reddel-server

Release 0.2.0

February 01, 2017

Contents

-
-
-
-
-

Table of Contents

Python EPC server component for reddel. It provides an easy way to send (python) source code from Emacs to the server, inspect or transform it via Redbaron and send the result back.

An example on how to expose a simple function to add arguments:

```
# file: myprovidermod.py
import reddel_server

class MyProvider(reddel_server.ProviderBase)
    @reddel_server.red_src()
    @reddel_server.red_validate([reddel_server.OptionalRegionValidator(),
                                 reddel_server.SingleNodeValidator(),
                                 reddel_server.TypeValidator(["def"])])
    def add_arg(self, red, start, end, index, arg):
        red.arguments.insert(index, arg)
        return red
```

Start the reddel server from Emacs:

```
>>> (require 'epc)

>>> (defvar my-epc (epc:start-epc "reddel" nil))

>>> ;; make sure myprovidermod is in a directory within the PYTHONPATH
>>> (epc:call-sync my-epc 'add_provider '("myprovidermod.MyProvider"))

>>> (message (epc:call-sync my-epc 'add_arg '("def foo(arg1, arg3): pass" nil nil 1 "arg2")))
"def foo(arg1, arg2, arg3): pass"
```

Redbaron provides a lossless format, so even formatting and comments are preserved.

CHAPTER 1

Installation

At the command line:

```
pip install reddel-server
```

CHAPTER 2

Usage

You can start a reddel server from within Emacs like shown above or start it from the command line. A executable `reddel` is provided by this project, which should suitable for most use cases.

```
$ reddel --help
Usage: reddel [OPTIONS]

Options:
  --address TEXT       address to bind the server to
  --port INTEGER       address to bind the server to
  -p, --provider TEXT  dotted path to a provider class
  -v, --verbosity LVL  Either CRITICAL, ERROR, WARNING, INFO or DEBUG
  --debug              Show tracebacks when erroring.
  --help               Show this message and exit.
```

`--address TEXT`	Default is `localhost`. Can be an IP or domain name.
`--port INTEGER`	Defaults to a random free port.
`-p, --provider TEXT`	Example: `mypkg.mymod.MyProviderClass` You can provide this multiple times. Defines additional providers that are available from the start. More providers can be added at runtime via `reddel_server.ChainedProvider.add_provider`.
`-v, --verbosity LVL`	Define the logging level.
`--debug`	By default all expected exceptions are only logged without the traceback. If this flag is set, the traceback is printed as well.
`--help`	Show the help message and exit.

Calling a function from within Emacs is very simple thanks to epc:

```
>>> (progn
...   (require 'epc)
...   (defvar my-epc (epc:start-epc "reddel" nil))
...   ;; list all methods compatible with the given source
...   (message (epc:call-sync my-epc 'list_methods '("def foo(arg1, arg3): pass"))))
```

`(epc:start-epc "reddel" nil)` starts the server by executing `reddel` without any arguments (`nil`). Then you can make calls to that server by referring to the manager returned from `epc:start-epc`. To execute a call, you can use `(epc:call-sync <manager> <method> <arguments>)`, where `<manager>` is the manager returned by `epc:start-epc`, `<method>` is the function and `<arguments>` is a list of arguments passed to `<method>`.

The Builtin Functions section in the documentation provides a guide through all functions that ship with this package. If you need advanced features check the reference documentation for help on how to write your own server.

CHAPTER 3

Documentation

https://reddel-server.readthedocs.io/

CHAPTER 4

Bugs/Requests

Please use the GitHub issue tracker to submit bugs or request features.

CHAPTER 5

Development

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER 6

License

6.1 Builtin Functions

Here is a list of all builtin functions the server provides. Some functions that work on code have a table like this in the docstring:

source input	outputs source	region	only single node	allowed types
Yes	No	Mandatory	No	Any

- `source input`: `Yes` means, that the function expects source code as first argument
- `outputs source`: `Yes` means that the function returns source code back
- `region`: `No` means that this function does not accept regions. `Optional` means that if a region is specified, only the region will be considered. If no region is specified, the whole source is used. `Mandatory` always requires a region.
- `only single node`: `Yes` means only a single node on root level is allowed.
- `allowed types`: a list of allowed identifiers on root level. E.g. `def` means only function definitions are valid.

See *reddel_server.red_src()*, *reddel_server.red_validate()*.

6.1.1 Basic Functions

The following functions are useful for introspection of reddel:

`ProviderBase.`**`list_methods`**(*src=None*)

Return a list of methods that this Provider exposes to clients

To get more information for each method use *reddel_server.ProviderBase.help()*.

By default this returns all available methods. But it can also be used to only get methods that actually work on a given source. This feature might be handy to dynamically build UIs that adapt to the current context.

To write your own methods that can be filtered in the same way, use the *reddel_server.red_validate()* decorators.

Parameters **source** (str) – if src return only compatible methods

Returns list of str

ProviderBase.**help**(*name*)
Return the docstring of the method

Parameters **name** (str) – the name of the method.

Example:

```
import reddel_server
server = reddel_server.Server()
p = reddel_server.ProviderBase(server)
for m in p.list_methods():
    print(m + ":")
    print(p.help(m))
```

ProviderBase.**echo**(*echo*)
Echo the given object

Can be used for simple tests. For example to test if certain values can be send to and received from the server.

Parameters **echo** – the object to echo

Returns the given echo

ProviderBase.**reddel_version**()
Return the reddel version

Some very simple logging control at runtime:

Server.**set_logging_level**(*level*)
Set logging level

Parameters **level** (str | int) – either DEBUG, INFO, WARNING, ERROR, CRITICAL or integer

Returns None

Raises None

Can be called with integer or a string:

```
>>> import logging
>>> import reddel_server
>>> server = reddel_server.Server()
>>> server.set_logging_level("DEBUG")
>>> server.set_logging_level(logging.INFO)
>>> server.set_logging_level(10)
>>> server.logger.level
10
```

The string has to be one of the builtin logging levels of logging, see Logging Levels.

Extend the provider at runtime:

ChainedProvider.**add_provider**(*dotted_path*)
Add a new provider

Parameters **dotted_path** (str) – dotted path to provider class. E.g. mypkg.mymod.MyProvider.

This provides a simple plug-in system. A client (e.g. Emacs) can call add_provider with a dotted path to a class within a module. The module has to be importable. So make sure you installed it or added the directory to the PYTHONPATH.

```
import reddel_server
cp = reddel_server.ChainedProvider(reddel_server.Server())
cp.add_provider('reddel_server.RedBaronProvider')
```

If the given provider has methods with the same name as the existing ones, it's methods will take precedence.

This will invalidate the cached methods on this instance and also on the server.

6.1.2 Code Introspection

The following functions are useful for inspecting source code:

Any source

RedBaronProvider.**analyze**(*src*, **args*, ***kwargs*)

Return the red baron help string for the given source

source input	outputs source	region	only single node	allowed types
Yes	No	No	No	Any

Parameters

- **red** (redbaron.RedBaron) – the red baron source
- **deep** (int) – how deep the nodes get printed
- **with_formatting** (bool) – also analyze formatting nodes

Returns the help text

Return type str

Example:

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> print(p.analyze("1+1"))
BinaryOperatorNode()
  # identifiers: binary_operator, binary_operator_, binaryoperator, binaryoperatornode
  value='+'
  first ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
  second ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
```

RedBaronProvider.**get_parents**(*src*, **args*, ***kwargs*)

Return a list of parents (scopes) relative to the given position

source input	outputs source	region	only single node	allowed types
Yes	No	Mandatory	No	Any

Parameters

- **red** (redbaron.RedBaron) – the red baron source

- **start** (*reddel_server.Position*) – the start position of the selected region.
- **end** (*reddel_server.Position*) – the end position of the selected region.

Returns a list of parents starting with the element at position first.

Return type `list` of the parents. A parent is represented by a `Parent` of the type, top-left, bottom-right position. Each position is a `Position` object.

```
>>> import reddel_server
>>> import pprint
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1):
...     arg2 = arg2 or ""
...     if Ture:
...         try:
...             pass
...         except:
...             func(subfunc(arg1="asdf"))
... """
>>> pprint.pprint(p.get_parents(src, reddel_server.Position(7, 32), reddel_server.Position(7, 32
[Parent(identifier='string', start=Position(row=7, column=31), end=Position(row=7, column=36)),
 Parent(identifier='call_argument', start=..., end=...),
 Parent(identifier='call', start=..., end=...),
 Parent(identifier='call_argument', start=..., end=...),
 Parent(identifier='call', start=..., end=...),
 Parent(identifier='atomtrailers', start=..., end=...),
 Parent(identifier='except', start=..., end=...),
 Parent(identifier='try', start=..., end=...),
 Parent(identifier='ifelseblock', start=..., end=...),
 Parent(identifier='def', start=..., end=...)]
```

Function Definitions

`RedBaronProvider.`**`get_args`**(*src*, **args*, ***kwargs*)

Return a list of args and their default value (if any) as source code

source input	outputs source	region	only single node	allowed types
Yes	No	Optional	Yes	def

Parameters

- **red** (`redbaron.RedBaron`) – the red baron source
- **start** (*reddel_server.Position* | `None`) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | `None`) – the end position of the selected region, if any.

Returns list of argument name and default value.

Return type `list` of `tuple` with `str` and `str` | `None` in it.

The default value is always a string, except for arguments without one which will be represented as None.

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1, arg2, kwarg1=None, kwarg2=1, kwarg3='None'):
```

```
...     arg2 = arg2 or ""
...     return arg2 + func(arg1, "arg2 arg2") + kwarg2
... """
>>> p.get_args(src, None, None)
[('arg1', None), ('arg2', None), ('kwarg1', 'None'), ('kwarg2', '1'), ('kwarg3', "'None'")]
```

6.1.3 Code Transformation

The following functions transform source code:

Function Definitions

RedBaronProvider.**rename_arg**(*src*, **args*, ***kwargs*)

Rename a argument

source input	outputs source	region	only single node	allowed types
Yes	Yes	Optional	Yes	def

Parameters

- **red** (redbaron.RedBaron) – the red baron source
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.
- **oldname** (str) – name of the argument to rename
- **newname** (str) – new name for the argument

Returns the transformed source code

Return type redbaron.RedBaron

Example:

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1, arg2, kwarg2=1):  # arg2
...     arg2 = arg2 or ""
...     return arg2 + func(arg1, "arg2 arg2") + kwarg2
... """
>>> print(p.rename_arg(src, None, None, "arg2", "renamed"))
def foo(arg1, renamed, kwarg2=1):  # arg2
    renamed = renamed or ""
    return renamed + func(arg1, "arg2 arg2") + kwarg2
```

RedBaronProvider.**add_arg**(*src*, **args*, ***kwargs*)

Add a argument at the given index

source input	outputs source	region	only single node	allowed types
Yes	Yes	Optional	Yes	Yes

Parameters

- **red** (redbaron.RedBaron) – the red baron source

- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.
- **index** (int) – position of the argument. 0 would mean to put the argument in the front.
- **arg** (str) – the argument to add

Returns the transformed source code

Return type redbaron.RedBaron

Example:

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1, arg2, kwarg2=1):
...     arg2 = arg2 or ""
...     return arg2 + func(arg1, "arg2 arg2") + kwarg2
... """
>>> print(p.add_arg(src, start=None, end=None, index=3, arg="kwarg3=123"))
def foo(arg1, arg2, kwarg2=1, kwarg3=123):
    arg2 = arg2 or ""
    return arg2 + func(arg1, "arg2 arg2") + kwarg2
```

6.2 Reference

The reddel_server package provides the complete supported API.

Note: It is not recommended to import any submodules as they are considered protected and might get breaking changes between minor and patch versions.

The API is divided into multiple domains:

- *Server*
- *Providers*
  - *RedBaron Provider*
- *Validators*
- *Exceptions*

6.2.1 Server

The *Server* is used to provide functions via EPC that a client like Emacs can call. The server is automatically created by the reddel script, which gets installed with this package. If you need a custom server with more functionality like threading or authentication (I don't know what kids need these days) you can use this as a base. Most functions the server provides come from a *provider* (see *Providers*).

See:

- *Server*

6.2.2 Providers

Providers are the heart of reddel. They provide methods that you can call remotely via the *Server*. *list_methods* can be used to get all available methods of a provider. You can also call it with a piece of source code to get all methods that can operate on it. This works by decorating a method with *red_validate* and providing some *Validators*. There are more useful decorators listed below.

The *ChainedProvider* is useful for combining multiple providers. The *Server* from the CLI uses such provider to combine a *RedBaronProvider* and any provider specified by the user. By calling *add_provider* (also remotely) with a dotted path you can add your own providers at runtime to extend reddel.

See:

- *ProviderBase*
- *ChainedProvider*

RedBaron Provider

The *RedBaronProvider* provides the built-in redbaron specific functionality. If you want to extend it or write your own provider, it's recommended to make use of the following decorators:

- *red_src*
- *red_validate*

These decorators are the mini framework that allows the server to tell the client what actions are available for a given piece of code.

There is also a small library with helper functions that might be useful when writing a provider:

- *get_parents*
- *get_node_of_region*

See:

- *RedBaronProvider*
- *redwraps*
- *red_src*
- *red_validate*
- *get_parents*
- *get_node_of_region*

6.2.3 Validators

Validators are used to get all methods compatible for processing a given source. E.g. if the source is a function, reddel can report to Emacs which functions can be applied to functions and Emacs can use the information to dynamically build a UI.

Validators can transform the source as well. The transformed source is passed onto the next validator when you use *reddel_server.red_validate()*. All validators provided by `reddel_server` can be used as mix-ins. When you create your own validator and you inherit from multiple builtin ones then they are effectively combined since all of them perform the appropriate super call.

See:

- *ValidatorInterface*
- *OptionalRegionValidator*
- *MandatoryRegionValidator*
- *SingleNodeValidator*
- *TypeValidator*

6.2.4 Exceptions

Here is a list of custom exceptions raised in reddel:

- *ValidationException*

6.2.5 API

class reddel_server.**Server**(*server_address=('localhost', 0)*, *RequestHandlerClass=<class epc.handler.EPCHandler>*)

Bases: epc.server.EPCServer

EPCServer that provides basic functionality.

This is a simple epc.server.EPCServer that exposes methods for clients to call remotely. You can use the python client to connect or call a method from within emacs. The exposed methods are defined by a *Provider*. Call *reddel_server.Server.set_provider()* to register the functions.

If a provider can change it's methods dynamically, make sure to call *reddel_server.Server.set_provider()* to reset the method cache.

Initialize server serving the given address with the given handler.

Parameters

- **server_address** (*tuple*) – URL/IP and port
- **RequestHandlerClass** (epc.server.EPCHandler) – the handler class to use

Raises None

allow_reuse_address = True

get_provider()

The *reddel_server.ProviderBase* instance that provides methods.

set_logging_level(*level*)

Set logging level

Parameters **level** (str | int) – either DEBUG, INFO, WARNING, ERROR, CRITICAL or integer

Returns None

Raises None

Can be called with integer or a string:

```
>>> import logging
>>> import reddel_server
>>> server = reddel_server.Server()
>>> server.set_logging_level("DEBUG")
>>> server.set_logging_level(logging.INFO)
```

```
>>> server.set_logging_level(10)
>>> server.logger.level
10
```

The string has to be one of the builtin logging levels of `logging`, see Logging Levels.

set_provider(*provider*)
Set the provider and reset the registered functions.

Parameters **provider** (*reddel_server.ProviderBase*) – the provider to set

class reddel_server.**ProviderBase**(*server*)
Bases: object

Base class for all Providers.

A provider exposes methods via the *reddel_server.Server* to clients. By default all public methods (that do not start with an underscore) are exposed.

Creating your own basic provider is very simple:

```
import reddel_server

class MyProvider(reddel_server.ProviderBase):
    def exposed(self):
        print("I'm exposed")
    def _private(self):
        print("I'm private")
```

```
server = reddel_server.Server()
provider = MyProvider(server)
server.set_provider(provider)
methods = provider.list_methods()
assert "exposed" in methods
assert "_private" not in methods
```

When starting reddel from the command line via the command `reddel`, it's automatically setup with a *reddel_server.ChainedProvider*, which combines multiple providers together. It also gives you the ability to call *reddel_server.ChainedProvider.add_provider()* from a client.

You can get a list of all methods provided by a provider by calling *reddel_server.ProviderBase.list_methods()*.

Initialize provider

Parameters **server** (*reddel_server.Server*) – the server that is using the provider

echo(*echo*)
Echo the given object

Can be used for simple tests. For example to test if certain values can be send to and received from the server.

Parameters **echo** – the object to echo

Returns the given echo

help(*name*)
Return the docstring of the method

Parameters **name** (str) – the name of the method.

Example:

```
        import reddel_server
        server = reddel_server.Server()
        p = reddel_server.ProviderBase(server)
        for m in p.list_methods():
            print(m + ":")
            print(p.help(m))
```

list_methods(*src=None*)
Return a list of methods that this Provider exposes to clients

To get more information for each method use *reddel_server.ProviderBase.help()*.

By default this returns all available methods. But it can also be used to only get methods that actually work on a given source. This feature might be handy to dynamically build UIs that adapt to the current context.

To write your own methods that can be filtered in the same way, use the *reddel_server.red_validate()* decorators.

Parameters **source** (str) – if src return only compatible methods

Returns list of str

reddel_version()
Return the reddel version

server

class reddel_server.**ChainedProvider**(*server*, *providers=None*)
Bases: reddel_server.provider.ProviderBase

Provider that can chain multiple other providers together

This is the provider used by the command line client to combine *reddel_server.RedBaronProvider* with third party providers. *reddel_server.ChainedProvider.add_provider()* is a function to provide a simple plug-in system.

Example:

```
import reddel_server

class FooProvider(reddel_server.ProviderBase):
    def foo(self): pass

class BarProvider(reddel_server.ProviderBase):
    def bar(self): pass

server = reddel_server.Server()
providers = [FooProvider(server), BarProvider(server)]
p = reddel_server.ChainedProvider(server, providers)
methods = p.list_methods()
assert "foo" in methods
assert "bar" in methods
```

Methods are cached in reddel_server.ChainedProvider._cached_methods. reddel_server.ChainedProvider._get_methods() will use the cached value unless it's None. *reddel_server.ChainedProvider.add_provider()* will reset the cache. Keep that in mind when building dynamic providers because the cache might become invalid.

Initialize a provider which acts as a combination of the given providers.

Parameters

- **server** (*reddel_server.Server*) – the server that is using the provider

- **providers** (list of *ProviderBase*) – list of providers. A provider's methods at the front of the list will take precedence.

add_provider(*dotted_path*)
Add a new provider

Parameters **dotted_path** (str) – dotted path to provider class. E.g. mypkg.mymod.MyProvider.

This provides a simple plug-in system. A client (e.g. Emacs) can call add_provider with a dotted path to a class within a module. The module has to be importable. So make sure you installed it or added the directory to the PYTHONPATH.

```
import reddel_server
cp = reddel_server.ChainedProvider(reddel_server.Server())
cp.add_provider('reddel_server.RedBaronProvider')
```

If the given provider has methods with the same name as the existing ones, it's methods will take precedence.

This will invalidate the cached methods on this instance and also on the server.

class reddel_server.**ValidatorInterface**
Bases: object

Validator interface

A validator checks a given source input and raises a *ValidationException* if the input is invalid. This can be used to check, which methods of a provider are compatible with a given input (see *reddel_server.red_validate()*).

Creating your own validator is simple. Just subclass from this class and override *reddel_server.ValidatorInterface.__call__()*.

```
import redbaron
import reddel_server

class MyValidator(reddel_server.ValidatorInterface):
    def __call__(self, red, start=None, end=None):
        if not (start and end):
            raise reddel_server.ValidationException("Expected a region.")
        if len(red) != 1:
            raise reddel_server.ValidationException("Expected only a single root node.")

val = MyValidator()

val(redbaron.RedBaron("a=2"), reddel_server.Position(1, 1), reddel_server.Position(1, 3))

try:
    val(redbaron.RedBaron("a=2+1\nb=3"))
except reddel_server.ValidationException:
    pass
else:
    assert False, 'Validator should have raised.'
```

A Validator can also implement a *transformation*. This transformation is used in *reddel_server.red_validate()*.

__call__(*red*, *start=None*, *end=None*)
Validate the given redbaron source

Parameters

- **red** (redbaron.RedBaron) – the source
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Raises *ValidationException*

transform(*red*, *start=None*, *end=None*)
Transform the given red baron

The base implementation just returns the source. See reddel_server.TypeValidator.transform() for an example.

Parameters

- **red** – a red baron source or other nodes
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Returns the transformed source, start and end

class reddel_server.**OptionalRegionValidator**
Bases: reddel_server.validators.ValidatorInterface

Used for functions that either use the given source code or only the specified region if a region is specified.

If a region is specified the source is transformed to only contain the region.

Examples:

```
>>> from redbaron import RedBaron
>>> import reddel_server
>>> val1 = reddel_server.OptionalRegionValidator()
>>> src, start, end = val1.transform(RedBaron('def foo(): pass'), start=None, end=None)
>>> src.dumps(), start, end
('def foo(): pass\n', None, None)
>>> src, start, end = val1.transform(RedBaron('a=1\nb=1'), start=(2,1), end=(2,3))
>>> src.dumps(), start, end
('b=1', Position(row=1, column=1), Position(row=1, column=3))
```

__call__(*red*, *start=None*, *end=None*)
Validate the given redbaron source

Parameters

- **red** (redbaron.RedBaron) – the source
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Raises *ValidationException*

transform(*red*, *start=None*, *end=None*)
Extract the region from red if any region is specified

Parameters

- **red** – a red baron source or other nodes
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Returns the transformed source, start and end

class reddel_server.**MandatoryRegionValidator**
Bases: reddel_server.validators.ValidatorInterface

Used for functions that expect a region

__call__(*red*, *start=None*, *end=None*)
Validate that a region is specified

Parameters

- **red** (redbaron.RedBaron) – the source
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Raises *ValidationException* if start or end is None.

class reddel_server.**SingleNodeValidator**
Bases: reddel_server.validators.ValidatorInterface

Validate that only one single node is provided.

If a list of nodes is provided, validate that it contains only one element. Transform the source to only a single node.

```
>>> from redbaron import RedBaron
>>> import reddel_server
>>> val1 = reddel_server.SingleNodeValidator()
>>> val1(redbaron.RedBaron("a=1+1"))
>>> val1.transform(redbaron.RedBaron("a=1+1"))
(a=1+1, None, None)
>>> try:
...     val1(redbaron.RedBaron("a=1+1\nb=2"))
... except reddel_server.ValidationException:
...     pass
... else:
...     assert False, "Validator should have raised"
```

By default, when creating a redbaron.RedBaron source, you always get a list even for a single expression. If you always want the single node, this validator will handle the transformation.

__call__(*red*, *start=None*, *end=None*)
Validate the given redbaron source

Parameters

- **red** (redbaron.RedBaron) – the source
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.

- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Raises *ValidationException*

transform(*red*, *start=None*, *end=None*)
Extract the single node red is a list

Parameters

- **red** – a red baron source or other nodes
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Returns the transformed source, start and end

class reddel_server.**TypeValidator**(*identifiers*)
Bases: reddel_server.validators.ValidatorInterface

Validate that the given source contains the correct type of nodes.

If a region is specified, only the region is checked. If a list of nodes is given, e.g. a redbaron.RedBaron object, all nodes will be checked.

Examples:

```
from redbaron import RedBaron
import reddel_server

val1 = reddel_server.TypeValidator(['def'])

# valid
val1(RedBaron('def foo(): pass'))
val1(RedBaron('def foo(): pass\ndef bar(): pass'))

# invalid
try:
    val1(RedBaron('def foo(): pass\na=1+1'))
except reddel_server.ValidationException:
    pass
else:
    assert False, "Validator should have raised"
```

Initialize the validator

Parameters **identifiers** (sequence of str) – allowed identifiers for the redbaron source

__call__(*red*, *start=None*, *end=None*)
Validate the given redbaron source

Parameters

- **red** (redbaron.RedBaron) – the source
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.

Raises *ValidationException*

exception reddel_server.**ValidationException**
Bases: exceptions.Exception

Raised when calling *reddel_server.ValidatorInterface* and a source is invalid.

class reddel_server.**RedBaronProvider**(*server*)
Bases: reddel_server.provider.ProviderBase

Provider for inspecting and transforming source code via redbaron.

Initialize provider

Parameters **server** (*reddel_server.Server*) – the server that is using the provider

add_arg(*src*, **args*, ***kwargs*)
Add a argument at the given index

source input	outputs source	region	only single node	allowed types
Yes	Yes	Optional	Yes	Yes

Parameters

- **red** (redbaron.RedBaron) – the red baron source
- **start** (*reddel_server.Position* | None) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | None) – the end position of the selected region, if any.
- **index** (int) – position of the argument. 0 would mean to put the argument in the front.
- **arg** (str) – the argument to add

Returns the transformed source code

Return type redbaron.RedBaron

Example:

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1, arg2, kwarg2=1):
...     arg2 = arg2 or ""
...     return arg2 + func(arg1, "arg2 arg2") + kwarg2
... """
>>> print(p.add_arg(src, start=None, end=None, index=3, arg="kwarg3=123"))
def foo(arg1, arg2, kwarg2=1, kwarg3=123):
    arg2 = arg2 or ""
    return arg2 + func(arg1, "arg2 arg2") + kwarg2
```

analyze(*src*, **args*, ***kwargs*)
Return the red baron help string for the given source

source input	outputs source	region	only single node	allowed types
Yes	No	No	No	Any

Parameters

- **red** (redbaron.RedBaron) – the red baron source
- **deep** (int) – how deep the nodes get printed
- **with_formatting** (bool) – also analyze formatting nodes

Returns the help text

Return type `str`

Example:

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> print(p.analyze("1+1"))
BinaryOperatorNode()
  # identifiers: binary_operator, binary_operator_, binaryoperator, binaryoperatornode
  value='+'
  first ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
  second ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
```

get_args (*src*, **args*, ***kwargs*)

Return a list of args and their default value (if any) as source code

source input	outputs source	region	only single node	allowed types
Yes	No	Optional	Yes	def

Parameters

- **red** (`redbaron.RedBaron`) – the red baron source
- **start** (*reddel_server.Position* | `None`) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | `None`) – the end position of the selected region, if any.

Returns list of argument name and default value.

Return type `list` of `tuple` with `str` and `str` | `None` in it.

The default value is always a string, except for arguments without one which will be represented as None.

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1, arg2, kwarg1=None, kwarg2=1, kwarg3='None'):
...     arg2 = arg2 or ""
...     return arg2 + func(arg1, "arg2 arg2") + kwarg2
... """
>>> p.get_args(src, None, None)
[('arg1', None), ('arg2', None), ('kwarg1', 'None'), ('kwarg2', '1'), ('kwarg3', "'None'")]
```

get_parents (*src*, **args*, ***kwargs*)

Return a list of parents (scopes) relative to the given position

source input	outputs source	region	only single node	allowed types
Yes	No	Mandatory	No	Any

Parameters

- **red** (`redbaron.RedBaron`) – the red baron source

- **start** (*reddel_server.Position*) – the start position of the selected region.
- **end** (*reddel_server.Position*) – the end position of the selected region.

Returns a list of parents starting with the element at position first.

Return type `list` of the parents. A parent is represented by a *Parent* of the type, top-left, bottom-right position. Each position is a *Position* object.

```
>>> import reddel_server
>>> import pprint
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1):
...     arg2 = arg2 or ""
...     if Ture:
...         try:
...             pass
...         except:
...             func(subfunc(arg1="asdf"))
... """
>>> pprint.pprint(p.get_parents(src, reddel_server.Position(7, 32), reddel_server.Position(7
[Parent(identifier='string', start=Position(row=7, column=31), end=Position(row=7, column=36
 Parent(identifier='call_argument', start=..., end=...),
 Parent(identifier='call', start=..., end=...),
 Parent(identifier='call_argument', start=..., end=...),
 Parent(identifier='call', start=..., end=...),
 Parent(identifier='atomtrailers', start=..., end=...),
 Parent(identifier='except', start=..., end=...),
 Parent(identifier='try', start=..., end=...),
 Parent(identifier='ifelseblock', start=..., end=...),
 Parent(identifier='def', start=..., end=...)]
```

rename_arg(*src*, **args*, ***kwargs*)

Rename a argument

source input	outputs source	region	only single node	allowed types
Yes	Yes	Optional	Yes	def

Parameters

- **red** (`redbaron.RedBaron`) – the red baron source
- **start** (*reddel_server.Position* | `None`) – the start position of the selected region, if any.
- **end** (*reddel_server.Position* | `None`) – the end position of the selected region, if any.
- **oldname** (`str`) – name of the argument to rename
- **newname** (`str`) – new name for the argument

Returns the transformed source code

Return type `redbaron.RedBaron`

Example:

```
>>> import reddel_server
>>> p = reddel_server.RedBaronProvider(reddel_server.Server())
>>> src = """def foo(arg1, arg2, kwarg2=1):  # arg2
...     arg2 = arg2 or ""
```

```
...     return arg2 + func(arg1, "arg2 arg2") + kwarg2
... """
>>> print(p.rename_arg(src, None, None, "arg2", "renamed"))
def foo(arg1, renamed, kwarg2=1):  # arg2
    renamed = renamed or ""
    return renamed + func(arg1, "arg2 arg2") + kwarg2
```

reddel_server.**red_src**(*dump=True*)

Create decorator that converts the first argument into a red baron source

Parameters **dump** (bool) – if True, dump the return value from the wrapped function. Expects the return type to be a redbaron.RedBaron object.

Returns the decorator

Return type types.FunctionType

Example:

```
import redbaron
import reddel_server

class MyProvider(reddel_server.ProviderBase):
    @reddel_server.red_src(dump=False)
    def inspect_red(self, red):
        assert isinstance(red, redbaron.RedBaron)
        red.help()

MyProvider(reddel_server.Server()).inspect_red("1+1")
```

```
0 -----------------------------------------------------
BinaryOperatorNode()
  # identifiers: binary_operator, binary_operator_, binaryoperator, binaryoperatornode
  value='+'
  first ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
  second ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
```

By default the return value is expected to be a transformed redbaron.RedBaron object that can be dumped. This is useful for taking a source as argument, transforming it and returning it back so that it the editor can replace the original source:

```
>>> import redbaron
>>> import reddel_server

>>> class MyProvider(reddel_server.ProviderBase):
...     @reddel_server.red_src(dump=True)
...     def echo(self, red):
...         assert isinstance(red, redbaron.RedBaron)
...         return red

>>> MyProvider(reddel_server.Server()).echo("1+1")
'1+1'
```

reddel_server.**red_validate**(*validators*)

Create decorator that adds the given validators to the wrapped function

Parameters **validators** (*reddel_server.ValidatorInterface*) – the validators

Validators can be used to provide sanity checks. *reddel_server.ProviderBase.list_methods()* uses them to filter out methods that are incompatible with the given input, which can be used to build dynamic UIs.

To use this validator is very simple. Create one or more validator instances (have to inherit from *reddel_server.ValidatorInterface*) and provide them as the argument:

```
import reddel_server

validator1 = reddel_server.SingleNodeValidator()
validator2 = reddel_server.TypeValidator(["def"])
class MyProvider(reddel_server.ProviderBase):
    @reddel_server.red_src()
    @reddel_server.red_validate([validator1, validator2])
    def foo(self, red, start, end):
        assert red.type == 'def'

provider = MyProvider(reddel_server.Server())

provider.foo("def bar(): pass", start=None, end=None)  # works

try:
    provider.foo("1+1", start=None, end=None)
except reddel_server.ValidationException:
    pass
else:
    assert False, "Validator should have raised"
```

On top of that validators also can implement a *reddel_server.ValidatorInterface.transform()* function to transform the red source on the way down. The transformed value is passed to the next validator and eventually into the function.

See *Validators*.

reddel_server.**redwraps**(*towrap*)

Use this when creating decorators instead of functools.wraps()

Parameters **towrap** (types.FunctionType) – the function to wrap

Returns the decorator

Return type types.FunctionType

Makes sure to transfer special reddel attributes to the wrapped function. On top of that uses functools.wraps().

Example:

```
import reddel_server

def my_decorator(func):
    @reddel_server.redwraps(func)  # here you would normally use functools.wraps
    def wrapped(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped
```

class reddel_server.**Position**

Bases: reddel_server.redlib.Position

Describes a position in the source code by line number and character position in that line.

Parameters

- **row** (int) – the line number
- **column** (int) – the position in the line

class reddel_server.**Parent**
Bases: reddel_server.redlib.Parent

Represents a node type with the bounding location.

Parameters

- **identifier** (str) – the node type
- **start** (*Position*) – the position where the node begins in the code
- **end** (*Position*) – the position where the node ends in the code

reddel_server.**get_parents**(*red*)
Yield the parents of the given red node

Parameters **red** (redbaron.base_nodes.Node | redbaron.RedBaron) – the red baron source

Returns each parent of the given node

Return type Generator[redbaron.base_nodes.Node]

Raises None

reddel_server.**get_node_of_region**(*red*, *start*, *end*)
Get the node that contains the given region

Parameters

- **red** (redbaron.RedBaron) – the red baron source
- **start** (*Position*) – position of the beginning of the region
- **end** (*Position*) – position of the end of the region

Returns the node that contains the region

Return type redbaron.base_nodes.Node

First the nodes at start and end are gathered. Then the common parent is selected. If the common parent is a list, the minimum slice is used. This makes it easier for the user because he can partially select nodes and still gets what he most likely intended to get.

For example if your region partially selects several lines in a for loop and you want to extract them (| shows the region bounds):

```
    for i in range(10):
        a = 1 + 2
        b |= 4
        c = 5
        d =| 7
```

then we expect to get back:

```
    b = 4
        c = 5
        d = 7
```

Note that the leading tab is missing because it doesn't belong to the `b = 4` node.

```
>>> import redbaron
>>> import reddel_server
>>> testsrc = ("for i in range(10):\n"
...            "    a = 1 + 2\n"
...            "    b = 4\n"
...            "    c = 5\n"
...            "    d = 7\n")
>>> start = (3, 7)
>>> end = (5, 8)
>>> reddel_server.get_node_of_region(redbaron.RedBaron(testsrc), start, end).dumps()
'b = 4\n    c = 5\n    d = 7'
```

You can also partially select a list:

```
>>> testsrc = "[1, 2, 3, 4, 5, 6]"
>>> start = (1, 8)  # "3"
>>> end = (1, 14)  # "5"
>>> reddel_server.get_node_of_region(redbaron.RedBaron(testsrc), start, end).dumps()
'3, 4, 5'
```

6.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

6.3.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.3.2 Documentation improvements

reddel-server could always use more documentation, whether as part of the official reddel-server docs, in docstrings, or even on the web in blog posts, articles, and such.

6.3.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/storax/reddel-server/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

6.3.4 Development

To set up *reddel-server* for local development:

1. Fork reddel-server (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/reddel-server.git
```

3. Create a branch for local development. Branch of `develop`:

```
git checkout -b name-of-your-bugfix-or-feature develop
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request against `develop` through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`) [1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

6.4 Authors

- David Zuber - https://github.com/storax

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

6.5 Changelog

6.5.1 0.2.0 (2017-01-02)

- You can optionally specify regions for methods that take in source code.
- Restructure the validators into several small classes.

6.5.2 0.1.0 (2016-11-17)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

r

Symbols

A

C

E

G

H

L

M

O

P

R

S

T

V