5.16 0.5.3 (Aug 1, 2014) .......................................................... 23
5.17 0.5.2 (Jul 24, 2014) .......................................................... 23
5.18 0.5.1 (Mar 4, 2014) .......................................................... 23
5.19 0.5 (Feb 17, 2014) .......................................................... 23
5.20 0.4.5 (Dec 16, 2013) ....................................................... 24
5.21 0.4.4 (Nov 20, 2013) ....................................................... 24
5.22 0.4.2 (Jun 25, 2013) .......................................................... 24
5.23 0.4.1 (Jun 16, 2013) .......................................................... 24
5.24 0.4 (Apr 26, 2013) .......................................................... 24
5.25 0.3 (Nov 2, 2012) ............................................................ 24
5.26 0.2 (Oct 11, 2012) ............................................................ 24
5.27 0.1 (Never) ................................................................. 25

6 Indices and tables ............................................................. 27

Bibliography ................................................................. 29

Python Module Index ......................................................... 31
Radon is a Python tool which computes various code metrics. Supported metrics are:

- raw metrics: SLOC, comment lines, blank lines, &c.
- Cyclomatic Complexity (i.e. McCabe’s Complexity)
- Halstead metrics (all of them)
- the Maintainability Index (a Visual Studio metric)

Radon can be used either from the command line or programatically through its API.
Introduction to Code Metrics

This section contains a brief explanations of the metrics that Radon can compute.

1.1 Cyclomatic Complexity

Cyclomatic Complexity corresponds to the number of decisions a block of code contains plus 1. This number (also called McCabe number) is equal to the number of linearly independent paths through the code. This number can be used as a guide when testing conditional logic in blocks.

Radon analyzes the AST tree of a Python program to compute Cyclomatic Complexity. Statements have the following effects on Cyclomatic Complexity:

<table>
<thead>
<tr>
<th>Construct</th>
<th>Effect on CC</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>+1</td>
<td>An if statement is a single decision.</td>
</tr>
<tr>
<td>elif</td>
<td>+1</td>
<td>The elif statement adds another decision.</td>
</tr>
<tr>
<td>else</td>
<td>+0</td>
<td>The else statement does not cause a new decision. The decision is at the if.</td>
</tr>
<tr>
<td>for</td>
<td>+1</td>
<td>There is a decision at the start of the loop.</td>
</tr>
<tr>
<td>while</td>
<td>+1</td>
<td>There is a decision at the while statement.</td>
</tr>
<tr>
<td>except</td>
<td>+1</td>
<td>Each except branch adds a new conditional path of execution.</td>
</tr>
<tr>
<td>finally</td>
<td>+0</td>
<td>The finally block is unconditionally executed.</td>
</tr>
<tr>
<td>with</td>
<td>+1</td>
<td>The with statement roughly corresponds to a try/except block (see PEP 343 for details).</td>
</tr>
<tr>
<td>assert</td>
<td>+1</td>
<td>The assert statement internally roughly equals a conditional statement.</td>
</tr>
<tr>
<td>Comprehension</td>
<td>+1</td>
<td>A list/set/dict comprehension of generator expression is equivalent to a for loop.</td>
</tr>
<tr>
<td>Boolean Operator</td>
<td>+1</td>
<td>Every boolean operator (and, or) adds a decision point.</td>
</tr>
</tbody>
</table>
1.2 Maintainability Index

Maintainability Index is a software metric which measures how maintainable (easy to support and change) the source code is. The maintainability index is calculated as a factored formula consisting of SLOC (Source Lines Of Code), Cyclomatic Complexity and Halstead volume. It is used in several automated software metric tools, including the Microsoft Visual Studio 2010 development environment, which uses a shifted scale (0 to 100) derivative.

Common formulas are:

- the original formula:
  \[ MI = 171 - 5.2 \ln V - 0.23G - 16.2 \ln L \]

- the derivative used by SEI:
  \[ MI = 171 - 5.2 \log_2 V - 0.23G - 16.2 \log_2 L + 50 \sin(\sqrt{2.4C}) \]

- the derivative used by Visual Studio:
  \[ MI = \max\left[0, 100 \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L}{171}\right] \]

Radon uses another derivative, computed from both SEI derivative and Visual Studio one:

\[ MI = \max\left[0, 100 \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L + 50 \sin(\sqrt{2.4C})}{171}\right] \]

Where:

- \( V \) is the Halstead Volume (see below);
- \( G \) is the total Cyclomatic Complexity;
- \( L \) is the number of Source Lines of Code (SLOC);
- \( C \) is the percent of comment lines (important: converted to radians).

Note: Maintainability Index is still a very experimental metric, and should not be taken into account as seriously as the other metrics.

1.3 Raw Metrics

The following are the definitions employed by Radon:

- **LOC**: The total number of lines of code. It does not necessarily correspond to the number of lines in the file.
- **LLOC**: The number of logical lines of code. Every logical line of code contains exactly one statement.
- **SLOC**: The number of source lines of code - not necessarily corresponding to the LLOC.
- Comments: The number of comment lines. Multi-line strings are not counted as comment since, to the Python interpreter, they are just strings.
- Multi: The number of lines which represent multi-line strings.
- Blanks: The number of blank lines (or whitespace-only ones).
The equation \( SLOC + \text{Multi} + \text{Single comments} + \text{Blank} = \text{LOC} \) should always hold. Additionally, comment stats are calculated:

- \( C\%L \): the ratio between number of comment lines and \( \text{LOC} \), expressed as a percentage;
- \( C\%S \): the ratio between number of comment lines and \( \text{SLOC} \), expressed as a percentage;
- \( C + M\%L \): the ratio between number of comment and multiline strings lines and \( \text{LOC} \), expressed as a percentage.

### 1.4 Halstead Metrics

Halstead’s goal was to identify measurable properties of software, and the relations between them. These numbers are statically computed from the source code:

- \( \eta_1 \) = the number of distinct operators
- \( \eta_2 \) = the number of distinct operands
- \( N_1 \) = the total number of operators
- \( N_2 \) = the total number of operands

From these numbers several measures can be calculated:

- Program vocabulary: \( \eta = \eta_1 + \eta_2 \)
- Program length: \( N = N_1 + N_2 \)
- Calculated program length: \( \tilde{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \)
- Volume: \( V = N \log_2 \eta \)
- Difficulty: \( D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2} \)
- Effort: \( E = D \cdot V \)
- Time required to program: \( T = \frac{E}{18} \) seconds
- Number of delivered bugs: \( B = \frac{V}{3000} \)

### 1.5 Further Reading

4. Arie van Deursen, Think Twice Before Using the “Maintainability Index”.
Radon currently has three commands:

- **cc**: compute Cyclomatic Complexity
- **raw**: compute raw metrics
- **mi**: compute Maintainability Index

**Note**: On some systems, such as Windows, the default encoding is not UTF-8. If you are using Unicode characters in your Python file and want to analyze it with Radon, you’ll have to set the `RADONFILESENCODING` environment variable to `UTF-8`.

### 2.1 The cc command

This command analyzes Python source files and compute Cyclomatic Complexity. The output can be filtered by specifying the `-n` and `-x` flags. By default, the complexity score is not displayed, the option `-s` (show complexity) toggles this behaviour. File or directories exclusion is supported through glob patterns. Every positional argument is interpreted as a path. The program then walks through its children and analyzes Python files. Every block will be ranked from A (best complexity score) to F (worst one). Ranks corresponds to complexity scores as follows:

<table>
<thead>
<tr>
<th>CC score</th>
<th>Rank</th>
<th>Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 5</td>
<td>A</td>
<td>low - simple block</td>
</tr>
<tr>
<td>6 - 10</td>
<td>B</td>
<td>low - well structured and stable block</td>
</tr>
<tr>
<td>11 - 20</td>
<td>C</td>
<td>moderate - slightly complex block</td>
</tr>
<tr>
<td>21 - 30</td>
<td>D</td>
<td>more than moderate - more complex block</td>
</tr>
<tr>
<td>31 - 40</td>
<td>E</td>
<td>high - complex block, alarming</td>
</tr>
<tr>
<td>41+</td>
<td>F</td>
<td>very high - error-prone, unstable block</td>
</tr>
</tbody>
</table>

Blocks are also classified into three types: functions, methods and classes. They’re listed by letter in the command output for convenience when scanning through a longer list of blocks:
### 2.1.1 Options

- **-x, --max**
  Set the maximum complexity rank to display.

- **-n, --min**
  Set the minimum complexity rank to display.

- **-a, --average**
  If given, at the end of the analysis show the average Cyclomatic Complexity. This option is influenced by -x, --max and -n, --min options.

- **--total-average**
  Like -a, --average, but it is not influenced by min and max. Every analyzed block is counted, no matter whether it is displayed or not.

- **-s, --show-complexity**
  If given, show the complexity score along with its rank.

- **-e, --exclude**
  Exclude files only when their path matches one of these glob patterns. Usually needs quoting at the command line.

- **-i, --ignore**
  Ignore directories when their name matches one of these glob patterns: radon won’t even descend into them. By default, hidden directories (starting with '.') are ignored.

- **-o, --order**
  The ordering function for the results. Can be one of:
  - **SCORE**: order by cyclomatic complexity (descending);
  - **LINES**: order by line numbers;
  - **ALPHA**: order by block names (alphabetically).

- **-j, --json**
  If given, the results will be converted into JSON. This is useful in case you need to export the results to another application.

- **--xml**
  If given, the results will be converted into XML. Note that not all the information is kept. This is specifically targeted to Jenkin’s plugin CCM.

- **--no-assert**
  Does not count assert statements when computing complexity. This is because Python can be run with an optimize flag which removes assert statements.

### 2.1.2 Examples

```
$ radon cc path
```
Radon will walk through the subdirectories of path and will analyze all child nodes (every Python file it encounters).

```bash
$ radon cc -e "path/tests*,path/docs/*" path
```

As in the above example, Radon will walk the directories, excluding paths matching `path/tests/*` and `path/docs/*`.

**Warning:** Remember to quote the patterns, otherwise your shell might expand them!

Depending on the single cases, a more suitable alternative might be this:

```bash
$ radon cc -i "docs,tests" path

$ cat path/to/file.py | radon cc -
```

Setting the path to “`-`” will cause Radon to analyze code from stdin

```bash
$ radon cc --min B --max E path
```

Here Radon will only display blocks ranked between B and E (i.e. from CC=6 to CC=40).

## 2.2 The mi command

This command analyzes Python source code files and compute the Maintainability Index score. Every positional argument is treated as a starting point from which to walk looking for Python files (as in the `cc` command). Paths can be excluded with the `--exclude` option. The Maintainability Index is always in the range 0-100. MI is ranked as follows:

<table>
<thead>
<tr>
<th>MI score</th>
<th>Rank</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 - 20</td>
<td>A</td>
<td>Very high</td>
</tr>
<tr>
<td>19 - 10</td>
<td>B</td>
<td>Medium</td>
</tr>
<tr>
<td>9 - 0</td>
<td>C</td>
<td>Extremely low</td>
</tr>
</tbody>
</table>

### 2.2.1 Options

- `-x`, `--max`
  Set the maximum MI to display.

- `-n`, `--min`
  Set the minimum MI to display.

- `-e`, `--exclude`
  Exclude files only when their path matches one of these glob patterns. Usually needs quoting at the command line.

- `-i`, `--ignore`
  Ignore directories when their name matches one of these glob patterns: radon won’t even descend into them. By default, hidden directories (starting with ‘.’) are ignored.

- `-m`, `--multi`
  If given, Radon will not count multiline strings as comments. Most of the time this is safe since multiline strings are used as functions docstrings, but one should be aware that their use is not limited to that and sometimes it would be wrong to count them as comment lines.

### 2.2. The mi command
-s, --show
If given, the actual MI value is shown in results, alongside the rank.

-j, --json
Format results in JSON.

### 2.2.2 Examples

```bash
$ radon mi path1 path2
```

Analyze every Python file under `path1` or `path2`. It checks recursively in every subdirectory.

```bash
$ radon mi path1 -e "path1/tests/"
```

Like the previous example, but excluding from the analysis every path that matches `path1/tests/*`.

```bash
$ radon mi -m path1
```

Like the previous examples, but does not count multiline strings as comments.

### 2.3 The raw command

This command analyzes the given Python modules in order to compute raw metrics. These include:

- **LOC**: the total number of lines of code
- **LLOC**: the number of logical lines of code
- **SLOC**: the number of source lines of code - not necessarily corresponding to the LLOC [Wikipedia]
- **comments**: the number of Python comment lines (i.e. only single-line comments `#`)
- **multi**: the number of lines representing multi-line strings
- **blank**: the number of blank lines (or whitespace-only ones)

The equation `sloc + multi + singlecomments + blank = loc` should always hold.

### 2.3.1 Options

- **-e**, **--exclude**
  Exclude files only when their path matches one of these glob patterns. Usually needs quoting at the command line.

- **-i**, **--ignore**
  Ignore directories when their name matches one of these glob patterns: radon won’t even descend into them. By default, hidden directories (starting with '.') are ignored.

- **-s**, **--summary**
  If given, at the end of the analysis a summary of the gathered metrics will be shown.

- **-j**, **--json**
  If given, the results will be converted into JSON.
2.3.2 Examples

$ radon raw path1 path2

Analyze every Python file under path1 or path2. It checks recursively in every subdirectory.

$ radon raw path1 -e "path1/tests/*"

Like the previous example, but excluding from the analysis every path that matches path1/tests/*.
Radon exposes a plugin for the Flake8 tool. In order to use it you will have to install both radon and flake8.

To enable the Radon checker, you will have to supply at least one of the following options:

--radon-max-cc <int>
   Set the cyclomatic complexity threshold. The default value is 10. Blocks with a greater complexity will be reported by the tool.

--radon-no-assert
   Instruct radon not to count assert statements towards cyclomatic complexity. The default behaviour is the opposite.

--radon-show-closures
   Instruct radon to add closures/inner classes to the output.

For more information visit the Flake8 documentation.
Using radon programmatically

Radon has a set of functions and classes that you can call from within your program to analyze files. Radon’s API is composed of three layers:

- at the very bottom (the lowest level) there are the Visitors: with these classes one can build an AST out of the code and get basic metrics. Currently, there are two available visitors: ComplexityVisitor and HalsteadVisitor. With the former one analyzes the cyclomatic complexity of the code, while the latter gathers the so-called Halstead metrics. With those and other raw metrics one can compute the Maintainability Index. Example:

```python
>>> from radon.visitors import ComplexityVisitor
>>> v = ComplexityVisitor.from_code('''
def factorial(n):
    if n < 2: return 1
    return n * factorial(n - 1)

def foo(bar):
    return sum(i for i in range(bar ** 2) if bar % i)
''')
>>> v.functions
[Function(name='factorial', lineno=2, col_offset=0, endline=4, is_method=False, classname=None, closures=[], complexity=2), Function(name='foo', lineno=6, col_offset=0, endline=7, is_method=False, ...
```

- at a higher level, there are helper functions residing in separate modules. For cyclomatic complexity, one can use those inside radon.complexity. For Halstead metrics and MI index those inside radon.metrics. Finally, for raw metrics (that includes SLOC, LLOC, LOC, &c.) one can use the function analyze() inside the radon.raw module. With the majority of these functions the result is an object (Module object in the case of raw metrics) or a list of objects (Function or Class objects for cyclomatic complexity). Example:

```python
>>> from radon.complexity import cc_rank, cc_visit
>>> cc_rank(4), cc_rank(9), cc_rank(14), cc_rank(23)
('A', 'B', 'C', 'D')
```
• at the highest level there are the **Harvesters**. A Harvester implements all the business logic of the CLI interface. To use a Harvester, it’s sufficient to create a **Config** object (which contains all the config values) and pass it to the Harvester instance along with a list of paths to analyze. An Harvester can then export its result to various formats (for cyclomatic complexity both JSON and XML are available). It’s possible to find an example for this in the **Xenon** project.

### 4.1 Cyclomatic Complexity

```python
>>> from radon.raw import analyze
>>> analyze('''
def _split_tokens(tokens, token, value):
    '''Split a list of tokens on the specified token pair (token, value),
    where *token* is the token type (i.e. its code) and *value* its actual
    value in the code.
    '''
    res = [[]]
    for token_values in tokens:
        if (token, value) == token_values[:2]:
            res.append([])
            continue
        res[-1].append(token_values)
    return res
'''
)
```

```python
>>> Module(loc=12, lloc=9, sloc=12, comments=0, multi=4, blank=0)
```

**radon.complexity.cc_visit**(code, **kwargs)

Visit the given code with **ComplexityVisitor**. All the keyword arguments are directly passed to the visitor.

**radon.complexity.cc_visit_ast**(ast_node, **kwargs)

Visit the AST node with **ComplexityVisitor**. All the keyword arguments are directly passed to the visitor.

**radon.complexity.cc_rank**(cc)

Rank the complexity score from A to F, where A stands for the simplest and best score and F the most complex and worst one:
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 5</td>
<td>A (low risk - simple block)</td>
</tr>
<tr>
<td>6 - 10</td>
<td>B (low risk - well structured and stable block)</td>
</tr>
<tr>
<td>11 - 20</td>
<td>C (moderate risk - slightly complex block)</td>
</tr>
<tr>
<td>21 - 30</td>
<td>D (more than moderate risk - more complex block)</td>
</tr>
<tr>
<td>31 - 40</td>
<td>E (high risk - complex block, alarming)</td>
</tr>
<tr>
<td>41+</td>
<td>F (very high risk - error-prone, unstable block)</td>
</tr>
</tbody>
</table>

Here block is used in place of function, method or class.

The formula used to convert the score into an index is the following:

\[
\text{rank} = \left\lceil \frac{\text{score}}{10} \right\rceil - H(5 - \text{score})
\]

where \( H(s) \) stands for the Heaviside Step Function. The rank is then associated to a letter (0 = A, 5 = F).

\[
\text{radon.complexity.sorted_results}(\text{blocks, order=SCORE})
\]

Given a ComplexityVisitor instance, returns a list of sorted blocks with respect to complexity. A block is a either Function object or a Class object. The blocks are sorted in descending order from the block with the highest complexity.

The optional order parameter indicates how to sort the blocks. It can be:

- \text{LINES}: sort by line numbering;
- \text{ALPHA}: sort by name (from A to Z);
- \text{SCORE}: sorty by score (descending).

Default is \text{SCORE}.

### 4.2 Raw metrics

\[
\text{radon.raw.analyze(source)}
\]

Analyze the source code and return a namedtuple with the following fields:

- \text{loc}: The number of lines of code (total)
- \text{lloc}: The number of logical lines of code
- \text{sloc}: The number of source lines of code (not necessarily corresponding to the LLOC)
- \text{comments}: The number of Python comment lines
- \text{multi}: The number of lines which represent multi-line strings
- \text{single_comments}: The number of lines which are just comments with no code
- \text{blank}: The number of blank lines (or whitespace-only ones)

The equation \( \text{sloc} + \text{blanks} + \text{multi} + \text{single_comments} = \text{loc} \) should always hold. Multiline strings are not counted as comments, since, to the Python interpreter, they are not comments but strings.

### 4.3 Other metrics

\[
\text{radon.metrics.h_visit(code)}
\]

Compile the code into an AST tree and then pass it to \( h\_visit\_ast() \).
radon.metrics.h_visit_ast(ast_node)
Visit the AST node using the HalsteadVisitor visitor. A namedtuple with the following fields is returned:

- h1: the number of distinct operators
- h2: the number of distinct operands
- N1: the total number of operators
- N2: the total number of operands
- h: the vocabulary, i.e. h1 + h2
- N: the length, i.e. N1 + N2
- calculated_length: h1 * \log_2(h1) + h2 * \log_2(h2)
- volume: V = N * \log_2(h)
- difficulty: D = h1 / 2 * N2 / h2
- effort: E = D * V
- time: T = E / 18 seconds
- bugs: B = V / 3000 - an estimate of the errors in the implementation

radon.metrics.mi_visit(code, multi)
Visit the code and compute the Maintainability Index (MI) from it.

radon.metrics.mi_rank(score)
Rank the score with a letter:

- A if score > 19;
- B if 9 < score ≤ 19;
- C if score ≤ 9.

radon.metrics.mi_parameters(code, count_multi=True)
Given a source code snippet, compute the necessary parameters to compute the Maintainability Index metric. These include:

- the Halstead Volume
- the Cyclomatic Complexity
- the number of LLOC (Logical Lines of Code)
- the percent of lines of comment

Parameters multi – If True, then count multiline strings as comment lines as well. This is not always safe because Python multiline strings are not always docstrings.

radon.metrics.mi_compute(halstead_volume, complexity, sloc, comments)
Compute the Maintainability Index (MI) given the Halstead Volume, the Cyclomatic Complexity, the SLOC number and the number of comment lines. Usually it is not used directly but instead mi_visit() is preferred.

4.4 Visitors

class radon.visitors.ComplexityVisitor(to_method=False, classname=None, off=True, no_assert=False)
A visitor that keeps track of the cyclomatic complexity of the elements.
Parameters

- **to_method** – If True, every function is treated as a method. In this case the `classname` parameter is used as class name.
- **classname** – Name of parent class.
- **off** – If True, the starting value for the complexity is set to 1, otherwise to 0.

```python
class radon.visitors.HalsteadVisitor(context=None):
    Visitor that keeps track of operators and operands, in order to compute Halstead metrics (see `radon.metrics.h_visit()`).
```

## 4.5 Harvesters

```python
class radon.cli.harvest.Harvester(paths, config):
    Base class defining the interface of a Harvester object.
```

A Harvester has the following lifecycle:

1. **Initialization:** `h = Harvester(paths, config)`
2. **Execution:** `r = h.results`. `results` holds an iterable object. The first time `results` is accessed, `h.run()` is called. This method should not be subclassed. Instead, the `gobble()` method should be implemented.
3. **Reporting:** the methods `as_json` and `as_xml` return a string with the corresponding format. The method `to_terminal` is a generator that yields the lines to be printed in the terminal.

This class is meant to be subclasses and cannot be used directly, since the methods `gobble()`, `as_xml()` and `to_terminal()` are not implemented.

```
__init__(paths, config)
    Initialize the Harvester.
    paths is a list of paths to analyze. config is a Config object holding the configuration values specific to the Harvester.

as_codeclimate_issues()
    Format the results as Code Climate issues.

as_json()
    Format the results as JSON.

as_xml()
    Format the results as XML.

gobble(fobj)
    Subclasses must implement this method to define behavior.
    This method is called for every file to analyze. fobj is the file object. This method should return the results from the analysis, preferably a dictionary.

results
    This property holds the results of the analysis.
    The first time it is accessed, an iterator is returned. Its elements are cached into a list as it is iterated over. Therefore, if results is accessed multiple times after the first one, a list will be returned.

run()
    Start the analysis. For every file, this method calls the `gobble()` method. Results are yielded as tuple: (filename, analysis_results).
```
**to_terminal()**

Yields tuples representing lines to be printed to a terminal.

The tuples have the following format: `(line, args, kwargs)`. The line is then formatted with `line.format(*args, **kwargs)`.

class radon.cli.harvest.CCHarvester(paths, config)

A class that analyzes Python modules’ Cyclomatic Complexity.

class radon.cli.harvest.RawHarvester(paths, config)

A class that analyzes Python modules’ raw metrics.

class radon.cli.harvest.MIH Harvester(paths, config)

A class that analyzes Python modules’ Maintainability Index.
5.1 2.2.0 (Jan 11, 2018)

• Add an option to show closures when Radon is used from flake8: #135

5.2 2.1.1 (Sep 07, 2017)

• Fix critical bug in Python 2 that prevented analysis of files with no comments at the start of the module: #133

5.3 2.0.3 (Aug 30, 2017)

• Fix encoding issues for all commands: #115 and #127

5.4 2.0.2 (Jun 04, 2017)

• Update mando dependency: #120

5.5 2.0.1 (Jun 02, 2017)

• Fix bug in the error message of Flake8’s plugin: #121

5.6 2.0.0 (May 30, 2017)

• Add --sort option to mi command: #100
• Add comment stats summary when using the \texttt{-s, --summary} option of the \texttt{raw} command: \#110
• Fix encoding bug: \#114
• Fix raw metrics (ensure that LOC = SLOC + Multi + Single Comments + Blank): \#118
• Python 2.6 is not supported anymore

5.7 1.5.0 (Mar 14, 2017)

• Use UTF-8 or RADONFILESENCODING to open files: \#86
• Fix raw analysis bug on docstrings: \#106

5.8 1.4.2 (Jul 26, 2016)

• Use flake8-polyfill in order to keep compatibility with Flake8 2.x and 3.x: \#92

5.9 1.4.0 (Jun 03, 2016)

• Add fingerprint to Code Climate issues: \#88.
• Ensure the Code Climate issues have integer location values: \#89.
• \texttt{Count} \texttt{async def, async for and async with} \texttt{towards CC}: \#90.

5.10 1.3.0 (Mar 02, 2016)

• Modify behaviour of \texttt{--show-closures}. Now inner classes are added to the output as well: \#79.
• Fix bug in \texttt{is_multiline_string}: \#81.

5.11 1.2.2 (Jul 09, 2015)

• Add plugin for flake8 tool: \#76.

5.12 1.2.1 (May 07, 2015)

• The XML output now contains the line numbers: \#75.

5.13 1.2 (Jan 16, 2015)

• \textbf{Backwards incompatible} change regarding to CC of lambda functions and nested functions: \#68.
• Fix the bug that caused classes with only one method have a CC of 2: \#70.
5.14 1.1 (Sep 6, 2014)

- Make \(-n, \textit{--min}\) and \(-x, \textit{--max}\) effective everywhere (in JSON and XML exporting too): \#62.
- Fix the bug that prevented JSON/XML export when one file had errors during the analysis: \#63.
- Add an explanations and various examples to the docs so that programmatical use of Radon is easier: \#64.

5.15 1.0 (Aug 15, 2014)

- Add \(--xml\) option to \textit{cc} command: \#49.
- Officialy support Python 3.4.
- Remove pathfinder: \#59.
- Reduce drastically unit-testing time: \#56.

5.16 0.5.3 (Aug 1, 2014)

- Encode the source code to bytes if that’s possible (Python 3).
- Show help if no command is given.
- Add support to read code from stdin (thanks @io41): \#55.
- Move the tests inside the radon directory: \#58.

5.17 0.5.2 (Jul 24, 2014)

- Fix \textit{while} \ldots \textit{else} bug: \#53.

5.18 0.5.1 (Mar 4, 2014)

- Fix \textit{--total-average} behavior.

5.19 0.5 (Feb 17, 2014)

- Add \(\textit{-i, --ignore}\) option to ignore directories: \#39.
- Add \(\textit{--no-assert}\) option to \textit{cc} command to avoid \textit{assert} statements: \#42.
- Add \(\textit{-j, --json}\) option to \textit{raw} command (thanks @cjav): \#45.
- Add \(\textit{--total-average}\) option to \textit{cc} command: \#44.
- Add \(\textit{--version}\) global option: \#47.
5.20 0.4.5 (Dec 16, 2013)

• Baker is replaced with mando: https://github.com/rubik/mando.

5.21 0.4.4 (Nov 20, 2013)

• Add -j option to cc command: #33.
• Use pathfinder and improve iter_filenames: #31.
• Complete the documentation: #18.
• Add -s, --summarize option to raw command (thanks @jsargiot): #36.

5.22 0.4.2 (Jun 25, 2013)

• raw command failed on almost-empty files: #29.

5.23 0.4.1 (Jun 16, 2013)

• Turn off colors when not printing to a tty (thanks @kennknowles): #26.
• Fixed #27 (endline could be float(‘-inf’) sometimes).

5.24 0.4 (Apr 26, 2013)

• Added -s option to mi command: #19.
• Added -o option to cc command to sort output: #20.
• Added endline attribute to Function and Class objects: #25.

5.25 0.3 (Nov 2, 2012)

• Code coverage to 100%, runs from Python 2.6 up to 3.3 and on PyPy as well.
• Created a documentation at https://radon.readthedocs.org: #5.
• Made the codebase compatible with PyPy: #9.
• Ported cli.py to Python 3: #14.
• More tests: #15.
• Minor fixes: #11, #12, #13, #17.

5.26 0.2 (Oct 11, 2012)

Initial version.
5.27 0.1 (Never)

There was no 0.1.
CHAPTER 6

Indices and tables

• genindex
• modindex
• search
Python Module Index

r

radon.cli.harvest, 19
radon.complexity, 16
radon.metrics, 17
radon.raw, 17
radon.visitors, 18
Index

Symbols

–no-assert
  cc command line option, 8
–radon-max-cc <int>
  flake8 command line option, 13
–radon-no-assert
  flake8 command line option, 13
–radon-show-closures
  flake8 command line option, 13
–total-average
  cc command line option, 8
–xml
  cc command line option, 8
-a, –average
  cc command line option, 8
-e, –exclude
  cc command line option, 8
  mi command line option, 9
  raw command line option, 10
-i, –ignore
  cc command line option, 8
  mi command line option, 9
  raw command line option, 10
-j, –json
  cc command line option, 8
  mi command line option, 9
  raw command line option, 10
-m, –multi
  mi command line option, 9
-n, –min
  cc command line option, 8
  mi command line option, 9
-o, –order
  cc command line option, 8
-s, –show
  mi command line option, 9
-s, –show-complexity
  cc command line option, 8
-s, –summary
  raw command line option, 10
-x, –max
  cc command line option, 8
  mi command line option, 9
  __init__() (radon.cli.harvest.Harvester method), 19

A

analyze() (in module radon.raw), 17
as_codeclimate_issues() (radon.cli.harvest.Harvester method), 19
as_json() (radon.cli.harvest.Harvester method), 19
as_xml() (radon.cli.harvest.Harvester method), 19

C

cc command line option
  –no-assert, 8
  –total-average, 8
  –xml, 8
-a, –average, 8
-e, –exclude, 8
-i, –ignore, 8
-j, –json, 8
-n, –min, 8
-o, –order, 8
-s, –show-complexity, 8
-x, –max, 8
cc_rank() (in module radon.complexity), 16
cc_visit() (in module radon.complexity), 16
cc_visit_ast() (in module radon.complexity), 16
CCHarvester (class in radon.cli.harvest), 20
ComplexityVisitor (class in radon.visitors), 18

F

flake8 command line option
  –radon-max-cc <int>, 13
  –radon-no-assert, 13
  –radon-show-closures, 13
Instruct radon to add closures/inner classes to the output., 13
G

gobble() (radon.cli.harvest.Harvester method), 19

H

h_visit() (in module radon.metrics), 17
h_visit_ast() (in module radon.metrics), 17
HalsteadVisitor (class in radon.visitors), 19
Harvester (class in radon.cli.harvest), 19

I

Instruct radon to add closures/inner classes to the output.
flake8 command line option, 13

M

mi command line option
-e, --exclude, 9
-i, --ignore, 9
-j, --json, 10
-m, --multi, 9
-n, --min, 9
-s, --show, 9
-x, --max, 9
mi_compute() (in module radon.metrics), 18
mi_parameters() (in module radon.metrics), 18
mi_rank() (in module radon.metrics), 18
mi_visit() (in module radon.metrics), 18
MIHarvester (class in radon.cli.harvest), 20

R

radon.cli.harvest (module), 19
radon.complexity (module), 16
radon.metrics (module), 17
radon.raw (module), 17
radon.visitors (module), 18
raw command line option
-e, --exclude, 10
-i, --ignore, 10
-j, --json, 10
-s, --summary, 10
RawHarvester (class in radon.cli.harvest), 20
results (radon.cli.harvest.Harvester attribute), 19
run() (radon.cli.harvest.Harvester method), 19

S

sorted_results() (in module radon.complexity), 17

T

to_terminal() (radon.cli.harvest.Harvester method), 19