian Lee (@IanLee1521) <IanLee1521@gmail.com>
- Farkhad Khatamov (@hatamov) <comsgn@gmail.com>
- Kevin Kelley (@kelleyk) <kelleyk@kelleyk.net>
- Sid Shanker (@squidarth) <sid.p.shanker@gmail.com>

- Reinoud Elhorst (@reinhrst)
- Guido van Rossum (@gvanrossum) <guido@python.org>
- Dmytro Sadovnychyi (@sadovnychyi) <jedi@dmit.ro>
- Cristi Burcă (@scribu)
- bstaint (@bstaint)
- Mathias Rav (@Mortal) <rav@cs.au.dk>
- Daniel Fiterman (@dfit99) <fitermandaniel2@gmail.com>
- Simon Ruggier (@sruggier)
- Élie Gouzien (@ElieGouzien)
- Robin Roth (@robinro)
- Malte Plath (@langsamer)
- Anton Zub (@zabulazza)
- Maksim Novikov (@m-novikov) <mnovikov.work@gmail.com>
- Tobias Rzepka (@TobiasRzepka)
- micbou (@micbou)
- Dima Gerasimov (@karlicoss) <karlicoss@gmail.com>
- Max Woerner Chase (@mwchase) <max.chase@gmail.com>
- Johannes Maria Frank (@jmfrank63) <jmfrank63@gmail.com>
- Shane Steinert-Threlkeld (@shanest) <ssshanest@gmail.com>
- Tim Gates (@timgates42) <tim.gates@iress.com>
- Lior Goldberg (@goldberglior)
- Ryan Clary (@mrclary)
- Max Mäusezahl (@mmaeusezahl) <maxmaeusezahl@googlemail.com>
- Vladislav Serebrennikov (@endilll)
- Andrii Kolomoiets (@muffinmad)
- Leo Ryu (@Leo-Ryu)

And a few more "anonymous" contributors.

Note: (@user) means a github user name.

1.10 Changelog

1.10.1 Unreleased

1.10.2 0.18.0 (2020-12-25)

- Dropped Python 2 and Python 3.5

- Using `pathlib.Path()` as an output instead of `str` in most places: - `Project.path` - `Script.path` - `Definition.module_path` - `Refactoring.get_renames` - `Refactoring.get_changed_files`
- Functions with `@property` now return `property` instead of `function` in `Name().type`
- Started using annotations
- Better support for the walrus operator
- Project attributes are now read accessible
- Removed all deprecations

This is likely going to be the last minor release before 1.0.

1.10.3 0.17.2 (2020-07-17)

- Added an option to pass environment variables to `Environment`
- `Project(...).path` exists now
- Support for Python 3.9
- A few bugfixes

This will be the last release that supports Python 2 and Python 3.5. `0.18.0` will be Python 3.6+.

1.10.4 0.17.1 (2020-06-20)

- Django `Model` meta class support
- Django Manager support (completion on Managers/QuerySets)
- Added Django Stubs to Jedi, thanks to all contributors of the Django Stubs project
- Added `SyntaxError.get_message`
- Python 3.9 support
- Bugfixes (mostly towards Generics)

1.10.5 0.17.0 (2020-04-14)

- Added `Project` support. This allows a user to specify which folders Jedi should work with.
- Added support for Refactoring. The following refactorings have been implemented: `Script.rename`, `Script.inline`, `Script.extract_variable` and `Script.extract_function`.
- Added `Script.get_syntax_errors` to display syntax errors in the current script.
- Added code search capabilities both for individual files and projects. The new functions are `Project.search`, `Project.complete_search`, `Script.search` and `Script.complete_search`.
- Added `Script.help` to make it easier to display a help window to people. Now returns pydoc information as well for Python keywords/operators. This means that on the class keyword it will now return the docstring of Python's builtin function `help('class')`.
- The API documentation is now way more readable and complete. Check it out under https://jedi.readthedocs.io. A lot of it has been rewritten.
- Removed Python 3.4 support

- Many bugfixes

This is likely going to be the last minor version that supports Python 2 and Python3.5. Bugfixes will be provided in 0.17.1+. The next minor/major version will probably be Jedi 1.0.0.

1.10.6 0.16.0 (2020-01-26)

- **Added** `Script.get_context` to get information where you currently are.
- Completions/type inference of **Pytest fixtures**.
- Tensorflow, Numpy and Pandas completions should now be about **4-10x faster** after the first time they are used.
- Dict key completions are working now. e.g. `d = {1000: 3}; d[10` will expand to `1000`.
- Completion for "proxies" works now. These are classes that have a `__getattr__(self, name)` method that does a `return getattr(x, name).` after loading them initially.
- Goto on a function/attribute in a class now goes to the definition in its super class.
- **Big Script API Changes:**
  - The line and column parameters of `jedi.Script` are now deprecated
  - `completions` deprecated, use `complete` instead
  - `goto_assignments` deprecated, use `goto` instead
  - `goto_definitions` deprecated, use `infer` instead
  - `call_signatures` deprecated, use `get_signatures` instead
  - `usages` deprecated, use `get_references` instead
  - `jedi.names` deprecated, use `jedi.Script(...).get_names()`
- `BaseName.goto_assignments` renamed to `BaseName.goto`
- Add follow_imports to `Name.goto`. Now its signature matches `Script.goto`.
- **Python 2 support deprecated**. For this release it is best effort. Python 2 has reached the end of its life and now it's just about a smooth transition. Bugs for Python 2 will not be fixed anymore and a third of the tests are already skipped.
- Removed `settings.no_completion_duplicates`. It wasn't tested and nobody was probably using it anyway.
- Removed `settings.use_filesystem_cache` and `settings.additional_dynamic_modules`, they have no usage anymore. Pretty much nobody was probably using them.

1.10.7 0.15.2 (2019-12-20)

- Signatures are now detected a lot better
- Add fuzzy completions with `Script(...).completions(fuzzy=True)`
- Files bigger than one MB (about 20kLOC) get cropped to avoid getting stuck completely.
- Many small Bugfixes
- A big refactoring around contexts/values

1.10.8 0.15.1 (2019-08-13)

- Small bugfix and removal of a print statement

1.10.9 0.15.0 (2019-08-11)

- Added file path completions, there's a **new** `Completion.type` now: `path`. Example: `'/ho` -> `'/home/`
- `*args`/`**kwargs` resolving. If possible Jedi replaces the parameters with the actual alternatives.
- Better support for enums/dataclasses
- When using Interpreter, properties are now executed, since a lot of people have complained about this. Discussion in #1299, #1347.

New APIs:

- `Name.get_signatures() -> List[Signature]`. Signatures are similar to `CallSignature`. `Name.params` is therefore deprecated.
- `Signature.to_string()` to format signatures.
- `Signature.params -> List[ParamName]`, ParamName has the following additional attributes `infer_default()`, `infer_annotation()`, `to_string()`, and `kind`.
- `Name.execute() -> List[Name]`, makes it possible to infer return values of functions.

1.10.10 0.14.1 (2019-07-13)

- CallSignature.index should now be working a lot better
- A couple of smaller bugfixes

1.10.11 0.14.0 (2019-06-20)

- Added `goto_*(prefer_stubs=True)` as well as `goto_*(prefer_stubs=True)`
- Stubs are used now for type inference
- Typeshed is used for better type inference
- Reworked Name.full_name, should have more correct return values

1.10.12 0.13.3 (2019-02-24)

- Fixed an issue with embedded Python, see https://github.com/davidhalter/jedi-vim/issues/870

1.10.13 0.13.2 (2018-12-15)

- Fixed a bug that led to Jedi spawning a lot of subprocesses.

1.10.14 0.13.1 (2018-10-02)

- Bugfixes, because tensorflow completions were still slow.

1.10.15 0.13.0 (2018-10-02)

- A small release. Some bug fixes.
- Remove Python 3.3 support. Python 3.3 support has been dropped by the Python foundation.
- Default environments are now using the same Python version as the Python process. In 0.12.x, we used to load the latest Python version on the system.
- Added `include_builtins` as a parameter to usages.
- `goto_assignments` has a new `follow_builtin_imports` parameter that changes the previous behavior slightly.

1.10.16 0.12.1 (2018-06-30)

- This release forces you to upgrade parso. If you don't, nothing will work anymore. Otherwise changes should be limited to bug fixes. Unfortunately Jedi still uses a few internals of parso that make it hard to keep compatibility over multiple releases. Parso >=0.3.0 is going to be needed.

1.10.17 0.12.0 (2018-04-15)

- Virtualenv/Environment support
- F-String Completion/Goto Support
- Cannot crash with segfaults anymore
- Cleaned up import logic
- Understand async/await and autocomplete it (including async generators)
- Better namespace completions
- Passing tests for Windows (including CI for Windows)
- Remove Python 2.6 support

1.10.18 0.11.1 (2017-12-14)

- Parso update - the caching layer was broken
- Better usages - a lot of internal code was ripped out and improved.

1.10.19 0.11.0 (2017-09-20)

- Split Jedi's parser into a separate project called `parso`.
- Avoiding side effects in REPL completion.
- Numpy docstring support should be much better.
- Moved the *settings.*recursion** away, they are no longer usable.

1.10.20 0.10.2 (2017-04-05)

- Python Packaging sucks. Some files were not included in 0.10.1.

1.10.21 0.10.1 (2017-04-05)

- Fixed a few very annoying bugs.
- Prepared the parser to be factored out of Jedi.

1.10.22 0.10.0 (2017-02-03)

- Actual semantic completions for the complete Python syntax.
- Basic type inference for `yield from` PEP 380.
- PEP 484 support (most of the important features of it). Thanks Claude! (@reinhrst)
- Added `get_line_code` to `Name` and `Completion` objects.
- Completely rewritten the type inference engine.
- A new and better parser for (fast) parsing diffs of Python code.

1.10.23 0.9.0 (2015-04-10)

- The import logic has been rewritten to look more like Python's. There is now an `InferState.modules` import cache, which resembles `sys.modules`.
- Integrated the parser of 2to3. This will make refactoring possible. It will also be possible to check for error messages (like compiling an AST would give) in the future.
- With the new parser, the type inference also completely changed. It's now simpler and more readable.
- Completely rewritten REPL completion.
- Added `jedi.names`, a command to do static analysis. Thanks to that sourcegraph guys for sponsoring this!
- Alpha version of the linter.

1.10.24 0.8.1 (2014-07-23)

- Bugfix release, the last release forgot to include files that improve autocompletion for builtin libraries. Fixed.

1.10.25 0.8.0 (2014-05-05)

- Memory Consumption for compiled modules (e.g. builtins, sys) has been reduced drastically. Loading times are down as well (it takes basically as long as an import).
- REPL completion is starting to become usable.
- Various small API changes. Generally this release focuses on stability and refactoring of internal APIs.
- Introducing operator precedence, which makes calculating correct Array indices and `__getattr__` strings possible.

1.10.26 0.7.0 (2013-08-09)

- Switched from LGPL to MIT license.
- Added an Interpreter class to the API to make autocompletion in REPL possible.
- Added autocompletion support for namespace packages.
- Add sith.py, a new random testing method.

1.10.27 0.6.0 (2013-05-14)

- Much faster parser with builtin part caching.
- A test suite, thanks @tkf.

1.10.28 0.5 versions (2012)

- Initial development.

CHAPTER 2

Resources

If you want to stay **up-to-date** with releases, please **subscribe** to this mailing list: https://groups.google.com/g/jedi-announce. To subscribe you can simply send an empty email to `jedi-announce+subscribe@googlegroups.com`.

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

Python Module Index

j

Index

U