RestrictedPython Documentation

Release 5.0

Alexander Loechel

Oct 07, 2020

Contents

1 Supported Python versions **3**

2 Contents **5**

2.1 The idea behind RestrictedPython 5
2.2 Install / Depend on RestrictedPython 7
2.3 Usage of RestrictedPython 7
2.4 API overview 14
2.5 Upgrade from 3.x 16
2.6 Roadmap 16
2.7 Contributing 17
2.8 Changes 29

3 Indices and tables **33**

Index **35**

RestrictedPython is a tool that helps to define a subset of the Python language which allows to provide a program input into a trusted environment. RestrictedPython is not a sandbox system or a secured environment, but it helps to define a trusted environment and execute untrusted code inside of it.

CHAPTER 1

Supported Python versions

RestrictedPython supports CPython 2.7, 3.5, 3.6, 3.7 and 3.8. It does _not_ support PyPy or other alternative Python implementations.

CHAPTER 2

Contents

2.1 The idea behind RestrictedPython

Python is a Turing complete programming language. To offer a Python interface for users in web context is a potential security risk. Web frameworks and Content Management Systems (CMS) want to offer their users as much extensibility as possible through the web (TTW). This also means to have permissions to add functionality via a Python script.

There should be additional preventive measures taken to ensure integrity of the application and the server itself, according to information security best practice and unrelated to RestrictedPython.

RestrictedPython defines a safe subset of the Python programming language. This is a common approach for securing a programming language. The Ada Ravenscar profile is another example of such an approach.

Defining a secure subset of the language involves restricting the EBNF elements and explicitly allowing or disallowing language features. Much of the power of a programming language derives from its standard and contributed libraries, so any calling of these methods must also be checked and potentially restricted. RestrictedPython generally disallows calls to any library that is not explicit whitelisted.

As Python is a scripting language that is executed by an interpreter any Python code that should be executed has to be explicitly checked before executing the generated byte code by the interpreter.

Python itself offers three methods that provide such a workflow:

- `compile()` which compiles source code to byte code
- `exec` / `exec()` which executes the byte code in the interpreter
- `eval` / `eval()` which executes a byte code expression

Therefore RestrictedPython offers a replacement for the python builtin function `compile()` (Python 2 / Python 3). This Python function is defined as following:

```
compile(source, filename, mode [, flags [, dont_inherit]])
```

The definition of the `compile()` method has changed over time, but its relevant parameters `source` and `mode` still remain.

There are three valid string values for `mode`:

- `'exec'`
- `'eval'`
- `'single'`

For RestrictedPython this `compile()` method is replaced by:

```
RestrictedPython.compile_restricted(source, filename, mode [, flags [, dont_inherit]])
```

The primary parameter `source` has to be a string or `ast.AST` instance. Both methods either return compiled byte code that the interpreter can execute or raise exceptions if the provided source code is invalid.

As `compile` and `compile_restricted` just compile the provided source code to byte code it is not sufficient as a sandbox environment, as all calls to libraries are still available.

The two methods / statements:

- `exec` / `exec()`
- `eval` / `eval()`

have two parameters:

- `globals`
- `locals`

which are references to the Python builtins.

By modifying and restricting the available modules, methods and constants from `globals` and `locals` we can limit the possible calls.

Additionally RestrictedPython offers a way to define a policy which allows developers to protect access to attributes. This works by defining a restricted version of:

- `print`
- `getattr`
- `setattr`
- `import`

Also RestrictedPython provides three predefined, limited versions of Python's `__builtins__`:

- `safe_builtins` (by Guards.py)
- `limited_builtins` (by Limits.py), which provides restricted sequence types
- `utilities_builtins` (by Utilities.py), which provides access for standard modules math, random, string and for sets.

One special shortcut:

- `safe_globals` for `{'__builtins__': safe_builtins}` (by Guards.py)

Additional there exist guard functions to make attributes of Python objects immutable –> `full_write_guard` (write and delete protected).

2.2 Install / Depend on RestrictedPython

RestrictedPython is usually not used stand alone, if you use it in context of your package add it to `install_requires` in your `setup.py` or a `requirement.txt` used by `pip`.

For a standalone usage:

```
$ pip install RestrictedPython
```

2.3 Usage of RestrictedPython

2.3.1 Basic usage

The general workflow to execute Python code that is loaded within a Python program is:

```
source_code = """
def do_something():
    pass
"""

byte_code = compile(source_code, filename='<inline code>', mode='exec')
exec(byte_code)
do_something()
```

With RestrictedPython that workflow should be as straight forward as possible:

```
from RestrictedPython import compile_restricted

source_code = """
def do_something():
    pass
"""

byte_code = compile_restricted(
    source_code,
    filename='<inline code>',
    mode='exec'
)
exec(byte_code)
do_something()
```

You might also use the replacement import:

```
from RestrictedPython import compile_restricted as compile
```

`compile_restricted` uses a predefined policy that checks and modify the source code and checks against a restricted subset of the Python language. The compiled source code is still executed against the full available set of library modules and methods.

The Python `exec()` takes three parameters:

- `code` which is the compiled byte code
- `globals` which is global dictionary
- `locals` which is the local dictionary

By limiting the entries in the `globals` and `locals` dictionaries you restrict the access to the available library modules and methods.

Providing defined dictionaries for `exec()` should be used in context of RestrictedPython.

```
byte_code = <code>
exec(byte_code, { ... }, { ... })
```

Typically there is a defined set of allowed modules, methods and constants used in that context. RestrictedPython provides three predefined built-ins for that (see *Predefined builtins* for details):

- `safe_builtins`
- `limited_builtins`
- `utility_builtins`

So you normally end up using:

```
from RestrictedPython import compile_restricted

from RestrictedPython import safe_builtins
from RestrictedPython import limited_builtins
from RestrictedPython import utility_builtins

source_code = """
def do_something():
    pass
"""

try:
    byte_code = compile_restricted(
        source_code,
        filename='<inline code>',
        mode='exec'
    )
    exec(byte_code, {'__builtins__': safe_builtins}, None)
except SyntaxError as e:
    pass
```

One common advanced usage would be to define an own restricted builtin dictionary.

There is a shortcut for `{'__builtins__': safe_builtins}` named `safe_globals` which can be imported from `RestrictedPython`.

2.3.2 Necessary setup

RestrictedPython requires some predefined names in globals in order to work properly.

To use classes in Python 3 `__metaclass__` must be set. Set it to `type` to use no custom metaclass.

To use `for` statements and comprehensions:

- `_getiter_` must point to an `iter` implementation. As an unguarded variant you might use `RestrictedPython.Eval.default_guarded_getiter()`.
- `_iter_unpack_sequence_` must point to `RestrictedPython.Guards.guarded_iter_unpack_sequence()`.

To use `getattr` you have to provide an implementation for it. `RestrictedPython.Guards.safer_getattr()` can be a starting point.

The usage of *RestrictedPython* in `AccessControl.ZopeGuards` can serve as example.

2.3.3 Usage in frameworks and Zope

One major issue with using `compile_restricted` directly in a framework is, that you have to use try-except statements to handle problems and it might be a bit harder to provide useful information to the user. RestrictedPython provides four specialized compile_restricted methods:

- `compile_restricted_exec`
- `compile_restricted_eval`
- `compile_restricted_single`
- `compile_restricted_function`

Those four methods return a named tuple (`CompileResult`) with four elements:

`code` `<code>` object or `None` if `errors` is not empty

`errors` a tuple with error messages

`warnings` a list with warnings

`used_names` a dictionary mapping collected used names to `True`.

These details can be used to inform the user about the compiled source code.

Modifying the builtins is straight forward, it is just a dictionary containing the available library elements. Modification normally means removing elements from existing builtins or adding allowed elements by copying from globals.

For frameworks it could possibly also be useful to change the handling of specific Python language elements. For that use case RestrictedPython provides the possibility to pass an own policy.

A policy is basically a special `NodeTransformer` that could be instantiated with three params for `errors`, `warnings` and `used_names`, it should be a subclass of RestrictedPython.RestrictingNodeTransformer.

```
from RestrictedPython import compile_restricted
from RestrictedPython import RestrictingNodeTransformer

class OwnRestrictingNodeTransformer(RestrictingNodeTransformer):
    pass

policy_instance = OwnRestrictingNodeTransformer(
    errors=[],
    warnings=[],
    used_names=[]
)
```

All `compile_restricted*` methods do have an optional parameter `policy`, where a specific policy could be provided.

```
source_code = """
def do_something():
    pass
"""

policy = OwnRestrictingNodeTransformer

byte_code = compile_restricted(
    source_code,
```

(continues on next page)

(continued from previous page)

```
    filename='<inline code>',
    mode='exec',
    policy=policy # policy class
)
exec(byte_code, globals(), None)
```

One special case "unrestricted RestrictedPython" (defined to unblock ports of Zope Packages to Python 3) is to actually use RestrictedPython in an unrestricted mode, by providing a Null-Policy (aka `None`). That special case would be written as:

```
from RestrictedPython import compile_restricted

source_code = """
def do_something():
    pass
"""

byte_code = compile_restricted(
    source_code,
    filename='<inline code>',
    mode='exec',
    policy=None # Null-Policy -> unrestricted
)
exec(byte_code, globals(), None)
```

2.3.4 Policies & builtins

RestrictedPython provides a way to define policies, by redefining restricted versions of `print`, `getattr`, `setattr`, `import`, etc.. As shortcuts it offers three stripped down versions of Python's `__builtins__`:

Predefined builtins

`safe_builtins` a safe set of builtin modules and functions

`limited_builtins` restricted sequence types (e. g. `range`, `list` and `tuple`)

`utility_builtins` access to standard modules like math, random, string and set.

`safe_globals` is a shortcut for `{'__builtins__': safe_builtins}` as this is the way globals have to be provided to the *exec* function to actually restrict the access to the builtins provided by Python.

Guards

Todo: Describe Guards and predefined guard methods in details

RestrictedPython predefines several guarded access and manipulation methods:

- `safer_getattr`
- `guarded_setattr`
- `guarded_delattr`
- `guarded_iter_unpack_sequence`

- `guarded_unpack_sequence`

Those and additional methods rely on a helper construct `full_write_guard`, which is intended to help implement immutable and semi mutable objects and attributes.

Todo: Describe full_write_guard more in detail and how it works.

2.3.5 Implementing a policy

RestrictedPython only provides the raw material for restricted execution. To actually enforce any restrictions, you need to supply a policy implementation by providing restricted versions of `print`, `getattr`, `setattr`, `import`, etc. These restricted implementations are hooked up by providing a set of specially named objects in the global dict that you use for execution of code. Specifically:

1. `_print_` is a callable object that returns a handler for print statements. This handler must have a `write()` method that accepts a single string argument, and must return a string when called. `RestrictedPython.PrintCollector.PrintCollector` is a suitable implementation.
2. `_write_` is a guard function taking a single argument. If the object passed to it may be written to, it should be returned, otherwise the guard function should raise an exception. `_write_` is typically called on an object before a `setattr` operation.
3. `_getattr_` and `_getitem_` are guard functions, each of which takes two arguments. The first is the base object to be accessed, while the second is the attribute name or item index that will be read. The guard function should return the attribute or subitem, or raise an exception. RestrictedPython ships with a default implementation for `_getattr_` which prevents the following actions:
   - accessing an attribute whose name start with an underscore
   - accessing the format method of strings as this is considered harmful.
4. `__import__` is the normal Python import hook, and should be used to control access to Python packages and modules.
5. `__builtins__` is the normal Python builtins dictionary, which should be weeded down to a set that cannot be used to get around your restrictions. A usable "safe" set is `RestrictedPython.Guards.safe_builtins`.

To help illustrate how this works under the covers, here's an example function:

```
def f(x):
    x.foo = x.foo + x[0]
    print x
    return printed
```

and (sort of) how it looks after restricted compilation:

```
def f(x):
    # Make local variables from globals.
    _print = _print_()
    _write = _write_
    _getattr = _getattr_
    _getitem = _getitem_

    # Translation of f(x) above
    _write(x).foo = _getattr(x, 'foo') + _getitem(x, 0)
```

(continues on next page)

(continued from previous page)

```
    print >>_print, x
    return _print()
```

2.3.6 Examples

print

To support the `print` statement in restricted code, we supply a `_print_` object (note that it's a *factory*, e.g. a class or a callable, from which the restricted machinery will create the object):

```
>>> from RestrictedPython.PrintCollector import PrintCollector
>>> _print_ = PrintCollector
>>> _getattr_ = getattr

>>> src = '''
... print("Hello World!")
... '''
>>> code = compile_restricted(src, '<string>', 'exec')
>>> exec(code)
```

As you can see, the text doesn't appear on stdout. The print collector collects it. We can have access to the text using the `printed` variable, though:

```
>>> src = '''
... print("Hello World!")
... result = printed
... '''
>>> code = compile_restricted(src, '<string>', 'exec')
>>> exec(code)

>>> result
'Hello World!\n'
```

Built-ins

By supplying a different `__builtins__` dictionary, we can rule out unsafe operations, such as opening files:

```
>>> from RestrictedPython.Guards import safe_builtins
>>> restricted_globals = dict(__builtins__=safe_builtins)

>>> src = '''
... open('/etc/passwd')
... '''
>>> code = compile_restricted(src, '<string>', 'exec')
>>> exec(code, restricted_globals)
Traceback (most recent call last):
  ...
NameError: name 'open' is not defined
```

Guards

Here's an example of a write guard that never lets restricted code modify (assign, delete an attribute or item) except dictionaries and lists:

```
>>> from RestrictedPython.Guards import full_write_guard
>>> _write_ = full_write_guard
>>> _getattr_ = getattr

>>> class BikeShed(object):
...     colour = 'green'
...
>>> shed = BikeShed()
```

Normally accessing attributes works as expected, because we're using the standard `getattr` function for the `_getattr_` guard:

```
>>> src = '''
... print(shed.colour)
... result = printed
... '''
>>> code = compile_restricted(src, '<string>', 'exec')
>>> exec(code)

>>> result
'green\n'
```

However, changing an attribute doesn't work:

```
>>> src = '''
... shed.colour = 'red'
... '''
>>> code = compile_restricted(src, '<string>', 'exec')
>>> exec(code)
Traceback (most recent call last):
  ...
TypeError: attribute-less object (assign or del)
```

As said, this particular write guard (`full_write_guard`) will allow restricted code to modify lists and dictionaries:

```
>>> fibonacci = [1, 1, 2, 3, 4]
>>> transl = dict(one=1, two=2, tres=3)
>>> src = '''
... # correct mistake in list
... fibonacci[-1] = 5
... # one item doesn't belong
... del transl['tres']
... '''
>>> code = compile_restricted(src, '<string>', 'exec')
>>> exec(code)

>>> fibonacci
[1, 1, 2, 3, 5]

>>> sorted(transl.keys())
['one', 'two']
```

2.4 API overview

2.4.1 `compile_restricted` methods

`RestrictedPython.`**`compile_restricted`**(*source*, *filename*, *mode*, *flags*, *dont_inherit*, *policy*)
Compiles source code into interpretable byte code.

Parameters

- **`source`** (str or unicode text or `ast.AST`) – (required). the source code that should be compiled
- **`filename`** (*`str or unicode text`*) – (optional). defaults to `'<unknown>'`
- **`mode`** (*`str or unicode text`*) – (optional). Use `'exec'`, `'eval'`, `'single'` or `'function'`. defaults to `'exec'`
- **`flags`** (*`int`*) – (optional). defaults to `0`
- **`dont_inherit`** (*`int`*) – (optional). defaults to `False`
- **`policy`** (*`RestrictingNodeTransformer class`*) – (optional). defaults to `RestrictingNodeTransformer`

Returns Python `code` object

`RestrictedPython.`**`compile_restricted_exec`**(*source*, *filename*, *flags*, *dont_inherit*, *policy*)
Compiles source code into interpretable byte code with `mode='exec'`. Use mode `'exec'` if the source contains a sequence of statements. The meaning and defaults of the parameters are the same as in `compile_restricted`.

Returns CompileResult (a namedtuple with code, errors, warnings, used_names)

`RestrictedPython.`**`compile_restricted_eval`**(*source*, *filename*, *flags*, *dont_inherit*, *policy*)
Compiles source code into interpretable byte code with `mode='eval'`. Use mode `'eval'` if the source contains a single expression. The meaning and defaults of the parameters are the same as in `compile_restricted`.

Returns CompileResult (a namedtuple with code, errors, warnings, used_names)

`RestrictedPython.`**`compile_restricted_single`**(*source*, *filename*, *flags*, *dont_inherit*, *policy*)
Compiles source code into interpretable byte code with `mode='eval'`. Use mode `'single'` if the source contains a single interactive statement. The meaning and defaults of the parameters are the same as in `compile_restricted`.

Returns CompileResult (a namedtuple with code, errors, warnings, used_names)

`RestrictedPython.`**`compile_restricted_function`**(*p*, *body*, *name*, *filename*, *globalize=None*)
Compiles source code into interpretable byte code with `mode='function'`. Use mode `'function'` for full functions.

Parameters

- **`p`** (*`str or unicode text`*) – (required). a string representing the function parameters
- **`body`** (*`str or unicode text`*) – (required). the function body
- **`name`** (*`str or unicode text`*) – (required). the function name
- **`filename`** (*`str or unicode text`*) – (optional). defaults to `'<string>'`
- **`globalize`** (*`None or list`*) – (optional). list of globals. defaults to `None`
- **`flags`** (*`int`*) – (optional). defaults to `0`

- **dont_inherit** (*int*) – (optional). defaults to `False`
- **policy** (*RestrictingNodeTransformer class*) – (optional). defaults to `RestrictingNodeTransformer`

Returns byte code

The globalize argument, if specified, is a list of variable names to be treated as globals (code is generated as if each name in the list appeared in a global statement at the top of the function). This allows to inject global variables into the generated function that feel like they are local variables, so the programmer who uses this doesn't have to understand that his code is executed inside a function scope instead of the global scope of a module.

To actually get an executable function, you need to execute this code and pull out the defined function out of the locals like this:

```
>>> from RestrictedPython import compile_restricted_function
>>> compiled = compile_restricted_function('', 'pass', 'function_name')
>>> safe_locals = {}
>>> safe_globals = {}
>>> exec(compiled.code, safe_globals, safe_locals)
>>> compiled_function = safe_locals['function_name']
>>> result = compiled_function(*[], **{})
```

Then if you want to control the globals for a specific call to this function, you can regenerate the function like this:

```
>>> my_call_specific_global_bindings = dict(foo='bar')
>>> safe_globals = safe_globals.copy()
>>> safe_globals.update(my_call_specific_global_bindings)
>>> import types
>>> new_function = types.FunctionType(
...     compiled_function.__code__,
...     safe_globals,
...     '<function_name>',
...     compiled_function.__defaults__ or ())
>>> result = new_function(*[], **{})
```

2.4.2 restricted builtins

- `safe_globals`
- `safe_builtins`
- `limited_builtins`
- `utility_builtins`

2.4.3 helper modules

- `PrintCollector`

2.4.4 RestrictingNodeTransformer

`RestrictingNodeTransformer` provides the base policy used by RestrictedPython itself.

It is a subclass of a `NodeTransformer` which has a set of `visit_<AST_Elem>` methods and a `generic_visit` method.

`generic_visit` is a predefined method of any `NodeVisitor` which sequentially visits all sub nodes. In RestrictedPython this behaviour is overwritten to always call a new internal method `not_allowed(node)`. This results in an implicit blacklisting of all not allowed AST elements.

Any possibly new introduced AST element in Python (new language element) will implicitly be blocked and not allowed in RestrictedPython.

So, if new elements should be introduced, an explicit `visit_<new AST elem>` is necessary.

2.5 Upgrade from 3.x

For packages that use RestrictedPython the upgrade path differs on the actual usage. If it uses pure RestrictedPython without any additional checks it should be enough to check the imports. RestrictedPython did move some of the imports to the base namespace, so you should only import directly from `RestrictedPython`.

- compile_restricted methods:
  - `from RestrictedPython import compile_restricted`
  - `from RestrictedPython import compile_restricted_eval`
  - `from RestrictedPython import compile_restricted_exec`
  - `from RestrictedPython import compile_restricted_function`
  - `from RestrictedPython import compile_restricted_single`
- predefined built-ins:
  - `from RestrictedPython import safe_globals`
  - `from RestrictedPython import safe_builtins`
  - `from RestrictedPython import limited_builtins`
  - `from RestrictedPython import utility_builtins`
- helper methods:
  - `from RestrictedPython import PrintCollector`

Any import from `RestrictedPython.RCompile` indicates that there have been advanced checks implemented. Those advanced checks where implemented via a `MutatingWalker`. Any checks needs to be reimplemented as a subclass of `RestrictingNodeTransformer`.

2.6 Roadmap

2.6.1 RestrictedPython 4.0

A feature complete rewrite of RestrictedPython using `ast` module instead of `compile` package. RestrictedPython 4.0 should not add any new or remove restrictions.

A detailed documentation that support usage and further development.

Full code coverage tests.

Todo: Complete documentation of all public API elements with docstyle comments https://www.python.org/dev/peps/pep-0257/ http://www.sphinx-doc.org/en/stable/ext/autodoc.html http://thomas-cokelaer.info/tutorials/sphinx/docstring_python.html

Todo: Resolve Discussion in https://github.com/zopefoundation/RestrictedPython/pull/39#issuecomment-283074699

compile_restricted optional params flags and dont_inherit will not work as expected with the current implementation.

stephan-hof did propose a solution, should be discussed and if approved implemented.

2.6.2 RestrictedPython 4.1+

Enhance RestrictedPython, declare deprecations and possible new restrictions.

2.6.3 RestrictedPython 5.0+

- Python 3+ only, no more support for Python 2.7
- mypy - Static Code Analysis Annotations

2.7 Contributing

Contributing to RestrictedPython

2.7.1 Todos

Todo: Complete documentation of all public API elements with docstyle comments https://www.python.org/dev/peps/pep-0257/ http://www.sphinx-doc.org/en/stable/ext/autodoc.html http://thomas-cokelaer.info/tutorials/sphinx/docstring_python.html

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/stable/docs/roadmap/inde
line 14.)

Todo: Resolve Discussion in https://github.com/zopefoundation/RestrictedPython/pull/39#issuecomment-283074699

compile_restricted optional params flags and dont_inherit will not work as expected with the current implementation.

stephan-hof did propose a solution, should be discussed and if approved implemented.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/stable/docs/roadmap/inde
line 21.)

Todo: Describe Guards and predefined guard methods in details

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/stable/docs/usage/policy.
line 28.)

Todo: Describe full_write_guard more in detail and how it works.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/stable/docs/usage/policy.
line 42.)

2.7.2 Understanding How RestrictedPython works internally

RestrictedPython is a classic approach of compiler construction to create a limited subset of an existing programming language.

Defining a programming language requires a regular grammar (Chomsky 3 / EBNF) definition. This grammar will be implemented in an abstract syntax tree (AST), which will be passed on to a code generator to produce a machine-readable version.

Code generation

As Python is a platform independent programming language, this machine readable version is a byte code which will be translated on the fly by an interpreter into machine code. This machine code then gets executed on the specific CPU architecture, with the standard operating system restrictions.

The byte code produced must be compatible with the execution environment that the Python interpreter is running in, so we do not generate the byte code directly from `compile_restricted` and the other `compile_restricted_*` methods manually, it may not match what the interpreter expects.

Thankfully, the Python `compile()` function introduced the capability to compile `ast.AST` code into byte code in Python 2.6, so we can return the platform-independent AST and keep byte code generation delegated to the interpreter.

`ast` module (Abstract Syntax Trees)

The `ast` module consists of four areas:

- `AST` (Basis of all Nodes) + all node class implementations
- `NodeVisitor` and `NodeTransformer` (tool to consume and modify the AST)
- Helper methods
  - `parse`
  - `walk`
  - `dump`
- Constants
  - `PyCF_ONLY_AST`

`NodeVisitor` & `NodeTransformer`

A `NodeVisitor` is a class of a node / AST consumer, it reads the data by stepping through the tree without modifying it. In contrast, a `NodeTransformer` (which inherits from a `NodeVisitor`) is allowed to modify the tree and nodes.

Technical Backgrounds - Links to External Documentation

- Concept of Immutable Types and Python Example
- Python 3 Standard Library Documentation on AST module
  - AST Grammar of Python
    - Python 3.8 AST
    - Python 3.7 AST
    - Python 3.6 AST
    - Python 3.5 AST
    - Python 3.4 AST (obsolete)
    - Python 3.3 AST (obsolete)
    - Python 3.2 AST (obsolete)
    - Python 3.1 AST (obsolete)
    - Python 3.0 AST (obsolete)
    - Python 2.7 AST
    - Python 2.6 AST (obsolete)
  - AST NodeVistiors Class (https://docs.python.org/3.5/library/ast.html#ast.NodeVisitor)
  - AST NodeTransformer Class (https://docs.python.org/3.5/library/ast.html#ast.NodeTransformer)
  - AST dump method (https://docs.python.org/3.5/library/ast.html#ast.dump)
- In detail Documentation on the Python AST module (Green Tree Snakes)
- Example how to Instrumenting the Python AST

2.7.3 Differences between different Python versions

A (modified style) Copy of all Abstract Grammar Definitions for the Python versions does live in this Documentation (ast Subfolder) to help finding difference quicker by comparing files.

Changes from Python 2.6 to Python 2.7

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python2_6.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python2_7.ast
@@ -1,7 +1,7 @@
--- Python 2.6 AST
+-- Python 2.7 AST
 -- ASDL's five builtin types are identifier, int, string, object, bool

-module Python version "2.6"
+module Python version "2.7"
 {
   mod = Module(stmt* body)
       | Interactive(stmt* body)
```

(continues on next page)

(continued from previous page)

```
@@ -14,7 +14,7 @@
                       arguments args,
                       stmt* body,
                       expr* decorator_list)
-         | ClassDef(identifier name, expr* bases, stmt* body, expr *decorator_list)
+         | ClassDef(identifier name, expr* bases, stmt* body, expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
@@ -37,7 +37,7 @@
          | Assert(expr test, expr? msg)

          | Import(alias* names)
-         | ImportFrom(identifier module, alias* names, int? level)
+         | ImportFrom(identifier? module, alias* names, int? level)

          -- Doesn't capture requirement that locals must be
          -- defined if globals is
@@ -59,7 +59,10 @@
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
+        | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
+        | SetComp(expr elt, comprehension* generators)
+        | DictComp(expr key, expr value, comprehension* generators)
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can occur
         | Yield(expr? value)
```

Changes from Python 3.0 to Python 3.1

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_0.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_1.ast
@@ -1,7 +1,7 @@
--- Python 3.0 AST
+-- Python 3.1 AST
 -- ASDL's four builtin types are identifier, int, string, object

-module Python version "3.0"
+module Python version "3.1"
 {
     mod = Module(stmt* body)
         | Interactive(stmt* body)
```

Changes from Python 3.1 to Python 3.2

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_1.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_2.ast
```

(continues on next page)

(continued from previous page)

```
@@ -1,7 +1,7 @@
--- Python 3.1 AST
+-- Python 3.2 AST
 -- ASDL's four builtin types are identifier, int, string, object

-module Python version "3.1"
+module Python version "3.2"
 {
     mod = Module(stmt* body)
         | Interactive(stmt* body)
@@ -21,7 +21,7 @@
                      expr? starargs,
                      expr? kwargs,
                      stmt* body,
-                     expr *decorator_list)
+                     expr* decorator_list)
           | Return(expr? value)

           | Delete(expr* targets)
@@ -40,7 +40,7 @@
           | Assert(expr test, expr? msg)

           | Import(alias* names)
-          | ImportFrom(identifier module, alias* names, int? level)
+          | ImportFrom(identifier? module, alias* names, int? level)

           | Global(identifier* names)
           | Nonlocal(identifier* names)
```

Changes from Python 3.2 to Python 3.3

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_2.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_3.ast
@@ -1,7 +1,7 @@
--- Python 3.2 AST
--- ASDL's four builtin types are identifier, int, string, object
+-- PYTHON 3.3 AST
+-- ASDL's five builtin types are identifier, int, string, bytes, object

-module Python version "3.2"
+module Python version "3.3"
 {
     mod = Module(stmt* body)
         | Interactive(stmt* body)
@@ -32,11 +32,10 @@
           | For(expr target, expr iter, stmt* body, stmt* orelse)
           | While(expr test, stmt* body, stmt* orelse)
           | If(expr test, stmt* body, stmt* orelse)
-          | With(expr context_expr, expr? optional_vars, stmt* body)
+          | With(withitem* items, stmt* body)

           | Raise(expr? exc, expr? cause)
-          | TryExcept(stmt* body, excepthandler* handlers, stmt* orelse)
```

(continues on next page)

(continued from previous page)

```
-          | TryFinally(stmt* body, stmt* finalbody)
+          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
           | Assert(expr test, expr? msg)

           | Import(alias* names)
@@ -67,6 +66,7 @@
           | GeneratorExp(expr elt, comprehension* generators)
           -- the grammar constrains where yield expressions can occur
           | Yield(expr? value)
+          | YieldFrom(expr value)
           -- need sequences for compare to distinguish between
           -- x < 4 < 3 and (x < 4) < 3
           | Compare(expr left, cmpop* ops, expr* comparators)
@@ -77,7 +77,7 @@
                  expr? kwargs)
           | Num(object n) -- a number as a PyObject.
           | Str(string s) -- need to specify raw, unicode, etc?
-          | Bytes(string s)
+          | Bytes(bytes s)
           | Ellipsis
           -- other literals? bools?

@@ -137,7 +137,6 @@

     comprehension = (expr target, expr iter, expr* ifs)

-    -- not sure what to call the first argument for raise and except
     excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                     attributes (int lineno, int col_offset)

@@ -156,4 +155,6 @@

     -- import name with optional 'as' alias.
     alias = (identifier name, identifier? asname)
+
+    withitem = (expr context_expr, expr? optional_vars)
 }
```

Changes from Python 3.3 to Python 3.4

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_3.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_4.ast
@@ -1,7 +1,7 @@
--- PYTHON 3.3 AST
--- ASDL's five builtin types are identifier, int, string, bytes, object
+-- Python 3.4 AST
+-- ASDL's six builtin types are identifier, int, string, bytes, object, singleton

-module Python version "3.3"
+module Python version "3.4"
 {
     mod = Module(stmt* body)
         | Interactive(stmt* body)
```

(continues on next page)

(continued from previous page)

```
@@ -78,8 +78,8 @@
          | Num(object n) -- a number as a PyObject.
          | Str(string s) -- need to specify raw, unicode, etc?
          | Bytes(bytes s)
+         | NameConstant(singleton value)
          | Ellipsis
-         -- other literals? bools?

          -- the following expression can appear in assignment context
          | Attribute(expr value, identifier attr, expr_context ctx)
@@ -141,14 +141,14 @@
                       attributes (int lineno, int col_offset)

     arguments = (arg* args,
-                 identifier? vararg,
-                 expr? varargannotation,
+                 arg? vararg,
                  arg* kwonlyargs,
-                 identifier? kwarg,
-                 expr? kwargannotation,
-                 expr* defaults,
-                 expr* kw_defaults)
+                 expr* kw_defaults,
+                 arg? kwarg,
+                 expr* defaults)
+
     arg = (identifier arg, expr? annotation)
+          attributes (int lineno, int col_offset)

     -- keyword arguments supplied to call
     keyword = (identifier arg, expr value)
```

Changes from Python 3.4 to Python 3.5

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_4.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_5.ast
@@ -1,7 +1,7 @@
--- Python 3.4 AST
+-- Python 3.5 AST
 -- ASDL's six builtin types are identifier, int, string, bytes, object, singleton

-module Python version "3.4"
+module Python version "3.5"
 {
     mod = Module(stmt* body)
         | Interactive(stmt* body)
@@ -15,11 +15,15 @@
                          stmt* body,
                          expr* decorator_list,
                          expr? returns)
+         | AsyncFunctionDef(identifier name,
+                            arguments args,
+                            stmt* body,
```

(continues on next page)

(continued from previous page)

```
+                           expr* decorator_list,
+                           expr? returns)
+
          | ClassDef(identifier name,
                     expr* bases,
                     keyword* keywords,
-                    expr? starargs,
-                    expr? kwargs,
                     stmt* body,
                     expr* decorator_list)
          | Return(expr? value)
@@ -30,9 +34,11 @@

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse)
+         | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body)
+         | AsyncWith(withitem* items, stmt* body)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
@@ -65,6 +71,7 @@
          | DictComp(expr key, expr value, comprehension* generators)
          | GeneratorExp(expr elt, comprehension* generators)
          -- the grammar constrains where yield expressions can occur
+         | Await(expr value)
          | Yield(expr? value)
          | YieldFrom(expr value)
          -- need sequences for compare to distinguish between
@@ -72,9 +79,7 @@
          | Compare(expr left, cmpop* ops, expr* comparators)
          | Call(expr func,
                 expr* args,
-                keyword* keywords,
-                expr? starargs,
-                expr? kwargs)
+                keyword* keywords)
          | Num(object n) -- a number as a PyObject.
          | Str(string s) -- need to specify raw, unicode, etc?
          | Bytes(bytes s)
@@ -109,6 +114,7 @@
     operator = Add
              | Sub
              | Mult
+             | MatMult
              | Div
              | Mod
              | Pow
@@ -150,8 +156,8 @@
     arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

-    -- keyword arguments supplied to call
-    keyword = (identifier arg, expr value)
+    -- keyword arguments supplied to call (NULL identifier for **kwargs)
```

(continues on next page)

(continued from previous page)

```
+    keyword = (identifier? arg, expr value)

     -- import name with optional 'as' alias.
     alias = (identifier name, identifier? asname)
```

Changes from Python 3.5 to Python 3.6

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_5.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_6.ast
@@ -1,7 +1,11 @@
--- Python 3.5 AST
--- ASDL's six builtin types are identifier, int, string, bytes, object, singleton
+-- Python 3.6 AST
+-- ASDL's 7 builtin types are:
+-- identifier, int, string, bytes, object, singleton, constant
+--
+-- singleton: None, True or False
+-- constant can be None, whereas None means "no value" for object.

-module Python version "3.5"
+module Python version "3.6"
 {
     mod = Module(stmt* body)
         | Interactive(stmt* body)
@@ -31,6 +35,8 @@
          | Delete(expr* targets)
          | Assign(expr* targets, expr value)
          | AugAssign(expr target, operator op, expr value)
+         -- 'simple' indicates that we annotate simple name without parens
+         | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse)
@@ -82,9 +88,12 @@
                  keyword* keywords)
          | Num(object n) -- a number as a PyObject.
          | Str(string s) -- need to specify raw, unicode, etc?
+         | FormattedValue(expr value, int? conversion, expr? format_spec)
+         | JoinedStr(expr* values)
          | Bytes(bytes s)
          | NameConstant(singleton value)
          | Ellipsis
+         | Constant(constant value)

          -- the following expression can appear in assignment context
          | Attribute(expr value, identifier attr, expr_context ctx)
@@ -141,7 +150,7 @@
           | In
           | NotIn

-    comprehension = (expr target, expr iter, expr* ifs)
+    comprehension = (expr target, expr iter, expr* ifs, int is_async)
```

(continues on next page)

(continued from previous page)

```
     excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                     attributes (int lineno, int col_offset)
```

Changes from Python 3.6 to Python 3.7

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_6.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_7.ast
@@ -1,11 +1,11 @@
--- Python 3.6 AST
+-- Python 3.7 AST
 -- ASDL's 7 builtin types are:
 -- identifier, int, string, bytes, object, singleton, constant
 --
 -- singleton: None, True or False
 -- constant can be None, whereas None means "no value" for object.

-module Python version "3.6"
+module Python version "3.7"
 {
     mod = Module(stmt* body)
         | Interactive(stmt* body)
```

Changes from Python 3.7 to Python 3.8

```
--- /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_7.ast
+++ /home/docs/checkouts/readthedocs.org/user_builds/restrictedpython/checkouts/
↪stable/docs/contributing/ast/python3_8.ast
@@ -1,15 +1,13 @@
--- Python 3.7 AST
--- ASDL's 7 builtin types are:
--- identifier, int, string, bytes, object, singleton, constant
---
--- singleton: None, True or False
--- constant can be None, whereas None means "no value" for object.
+-- Python 3.8 AST
+-- ASDL's 5 builtin types are:
+-- identifier, int, string, object, constant

-module Python version "3.7"
+module Python version "3.8"
 {
-    mod = Module(stmt* body)
+    mod = Module(stmt* body, type_ignore *type_ignores)
         | Interactive(stmt* body)
         | Expression(expr body)
+        | FunctionType(expr* argtypes, expr returns)

         -- not really an actual node but useful in Jython's typesystem.
         | Suite(stmt* body)
@@ -18,12 +16,14 @@
```

(continues on next page)

(continued from previous page)

```
                        arguments args,
                        stmt* body,
                        expr* decorator_list,
-                       expr? returns)
+                       expr? returns,
+                       string? type_comment)
          | AsyncFunctionDef(identifier name,
                             arguments args,
                             stmt* body,
                             expr* decorator_list,
-                            expr? returns)
+                            expr? returns,
+                            string? type_comment)

          | ClassDef(identifier name,
                     expr* bases,
@@ -33,18 +33,18 @@
          | Return(expr? value)

          | Delete(expr* targets)
-         | Assign(expr* targets, expr value)
+         | Assign(expr* targets, expr value, string? type_comment)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
-         | For(expr target, expr iter, stmt* body, stmt* orelse)
-         | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
+         | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
+         | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
-         | With(withitem* items, stmt* body)
-         | AsyncWith(withitem* items, stmt* body)
+         | With(withitem* items, stmt* body, string? type_comment)
+         | AsyncWith(withitem* items, stmt* body, string? type_comment)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
@@ -62,10 +62,11 @@

          -- XXX Jython will be different
          -- col_offset is the byte offset in the utf8 string the parser uses
-         attributes (int lineno, int col_offset)
+         attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

          -- BoolOp() can use left & right?
     expr = BoolOp(boolop op, expr* values)
+          | NamedExpr(expr target, expr value)
          | BinOp(expr left, operator op, expr right)
          | UnaryOp(unaryop op, expr operand)
          | Lambda(arguments args, expr body)
@@ -86,14 +87,9 @@
```

(continues on next page)

(continued from previous page)

```
          | Call(expr func,
                 expr* args,
                 keyword* keywords)
-         | Num(object n) -- a number as a PyObject.
-         | Str(string s) -- need to specify raw, unicode, etc?
          | FormattedValue(expr value, int? conversion, expr? format_spec)
          | JoinedStr(expr* values)
-         | Bytes(bytes s)
-         | NameConstant(singleton value)
-         | Ellipsis
-         | Constant(constant value)
+         | Constant(constant value, string? kind)

          -- the following expression can appear in assignment context
          | Attribute(expr value, identifier attr, expr_context ctx)
@@ -104,7 +100,7 @@
          | Tuple(expr* elts, expr_context ctx)

          -- col_offset is the byte offset in the utf8 string the parser uses
-         attributes (int lineno, int col_offset)
+         attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

     expr_context = Load
                  | Store
@@ -153,17 +149,18 @@
     comprehension = (expr target, expr iter, expr* ifs, int is_async)

     excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
-                    attributes (int lineno, int col_offset)
+                    attributes (int lineno, int col_offset, int? end_lineno, int?␣
↪end_col_offset)

-    arguments = (arg* args,
+    arguments = (arg* posonlyargs,
+                 arg* args,
                  arg? vararg,
                  arg* kwonlyargs,
                  expr* kw_defaults,
                  arg? kwarg,
                  expr* defaults)

-    arg = (identifier arg, expr? annotation)
-          attributes (int lineno, int col_offset)
+    arg = (identifier arg, expr? annotation, string? type_comment)
+          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

     -- keyword arguments supplied to call (NULL identifier for **kwargs)
     keyword = (identifier? arg, expr value)
@@ -172,4 +169,6 @@
     alias = (identifier name, identifier? asname)

     withitem = (expr context_expr, expr? optional_vars)
+
+    type_ignore = TypeIgnore(int lineno, string tag)
 }
```

2.8 Changes

2.8.1 5.1 (2020-10-07)

Features

- Add support for (Python 3.8+) assignment expressions (i.e. the `:=` operator)
- Add support for Python 3.9 after checking the security implications of the syntax changes made in that version.
- Add support for the `bytes` and `sorted` builtins (#186)

Documentation

- Document parameter `mode` for the `compile_restricted` functions (#157)
- Fix documentation for `compile_restricted_function` (#158)

Fixes

- Fix `compile_restricted_function` with SyntaxErrors that have no text (#181)
- Drop install dependency on `setuptools`. (#189)

2.8.2 5.0 (2019-09-03)

Breaking changes

- Revert the Allowance of the `...` (Ellipsis) statement, as of 4.0. It is not needed to support Python 3.8. The security implications of the Ellipsis Statement is not 100 % clear and is not checked. `...` (Ellipsis) is disallowed again.

Features

- Add support for f-strings in Python 3.6+. (#123)

2.8.3 4.0 (2019-05-10)

Changes since 3.6.0:

Breaking changes

- The `compile_restricted*` functions now return a `namedtuple CompileResult` instead of a simple `tuple`.
- Drop the old implementation of version 3.x: *RCompile.py*, *SelectCompiler.py*, *MutatingWorker.py*, *RestrictionMutator.py* and *tests/verify.py*.
- Drop support for long-deprecated `sets` module.

Security related issues

- RestrictedPython now ships with a default implementation for `_getattr_` which prevents from using the `format()` method on str/unicode as it is not safe, see: http://lucumr.pocoo.org/2016/12/29/careful-with-str-format/

  Caution: If you do not already have secured the access to this `format()` method in your `_getattr_` implementation use `RestrictedPython.Guards.safer_getattr()` in your implementation to benefit from this fix.

Features

- Mostly complete rewrite based on Python AST module. [loechel (Alexander Loechel), icemac (Michael Howitz), stephan-hof (Stephan Hofmockel), tlotze (Thomas Lotze)]
- Add support for Python 3.5, 3.6, 3.7.
- Add preliminary support for Python 3.8. as of 3.8.0a3 is released.
- Warn when using another Python implementation than CPython as it is not safe to use RestrictedPython with other versions than CPyton. See https://bitbucket.org/pypy/pypy/issues/2653 for PyPy.
- Allow the `...` (Ellipsis) statement. It is needed to support Python 3.8.
- Allow *yield* and *yield from* statements. Generator functions would now work in RestrictedPython.
- Allow the following magic methods to be defined on classes. (#104) They cannot be called directly but by the built-in way to use them (e. g. class instantiation, or comparison):
  - `__init__`
  - `__contains__`
  - `__lt__`
  - `__le__`
  - `__eq__`
  - `__ne__`
  - `__gt__`
  - `__ge__`
- Imports like `from a import *` (so called star imports) are now forbidden as they allow to import names starting with an underscore which could override protected build-ins. (#102)
- Allow to use list comprehensions in the default implementation of `RestrictionCapableEval.eval()`.
- Switch to pytest as test runner.
- Bring test coverage to 100 %.

Bug fixes

- Improve *.Guards.safer_getattr* to prevent accessing names starting with underscore. (#142)

2.8.4 3.6.0 (2010-07-09)

- Add name check for names assigned during imports using the `from x import y` format.
- Add test for name check when assigning an alias using multiple-context `with` statements in Python 2.7.
- Add tests for protection of the iterators for dict and set comprehensions in Python 2.7.

2.8.5 3.6.0a1 (2010-06-05)

- Remove support for `DocumentTemplate.sequence` - this is handled in the DocumentTemplate package itself.

2.8.6 3.5.2 (2010-04-30)

- Remove a testing dependency on `zope.testing`.

2.8.7 3.5.1 (2009-03-17)

- Add tests for `Utilities` module.
- Filter DeprecationWarnings when importing Python's `sets` module.

2.8.8 3.5.0 (2009-02-09)

- Drop legacy support for Python 2.1 / 2.2 (`__future__` imports of `nested_scopes` / `generators`.).

2.8.9 3.4.3 (2008-10-26)

- Fix deprecation warning: `with` is now a reserved keyword on Python 2.6. That means RestrictedPython should run on Python 2.6 now. Thanks to Ranjith Kannikara, GSoC Student for the patch.
- Add tests for ternary if expression and for `with` keyword and context managers.

2.8.10 3.4.2 (2007-07-28)

- Changed homepage URL to the PyPI site
- Improve `README.txt`.

2.8.11 3.4.1 (2007-06-23)

- Fix http://www.zope.org/Collectors/Zope/2295: Bare conditional in a Zope 2 PythonScript followed by a comment causes SyntaxError.

2.8.12 3.4.0 (2007-06-04)

- RestrictedPython now has its own release cycle as a separate project.
- Synchronized with RestrictedPython from Zope 2 tree.

2.8.13 3.2.0 (2006-01-05)

- Corresponds to the verison of the RestrictedPython package shipped as part of the Zope 3.2.0 release.
- No changes from 3.1.0.

2.8.14 3.1.0 (2005-10-03)

- Corresponds to the verison of the RestrictedPython package shipped as part of the Zope 3.1.0 release.
- Remove unused fossil module, `SafeMapping`.
- Replaced use of deprecated `whrandom` module with `random` (aliased to `whrandom` for backward compatibility).

2.8.15 3.0.0 (2004-11-07)

- Corresponds to the verison of the RestrictedPython package shipped as part of the Zope X3.0.0 release.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

C

compile_restricted() (*in module RestrictedPython*), 14
compile_restricted_eval() (*in module RestrictedPython*), 14
compile_restricted_exec() (*in module RestrictedPython*), 14
compile_restricted_function() (*in module RestrictedPython*), 14
compile_restricted_single() (*in module RestrictedPython*), 14