Coverage.py

Release 7.3.1

unknown

CONTENTS

Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.

Coverage measurement is typically used to gauge the effectiveness of tests. It can show which parts of your code are being exercised by tests, and which are not.

The latest version is coverage.py 7.3.1, released September 6, 2023. It is supported on:

- Python versions 3.8 through 3.12.0rc1.
- PyPy3 versions 3.8 through 3.10.

ONE

FOR ENTERPRISE

Available as part of the Tidelift Subscription. Coverage and thousands of other packages are working with Tidelift to deliver one enterprise subscription that covers all of the open source you use. If you want the flexibility of open source and the confidence of commercial-grade software, this is for you. Learn more.

QUICK START

Getting started is easy:

1. Install coverage.py:

   ```
   $ python3 -m pip install coverage
   ```

   For more details, see *Installation*.

2. Use `coverage run` to run your test suite and gather data. However you normally run your test suite, you can use your test runner under coverage.

   Tip: If your test runner command starts with "python", just replace the initial "python" with "coverage run".

   `python something.py` becomes `coverage run something.py`

   `python -m amodule` becomes `coverage run -m amodule`

   Other instructions for specific test runners:

   Listing 1: pytest

   If you usually use:

   Listing 1: pytest

   ```
   $ pytest arg1 arg2 arg3
   ```

   then you can run your tests under coverage with:

   Listing 1: pytest

   ```
   $ coverage run -m pytest arg1 arg2 arg3
   ```

   Many people choose to use the pytest-cov plugin, but for most purposes, it is unnecessary.

   Listing 2: unittest

   Change "python" to "coverage run", so this:

   Listing 2: unittest

   ```
   $ python3 -m unittest discover
   ```

   becomes:

   Listing 2: unittest

   ```
   $ coverage run -m unittest discover
   ```

Listing 3: nosetest

Note: Nose has been unmaintained since at least 2015. *You should seriously consider using a different test runner.*

Change this:

Listing 3: nosetest

```
$ nosetests arg1 arg2
```

to:

Listing 3: nosetest

```
$ coverage run -m nose arg1 arg2
```

To limit coverage measurement to code in the current directory, and also find files that weren't executed at all, add the `--source=.` argument to your coverage command line.

3. Use `coverage report` to report on the results:

```
$ coverage report -m
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
my_program.py                20      4    80%   33-35, 39
my_other_module.py           56      6    89%   17-23
-------------------------------------------------------
TOTAL                        76     10    87%
```

4. For a nicer presentation, use `coverage html` to get annotated HTML listings detailing missed lines:

```
$ coverage html
```

Then open htmlcov/index.html in your browser, to see a report like this.

CAPABILITIES

Coverage.py can do a number of things:

- By default it will measure line (statement) coverage.

- It can also measure *branch coverage*.

- It can tell you *what tests ran which lines*.

- It can produce reports in a number of formats: *text*, *HTML*, *XML*, *LCOV*, and *JSON*.

- For advanced uses, there's an *API*, and the result data is available in a *SQLite database*.

USING COVERAGE.PY

There are a few different ways to use coverage.py. The simplest is the *command line*, which lets you run your program and see the results. If you need more control over how your project is measured, you can use the *API*.

Some test runners provide coverage integration to make it easy to use coverage.py while running tests. For example, pytest has the pytest-cov plugin.

You can fine-tune coverage.py's view of your code by directing it to ignore parts that you know aren't interesting. See *Specifying source files* and *Excluding code from coverage.py* for details.

GETTING HELP

If the *FAQ* doesn't answer your question, you can discuss coverage.py or get help using it on the Python discussion forums. If you ping me (@nedbat), there's a higher chance I'll see the post.

Bug reports are gladly accepted at the GitHub issue tracker. GitHub also hosts the code repository.

Professional support for coverage.py is available as part of the Tidelift Subscription.

I can be reached in a number of ways. I'm happy to answer questions about using coverage.py.

SIX

MORE INFORMATION

6.1 Installation

You can install coverage.py in the usual ways. The simplest way is with pip:

```
$ python3 -m pip install coverage
```

6.1.1 C Extension

Coverage.py includes a C extension for speed. It is strongly recommended to use this extension: it is much faster, and is needed to support a number of coverage.py features. Most of the time, the C extension will be installed without any special action on your part.

You can determine if you are using the extension by looking at the output of `coverage --version`:

```
$ coverage --version
Coverage.py, version 7.3.1 with C extension
Documentation at https://coverage.readthedocs.io/en/7.3.1
```

The first line will either say "with C extension," or "without C extension."

If you are missing the extension, first make sure you have the latest version of pip in use when installing coverage.

If you are installing on Linux, you may need to install the python-dev and gcc support files before installing coverage via pip. The exact commands depend on which package manager you use, which Python version you are using, and the names of the packages for your distribution. For example:

```
$ sudo apt-get install python-dev gcc
$ sudo yum install python-devel gcc

$ sudo apt-get install python3-dev gcc
$ sudo yum install python3-devel gcc
```

A few features of coverage.py aren't supported without the C extension, such as concurrency and plugins.

6.1.2 Checking the installation

If all went well, you should be able to open a command prompt, and see coverage.py installed properly:

```
$ coverage --version
Coverage.py, version 7.3.1 with C extension
Documentation at https://coverage.readthedocs.io/en/7.3.1
```

You can also invoke coverage.py as a module:

```
$ python3 -m coverage --version
Coverage.py, version 7.3.1 with C extension
Documentation at https://coverage.readthedocs.io/en/7.3.1
```

6.2 Command line usage

When you install coverage.py, a command-line script called `coverage` is placed on your path. To help with multi-version installs, it will also create a `coverage3` alias, and a `coverage-X.Y` alias, depending on the version of Python you're using. For example, when installing on Python 3.10, you will be able to use `coverage`, `coverage3`, or `coverage-3.10` on the command line.

Coverage.py has a number of commands:

- **run** – *Run a Python program and collect execution data*.

- **combine** – *Combine together a number of data files*.

- **erase** – *Erase previously collected coverage data*.

- **report** – *Report coverage results*.

- **html** – *Produce annotated HTML listings with coverage results*.

- **xml** – *Produce an XML report with coverage results*.

- **json** – *Produce a JSON report with coverage results*.

- **lcov** – *Produce an LCOV report with coverage results*.

- **annotate** – *Annotate source files with coverage results*.

- **debug** – *Get diagnostic information*.

Help is available with the **help** command, or with the `--help` switch on any other command:

```
$ coverage help
$ coverage help run
$ coverage run --help
```

Version information for coverage.py can be displayed with `coverage --version`:

```
$ coverage --version
Coverage.py, version 7.3.1 with C extension
Documentation at https://coverage.readthedocs.io/en/7.3.1
```

Any command can use a configuration file by specifying it with the `--rcfile=FILE` command-line switch. Any option you can set on the command line can also be set in the configuration file. This can be a better way to control coverage.py since the configuration file can be checked into source control, and can provide options that other invocation techniques (like test runner plugins) may not offer. See *Configuration reference* for more details.

6.2.1 Execution: `coverage run`

You collect execution data by running your Python program with the **run** command:

```
$ coverage run my_program.py arg1 arg2
blah blah ..your program's output.. blah blah
```

Your program runs just as if it had been invoked with the Python command line. Arguments after your file name are passed to your program as usual in `sys.argv`. Rather than providing a file name, you can use the `-m` switch and specify an importable module name instead, just as you can with the Python `-m` switch:

```
$ coverage run -m packagename.modulename arg1 arg2
blah blah ..your program's output.. blah blah
```

Note: In most cases, the program to use here is a test runner, not your program you are trying to measure. The test runner will run your tests and coverage will measure the coverage of your code along the way.

There are many options:

```
$ coverage run --help
Usage: coverage run [options] <pyfile> [program options]

Run a Python program, measuring code execution.

Options:
  -a, --append         Append coverage data to .coverage, otherwise it starts
                       clean each time.
  --branch             Measure branch coverage in addition to statement
                       coverage.
  --concurrency=LIBS   Properly measure code using a concurrency library.
                       Valid values are: eventlet, gevent, greenlet,
                       multiprocessing, thread, or a comma-list of them.
  --context=LABEL      The context label to record for this coverage run.
  --data-file=OUTFILE  Write the recorded coverage data to this file.
                       Defaults to '.coverage'. [env: COVERAGE_FILE]
  --include=PAT1,PAT2,...
                       Include only files whose paths match one of these
                       patterns. Accepts shell-style wildcards, which must be
                       quoted.
  -m, --module         <pyfile> is an importable Python module, not a script
                       path, to be run as 'python -m' would run it.
  --omit=PAT1,PAT2,... Omit files whose paths match one of these patterns.
                       Accepts shell-style wildcards, which must be quoted.
  -L, --pylib          Measure coverage even inside the Python installed
                       library, which isn't done by default.
  -p, --parallel-mode  Append the machine name, process id and random number
                       to the data file name to simplify collecting data from
                       many processes.
  --source=SRC1,SRC2,...
                       A list of directories or importable names of code to
                       measure.
  --timid              Use a simpler but slower trace method. Try this if you
```

```
                           get seemingly impossible results!
 --debug=OPTS              Debug options, separated by commas. [env:
                           COVERAGE_DEBUG]
 -h, --help                Get help on this command.
 --rcfile=RCFILE           Specify configuration file. By default '.coveragerc',
                           'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                           tried. [env: COVERAGE_RCFILE]
```

If you want *branch coverage* measurement, use the `--branch` flag. Otherwise only statement coverage is measured.

You can specify the code to measure with the `--source`, `--include`, and `--omit` switches. See *Specifying source files* for details of their interpretation. Remember to put options for run after "run", but before the program invocation:

```
$ coverage run --source=dir1,dir2 my_program.py arg1 arg2
$ coverage run --source=dir1,dir2 -m packagename.modulename arg1 arg2
```

Note: Specifying `--source` on the `coverage run` command line won't affect subsequent reporting commands like `coverage xml`. Use the *source* setting in the configuration file to apply the setting uniformly to all commands.

Coverage.py can measure multi-threaded programs by default. If you are using more other concurrency support, with the multiprocessing, greenlet, eventlet, or gevent libraries, then coverage.py can get confused. Use the `--concurrency` switch to properly measure programs using these libraries. Give it a value of `multiprocessing`, `thread`, `greenlet`, `eventlet`, or `gevent`. Values other than `thread` require the *C extension*.

You can combine multiple values for `--concurrency`, separated with commas. You can specify `thread` and also one of `eventlet`, `gevent`, or `greenlet`.

If you are using `--concurrency=multiprocessing`, you must set other options in the configuration file. Options on the command line will not be passed to the processes that multiprocessing creates. Best practice is to use the configuration file for all options.

If you are measuring coverage in a multi-process program, or across a number of machines, you'll want the `--parallel-mode` switch to keep the data separate during measurement. See *Combining data files: coverage combine* below.

You can specify a *static context* for a coverage run with `--context`. This can be any label you want, and will be recorded with the data. See *Measurement contexts* for more information.

By default, coverage.py does not measure code installed with the Python interpreter, for example, the standard library. If you want to measure that code as well as your own, add the `-L` (or `--pylib`) flag.

If your coverage results seem to be overlooking code that you know has been executed, try running coverage.py again with the `--timid` flag. This uses a simpler but slower trace method, and might be needed in rare cases.

Coverage.py sets an environment variable, `COVERAGE_RUN` to indicate that your code is running under coverage measurement. The value is not relevant, and may change in the future.

These options can also be set in the *[run]* section of your .coveragerc file.

Warnings

During execution, coverage.py may warn you about conditions it detects that could affect the measurement process. The possible warnings include:

Couldn't parse Python file XXX (couldnt-parse)
> During reporting, a file was thought to be Python, but it couldn't be parsed as Python.

Trace function changed, data is likely wrong: XXX (trace-changed)
> Coverage measurement depends on a Python setting called the trace function. Other Python code in your product might change that function, which will disrupt coverage.py's measurement. This warning indicates that has happened. The XXX in the message is the new trace function value, which might provide a clue to the cause.

Module XXX has no Python source (module-not-python)
> You asked coverage.py to measure module XXX, but once it was imported, it turned out not to have a corresponding .py file. Without a .py file, coverage.py can't report on missing lines.

Module XXX was never imported (module-not-imported)
> You asked coverage.py to measure module XXX, but it was never imported by your program.

No data was collected (no-data-collected)
> Coverage.py ran your program, but didn't measure any lines as executed. This could be because you asked to measure only modules that never ran, or for other reasons.
>
> To debug this problem, try using `run --debug=trace` to see the tracing decision made for each file.

Module XXX was previously imported, but not measured (module-not-measured)
> You asked coverage.py to measure module XXX, but it had already been imported when coverage started. This meant coverage.py couldn't monitor its execution.

Already imported a file that will be measured: XXX (already-imported)
> File XXX had already been imported when coverage.py started measurement. Your setting for `--source` or `--include` indicates that you wanted to measure that file. Lines will be missing from the coverage report since the execution during import hadn't been measured.

--include is ignored because --source is set (include-ignored)
> Both `--include` and `--source` were specified while running code. Both are meant to focus measurement on a particular part of your source code, so `--include` is ignored in favor of `--source`.

Conflicting dynamic contexts (dynamic-conflict)
> The `[run] dynamic_context` option is set in the configuration file, but something (probably a test runner plugin) is also calling the *Coverage.switch_context()* function to change the context. Only one of these mechanisms should be in use at a time.

Individual warnings can be disabled with the *disable_warnings* configuration setting. To silence "No data was collected," add this to your configuration file:

Listing 1: .coveragerc

```
[run]
disable_warnings = no-data-collected
```

Listing 2: pyproject.toml

```
[tool.coverage.run]
disable_warnings = ["no-data-collected"]
```

Listing 3: setup.cfg, tox.ini

```
[coverage:run]
disable_warnings = no-data-collected
```

Data file

Coverage.py collects execution data in a file called ".coverage". If need be, you can set a new file name with the COVERAGE_FILE environment variable. This can include a path to another directory.

By default, each run of your program starts with an empty data set. If you need to run your program multiple times to get complete data (for example, because you need to supply different options), you can accumulate data across runs with the `--append` flag on the **run** command.

6.2.2 Combining data files: `coverage combine`

Often test suites are run under different conditions, for example, with different versions of Python, or dependencies, or on different operating systems. In these cases, you can collect coverage data for each test run, and then combine all the separate data files into one combined file for reporting.

The **combine** command reads a number of separate data files, matches the data by source file name, and writes a combined data file with all of the data.

Coverage normally writes data to a filed named ".coverage". The `run --parallel-mode` switch (or `[run] parallel=True` configuration option) tells coverage to expand the file name to include machine name, process id, and a random number so that every data file is distinct:

```
.coverage.Neds-MacBook-Pro.local.88335.316857
.coverage.Geometer.8044.799674
```

You can also define a new data file name with the `[run] data_file` option.

Once you have created a number of these files, you can copy them all to a single directory, and use the **combine** command to combine them into one .coverage data file:

```
$ coverage combine
```

You can also name directories or files to be combined on the command line:

```
$ coverage combine data1.dat windows_data_files/
```

Coverage.py will collect the data from those places and combine them. The current directory isn't searched if you use command-line arguments. If you also want data from the current directory, name it explicitly on the command line.

When coverage.py combines data files, it looks for files named the same as the data file (defaulting to ".coverage"), with a dotted suffix. Here are some examples of data files that can be combined:

```
.coverage.machine1
.coverage.20120807T212300
.coverage.last_good_run.ok
```

An existing combined data file is ignored and re-written. If you want to use **combine** to accumulate results into the .coverage data file over a number of runs, use the `--append` switch on the **combine** command. This behavior was the default before version 4.2.

If any of the data files can't be read, coverage.py will print a warning indicating the file and the problem.

The original input data files are deleted once they've been combined. If you want to keep those files, use the `--keep` command-line option.

```
$ coverage combine --help
Usage: coverage combine [options] <path1> <path2> ... <pathN>

Combine data from multiple coverage files. The combined results are written to
a single file representing the union of the data. The positional arguments are
data files or directories containing data files. If no paths are provided,
data files in the default data file's directory are combined.

Options:
  -a, --append          Append coverage data to .coverage, otherwise it starts
                        clean each time.
  --data-file=DATAFILE  Base name of the data files to operate on. Defaults to
                        '.coverage'. [env: COVERAGE_FILE]
  --keep                Keep original coverage files, otherwise they are
                        deleted.
  -q, --quiet           Don't print messages about what is happening.
  --debug=OPTS          Debug options, separated by commas. [env:
                        COVERAGE_DEBUG]
  -h, --help            Get help on this command.
  --rcfile=RCFILE       Specify configuration file. By default '.coveragerc',
                        'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                        tried. [env: COVERAGE_RCFILE]
```

Re-mapping paths

To combine data for a source file, coverage has to find its data in each of the data files. Different test runs may run the same source file from different locations. For example, different operating systems will use different paths for the same file, or perhaps each Python version is run from a different subdirectory. Coverage needs to know that different file paths are actually the same source file for reporting purposes.

You can tell coverage.py how different source locations relate with a `[paths]` section in your configuration file (see *[paths]*). It might be more convenient to use the `[run] relative_files` setting to store relative file paths (see *relative_files*).

If data isn't combining properly, you can see details about the inner workings with `--debug=pathmap`.

6.2.3 Erase data: `coverage erase`

To erase the collected data, use the **erase** command:

```
$ coverage erase --help
Usage: coverage erase [options]

Erase previously collected coverage data.

Options:
  --data-file=DATAFILE  Base name of the data files to operate on. Defaults to
                        '.coverage'. [env: COVERAGE_FILE]
```

```
--debug=OPTS          Debug options, separated by commas. [env:
                      COVERAGE_DEBUG]
-h, --help            Get help on this command.
--rcfile=RCFILE       Specify configuration file. By default '.coveragerc',
                      'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                      tried. [env: COVERAGE_RCFILE]
```

If your configuration file indicates parallel data collection, **erase** will remove all of the data files.

6.2.4 Reporting

Coverage.py provides a few styles of reporting, with the **report**, **html**, **annotate**, **json**, **lcov**, and **xml** commands. They share a number of common options.

The command-line arguments are module or file names to report on, if you'd like to report on a subset of the data collected.

The `--include` and `--omit` flags specify lists of file name patterns. They control which files to report on, and are described in more detail in *Specifying source files*.

The `-i` or `--ignore-errors` switch tells coverage.py to ignore problems encountered trying to find source files to report on. This can be useful if some files are missing, or if your Python execution is tricky enough that file names are synthesized without real source files.

If you provide a `--fail-under` value, the total percentage covered will be compared to that value. If it is less, the command will exit with a status code of 2, indicating that the total coverage was less than your target. This can be used as part of a pass/fail condition, for example in a continuous integration server. This option isn't available for **annotate**.

These options can also be set in your .coveragerc file. See *Configuration: [report]*.

6.2.5 Coverage summary: `coverage report`

The simplest reporting is a textual summary produced with **report**:

```
$ coverage report
Name                   Stmts   Miss  Cover
--------------------------------------------
my_program.py             20      4    80%
my_module.py              15      2    86%
my_other_module.py        56      6    89%
--------------------------------------------
TOTAL                     91     12    87%
```

For each module executed, the report shows the count of executable statements, the number of those statements missed, and the resulting coverage, expressed as a percentage.

```
$ coverage report --help
Usage: coverage report [options] [modules]

Report coverage statistics on modules.

Options:
  --contexts=REGEX1,REGEX2,...
```

```
                         Only display data from lines covered in the given
                         contexts. Accepts Python regexes, which must be
                         quoted.
  --data-file=INFILE     Read coverage data for report generation from this
                         file. Defaults to '.coverage'. [env: COVERAGE_FILE]
  --fail-under=MIN       Exit with a status of 2 if the total coverage is less
                         than MIN.
  --format=FORMAT        Output format, either text (default), markdown, or
                         total.
  -i, --ignore-errors    Ignore errors while reading source files.
  --include=PAT1,PAT2,...
                         Include only files whose paths match one of these
                         patterns. Accepts shell-style wildcards, which must be
                         quoted.
  --omit=PAT1,PAT2,...   Omit files whose paths match one of these patterns.
                         Accepts shell-style wildcards, which must be quoted.
  --precision=N          Number of digits after the decimal point to display
                         for reported coverage percentages.
  --sort=COLUMN          Sort the report by the named column: name, stmts,
                         miss, branch, brpart, or cover. Default is name.
  -m, --show-missing     Show line numbers of statements in each module that
                         weren't executed.
  --skip-covered         Skip files with 100% coverage.
  --no-skip-covered      Disable --skip-covered.
  --skip-empty           Skip files with no code.
  --debug=OPTS           Debug options, separated by commas. [env:
                         COVERAGE_DEBUG]
  -h, --help             Get help on this command.
  --rcfile=RCFILE        Specify configuration file. By default '.coveragerc',
                         'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                         tried. [env: COVERAGE_RCFILE]
```

The `-m` flag also shows the line numbers of missing statements:

```
$ coverage report -m
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
my_program.py                20      4    80%   33-35, 39
my_module.py                 15      2    86%   8, 12
my_other_module.py           56      6    89%   17-23
-------------------------------------------------------
TOTAL                        91     12    87%
```

If you are using branch coverage, then branch statistics will be reported in the Branch and BrPart (for Partial Branch) columns, the Missing column will detail the missed branches:

```
$ coverage report -m
Name                      Stmts   Miss Branch BrPart  Cover   Missing
---------------------------------------------------------------------
my_program.py                20      4     10      2    80%   33-35, 36->38, 39
my_module.py                 15      2      3      0    86%   8, 12
my_other_module.py           56      6      5      1    89%   17-23, 40->45
```

```
--------------------------------------------------------------------
TOTAL                        91    12    18     3    87%
```

You can restrict the report to only certain files by naming them on the command line:

```
$ coverage report -m my_program.py my_other_module.py
Name                      Stmts   Miss  Cover   Missing
-----------------------------------------------------
my_program.py                20      4    80%   33-35, 39
my_other_module.py           56      6    89%   17-23
-----------------------------------------------------
TOTAL                        76     10    87%
```

The `--skip-covered` switch will skip any file with 100% coverage, letting you focus on the files that still need attention. The `--no-skip-covered` option can be used if needed to see all the files. The `--skip-empty` switch will skip any file with no executable statements.

If you have *recorded contexts*, the `--contexts` option lets you choose which contexts to report on. See *Context reporting* for details.

The `--precision` option controls the number of digits displayed after the decimal point in coverage percentages, defaulting to none.

The `--sort` option is the name of a column to sort the report by.

The `--format` option controls the style of the report. `--format=text` creates plain text tables as shown above. `--format=markdown` creates Markdown tables. `--format=total` writes out a single number, the total coverage percentage as shown at the end of the tables, but without a percent sign.

Other common reporting options are described above in *Reporting*. These options can also be set in your .coveragerc file. See *Configuration: [report]*.

6.2.6 HTML reporting: `coverage html`

Coverage.py can annotate your source code to show which lines were executed and which were not. The **html** command creates an HTML report similar to the **report** summary, but as an HTML file. Each module name links to the source file decorated to show the status of each line.

Here's a sample report.

Lines are highlighted: green for executed, red for missing, and gray for excluded. If you've used branch coverage, partial branches are yellow. The colored counts at the top of the file are buttons to turn on and off the highlighting.

A number of keyboard shortcuts are available for navigating the report. Click the keyboard icon in the upper right to see the complete list.

```
$ coverage html --help
Usage: coverage html [options] [modules]

Create an HTML report of the coverage of the files.  Each file gets its own
page, with the source decorated to show executed, excluded, and missed lines.

Options:
  --contexts=REGEX1,REGEX2,...
                        Only display data from lines covered in the given
```

```
                            contexts. Accepts Python regexes, which must be
                            quoted.
  -d DIR, --directory=DIR
                            Write the output files to DIR.
  --data-file=INFILE        Read coverage data for report generation from this
                            file. Defaults to '.coverage'. [env: COVERAGE_FILE]
  --fail-under=MIN          Exit with a status of 2 if the total coverage is less
                            than MIN.
  -i, --ignore-errors       Ignore errors while reading source files.
  --include=PAT1,PAT2,...
                            Include only files whose paths match one of these
                            patterns. Accepts shell-style wildcards, which must be
                            quoted.
  --omit=PAT1,PAT2,...      Omit files whose paths match one of these patterns.
                            Accepts shell-style wildcards, which must be quoted.
  --precision=N             Number of digits after the decimal point to display
                            for reported coverage percentages.
  -q, --quiet               Don't print messages about what is happening.
  --show-contexts           Show contexts for covered lines.
  --skip-covered            Skip files with 100% coverage.
  --no-skip-covered         Disable --skip-covered.
  --skip-empty              Skip files with no code.
  --title=TITLE             A text string to use as the title on the HTML.
  --debug=OPTS              Debug options, separated by commas. [env:
                            COVERAGE_DEBUG]
  -h, --help                Get help on this command.
  --rcfile=RCFILE           Specify configuration file. By default '.coveragerc',
                            'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                            tried. [env: COVERAGE_RCFILE]
```

The title of the report can be set with the `title` setting in the `[html]` section of the configuration file, or the `--title` switch on the command line.

If you prefer a different style for your HTML report, you can provide your own CSS file to apply, by specifying a CSS file in the `[html]` section of the configuration file. See *[html] extra_css* for details.

The `-d` argument specifies an output directory, defaulting to "htmlcov":

```
$ coverage html -d coverage_html
```

Other common reporting options are described above in *Reporting*.

Generating the HTML report can be time-consuming. Stored with the HTML report is a data file that is used to speed up reporting the next time. If you generate a new report into the same directory, coverage.py will skip generating unchanged pages, making the process faster.

The `--skip-covered` switch will skip any file with 100% coverage, letting you focus on the files that still need attention. The `--skip-empty` switch will skip any file with no executable statements.

The `--precision` option controls the number of digits displayed after the decimal point in coverage percentages, defaulting to none.

If you have *recorded contexts*, the `--contexts` option lets you choose which contexts to report on, and the `--show-contexts` option will annotate lines with the contexts that ran them. See *Context reporting* for details.

These options can also be set in your .coveragerc file. See *Configuration: [html]*.

6.2.7 XML reporting: `coverage xml`

The **xml** command writes coverage data to a "coverage.xml" file in a format compatible with Cobertura.

```
$ coverage xml --help
Usage: coverage xml [options] [modules]

Generate an XML report of coverage results.

Options:
  --data-file=INFILE    Read coverage data for report generation from this
                        file. Defaults to '.coverage'. [env: COVERAGE_FILE]
  --fail-under=MIN      Exit with a status of 2 if the total coverage is less
                        than MIN.
  -i, --ignore-errors   Ignore errors while reading source files.
  --include=PAT1,PAT2,...
                        Include only files whose paths match one of these
                        patterns. Accepts shell-style wildcards, which must be
                        quoted.
  --omit=PAT1,PAT2,...  Omit files whose paths match one of these patterns.
                        Accepts shell-style wildcards, which must be quoted.
  -o OUTFILE            Write the XML report to this file. Defaults to
                        'coverage.xml'
  -q, --quiet          Don't print messages about what is happening.
  --skip-empty         Skip files with no code.
  --debug=OPTS         Debug options, separated by commas. [env:
                        COVERAGE_DEBUG]
  -h, --help           Get help on this command.
  --rcfile=RCFILE      Specify configuration file. By default '.coveragerc',
                        'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                        tried. [env: COVERAGE_RCFILE]
```

You can specify the name of the output file with the `-o` switch.

Other common reporting options are described above in *Reporting*.

To include complete file paths in the output file, rather than just the file name, use [include] vs [source] in your ".coveragerc" file.

For example, use this:

```
[run]
include =
    foo/*
    bar/*
```

which will result in

```
<class filename="bar/hello.py">
<class filename="bar/baz/hello.py">
<class filename="foo/hello.py">
```

in place of this:

```
[run]
source =
```

```
    foo
    bar
```

which may result in

```
<class filename="hello.py">
<class filename="baz/hello.py">
```

These options can also be set in your .coveragerc file. See *Configuration: [xml]*.

6.2.8 JSON reporting: `coverage json`

The **json** command writes coverage data to a "coverage.json" file.

```
$ coverage json --help
Usage: coverage json [options] [modules]

Generate a JSON report of coverage results.

Options:
  --contexts=REGEX1,REGEX2,...
                        Only display data from lines covered in the given
                        contexts. Accepts Python regexes, which must be
                        quoted.
  --data-file=INFILE    Read coverage data for report generation from this
                        file. Defaults to '.coverage'. [env: COVERAGE_FILE]
  --fail-under=MIN      Exit with a status of 2 if the total coverage is less
                        than MIN.
  -i, --ignore-errors   Ignore errors while reading source files.
  --include=PAT1,PAT2,...
                        Include only files whose paths match one of these
                        patterns. Accepts shell-style wildcards, which must be
                        quoted.
  --omit=PAT1,PAT2,...  Omit files whose paths match one of these patterns.
                        Accepts shell-style wildcards, which must be quoted.
  -o OUTFILE            Write the JSON report to this file. Defaults to
                        'coverage.json'
  --pretty-print       Format the JSON for human readers.
  -q, --quiet          Don't print messages about what is happening.
  --show-contexts      Show contexts for covered lines.
  --debug=OPTS         Debug options, separated by commas. [env:
                        COVERAGE_DEBUG]
  -h, --help           Get help on this command.
  --rcfile=RCFILE      Specify configuration file. By default '.coveragerc',
                        'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                        tried. [env: COVERAGE_RCFILE]
```

You can specify the name of the output file with the `-o` switch. The JSON can be nicely formatted by specifying the `--pretty-print` switch.

Other common reporting options are described above in *Reporting*. These options can also be set in your .coveragerc file. See *Configuration: [json]*.

6.2.9 LCOV reporting: `coverage lcov`

The **lcov** command writes coverage data to a "coverage.lcov" file.

```
$ coverage lcov --help
Usage: coverage lcov [options] [modules]

Generate an LCOV report of coverage results.

Options:
  --data-file=INFILE    Read coverage data for report generation from this
                        file. Defaults to '.coverage'. [env: COVERAGE_FILE]
  --fail-under=MIN      Exit with a status of 2 if the total coverage is less
                        than MIN.
  -i, --ignore-errors   Ignore errors while reading source files.
  --include=PAT1,PAT2,...
                        Include only files whose paths match one of these
                        patterns. Accepts shell-style wildcards, which must be
                        quoted.
  -o OUTFILE           Write the LCOV report to this file. Defaults to
                        'coverage.lcov'
  --omit=PAT1,PAT2,...  Omit files whose paths match one of these patterns.
                        Accepts shell-style wildcards, which must be quoted.
  -q, --quiet          Don't print messages about what is happening.
  --debug=OPTS         Debug options, separated by commas. [env:
                        COVERAGE_DEBUG]
  -h, --help           Get help on this command.
  --rcfile=RCFILE      Specify configuration file. By default '.coveragerc',
                        'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                        tried. [env: COVERAGE_RCFILE]
```

Common reporting options are described above in *Reporting*. Also see *Configuration: [lcov]*.

New in version 6.3.

6.2.10 Text annotation: `coverage annotate`

Note: The **annotate** command has been obsoleted by more modern reporting tools, including the **html** command. **annotate** will be removed in a future version.

The **annotate** command produces a text annotation of your source code. With a `-d` argument specifying an output directory, each Python file becomes a text file in that directory. Without `-d`, the files are written into the same directories as the original Python files.

Coverage status for each line of source is indicated with a character prefix:

```
> executed
! missing (not executed)
- excluded
```

For example:

```
  # A simple function, never called with x==1

> def h(x):
      """Silly function."""
-     if 0:  # pragma: no cover
-         pass
>     if x == 1:
!         a = 1
>     else:
>         a = 2
```

```
$ coverage annotate --help
Usage: coverage annotate [options] [modules]

Make annotated copies of the given files, marking statements that are executed
with > and statements that are missed with !.

Options:
  -d DIR, --directory=DIR
                        Write the output files to DIR.
  --data-file=INFILE    Read coverage data for report generation from this
                        file. Defaults to '.coverage'. [env: COVERAGE_FILE]
  -i, --ignore-errors   Ignore errors while reading source files.
  --include=PAT1,PAT2,...
                        Include only files whose paths match one of these
                        patterns. Accepts shell-style wildcards, which must be
                        quoted.
  --omit=PAT1,PAT2,...  Omit files whose paths match one of these patterns.
                        Accepts shell-style wildcards, which must be quoted.
  --debug=OPTS          Debug options, separated by commas. [env:
                        COVERAGE_DEBUG]
  -h, --help            Get help on this command.
  --rcfile=RCFILE       Specify configuration file. By default '.coveragerc',
                        'setup.cfg', 'tox.ini', and 'pyproject.toml' are
                        tried. [env: COVERAGE_RCFILE]
```

Other common reporting options are described above in *Reporting*.

6.2.11 Diagnostics: `coverage debug`

The **debug** command shows internal information to help diagnose problems. If you are reporting a bug about coverage.py, including the output of this command can often help:

```
$ coverage debug sys > please_attach_to_bug_report.txt
```

A few types of information are available:

- `config`: show coverage's configuration

- `sys`: show system configuration

- `data`: show a summary of the collected coverage data

- `premain`: show the call stack invoking coverage

• `pybehave`: show internal flags describing Python behavior

```
$ coverage debug --help
Usage: coverage debug <topic>

Display information about the internals of coverage.py, for diagnosing
problems. Topics are: 'data' to show a summary of the collected data; 'sys' to
show installation information; 'config' to show the configuration; 'premain'
to show what is calling coverage; 'pybehave' to show internal flags describing
Python behavior.

Options:
  --debug=OPTS      Debug options, separated by commas. [env: COVERAGE_DEBUG]
  -h, --help        Get help on this command.
  --rcfile=RCFILE   Specify configuration file. By default '.coveragerc',
                    'setup.cfg', 'tox.ini', and 'pyproject.toml' are tried.
                    [env: COVERAGE_RCFILE]
```

--debug

The `--debug` option is also available on all commands. It instructs coverage.py to log internal details of its operation, to help with diagnosing problems. It takes a comma-separated list of options, each indicating a facet of operation to log:

• `callers`: annotate each debug message with a stack trace of the callers to that point.

• `config`: before starting, dump all the *configuration* values.

• `dataio`: log when reading or writing any data file.

• `dataop`: log when data is added to the CoverageData object.

• `lock`: log operations acquiring locks in the data layer.

• `multiproc`: log the start and stop of multiprocessing processes.

• `pathmap`: log the remapping of paths that happens during `coverage combine`. See *[paths]*.

• `pid`: annotate all warnings and debug output with the process and thread ids.

• `plugin`: print information about plugin operations.

• `process`: show process creation information, and changes in the current directory. This also writes a time stamp and command arguments into the data file.

• `pybehave`: show the values of internal flags describing the behavior of the current version of Python.

• `self`: annotate each debug message with the object printing the message.

• `sql`: log the SQL statements used for recording data.

• `sqldata`: when used with `debug=sql`, also log the full data being used in SQL statements.

• `sys`: before starting, dump all the system and environment information, as with *coverage debug sys*.

• `trace`: print every decision about whether to trace a file or not. For files not being traced, the reason is also given.

Debug options can also be set with the `COVERAGE_DEBUG` environment variable, a comma-separated list of these options, or in the *[run] debug* section of the .coveragerc file.

The debug output goes to stderr, unless the *[run] debug_file* setting or the COVERAGE_DEBUG_FILE environment variable names a different file, which will be appended to. This can be useful because many test runners capture output, which could hide important details. COVERAGE_DEBUG_FILE accepts the special names stdout and stderr to write to those destinations.

6.3 Configuration reference

Coverage.py options can be specified in a configuration file. This makes it easier to re-run coverage.py with consistent settings, and also allows for specification of options that are otherwise only available in the *API*.

Configuration files also make it easier to get coverage testing of spawned sub-processes. See *Measuring sub-processes* for more details.

The default name for the configuration file is .coveragerc, in the same directory coverage.py is being run in. Most of the settings in the configuration file are tied to your source code and how it should be measured, so it should be stored with your source, and checked into source control, rather than put in your home directory.

A different location for the configuration file can be specified with the --rcfile=FILE command line option or with the COVERAGE_RCFILE environment variable.

Coverage.py will read settings from other usual configuration files if no other configuration file is used. It will automatically read from "setup.cfg" or "tox.ini" if they exist. In this case, the section names have "coverage:" prefixed, so the [run] options described below will be found in the [coverage:run] section of the file.

Coverage.py will read from "pyproject.toml" if TOML support is available, either because you are running on Python 3.11 or later, or because you installed with the toml extra (pip install coverage[toml]).

6.3.1 Syntax

The specific syntax of a configuration file depends on what type it is. All configuration files are assumed to be in INI format, unless their file extension is .toml, which are TOML.

INI Syntax

A coverage.py configuration file is in classic .ini file format: sections are introduced by a [section] header, and contain name = value entries. Lines beginning with # or ; are ignored as comments.

Strings don't need quotes. Multi-valued strings can be created by indenting values on multiple lines.

Boolean values can be specified as on, off, true, false, 1, or 0 and are case-insensitive.

TOML Syntax

TOML syntax uses explicit lists with brackets, and strings with quotes. Booleans are in true or false.

Configuration must be within the [tool.coverage] section, for example, [tool.coverage.run]. Environment variable expansion in values is available, but only within quoted strings, even for non-string values.

Environment variables

Environment variables can be substituted in by using dollar signs: `$WORD` or `${WORD}` will be replaced with the value of `WORD` in the environment. A dollar sign can be inserted with `$$`. Special forms can be used to control what happens if the variable isn't defined in the environment:

- If you want to raise an error if an environment variable is undefined, use a question mark suffix: `${WORD?}`.

- If you want to provide a default for missing variables, use a dash with a default value: `${WORD-default value}`.

- Otherwise, missing environment variables will result in empty strings with no error.

Sample file

Here's a sample configuration file, in each syntax:

Listing 4: .coveragerc

```
[run]
branch = True

[report]
; Regexes for lines to exclude from consideration
exclude_also =
    ; Don't complain about missing debug-only code:
    def __repr__
    if self\.debug

    ; Don't complain if tests don't hit defensive assertion code:
    raise AssertionError
    raise NotImplementedError

    ; Don't complain if non-runnable code isn't run:
    if 0:
    if __name__ == .__main__.:

    ; Don't complain about abstract methods, they aren't run:
    @(abc\.)?abstractmethod

ignore_errors = True

[html]
directory = coverage_html_report
```

Listing 5: pyproject.toml

```
[tool.coverage.run]
branch = true

[tool.coverage.report]
# Regexes for lines to exclude from consideration
exclude_also = [
    # Don't complain about missing debug-only code:
    "def __repr__",
```

```
    "if self\\.debug",

    # Don't complain if tests don't hit defensive assertion code:
    "raise AssertionError",
    "raise NotImplementedError",

    # Don't complain if non-runnable code isn't run:
    "if 0:",
    "if __name__ == .__main__.:",

    # Don't complain about abstract methods, they aren't run:
    "@(abc\\.)?abstractmethod",
    ]

ignore_errors = true

[tool.coverage.html]
directory = "coverage_html_report"
```

Listing 6: setup.cfg, tox.ini

```
[coverage:run]
branch = True

[coverage:report]
; Regexes for lines to exclude from consideration
exclude_also =
    ; Don't complain about missing debug-only code:
    def __repr__
    if self\.debug

    ; Don't complain if tests don't hit defensive assertion code:
    raise AssertionError
    raise NotImplementedError

    ; Don't complain if non-runnable code isn't run:
    if 0:
    if __name__ == .__main__.:

    ; Don't complain about abstract methods, they aren't run:
    @(abc\.)?abstractmethod

ignore_errors = True

[coverage:html]
directory = coverage_html_report
```

The specific configuration settings are described below. Many sections and settings correspond roughly to commands and options in the *command-line interface*.

6.3.2 [run]

These settings are generally used when running product code, though some apply to more than one command.

[run] branch

(boolean, default False) Whether to measure *branch coverage* in addition to statement coverage.

[run] command_line

(string) The command-line to run your program. This will be used if you run `coverage run` with no further arguments. Coverage.py options cannot be specified here, other than `-m` to indicate the module to run.

New in version 5.0.

[run] concurrency

(multi-string, default "thread") The concurrency libraries in use by the product code. If your program uses multiprocessing, gevent, greenlet, or eventlet, you must name that library in this option, or coverage.py will produce very wrong results.

See *Measuring sub-processes* for details of multi-process measurement.

Before version 4.2, this option only accepted a single string.

New in version 4.0.

[run] context

(string) The static context to record for this coverage run. See *Measurement contexts* for more information

New in version 5.0.

[run] cover_pylib

(boolean, default False) Whether to measure the Python standard library.

[run] data_file

(string, default ".coverage") The name of the data file to use for storing or reporting coverage. This value can include a path to another directory.

[run] disable_warnings

(multi-string) A list of warnings to disable. Warnings that can be disabled include a short string at the end, the name of the warning. See *Warnings* for specific warnings.

[run] debug

(multi-string) A list of debug options. See *the run –debug option* for details.

[run] debug_file

(string) A file name to write debug output to. See *the run –debug option* for details.

[run] dynamic_context

(string) The name of a strategy for setting the dynamic context during execution. See *Dynamic contexts* for details.

[run] include

(multi-string) A list of file name patterns, the files to include in measurement or reporting. Ignored if `source` is set. See *Specifying source files* for details.

[run] omit

(multi-string) A list of file name patterns, the files to leave out of measurement or reporting. See *Specifying source files* for details.

[run] parallel

(boolean, default False) Append the machine name, process id and random number to the data file name to simplify collecting data from many processes. See *Combining data files: coverage combine* for more information.

[run] plugins

(multi-string) A list of plugin package names. See *Plug-ins* for more information.

[run] relative_files

(boolean, default False) store relative file paths in the data file. This makes it easier to measure code in one (or multiple) environments, and then report in another. See *Combining data files: coverage combine* for details.

Note that setting `source` has to be done in the configuration file rather than the command line for this option to work, since the reporting commands need to know the source origin.

New in version 5.0.

[run] sigterm

(boolean, default False) if true, register a SIGTERM signal handler to capture data when the process ends due to a SIGTERM signal. This includes `Process.terminate`, and other ways to terminate a process. This can help when collecting data in usual situations, but can also introduce problems (see issue 1310).

Only on Linux and Mac.

New in version 6.4: (in 6.3 this was always enabled)

[run] source

(multi-string) A list of packages or directories, the source to measure during execution. If set, `include` is ignored. See *Specifying source files* for details.

[run] source_pkgs

(multi-string) A list of packages, the source to measure during execution. Operates the same as `source`, but only names packages, for resolving ambiguities between packages and directories.

New in version 5.3.

[run] timid

(boolean, default False) Use a simpler but slower trace method. This uses PyTracer instead of CTracer, and is only needed in very unusual circumstances. Try this if you get seemingly impossible results.

6.3.3 [paths]

The entries in this section are lists of file paths that should be considered equivalent when combining data from different machines:

Listing 7: .coveragerc

```
[paths]
source =
    src/
    /jenkins/build/*/src
    c:\myproj\src
```

Listing 8: pyproject.toml

```
[tool.coverage.paths]
source = [
    "src/",
    "/jenkins/build/*/src",
    "c:\\myproj\\src",
    ]
```

Listing 9: setup.cfg, tox.ini

```
[coverage:paths]
source =
    src/
    /jenkins/build/*/src
    c:\myproj\src
```

The names of the entries ("source" in this example) are ignored, you may choose any name that you like. The value is a list of strings. When combining data with the `combine` command, two file paths will be combined if they start with paths from the same list.

The first value must be an actual file path on the machine where the reporting will happen, so that source code can be found. The other values can be file patterns to match against the paths of collected data, or they can be absolute or relative file paths on the current machine.

In this example, data collected for "/jenkins/build/1234/src/module.py" will be combined with data for "c:\myproj\src\module.py", and will be reported against the source file found at "src/module.py".

If you specify more than one list of paths, they will be considered in order. A file path will only be remapped if the result exists. If a path matches a list, but the result doesn't exist, the next list will be tried. The first list that has an existing result will be used.

Remapping will also be done during reporting, but only within the single data file being reported. Combining multiple files requires the `combine` command.

The `--debug=pathmap` option can be used to log details of the re-mapping of paths. See *the –debug option*.

See *Re-mapping paths* and *File patterns* for more information.

6.3.4 [report]

Settings common to many kinds of reporting.

[report] exclude_also

(multi-string) A list of regular expressions. This setting is similar to *[report] exclude_lines*: it specifies patterns for lines to exclude from reporting. This setting is preferred, because it will preserve the default exclude patterns instead of overwriting them.

New in version 7.2.0.

[report] exclude_lines

(multi-string) A list of regular expressions. Any line of your source code containing a match for one of these regexes is excluded from being reported as missing. More details are in *Excluding code from coverage.py*. If you use this option, you are replacing all the exclude regexes, so you'll need to also supply the "pragma: no cover" regex if you still want to use it. The *[report] exclude_also* setting can be used to specify patterns without overwriting the default set.

You can exclude lines introducing blocks, and the entire block is excluded. If you exclude a `def` line or decorator line, the entire function is excluded.

Be careful when writing this setting: the values are regular expressions that only have to match a portion of the line. For example, if you write `...`, you'll exclude any line with three or more of any character. If you write `pass`, you'll also exclude the line `my_pass="foo"`, and so on.

[report] fail_under

(float) A target coverage percentage. If the total coverage measurement is under this value, then exit with a status code of 2. If you specify a non-integral value, you must also set `[report] precision` properly to make use of the decimal places. A setting of 100 will fail any value under 100, regardless of the number of decimal places of precision.

[report] ignore_errors

(boolean, default False) Ignore source code that can't be found, emitting a warning instead of an exception.

[report] include

(multi-string) A list of file name patterns, the files to include in reporting. See *Specifying source files* for details.

[report] include_namespace_packages

(boolean, default False) When searching for completely un-executed files, include directories without `__init__.py` files. These are implicit namespace packages, and are usually skipped.

New in version 7.0.

[report] omit

(multi-string) A list of file name patterns, the files to leave out of reporting. See *Specifying source files* for details.

[report] partial_branches

(multi-string) A list of regular expressions. Any line of code that matches one of these regexes is excused from being reported as a partial branch. More details are in *Branch coverage measurement*. If you use this option, you are replacing all the partial branch regexes so you'll need to also supply the "pragma: no branch" regex if you still want to use it.

[report] precision

(integer) The number of digits after the decimal point to display for reported coverage percentages. The default is 0, displaying for example "87%". A value of 2 will display percentages like "87.32%". This setting also affects the interpretation of the `fail_under` setting.

[report] show_missing

(boolean, default False) When running a summary report, show missing lines. See *Coverage summary: coverage report* for more information.

[report] skip_covered

(boolean, default False) Don't report files that are 100% covered. This helps you focus on files that need attention.

[report] skip_empty

(boolean, default False) Don't report files that have no executable code (such as `__init__.py` files).

[report] sort

(string, default "Name") Sort the text report by the named column. Allowed values are "Name", "Stmts", "Miss", "Branch", "BrPart", or "Cover". Prefix with – for descending sort (for example, "-cover").

6.3.5 [html]

Settings particular to HTML reporting. The settings in the `[report]` section also apply to HTML output, where appropriate.

[html] directory

(string, default "htmlcov") Where to write the HTML report files.

[html] extra_css

(string) The path to a file of CSS to apply to the HTML report. The file will be copied into the HTML output directory. Don't name it "style.css". This CSS is in addition to the CSS normally used, though you can overwrite as many of the rules as you like.

[html] show_contexts

(boolean) Should the HTML report include an indication on each line of which contexts executed the line. See *Dynamic contexts* for details.

[html] skip_covered

(boolean, defaulted from `[report] skip_covered`) Don't include files in the report that are 100% covered files. See *Coverage summary: coverage report* for more information.

New in version 5.4.

[html] skip_empty

(boolean, defaulted from `[report] skip_empty`) Don't include empty files (those that have 0 statements) in the report. See *Coverage summary: coverage report* for more information.

New in version 5.4.

[html] title

(string, default "Coverage report") The title to use for the report. Note this is text, not HTML.

6.3.6 [xml]

Settings particular to XML reporting. The settings in the `[report]` section also apply to XML output, where appropriate.

[xml] output

(string, default "coverage.xml") Where to write the XML report.

[xml] package_depth

(integer, default 99) Controls which directories are identified as packages in the report. Directories deeper than this depth are not reported as packages. The default is that all directories are reported as packages.

6.3.7 [json]

Settings particular to JSON reporting. The settings in the `[report]` section also apply to JSON output, where appropriate.

New in version 5.0.

[json] output

(string, default "coverage.json") Where to write the JSON file.

[json] pretty_print

(boolean, default false) Controls if the JSON is outputted with white space formatted for human consumption (True) or for minimum file size (False).

[json] show_contexts

(boolean, default false) Should the JSON report include an indication of which contexts executed each line. See *Dynamic contexts* for details.

6.3.8 [lcov]

Settings particular to LCOV reporting (see *LCOV reporting: coverage lcov*).

New in version 6.3.

[lcov] output

(string, default "coverage.lcov") Where to write the LCOV file.

6.4 Specifying source files

When coverage.py is running your program and measuring its execution, it needs to know what code to measure and what code not to. Measurement imposes a speed penalty, and the collected data must be stored in memory and then on disk. More importantly, when reviewing your coverage reports, you don't want to be distracted with modules that aren't your concern.

Coverage.py has a number of ways you can focus it in on the code you care about.

6.4.1 Execution

When running your code, the `coverage run` command will by default measure all code, unless it is part of the Python standard library.

You can specify source to measure with the `--source` command-line switch, or the `[run] source` configuration value. The value is a comma- or newline-separated list of directories or importable names (packages or modules).

If the source option is specified, only code in those locations will be measured. Specifying the source option also enables coverage.py to report on un-executed files, since it can search the source tree for files that haven't been measured at all. Only importable files (ones at the root of the tree, or in directories with a `__init__.py` file) will be considered. Files with unusual punctuation in their names will be skipped (they are assumed to be scratch files written by text editors). Files that do not end with `.py`, `.pyw`, `.pyo`, or `.pyc` will also be skipped.

Note: Modules named as sources may be imported twice, once by coverage.py to find their location, then again by your own code or test suite. Usually this isn't a problem, but could cause trouble if a module has side-effects at import time.

Exceptions during the early import are suppressed and ignored.

You can further fine-tune coverage.py's attention with the `--include` and `--omit` switches (or `[run] include` and `[run] omit` configuration values). `--include` is a list of file name patterns. If specified, only files matching those patterns will be measured. `--omit` is also a list of file name patterns, specifying files not to measure. If both `include` and `omit` are specified, first the set of files is reduced to only those that match the include patterns, then any files that match the omit pattern are removed from the set.

The `include` and `omit` file name patterns follow common shell syntax, described below in *File patterns*. Patterns that start with a wildcard character are used as-is, other patterns are interpreted relative to the current directory:

Listing 10: .coveragerc

```
[run]
omit =
    # omit anything in a .local directory anywhere
    */.local/*
    # omit everything in /usr
    /usr/*
    # omit this single file
    utils/tirefire.py
```

Listing 11: pyproject.toml

```
[tool.coverage.run]
omit = [
    # omit anything in a .local directory anywhere
    "*/.local/*",
    # omit everything in /usr
    "/usr/*",
    # omit this single file
    "utils/tirefire.py",
    ]
```

Listing 12: setup.cfg, tox.ini

```
[coverage:run]
omit =
    # omit anything in a .local directory anywhere
    */.local/*
    # omit everything in /usr
    /usr/*
    # omit this single file
    utils/tirefire.py
```

The `source`, `include`, and `omit` values all work together to determine the source that will be measured.

If both `source` and `include` are set, the `include` value is ignored and a warning is issued.

6.4.2 Reporting

Once your program is measured, you can specify the source files you want reported. Usually you want to see all the code that was measured, but if you are measuring a large project, you may want to get reports for just certain parts.

The report commands (`report`, `html`, `json`, `lcov`, `annotate`, and `xml`) all take optional `modules` arguments, and `--include` and `--omit` switches. The `modules` arguments specify particular modules to report on. The `include` and `omit` values are lists of file name patterns, just as with the `run` command.

Remember that the reporting commands can only report on the data that has been collected, so the data you're looking for may not be in the data available for reporting.

Note that these are ways of specifying files to measure. You can also exclude individual source lines. See *Excluding code from coverage.py* for details.

6.4.3 File patterns

File path patterns are used for *include* and *omit*, and for *combining path remapping*. They follow common shell syntax:

- ? matches a single file name character.

- * matches any number of file name characters, not including the directory separator. As a special case, if a pattern starts with */, it is treated as **/, and if a pattern ends with /*, it is treated as /**.

- ** matches any number of nested directory names, including none. It must be used as a full component of the path, not as part of a word: /**/ is allowed, but /a**/ is not.

- Both / and \ will match either a slash or a backslash, to make cross-platform matching easier.

- A pattern with no directory separators matches the file name in any directory.

Some examples:

Pattern	Matches	Doesn't Match
`a*.py`	anything.py sub1/sub2/another.py	cat.py
`sub/*/*.py`	sub/a/main.py sub/b/another.py	sub/foo.py sub/m1/m2/foo.py
`sub/**/*.py`	sub/something.py sub/a/main.py sub/b/another.py sub/m1/m2/foo.py	sub1/anything.py sub1/more/code/main.py
`*/sub/*`	some/where/sub/more/something.py sub/hello.py	sub1/anything.py
`*/sub*/*`	some/where/sub/more/something.py sub/hello.py sub1/anything.py	some/more/something.py
`*/*sub/test_*.py`	some/where/sub/test_everything.py moresub/test_things.py	some/where/sub/more/test_everything.py more/test_things.py
`*/*sub/*sub/**`	sub/sub/something.py asub/bsub/more/thing.py code/sub/sub/code.py	sub/something.py

6.5 Excluding code from coverage.py

You may have code in your project that you know won't be executed, and you want to tell coverage.py to ignore it. For example, you may have debugging-only code that won't be executed during your unit tests. You can tell coverage.py to exclude this code during reporting so that it doesn't clutter your reports with noise about code that you don't need to hear about.

Coverage.py will look for comments marking clauses for exclusion. In this code, the "if debug" clause is excluded from reporting:

```
a = my_function1()
if debug:   # pragma: no cover
    msg = "blah blah"
    log_message(msg, a)
b = my_function2()
```

Any line with a comment of "pragma: no cover" is excluded. If that line introduces a clause, for example, an if clause, or a function or class definition, then the entire clause is also excluded. Here the __repr__ function is not reported as missing:

```
class MyObject(object):
    def __init__(self):
        blah1()
        blah2()

    def __repr__(self):   # pragma: no cover
        return "<MyObject>"
```

Excluded code is executed as usual, and its execution is recorded in the coverage data as usual. When producing reports though, coverage.py excludes it from the list of missing code.

6.5.1 Branch coverage

When measuring *branch coverage*, a conditional will not be counted as a branch if one of its choices is excluded:

```
def only_one_choice(x):
    if x:
        blah1()
        blah2()
    else:   # pragma: no cover
        # x is always true.
        blah3()
```

Because the `else` clause is excluded, the `if` only has one possible next line, so it isn't considered a branch at all.

6.5.2 Advanced exclusion

Coverage.py identifies exclusions by matching lines against a list of regular expressions. Using *configuration files* or the coverage *API*, you can add to that list. This is useful if you have often-used constructs to exclude that can be matched with a regex. You can exclude them all at once without littering your code with exclusion pragmas.

If the matched line introduces a block, the entire block is excluded from reporting. Matching a `def` line or decorator line will exclude an entire function.

For example, you might decide that __repr__ functions are usually only used in debugging code, and are uninteresting to test themselves. You could exclude all of them by adding a regex to the exclusion list:

Listing 13: .coveragerc

```
[report]
exclude_also =
    def __repr__
```

6.5. Excluding code from coverage.py

Listing 14: pyproject.toml

```
[tool.coverage.report]
exclude_also = [
    "def __repr__",
    ]
```

Listing 15: setup.cfg, tox.ini

```
[coverage:report]
exclude_also =
    def __repr__
```

For example, here's a list of exclusions I've used:

Listing 16: .coveragerc

```
[report]
exclude_also =
    def __repr__
    if self.debug:
    if settings.DEBUG
    raise AssertionError
    raise NotImplementedError
    if 0:
    if __name__ == .__main__.:
    if TYPE_CHECKING:
    class .*\bProtocol\):
    @(abc\.)?abstractmethod
```

Listing 17: pyproject.toml

```
[tool.coverage.report]
exclude_also = [
    "def __repr__",
    "if self.debug:",
    "if settings.DEBUG",
    "raise AssertionError",
    "raise NotImplementedError",
    "if 0:",
    "if __name__ == .__main__.:",
    "if TYPE_CHECKING:",
    "class .*\\bProtocol\\):",
    "@(abc\\.)?abstractmethod",
    ]
```

Listing 18: setup.cfg, tox.ini

```
[coverage:report]
exclude_also =
    def __repr__
    if self.debug:
```

(continues on next page)

```
if settings.DEBUG
raise AssertionError
raise NotImplementedError
if 0:
if __name__ == .__main__.:
if TYPE_CHECKING:
class .*\bProtocol\):
@(abc\.)?abstractmethod
```

The *[report] exclude_also* option adds regexes to the built-in default list so that you can add your own exclusions. The older *[report] exclude_lines* option completely overwrites the list of regexes.

The regexes only have to match part of a line. Be careful not to over-match. A value of `...` will match any line with more than three characters in it.

A similar pragma, "no branch", can be used to tailor branch coverage measurement. See *Branch coverage measurement* for details.

6.5.3 Excluding source files

See *Specifying source files* for ways to limit what files coverage.py measures or reports on.

6.6 Branch coverage measurement

In addition to the usual statement coverage, coverage.py also supports branch coverage measurement. Where a line in your program could jump to more than one next line, coverage.py tracks which of those destinations are actually visited, and flags lines that haven't visited all of their possible destinations.

For example:

```
1  def my_partial_fn(x):
2      if x:
3          y = 10
4      return y
5
6  my_partial_fn(1)
```

In this code, line 2 is an `if` statement which can go next to either line 3 or line 4. Statement coverage would show all lines of the function as executed. But the if was never evaluated as false, so line 2 never jumps to line 4.

Branch coverage will flag this code as not fully covered because of the missing jump from line 2 to line 4. This is known as a partial branch.

6.6.1 How to measure branch coverage

To measure branch coverage, run coverage.py with the `--branch` flag:

```
coverage run --branch myprog.py
```

When you report on the results with `coverage report` or `coverage html`, the percentage of branch possibilities taken will be included in the percentage covered total for each file. The coverage percentage for a file is the actual executions divided by the execution opportunities. Each line in the file is an execution opportunity, as is each branch destination.

The HTML report gives information about which lines had missing branches. Lines that were missing some branches are shown in yellow, with an annotation at the far right showing branch destination line numbers that were not exercised.

The XML and JSON reports produced by `coverage xml` and `coverage json` also include branch information, including separate statement and branch coverage percentages.

6.6.2 How it works

When measuring branches, coverage.py collects pairs of line numbers, a source and destination for each transition from one line to another. Static analysis of the source provides a list of possible transitions. Comparing the measured to the possible indicates missing branches.

The idea of tracking how lines follow each other was from Titus Brown. Thanks, Titus!

6.6.3 Excluding code

If you have *excluded code*, a conditional will not be counted as a branch if one of its choices is excluded:

```python
1  def only_one_choice(x):
2      if x:
3          blah1()
4          blah2()
5      else:  # pragma: no cover
6          # x is always true.
7          blah3()
```

Because the `else` clause is excluded, the `if` only has one possible next line, so it isn't considered a branch at all.

6.6.4 Structurally partial branches

Sometimes branching constructs are used in unusual ways that don't actually branch. For example:

```python
while True:
    if cond:
        break
    do_something()
```

Here the while loop will never exit normally, so it doesn't take both of its "possible" branches. For some of these constructs, such as "while True:" and "if 0:", coverage.py understands what is going on. In these cases, the line will not be marked as a partial branch.

But there are many ways in your own code to write intentionally partial branches, and you don't want coverage.py pestering you about them. You can tell coverage.py that you don't want them flagged by marking them with a pragma:

```
i = 0
while i < 999999999:   # pragma: no branch
    if eventually():
        break
```

Here the while loop will never complete because the break will always be taken at some point. Coverage.py can't work that out on its own, but the "no branch" pragma indicates that the branch is known to be partial, and the line is not flagged.

6.7 Measuring sub-processes

Complex test suites may spawn sub-processes to run tests, either to run them in parallel, or because sub-process behavior is an important part of the system under test. Measuring coverage in those sub-processes can be tricky because you have to modify the code spawning the process to invoke coverage.py.

There's an easier way to do it: coverage.py includes a function, *coverage.process_startup()* designed to be invoked when Python starts. It examines the `COVERAGE_PROCESS_START` environment variable, and if it is set, begins coverage measurement. The environment variable's value will be used as the name of the *configuration file* to use.

Note: The subprocess only sees options in the configuration file. Options set on the command line will not be used in the subprocesses.

Note: If you have subprocesses created with `multiprocessing`, the `--concurrency=multiprocessing` command-line option should take care of everything for you. See *Execution: coverage run* for details.

When using this technique, be sure to set the parallel option to true so that multiple coverage.py runs will each write their data to a distinct file.

6.7.1 Configuring Python for sub-process measurement

Measuring coverage in sub-processes is a little tricky. When you spawn a sub-process, you are invoking Python to run your program. Usually, to get coverage measurement, you have to use coverage.py to run your program. Your sub-process won't be using coverage.py, so we have to convince Python to use coverage.py even when not explicitly invoked.

To do that, we'll configure Python to run a little coverage.py code when it starts. That code will look for an environment variable that tells it to start coverage measurement at the start of the process.

To arrange all this, you have to do two things: set a value for the `COVERAGE_PROCESS_START` environment variable, and then configure Python to invoke *coverage.process_startup()* when Python processes start.

How you set `COVERAGE_PROCESS_START` depends on the details of how you create sub-processes. As long as the environment variable is visible in your sub-process, it will work.

You can configure your Python installation to invoke the `process_startup` function in two ways:

1. Create or append to sitecustomize.py to add these lines:

   ```
   import coverage
   coverage.process_startup()
   ```

2. Create a .pth file in your Python installation containing:

```
import coverage; coverage.process_startup()
```

The sitecustomize.py technique is cleaner, but may involve modifying an existing sitecustomize.py, since there can be only one. If there is no sitecustomize.py already, you can create it in any directory on the Python path.

The .pth technique seems like a hack, but works, and is documented behavior. On the plus side, you can create the file with any name you like so you don't have to coordinate with other .pth files. On the minus side, you have to create the file in a system-defined directory, so you may need privileges to write it.

Note that if you use one of these techniques, you must undo them if you uninstall coverage.py, since you will be trying to import it during Python start-up. Be sure to remove the change when you uninstall coverage.py, or use a more defensive approach to importing it.

6.7.2 Process termination

To successfully write a coverage data file, the Python sub-process under analysis must shut down cleanly and have a chance for coverage.py to run its termination code. It will do that when the process ends naturally, or when a SIGTERM signal is received.

Coverage.py uses `atexit` to handle usual process ends, and a `signal` handler to catch SIGTERM signals.

Other ways of ending a process, like SIGKILL or `os._exit`, will prevent coverage.py from writing its data file, leaving you with incomplete or non-existent coverage data.

6.8 Measurement contexts

New in version 5.0.

Coverage.py measures whether code was run, but it can also record the context in which it was run. This can provide more information to help you understand the behavior of your tests.

There are two kinds of context: static and dynamic. Static contexts are fixed for an entire run, and are set explicitly with an option. Dynamic contexts change over the course of a single run.

6.8.1 Static contexts

A static context is set by an option when you run coverage.py. The value is fixed for the duration of a run. They can be any text you like, for example, "python3" or "with_numpy". The context is recorded with the data.

When you *combine multiple data files* together, they can have differing contexts. All of the information is retained, so that the different contexts are correctly recorded in the combined file.

A static context is specified with the `--context=CONTEXT` option to *the coverage run command*, or the `[run]` `context` setting in the configuration file.

6.8.2 Dynamic contexts

Dynamic contexts are found during execution. They are most commonly used to answer the question "what test ran this line?," but have been generalized to allow any kind of context tracking. As execution proceeds, the dynamic context changes to record the context of execution. Separate data is recorded for each context, so that it can be analyzed later.

There are three ways to enable dynamic contexts:

- you can set the `[run] dynamic_context` option in your .coveragerc file, or

- you can enable a *dynamic context switcher* plugin, or

- another tool (such as a test runner) can call the `Coverage.switch_context()` method to set the context explicitly. The pytest plugin pytest-cov has a `--cov-context` option that uses this to set the dynamic context for each test.

The `[run] dynamic_context` setting has only one option now. Set it to `test_function` to start a new dynamic context for every test function:

Listing 19: .coveragerc

```
[run]
dynamic_context = test_function
```

Listing 20: pyproject.toml

```
[tool.coverage.run]
dynamic_context = "test_function"
```

Listing 21: setup.cfg, tox.ini

```
[coverage:run]
dynamic_context = test_function
```

Each test function you run will be considered a separate dynamic context, and coverage data will be segregated for each. A test function is any function whose name starts with "test".

If you have both a static context and a dynamic context, they are joined with a pipe symbol to be recorded as a single string.

Initially, when your program starts running, the dynamic context is an empty string. Any code measured before a dynamic context is set will be recorded in this empty context. For example, if you are recording test names as contexts, then the code run by the test runner before (and between) tests will be in the empty context.

Dynamic contexts can be explicitly disabled by setting `dynamic_context` to `none`.

6.8.3 Context reporting

The `coverage report` and `coverage html` commands both accept `--contexts` option, a comma-separated list of regular expressions. The report will be limited to the contexts that match one of those patterns.

The `coverage html` command also has `--show-contexts`. If set, the HTML report will include an annotation on each covered line indicating the number of contexts that executed the line. Clicking the annotation displays a list of the contexts.

6.8.4 Raw data

For more advanced reporting or analysis, the .coverage data file is a SQLite database. See *Coverage.py database schema* for details.

6.9 Coverage.py API

There are a few different ways to use coverage.py programmatically.

The API to coverage.py is in a module called `coverage`. Most of the interface is in the `coverage.Coverage` class. Methods on the Coverage object correspond roughly to operations available in the command line interface. For example, a simple use would be:

```python
import coverage

cov = coverage.Coverage()
cov.start()

# .. call your code ..

cov.stop()
cov.save()

cov.html_report()
```

Any of the methods can raise specialized exceptions described in *Coverage exceptions*.

Coverage.py supports plugins that can change its behavior, to collect information from non-Python files, or to perform complex configuration. See *Plug-in classes* for details.

If you want to access the data that coverage.py has collected, the `coverage.CoverageData` class provides an API to read coverage.py data files.

Note: Only the documented portions of the API are supported. Other names you may find in modules or objects can change their behavior at any time. Please limit yourself to documented methods to avoid problems.

For more intensive data use, you might want to access the coverage.py database file directly. The schema is subject to change, so this is for advanced uses only. *Coverage.py database schema* explains more.

6.9.1 The Coverage class

class coverage.**Coverage**(*data_file=MISSING*, *data_suffix=None*, *cover_pylib=None*, *auto_data=False*, *timid=None*, *branch=None*, *config_file=True*, *source=None*, *source_pkgs=None*, *omit=None*, *include=None*, *debug=None*, *concurrency=None*, *check_preimported=False*, *context=None*, *messages=False*)

Programmatic access to coverage.py.

To use:

```python
from coverage import Coverage

cov = Coverage()
```

```
cov.start()
#.. call your code ..
cov.stop()
cov.html_report(directory="covhtml")
```

A context manager is available to do the same thing:

```
cov = Coverage()
with cov.collect():
    #.. call your code ..
cov.html_report(directory="covhtml")
```

Note: in keeping with Python custom, names starting with underscore are not part of the public API. They might stop working at any point. Please limit yourself to documented methods to avoid problems.

Methods can raise any of the exceptions described in *Coverage exceptions*.

> **Parameters**
>
> > - **data_file** (*Optional[Union[FilePath, DefaultValue]]*) –
> > - **data_suffix** (*Optional[Union[*str*, *bool*]]*) –
> > - **cover_pylib** (*Optional[*bool*]*) –
> > - **auto_data** (*bool*) –
> > - **timid** (*Optional[*bool*]*) –
> > - **branch** (*Optional[*bool*]*) –
> > - **config_file** (*Union[FilePath, *bool*]*) –
> > - **source** (*Optional[Iterable[*str*]]*) –
> > - **source_pkgs** (*Optional[Iterable[*str*]]*) –
> > - **omit** (*Optional[Union[*str*, Iterable[*str*]]]*) –
> > - **include** (*Optional[Union[*str*, Iterable[*str*]]]*) –
> > - **debug** (*Optional[Iterable[*str*]]*) –
> > - **concurrency** (*Optional[Union[*str*, Iterable[*str*]]]*) –
> > - **check_preimported** (*bool*) –
> > - **context** (*Optional[*str*]*) –
> > - **messages** (*bool*) –

__init__(*data_file=MISSING*, *data_suffix=None*, *cover_pylib=None*, *auto_data=False*, *timid=None*, *branch=None*, *config_file=True*, *source=None*, *source_pkgs=None*, *omit=None*, *include=None*, *debug=None*, *concurrency=None*, *check_preimported=False*, *context=None*, *messages=False*)

> Many of these arguments duplicate and override values that can be provided in a configuration file. Parameters that are missing here will use values from the config file.
>
> *data_file* is the base name of the data file to use. The config value defaults to ".coverage". None can be provided to prevent writing a data file. *data_suffix* is appended (with a dot) to *data_file* to create the final file name. If *data_suffix* is simply True, then a suffix is created with the machine and process identity included.
>
> *cover_pylib* is a boolean determining whether Python code installed with the Python interpreter is measured. This includes the Python standard library and any packages installed with the interpreter.

If *auto_data* is true, then any existing data file will be read when coverage measurement starts, and data will be saved automatically when measurement stops.

If *timid* is true, then a slower and simpler trace function will be used. This is important for some environments where manipulation of tracing functions breaks the faster trace function.

If *branch* is true, then branch coverage will be measured in addition to the usual statement coverage.

config_file determines what configuration file to read:

- If it is ".coveragerc", it is interpreted as if it were True, for backward compatibility.

- If it is a string, it is the name of the file to read. If the file can't be read, it is an error.

- If it is True, then a few standard files names are tried (".coveragerc", "setup.cfg", "tox.ini"). It is not an error for these files to not be found.

- If it is False, then no configuration file is read.

source is a list of file paths or package names. Only code located in the trees indicated by the file paths or package names will be measured.

source_pkgs is a list of package names. It works the same as *source*, but can be used to name packages where the name can also be interpreted as a file path.

include and *omit* are lists of file name patterns. Files that match *include* will be measured, files that match *omit* will not. Each will also accept a single string argument.

debug is a list of strings indicating what debugging information is desired.

concurrency is a string indicating the concurrency library being used in the measured code. Without this, coverage.py will get incorrect results if these libraries are in use. Valid strings are "greenlet", "eventlet", "gevent", "multiprocessing", or "thread" (the default). This can also be a list of these strings.

If *check_preimported* is true, then when coverage is started, the already-imported files will be checked to see if they should be measured by coverage. Importing measured files before coverage is started can mean that code is missed.

context is a string to use as the *static context* label for collected data.

If *messages* is true, some messages will be printed to stdout indicating what is happening.

New in version 4.0: The *concurrency* parameter.

New in version 4.2: The *concurrency* parameter can now be a list of strings.

New in version 5.0: The *check_preimported* and *context* parameters.

New in version 5.3: The *source_pkgs* parameter.

New in version 6.0: The *messages* parameter.

> **Parameters**
>
> - **data_file** (*str | PathLike | DefaultValue | None*) –
> - **data_suffix** (*str | bool | None*) –
> - **cover_pylib** (*bool | None*) –
> - **auto_data** (*bool*) –
> - **timid** (*bool | None*) –
> - **branch** (*bool | None*) –
> - **config_file** (*str | PathLike | bool*) –

- **source** (*Iterable[str] | None*) –

- **source_pkgs** (*Iterable[str] | None*) –

- **omit** (*str | Iterable[str] | None*) –

- **include** (*str | Iterable[str] | None*) –

- **debug** (*Iterable[str] | None*) –

- **concurrency** (*str | Iterable[str] | None*) –

- **check_preimported** (*bool*) –

- **context** (*str | None*) –

- **messages** (*bool*) –

> **Return type**
>> None

analysis(*morf*)

> Like *analysis2* but doesn't return excluded line numbers.
>
> > **Parameters**
> >> **morf** (*module | str*) –
> >
> > **Return type**
> >> *Tuple*[str, *List*[int], *List*[int], str]

analysis2(*morf*)

> Analyze a module.
>
> *morf* is a module or a file name. It will be analyzed to determine its coverage statistics. The return value is a 5-tuple:
>
> - The file name for the module.
>
> - A list of line numbers of executable statements.
>
> - A list of line numbers of excluded statements.
>
> - A list of line numbers of statements not run (missing from execution).
>
> - A readable formatted string of the missing line numbers.
>
> The analysis uses the source file itself and the current measured coverage data.
>
> > **Parameters**
> >> **morf** (*module | str*) –
> >
> > **Return type**
> >> *Tuple*[str, *List*[int], *List*[int], *List*[int], str]

annotate(*morfs=None*, *directory=None*, *ignore_errors=None*, *omit=None*, *include=None*, *contexts=None*)

> Annotate a list of modules.
>
> ---
>
> **Note:** This method has been obsoleted by more modern reporting tools, including the *html_report()* method. It will be removed in a future version.
>
> ---
>
> Each module in *morfs* is annotated. The source is written to a new file, named with a ",cover" suffix, with each line prefixed with a marker to indicate the coverage of the line. Covered lines have ">", excluded lines have "-", and missing lines have "!".

See `report()` for other arguments.

> **Parameters**
>
> - **morfs** (`Iterable[module | str] | None`) –
> - **directory** (`str | None`) –
> - **ignore_errors** (`bool | None`) –
> - **omit** (`str | List[str] | None`) –
> - **include** (`str | List[str] | None`) –
> - **contexts** (`List[str] | None`) –
>
> **Return type**
> > None

clear_exclude(*which='exclude'*)

> Clear the exclude list.
>
> > **Parameters**
> > > **which** (`str`) –
> >
> > **Return type**
> > > None

collect()

> A context manager to start/stop coverage measurement collection.
>
> New in version 7.3.
>
> > **Return type**
> > > *Iterator*[None]

combine(*data_paths=None*, *strict=False*, *keep=False*)

> Combine together a number of similarly-named coverage data files.
>
> All coverage data files whose name starts with *data_file* (from the coverage() constructor) will be read, and combined together into the current measurements.
>
> *data_paths* is a list of files or directories from which data should be combined. If no list is passed, then the data files from the directory indicated by the current data file (probably the current directory) will be combined.
>
> If *strict* is true, then it is an error to attempt to combine when there are no data files to combine.
>
> If *keep* is true, then original input data files won't be deleted.
>
> New in version 4.0: The *data_paths* parameter.
>
> New in version 4.3: The *strict* parameter.
>
> > **Parameters**
> >
> > - **data_paths** (`Iterable[str] | None`) –
> > - **strict** (`bool`) –
> > - **keep** (`bool`) –
> >
> > **Return type**
> > > None

classmethod current()

> Get the latest started *Coverage* instance, if any.
>
> Returns: a *Coverage* instance, or None.
>
> New in version 5.0.
>
> > **Return type**
> > Coverage | None

erase()

> Erase previously collected coverage data.
>
> This removes the in-memory data collected in this session as well as discarding the data file.
>
> > **Return type**
> > None

exclude(*regex*, *which='exclude'*)

> Exclude source lines from execution consideration.
>
> A number of lists of regular expressions are maintained. Each list selects lines that are treated differently during reporting.
>
> *which* determines which list is modified. The "exclude" list selects lines that are not considered executable at all. The "partial" list indicates lines with branches that are not taken.
>
> *regex* is a regular expression. The regex is added to the specified list. If any of the regexes in the list is found in a line, the line is marked for special treatment during reporting.
>
> > **Parameters**
> >
> > - **regex** (*str*) –
> >
> > - **which** (*str*) –
> >
> > **Return type**
> > None

get_data()

> Get the collected data.
>
> Also warn about various problems collecting data.
>
> Returns a *coverage.CoverageData*, the collected coverage data.
>
> New in version 4.0.
>
> > **Return type**
> > CoverageData

get_exclude_list(*which='exclude'*)

> Return a list of excluded regex strings.
>
> *which* indicates which list is desired. See exclude() for the lists that are available, and their meaning.
>
> > **Parameters**
> > **which** (*str*) –
> >
> > **Return type**
> > *List*[str]

get_option(*option_name*)

> Get an option from the configuration.
>
> *option_name* is a colon-separated string indicating the section and option name. For example, the `branch` option in the `[run]` section of the config file would be indicated with *"run:branch"*.
>
> Returns the value of the option. The type depends on the option selected.
>
> As a special case, an *option_name* of `"paths"` will return an dictionary with the entire `[paths]` section value.
>
> New in version 4.0.
>
> > **Parameters**
> > **option_name** (`str`) –
> >
> > **Return type**
> > bool | int | float | str | *List*[str] | None

html_report(*morfs=None*, *directory=None*, *ignore_errors=None*, *omit=None*, *include=None*, *extra_css=None*, *title=None*, *skip_covered=None*, *show_contexts=None*, *contexts=None*, *skip_empty=None*, *precision=None*)

> Generate an HTML report.
>
> The HTML is written to *directory*. The file "index.html" is the overview starting point, with links to more detailed pages for individual modules.
>
> *extra_css* is a path to a file of other CSS to apply on the page. It will be copied into the HTML directory.
>
> *title* is a text string (not HTML) to use as the title of the HTML report.
>
> See *report()* for other arguments.
>
> Returns a float, the total percentage covered.

Note: The HTML report files are generated incrementally based on the source files and coverage results. If you modify the report files, the changes will not be considered. You should be careful about changing the files in the report folder.

> **Parameters**
> - **morfs** (`Iterable[module | str] | None`) –
> - **directory** (`str | None`) –
> - **ignore_errors** (`bool | None`) –
> - **omit** (`str | List[str] | None`) –
> - **include** (`str | List[str] | None`) –
> - **extra_css** (`str | None`) –
> - **title** (`str | None`) –
> - **skip_covered** (`bool | None`) –
> - **show_contexts** (`bool | None`) –
> - **contexts** (`List[str] | None`) –
> - **skip_empty** (`bool | None`) –
> - **precision** (`int | None`) –

> **Return type**
>> float

json_report(*morfs=None*, *outfile=None*, *ignore_errors=None*, *omit=None*, *include=None*, *contexts=None*, *pretty_print=None*, *show_contexts=None*)

> Generate a JSON report of coverage results.

> Each module in *morfs* is included in the report. *outfile* is the path to write the file to, "-" will write to stdout.

> *pretty_print* is a boolean, whether to pretty-print the JSON output or not.

> See `report()` for other arguments.

> Returns a float, the total percentage covered.

> New in version 5.0.

>> **Parameters**
>>
>> - **morfs** (*Iterable[module | str] | None*) –
>> - **outfile** (*str | None*) –
>> - **ignore_errors** (*bool | None*) –
>> - **omit** (*str | List[str] | None*) –
>> - **include** (*str | List[str] | None*) –
>> - **contexts** (*List[str] | None*) –
>> - **pretty_print** (*bool | None*) –
>> - **show_contexts** (*bool | None*) –
>>
>> **Return type**
>>> float

lcov_report(*morfs=None*, *outfile=None*, *ignore_errors=None*, *omit=None*, *include=None*, *contexts=None*)

> Generate an LCOV report of coverage results.

> Each module in *morfs* is included in the report. *outfile* is the path to write the file to, "-" will write to stdout.

> See `report()` for other arguments.

> New in version 6.3.

>> **Parameters**
>>
>> - **morfs** (*Iterable[module | str] | None*) –
>> - **outfile** (*str | None*) –
>> - **ignore_errors** (*bool | None*) –
>> - **omit** (*str | List[str] | None*) –
>> - **include** (*str | List[str] | None*) –
>> - **contexts** (*List[str] | None*) –
>>
>> **Return type**
>>> float

load()

> Load previously-collected coverage data from the data file.
>
>> **Return type**
>>> None

report(*morfs=None*, *show_missing=None*, *ignore_errors=None*, *file=None*, *omit=None*, *include=None*, *skip_covered=None*, *contexts=None*, *skip_empty=None*, *precision=None*, *sort=None*, *output_format=None*)

> Write a textual summary report to *file*.
>
> Each module in *morfs* is listed, with counts of statements, executed statements, missing statements, and a list of lines missed.
>
> If *show_missing* is true, then details of which lines or branches are missing will be included in the report. If *ignore_errors* is true, then a failure while reporting a single file will not stop the entire report.
>
> *file* is a file-like object, suitable for writing.
>
> *output_format* determines the format, either "text" (the default), "markdown", or "total".
>
> *include* is a list of file name patterns. Files that match will be included in the report. Files matching *omit* will not be included in the report.
>
> If *skip_covered* is true, don't report on files with 100% coverage.
>
> If *skip_empty* is true, don't report on empty files (those that have no statements).
>
> *contexts* is a list of regular expression strings. Only data from *dynamic contexts* that match one of those expressions (using `re.search`) will be included in the report.
>
> *precision* is the number of digits to display after the decimal point for percentages.
>
> All of the arguments default to the settings read from the *configuration file*.
>
> Returns a float, the total percentage covered.
>
> New in version 4.0: The *skip_covered* parameter.
>
> New in version 5.0: The *contexts* and *skip_empty* parameters.
>
> New in version 5.2: The *precision* parameter.
>
> New in version 7.0: The *format* parameter.
>
>> **Parameters**
>>
>>> - **morfs** (`Iterable[module | str] | None`) –
>>> - **show_missing** (`bool | None`) –
>>> - **ignore_errors** (`bool | None`) –
>>> - **file** (`IO[str] | None`) –
>>> - **omit** (`str | List[str] | None`) –
>>> - **include** (`str | List[str] | None`) –
>>> - **skip_covered** (`bool | None`) –
>>> - **contexts** (`List[str] | None`) –
>>> - **skip_empty** (`bool | None`) –
>>> - **precision** (`int | None`) –
>>> - **sort** (`str | None`) –

> • **output_format** (*str | None*) –

>> **Return type**
>>> float

save()

> Save the collected coverage data to the data file.

>> **Return type**
>>> None

set_option(*option_name*, *value*)

> Set an option in the configuration.

> *option_name* is a colon-separated string indicating the section and option name. For example, the `branch` option in the `[run]` section of the config file would be indicated with `"run:branch"`.

> *value* is the new value for the option. This should be an appropriate Python value. For example, use True for booleans, not the string `"True"`.

> As an example, calling:

```python
cov.set_option("run:branch", True)
```

> has the same effect as this configuration file:

```ini
[run]
branch = True
```

> As a special case, an *option_name* of `"paths"` will replace the entire `[paths]` section. The value should be a dictionary.

> New in version 4.0.

>> **Parameters**

>>> • **option_name** (*str*) –

>>> • **value** (*bool | int | float | str | Iterable[str] | None | Mapping[str, bool | int | float | str | Iterable[str] | None]*) –

>> **Return type**
>>> None

start()

> Start measuring code coverage.

> Coverage measurement is only collected in functions called after *start()* is invoked. Statements in the same scope as *start()* won't be measured.

> Once you invoke *start()*, you must also call *stop()* eventually, or your process might not shut down cleanly.

> The *collect()* method is a context manager to handle both starting and stopping collection.

>> **Return type**
>>> None

stop()

> Stop measuring code coverage.

>> **Return type**
>>> None

switch_context(*new_context*)

> Switch to a new dynamic context.
>
> *new_context* is a string to use as the *dynamic context* label for collected data. If a *static context* is in use, the static and dynamic context labels will be joined together with a pipe character.
>
> Coverage collection must be started already.
>
> New in version 5.0.
>
> > **Parameters**
> > > **new_context** (`str`) –
> >
> > **Return type**
> > > None

xml_report(*morfs=None*, *outfile=None*, *ignore_errors=None*, *omit=None*, *include=None*, *contexts=None*, *skip_empty=None*)

> Generate an XML report of coverage results.
>
> The report is compatible with Cobertura reports.
>
> Each module in *morfs* is included in the report. *outfile* is the path to write the file to, "-" will write to stdout.
>
> See `report()` for other arguments.
>
> Returns a float, the total percentage covered.
>
> > **Parameters**
> > > - **morfs** (`Iterable[module | str] | None`) –
> > > - **outfile** (`str | None`) –
> > > - **ignore_errors** (`bool | None`) –
> > > - **omit** (`str | List[str] | None`) –
> > > - **include** (`str | List[str] | None`) –
> > > - **contexts** (`List[str] | None`) –
> > > - **skip_empty** (`bool | None`) –
> >
> > **Return type**
> > > float

6.9.2 Coverage exceptions

class coverage.exceptions.**CoverageException**

> The base class of all exceptions raised by Coverage.py.

Exceptions coverage.py can raise.

exception coverage.exceptions.**ConfigError**

> A problem with a config file, or a value in one.

exception coverage.exceptions.**CoverageWarning**

> A warning from Coverage.py.

exception coverage.exceptions.**DataError**

> An error in using a data file.

exception `coverage.exceptions.`**`NoCode`**

> We couldn't find any code at all.

exception `coverage.exceptions.`**`NoDataError`**

> We didn't have data to work with.

exception `coverage.exceptions.`**`NoSource`**

> We couldn't find the source for a module.

exception `coverage.exceptions.`**`NotPython`**

> A source file turned out not to be parsable Python.

exception `coverage.exceptions.`**`PluginError`**

> A plugin misbehaved.

6.9.3 coverage module

The most important thing in the coverage module is the `coverage.Coverage` class, described in *The Coverage class*, but there are a few other things also.

`coverage.`**`version_info`**

A tuple of five elements, similar to `sys.version_info`: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. Unlike `sys.version_info`, the elements are not available by name.

`coverage.`**`__version__`**

A string with the version of coverage.py, for example, `"5.0b2"`.

class `coverage.`**`CoverageException`**

> The base class of all exceptions raised by Coverage.py.

Starting coverage.py automatically

This function is used to start coverage measurement automatically when Python starts. See *Measuring sub-processes* for details.

`coverage.`**`process_startup`**`()`

> Call this at Python start-up to perhaps measure coverage.
>
> If the environment variable COVERAGE_PROCESS_START is defined, coverage measurement is started. The value of the variable is the config file to use.
>
> There are two ways to configure your Python installation to invoke this function when Python starts:
>
> 1. Create or append to sitecustomize.py to add these lines:
>
>    ```python
>    import coverage
>    coverage.process_startup()
>    ```
>
> 2. Create a .pth file in your Python installation containing:
>
>    ```python
>    import coverage; coverage.process_startup()
>    ```
>
> Returns the *Coverage* instance that was started, or None if it was not started by this call.

> **Return type**
>> Coverage | None

6.9.4 Plug-in classes

New in version 4.0.

Plug-in interfaces for coverage.py.

Coverage.py supports a few different kinds of plug-ins that change its behavior:

- File tracers implement tracing of non-Python file types.

- Configurers add custom configuration, using Python code to change the configuration.

- Dynamic context switchers decide when the dynamic context has changed, for example, to record what test function produced the coverage.

To write a coverage.py plug-in, create a module with a subclass of *CoveragePlugin*. You will override methods in your class to participate in various aspects of coverage.py's processing. Different types of plug-ins have to override different methods.

Any plug-in can optionally implement *sys_info()* to provide debugging information about their operation.

Your module must also contain a `coverage_init` function that registers an instance of your plug-in class:

```python
import coverage

class MyPlugin(coverage.CoveragePlugin):
    ...

def coverage_init(reg, options):
    reg.add_file_tracer(MyPlugin())
```

You use the *reg* parameter passed to your `coverage_init` function to register your plug-in object. The registration method you call depends on what kind of plug-in it is.

If your plug-in takes options, the *options* parameter is a dictionary of your plug-in's options from the coverage.py configuration file. Use them however you want to configure your object before registering it.

Coverage.py will store its own information on your plug-in object, using attributes whose names start with `_coverage_`. Don't be startled.

> **Warning:** Plug-ins are imported by coverage.py before it begins measuring code. If you write a plugin in your own project, it might import your product code before coverage.py can start measuring. This can result in your own code being reported as missing.
>
> One solution is to put your plugins in your project tree, but not in your importable Python package.

File Tracers

File tracers implement measurement support for non-Python files. File tracers implement the `file_tracer()` method to claim files and the `file_reporter()` method to report on those files.

In your `coverage_init` function, use the `add_file_tracer` method to register your file tracer.

Configurers

New in version 4.5.

Configurers modify the configuration of coverage.py during start-up. Configurers implement the `configure()` method to change the configuration.

In your `coverage_init` function, use the `add_configurer` method to register your configurer.

Dynamic Context Switchers

New in version 5.0.

Dynamic context switcher plugins implement the `dynamic_context()` method to dynamically compute the context label for each measured frame.

Computed context labels are useful when you want to group measured data without modifying the source code.

For example, you could write a plugin that checks *frame.f_code* to inspect the currently executed method, and set the context label to a fully qualified method name if it's an instance method of *unittest.TestCase* and the method name starts with 'test'. Such a plugin would provide basic coverage grouping by test and could be used with test runners that have no built-in coveragepy support.

In your `coverage_init` function, use the `add_dynamic_context` method to register your dynamic context switcher.

The CoveragePlugin class

class coverage.**CoveragePlugin**

> Base class for coverage.py plug-ins.

> **file_tracer**(*filename*)
>
> > Get a `FileTracer` object for a file.
> >
> > Plug-in type: file tracer.
> >
> > Every Python source file is offered to your plug-in to give it a chance to take responsibility for tracing the file. If your plug-in can handle the file, it should return a `FileTracer` object. Otherwise return None.
> >
> > There is no way to register your plug-in for particular files. Instead, this method is invoked for all files as they are executed, and the plug-in decides whether it can trace the file or not. Be prepared for *filename* to refer to all kinds of files that have nothing to do with your plug-in.
> >
> > The file name will be a Python file being executed. There are two broad categories of behavior for a plug-in, depending on the kind of files your plug-in supports:
> >
> > - Static file names: each of your original source files has been converted into a distinct Python file. Your plug-in is invoked with the Python file name, and it maps it back to its original source file.
> >
> > - Dynamic file names: all of your source files are executed by the same Python file. In this case, your plug-in implements `FileTracer.dynamic_source_filename()` to provide the actual source file for each execution frame.

filename is a string, the path to the file being considered. This is the absolute real path to the file. If you are comparing to other paths, be sure to take this into account.

Returns a `FileTracer` object to use to trace *filename*, or None if this plug-in cannot trace this file.

> **Parameters**
>   **filename** (`str`) –
>
> **Return type**
>   FileTracer | None

file_reporter(*filename*)

> Get the `FileReporter` class to use for a file.
>
> Plug-in type: file tracer.
>
> This will only be invoked if *filename* returns non-None from `file_tracer()`. It's an error to return None from this method.
>
> Returns a `FileReporter` object to use to report on *filename*, or the string *"python"* to have coverage.py treat the file as Python.
>
> > **Parameters**
> >   **filename** (`str`) –
> >
> > **Return type**
> >   FileReporter | str

dynamic_context(*frame*)

> Get the dynamically computed context label for *frame*.
>
> Plug-in type: dynamic context.
>
> This method is invoked for each frame when outside of a dynamic context, to see if a new dynamic context should be started. If it returns a string, a new context label is set for this and deeper frames. The dynamic context ends when this frame returns.
>
> Returns a string to start a new dynamic context, or None if no new context should be started.
>
> > **Parameters**
> >   **frame** (*frame*) –
> >
> > **Return type**
> >   str | None

find_executable_files(*src_dir*)

> Yield all of the executable files in *src_dir*, recursively.
>
> Plug-in type: file tracer.
>
> Executability is a plug-in-specific property, but generally means files which would have been considered for coverage analysis, had they been included automatically.
>
> Returns or yields a sequence of strings, the paths to files that could have been executed, including files that had been executed.
>
> > **Parameters**
> >   **src_dir** (`str`) –
> >
> > **Return type**
> >   *Iterable*[str]

configure(*config*)

>   Modify the configuration of coverage.py.

>   Plug-in type: configurer.

>   This method is called during coverage.py start-up, to give your plug-in a chance to change the configuration. The *config* parameter is an object with `get_option()` and `set_option()` methods. Do not call any other methods on the *config* object.

>> **Parameters**
>>> **config** (*TConfigurable*) –

>> **Return type**
>>> None

sys_info()

>   Get a list of information useful for debugging.

>   Plug-in type: any.

>   This method will be invoked for `--debug=sys`. Your plug-in can return any information it wants to be displayed.

>   Returns a list of pairs: *[(name, value), ...]*.

>> **Return type**
>>> *Iterable*[*Tuple*[str, *Any*]]

The FileTracer class

class coverage.**FileTracer**

>   Support needed for files during the execution phase.

>   File tracer plug-ins implement subclasses of FileTracer to return from their `file_tracer()` method.

>   You may construct this object from `CoveragePlugin.file_tracer()` any way you like. A natural choice would be to pass the file name given to *file_tracer*.

>   *FileTracer* objects should only be created in the `CoveragePlugin.file_tracer()` method.

>   See *How coverage.py works* for details of the different coverage.py phases.

>   **source_filename**()

>>   The source file name for this file.

>>   This may be any file name you like. A key responsibility of a plug-in is to own the mapping from Python execution back to whatever source file name was originally the source of the code.

>>   See `CoveragePlugin.file_tracer()` for details about static and dynamic file names.

>>   Returns the file name to credit with this execution.

>>> **Return type**
>>>> str

has_dynamic_source_filename()

>   Does this FileTracer have dynamic source file names?

>   FileTracers can provide dynamically determined file names by implementing `dynamic_source_filename()`. Invoking that function is expensive. To determine whether to invoke it, coverage.py uses the result of this function to know if it needs to bother invoking `dynamic_source_filename()`.

See `CoveragePlugin.file_tracer()` for details about static and dynamic file names.

Returns True if `dynamic_source_filename()` should be called to get dynamic source file names.

> **Return type**
> bool

dynamic_source_filename(*filename*, *frame*)

Get a dynamically computed source file name.

Some plug-ins need to compute the source file name dynamically for each frame.

This function will not be invoked if `has_dynamic_source_filename()` returns False.

Returns the source file name for this frame, or None if this frame shouldn't be measured.

> **Parameters**
>
> • **filename** (`str`) –
>
> • **frame** (*frame*) –
>
> **Return type**
> str | None

line_number_range(*frame*)

Get the range of source line numbers for a given a call frame.

The call frame is examined, and the source line number in the original file is returned. The return value is a pair of numbers, the starting line number and the ending line number, both inclusive. For example, returning (5, 7) means that lines 5, 6, and 7 should be considered executed.

This function might decide that the frame doesn't indicate any lines from the source file were executed. Return (-1, -1) in this case to tell coverage.py that no lines should be recorded for this frame.

> **Parameters**
> **frame** (*frame*) –
>
> **Return type**
> *Tuple*[int, int]

The FileReporter class

class coverage.**FileReporter**(*filename*)

Support needed for files during the analysis and reporting phases.

File tracer plug-ins implement a subclass of *FileReporter*, and return instances from their `CoveragePlugin.file_reporter()` method.

There are many methods here, but only `lines()` is required, to provide the set of executable lines in the file.

See *How coverage.py works* for details of the different coverage.py phases.

> **Parameters**
> **filename** (`str`) –

relative_filename()

Get the relative file name for this file.

This file path will be displayed in reports. The default implementation will supply the actual project-relative file path. You only need to supply this method if you have an unusual syntax for file paths.

> > > **Return type**
> > > > str

> > **source()**
> > > Get the source for the file.

> > > Returns a Unicode string.

> > > The base implementation simply reads the *self.filename* file and decodes it as UTF-8. Override this method if your file isn't readable as a text file, or if you need other encoding support.

> > > > **Return type**
> > > > > str

> > **lines()**
> > > Get the executable lines in this file.

> > > Your plug-in must determine which lines in the file were possibly executable. This method returns a set of those line numbers.

> > > Returns a set of line numbers.

> > > > **Return type**
> > > > > *Set*[int]

> > **excluded_lines()**
> > > Get the excluded executable lines in this file.

> > > Your plug-in can use any method it likes to allow the user to exclude executable lines from consideration.

> > > Returns a set of line numbers.

> > > The base implementation returns the empty set.

> > > > **Return type**
> > > > > *Set*[int]

> > **translate_lines**(*lines*)
> > > Translate recorded lines into reported lines.

> > > Some file formats will want to report lines slightly differently than they are recorded. For example, Python records the last line of a multi-line statement, but reports are nicer if they mention the first line.

> > > Your plug-in can optionally define this method to perform these kinds of adjustment.

> > > *lines* is a sequence of integers, the recorded line numbers.

> > > Returns a set of integers, the adjusted line numbers.

> > > The base implementation returns the numbers unchanged.

> > > > **Parameters**
> > > > > **lines** (`Iterable[int]`) –

> > > > **Return type**
> > > > > *Set*[int]

> > **arcs()**
> > > Get the executable arcs in this file.

> > > To support branch coverage, your plug-in needs to be able to indicate possible execution paths, as a set of line number pairs. Each pair is a *(prev, next)* pair indicating that execution can transition from the *prev* line number to the *next* line number.

> > > Returns a set of pairs of line numbers. The default implementation returns an empty set.

> **Return type**
>> *Set*[*Tuple*[int, int]]

no_branch_lines()

> Get the lines excused from branch coverage in this file.
>
> Your plug-in can use any method it likes to allow the user to exclude lines from consideration of branch coverage.
>
> Returns a set of line numbers.
>
> The base implementation returns the empty set.
>
>> **Return type**
>>> *Set*[int]

translate_arcs(*arcs*)

> Translate recorded arcs into reported arcs.
>
> Similar to `translate_lines()`, but for arcs. *arcs* is a set of line number pairs.
>
> Returns a set of line number pairs.
>
> The default implementation returns *arcs* unchanged.
>
>> **Parameters**
>>> **arcs** (`Iterable[Tuple[int, int]]`) –
>>
>> **Return type**
>>> *Set*[*Tuple*[int, int]]

exit_counts()

> Get a count of exits from that each line.
>
> To determine which lines are branches, coverage.py looks for lines that have more than one exit. This function creates a dict mapping each executable line number to a count of how many exits it has.
>
> To be honest, this feels wrong, and should be refactored. Let me know if you attempt to implement this method in your plug-in…
>
>> **Return type**
>>> *Dict*[int, int]

missing_arc_description(*start*, *end*, *executed_arcs=None*)

> Provide an English sentence describing a missing arc.
>
> The *start* and *end* arguments are the line numbers of the missing arc. Negative numbers indicate entering or exiting code objects.
>
> The *executed_arcs* argument is a set of line number pairs, the arcs that were executed in this file.
>
> By default, this simply returns the string "Line {start} didn't jump to {end}".
>
>> **Parameters**
>>
>>> • **start** (`int`) –
>>>
>>> • **end** (`int`) –
>>>
>>> • **executed_arcs** (`Iterable[Tuple[int, int]] | None`) –
>>
>> **Return type**
>>> str

`source_token_lines()`

Generate a series of tokenized lines, one for each line in *source*.

These tokens are used for syntax-colored reports.

Each line is a list of pairs, each pair is a token:

```
[("key", "def"), ("ws", " "), ("nam", "hello"), ("op", "("), ... ]
```

Each pair has a token class, and the token text. The token classes are:

- `"com"`: a comment
- `"key"`: a keyword
- `"nam"`: a name, or identifier
- `"num"`: a number
- `"op"`: an operator
- `"str"`: a string literal
- `"ws"`: some white space
- `"txt"`: some other kind of text

If you concatenate all the token texts, and then join them with newlines, you should have your original source back.

The default implementation simply returns each line tagged as `"txt"`.

> **Return type**
> *Iterable*[*List*[*Tuple*[str, str]]]

6.9.5 The CoverageData class

New in version 4.0.

class coverage.**CoverageData**(*basename=None*, *suffix=None*, *no_disk=False*, *warn=None*, *debug=None*)

Manages collected coverage data, including file storage.

This class is the public supported API to the data that coverage.py collects during program execution. It includes information about what code was executed. It does not include information from the analysis phase, to determine what lines could have been executed, or what lines were not executed.

Note: The data file is currently a SQLite database file, with a *documented schema*. The schema is subject to change though, so be careful about querying it directly. Use this API if you can to isolate yourself from changes.

There are a number of kinds of data that can be collected:

- **lines**: the line numbers of source lines that were executed. These are always available.
- **arcs**: pairs of source and destination line numbers for transitions between source lines. These are only available if branch coverage was used.
- **file tracer names**: the module names of the file tracer plugins that handled each file in the data.

Lines, arcs, and file tracer names are stored for each source file. File names in this API are case-sensitive, even on platforms with case-insensitive file systems.

A data file either stores lines, or arcs, but not both.

A data file is associated with the data when the `CoverageData` is created, using the parameters *basename*, *suffix*, and *no_disk*. The base name can be queried with `base_filename()`, and the actual file name being used is available from `data_filename()`.

To read an existing coverage.py data file, use `read()`. You can then access the line, arc, or file tracer data with `lines()`, `arcs()`, or `file_tracer()`.

The `has_arcs()` method indicates whether arc data is available. You can get a set of the files in the data with `measured_files()`. As with most Python containers, you can determine if there is any data at all by using this object as a boolean value.

The contexts for each line in a file can be read with `contexts_by_lineno()`.

To limit querying to certain contexts, use `set_query_context()` or `set_query_contexts()`. These will narrow the focus of subsequent `lines()`, `arcs()`, and `contexts_by_lineno()` calls. The set of all measured context names can be retrieved with `measured_contexts()`.

Most data files will be created by coverage.py itself, but you can use methods here to create data files if you like. The `add_lines()`, `add_arcs()`, and `add_file_tracers()` methods add data, in ways that are convenient for coverage.py.

To record data for contexts, use `set_context()` to set a context to be used for subsequent `add_lines()` and `add_arcs()` calls.

To add a source file without any measured data, use `touch_file()`, or `touch_files()` for a list of such files.

Write the data to its file with `write()`.

You can clear the data in memory with `erase()`. Data for specific files can be removed from the database with `purge_files()`.

Two data collections can be combined by using `update()` on one `CoverageData`, passing it the other.

Data in a `CoverageData` can be serialized and deserialized with `dumps()` and `loads()`.

The methods used during the coverage.py collection phase (`add_lines()`, `add_arcs()`, `set_context()`, and `add_file_tracers()`) are thread-safe. Other methods may not be.

> **Parameters**
>
> - **basename** (`Optional[FilePath]`) –
> - **suffix** (`Optional[Union[str, bool]]`) –
> - **no_disk** (`bool`) –
> - **warn** (`Optional[TWarnFn]`) –
> - **debug** (`Optional[TDebugCtl]`) –

`__init__`(*basename=None*, *suffix=None*, *no_disk=False*, *warn=None*, *debug=None*)

> Create a `CoverageData` object to hold coverage-measured data.
>
> **Parameters**
>
> - **basename** (`str`) – the base name of the data file, defaulting to ".coverage". This can be a path to a file in another directory.
> - **suffix** (`str or bool`) – has the same meaning as the *data_suffix* argument to `coverage.Coverage`.
> - **no_disk** (`bool`) – if True, keep all data in memory, and don't write any disk file.
> - **warn** (`TWarnFn | None`) – a warning callback function, accepting a warning message argument.

- **debug** (*TDebugCtl | None*) – a *DebugControl* object (optional)

> **Return type**
> > None

add_arcs(*arc_data*)

> Add measured arc data.
>
> *arc_data* is a dictionary mapping file names to iterables of pairs of ints:

```
{ filename: { (l1,l2), (l1,l2), ... }, ...}
```

> > **Parameters**
> > > **arc_data** (`Mapping[str, Collection[Tuple[int, int]]]`) –
> >
> > **Return type**
> > > None

add_file_tracers(*file_tracers*)

> Add per-file plugin information.
>
> *file_tracers* is { filename: plugin_name, … }

> > **Parameters**
> > > **file_tracers** (`Mapping[str, str]`) –
> >
> > **Return type**
> > > None

add_lines(*line_data*)

> Add measured line data.
>
> *line_data* is a dictionary mapping file names to iterables of ints:

```
{ filename: { line1, line2, ... }, ...}
```

> > **Parameters**
> > > **line_data** (`Mapping[str, Collection[int]]`) –
> >
> > **Return type**
> > > None

arcs(*filename*)

> Get the list of arcs executed for a file.
>
> If the file was not measured, returns None. A file might be measured, and have no arcs executed, in which case an empty list is returned.
>
> If the file was executed, returns a list of 2-tuples of integers. Each pair is a starting line number and an ending line number for a transition from one line to another. The list is in no particular order.
>
> Negative numbers have special meaning. If the starting line number is -N, it represents an entry to the code object that starts at line N. If the ending ling number is -N, it's an exit from the code object that starts at line N.

> > **Parameters**
> > > **filename** (`str`) –
> >
> > **Return type**
> > > *List*[*Tuple*[int, int]] | None

---

base_filename()

> The base filename for storing data.
>
> New in version 5.0.
>
> > **Return type**
> >
> > > str

contexts_by_lineno(*filename*)

> Get the contexts for each line in a file.
>
> > **Returns**
> >
> > > A dict mapping line numbers to a list of context names.
> >
> > **Parameters**
> >
> > > **filename** (*str*) –
> >
> > **Return type**
> >
> > > *Dict*[int, *List*[str]]
>
> New in version 5.0.

data_filename()

> Where is the data stored?
>
> New in version 5.0.
>
> > **Return type**
> >
> > > str

dumps()

> Serialize the current data to a byte string.
>
> The format of the serialized data is not documented. It is only suitable for use with *loads()* in the same version of coverage.py.
>
> Note that this serialization is not what gets stored in coverage data files. This method is meant to produce bytes that can be transmitted elsewhere and then deserialized with *loads()*.
>
> > **Returns**
> >
> > > A byte string of serialized data.
> >
> > **Return type**
> >
> > > bytes
>
> New in version 5.0.

erase(*parallel=False*)

> Erase the data in this object.
>
> If *parallel* is true, then also deletes data files created from the basename by parallel-mode.
>
> > **Parameters**
> >
> > > **parallel** (*bool*) –
> >
> > **Return type**
> >
> > > None

file_tracer(*filename*)

> Get the plugin name of the file tracer for a file.
>
> Returns the name of the plugin that handles this file. If the file was measured, but didn't use a plugin, then "" is returned. If the file was not measured, then None is returned.

> > **Parameters**
> > > **filename** (*str*) –
> >
> > **Return type**
> > > str | None

has_arcs()

> Does the database have arcs (True) or lines (False).
>
> > **Return type**
> > > bool

lines(*filename*)

> Get the list of lines executed for a source file.
>
> If the file was not measured, returns None. A file might be measured, and have no lines executed, in which case an empty list is returned.
>
> If the file was executed, returns a list of integers, the line numbers executed in the file. The list is in no particular order.
>
> > **Parameters**
> > > **filename** (*str*) –
> >
> > **Return type**
> > > *List*[int] | None

loads(*data*)

> Deserialize data from *dumps()*.
>
> Use with a newly-created empty *CoverageData* object. It's undefined what happens if the object already has data in it.
>
> Note that this is not for reading data from a coverage data file. It is only for use on data you produced with *dumps()*.
>
> > **Parameters**
> > > **data** (*bytes*) – A byte string of serialized data produced by *dumps()*.
> >
> > **Return type**
> > > None
>
> New in version 5.0.

measured_contexts()

> A set of all contexts that have been measured.
>
> New in version 5.0.
>
> > **Return type**
> > > *Set*[str]

measured_files()

> A set of all files that have been measured.
>
> Note that a file may be mentioned as measured even though no lines or arcs for that file are present in the data.
>
> > **Return type**
> > > *Set*[str]

purge_files(*filenames*)

> Purge any existing coverage data for the given *filenames*.
>
> New in version 7.2.
>
> > **Parameters**
> > > **filenames** (`Collection[str]`) –
> >
> > **Return type**
> > > None

read()

> Start using an existing data file.
>
> > **Return type**
> > > None

set_context(*context*)

> Set the current context for future *add_lines()* etc.
>
> *context* is a str, the name of the context to use for the next data additions. The context persists until the next *set_context()*.
>
> New in version 5.0.
>
> > **Parameters**
> > > **context** (`str | None`) –
> >
> > **Return type**
> > > None

set_query_context(*context*)

> Set a context for subsequent querying.
>
> The next *lines()*, *arcs()*, or *contexts_by_lineno()* calls will be limited to only one context. *context* is a string which must match a context exactly. If it does not, no exception is raised, but queries will return no data.
>
> New in version 5.0.
>
> > **Parameters**
> > > **context** (`str`) –
> >
> > **Return type**
> > > None

set_query_contexts(*contexts*)

> Set a number of contexts for subsequent querying.
>
> The next *lines()*, *arcs()*, or *contexts_by_lineno()* calls will be limited to the specified contexts. *contexts* is a list of Python regular expressions. Contexts will be matched using `re.search`. Data will be included in query results if they are part of any of the contexts matched.
>
> New in version 5.0.
>
> > **Parameters**
> > > **contexts** (`Sequence[str] | None`) –
> >
> > **Return type**
> > > None

classmethod sys_info()

Our information for *Coverage.sys_info*.

Returns a list of (key, value) pairs.

> **Return type**
>> *List*[*Tuple*[str, *Any*]]

touch_file(*filename*, *plugin_name=''*)

Ensure that *filename* appears in the data, empty if needed.

plugin_name is the name of the plugin responsible for this file. It is used to associate the right filereporter, etc.

> **Parameters**
>> • **filename** (*str*) –
>>
>> • **plugin_name** (*str*) –
>
> **Return type**
>> None

touch_files(*filenames*, *plugin_name=None*)

Ensure that *filenames* appear in the data, empty if needed.

plugin_name is the name of the plugin responsible for these files. It is used to associate the right filereporter, etc.

> **Parameters**
>> • **filenames** (*Collection[str]*) –
>>
>> • **plugin_name** (*str | None*) –
>
> **Return type**
>> None

update(*other_data*, *aliases=None*)

Update this data with data from several other *CoverageData* instances.

If *aliases* is provided, it's a *PathAliases* object that is used to re-map paths to match the local machine's. Note: *aliases* is None only when called directly from the test suite.

> **Parameters**
>> • **other_data** (*CoverageData*) –
>>
>> • **aliases** (*PathAliases | None*) –
>
> **Return type**
>> None

write()

Ensure the data is written to the data file.

> **Return type**
>> None

6.9.6 Coverage.py database schema

New in version 5.0.

Coverage.py stores data in a SQLite database, by default called `.coverage`. For most needs, the *CoverageData* API will be sufficient, and should be preferred to accessing the database directly. Only advanced uses will need to use the database.

The schema can change without changing the major version of coverage.py, so be careful when accessing the database directly. The `coverage_schema` table has the schema number of the database. The schema described here corresponds to:

```
SCHEMA_VERSION = 7
```

You can use SQLite tools such as the `sqlite3` module in the Python standard library to access the data. Some data is stored in a packed format that will need custom functions to access. See *register_sqlite_functions()*.

Database schema

This is the database schema:

```sql
CREATE TABLE coverage_schema (
    -- One row, to record the version of the schema in this db.
    version integer
);

CREATE TABLE meta (
    -- Key-value pairs, to record metadata about the data
    key text,
    value text,
    unique (key)
    -- Possible keys:
    --  'has_arcs' boolean      -- Is this data recording branches?
    --  'sys_argv' text         -- The coverage command line that recorded the data.
    --  'version' text          -- The version of coverage.py that made the file.
    --  'when' text             -- Datetime when the file was created.
);

CREATE TABLE file (
    -- A row per file measured.
    id integer primary key,
    path text,
    unique (path)
);

CREATE TABLE context (
    -- A row per context measured.
    id integer primary key,
    context text,
    unique (context)
);

CREATE TABLE line_bits (
    -- If recording lines, a row per context per file executed.
```

(continues on next page)

```
    -- All of the line numbers for that file/context are in one numbits.
    file_id integer,            -- foreign key to `file`.
    context_id integer,         -- foreign key to `context`.
    numbits blob,               -- see the numbits functions in coverage.numbits
    foreign key (file_id) references file (id),
    foreign key (context_id) references context (id),
    unique (file_id, context_id)
);

CREATE TABLE arc (
    -- If recording branches, a row per context per from/to line transition executed.
    file_id integer,            -- foreign key to `file`.
    context_id integer,         -- foreign key to `context`.
    fromno integer,             -- line number jumped from.
    tono integer,               -- line number jumped to.
    foreign key (file_id) references file (id),
    foreign key (context_id) references context (id),
    unique (file_id, context_id, fromno, tono)
);

CREATE TABLE tracer (
    -- A row per file indicating the tracer used for that file.
    file_id integer primary key,
    tracer text,
    foreign key (file_id) references file (id)
);
```

Numbits

Functions to manipulate packed binary representations of number sets.

To save space, coverage stores sets of line numbers in SQLite using a packed binary representation called a numbits.
A numbits is a set of positive integers.

A numbits is stored as a blob in the database. The exact meaning of the bytes in the blobs should be considered an
implementation detail that might change in the future. Use these functions to work with those binary blobs of data.

coverage.numbits.**num_in_numbits**(*num*, *numbits*)

> Does the integer *num* appear in *numbits*?

> > **Returns**
> > > A bool, True if *num* is a member of *numbits*.

> > **Parameters**
> > > - **num** (*int*) –
> > > - **numbits** (*bytes*) –

> > **Return type**
> > > bool

coverage.numbits.**numbits_any_intersection**(*numbits1*, *numbits2*)

> Is there any number that appears in both numbits?

> Determine whether two number sets have a non-empty intersection. This is faster than computing the intersection.

> **Returns**
>> A bool, True if there is any number in both *numbits1* and *numbits2*.
>
> **Parameters**
>> - **numbits1** (*bytes*) –
>>
>> - **numbits2** (*bytes*) –
>
> **Return type**
>> bool

coverage.numbits.**numbits_intersection**(*numbits1*, *numbits2*)

> Compute the intersection of two numbits.
>
>> **Returns**
>>> A new numbits, the intersection *numbits1* and *numbits2*.
>>
>> **Parameters**
>>> - **numbits1** (*bytes*) –
>>>
>>> - **numbits2** (*bytes*) –
>>
>> **Return type**
>>> bytes

coverage.numbits.**numbits_to_nums**(*numbits*)

> Convert a numbits into a list of numbers.
>
>> **Parameters**
>>> **numbits** (*bytes*) – a binary blob, the packed number set.
>>
>> **Returns**
>>> A list of ints.
>>
>> **Return type**
>>> *List*[int]
>
> When registered as a SQLite function by `register_sqlite_functions()`, this returns a string, a JSON-encoded list of ints.

coverage.numbits.**numbits_union**(*numbits1*, *numbits2*)

> Compute the union of two numbits.
>
>> **Returns**
>>> A new numbits, the union of *numbits1* and *numbits2*.
>>
>> **Parameters**
>>> - **numbits1** (*bytes*) –
>>>
>>> - **numbits2** (*bytes*) –
>>
>> **Return type**
>>> bytes

coverage.numbits.**nums_to_numbits**(*nums*)

> Convert *nums* into a numbits.
>
>> **Parameters**
>>> **nums** (*Iterable[int]*) – a reusable iterable of integers, the line numbers to store.
>>
>> **Returns**
>>> A binary blob.

> **Return type**
>> bytes

coverage.numbits.**register_sqlite_functions**(*connection*)

> Define numbits functions in a SQLite connection.
>
> This defines these functions for use in SQLite statements:
>
> - *numbits_union()*
> - *numbits_intersection()*
> - *numbits_any_intersection()*
> - *num_in_numbits()*
> - *numbits_to_nums()*
>
> *connection* is a sqlite3.Connection object. After creating the connection, pass it to this function to register the numbits functions. Then you can use numbits functions in your queries:

```python
import sqlite3
from coverage.numbits import register_sqlite_functions

conn = sqlite3.connect("example.db")
register_sqlite_functions(conn)
c = conn.cursor()
# Kind of a nonsense query:
# Find all the files and contexts that executed line 47 in any file:
c.execute(
    "select file_id, context_id from line_bits where num_in_numbits(?, numbits)",
    (47,)
)
```

> **Parameters**
>> **connection** (*Connection*) –
>
> **Return type**
>> None

6.10 How coverage.py works

For advanced use of coverage.py, or just because you are curious, it helps to understand what's happening behind the scenes.

Coverage.py works in three phases:

- **Execution**: Coverage.py runs your code, and monitors it to see what lines were executed.
- **Analysis**: Coverage.py examines your code to determine what lines could have run.
- **Reporting**: Coverage.py combines the results of execution and analysis to produce a coverage number and an indication of missing execution.

The execution phase is handled by the `coverage run` command. The analysis and reporting phases are handled by the reporting commands like `coverage report` or `coverage html`.

As a short-hand, I say that coverage.py measures what lines were executed. But it collects more information than that. It can measure what branches were taken, and if you have contexts enabled, for each line or branch, it will also measure what contexts they were executed in.

Let's look at each phase in more detail.

6.10.1 Execution

At the heart of the execution phase is a trace function. This is a function that the Python interpreter invokes for each line executed in a program. Coverage.py implements a trace function that records each file and line number as it is executed.

For more details of trace functions, see the Python docs for sys.settrace, or if you are really brave, How C trace functions really work.

Executing a function for every line in your program can make execution very slow. Coverage.py's trace function is implemented in C to reduce that overhead. It also takes care to not trace code that you aren't interested in.

When measuring branch coverage, the same trace function is used, but instead of recording line numbers, coverage.py records pairs of line numbers. Each invocation of the trace function remembers the line number, then the next invocation records the pair *(prev, this)* to indicate that execution transitioned from the previous line to this line. Internally, these are called arcs.

As the data is being collected, coverage.py writes the data to a file, usually named `.coverage`. This is a *SQLite database* containing all of the measured data.

Plugins

Of course coverage.py mostly measures execution of Python files. But it can also be used to analyze other kinds of execution. *File tracer plugins* provide support for non-Python files. For example, Django HTML templates result in Python code being executed somewhere, but as a developer, you want that execution mapped back to your .html template file.

During execution, each new Python file encountered is provided to the plugins to consider. A plugin can claim the file and then convert the runtime Python execution into source-level data to be recorded.

Dynamic contexts

When using *dynamic contexts*, there is a current dynamic context that changes over the course of execution. It starts as empty. While it is empty, every time a new function is entered, a check is made to see if the dynamic context should change. While a non-empty dynamic context is current, the check is skipped until the function that started the context returns.

6.10.2 Analysis

After your program has been executed and the line numbers recorded, coverage.py needs to determine what lines could have been executed. Luckily, compiled Python files (.pyc files) have a table of line numbers in them. Coverage.py reads this table to get the set of executable lines, with a little more source analysis to leave out things like docstrings.

The data file is read to get the set of lines that were executed. The difference between the executable lines and the executed lines are the lines that were not executed.

The same principle applies for branch measurement, though the process for determining possible branches is more involved. Coverage.py uses the abstract syntax tree of the Python source file to determine the set of possible branches.

6.10.3 Reporting

Once we have the set of executed lines and missing lines, reporting is just a matter of formatting that information in a useful way. Each reporting method (text, HTML, JSON, annotated source, XML) has a different output format, but the process is the same: write out the information in the particular format, possibly including the source code itself.

6.11 Plug-ins

Coverage.py's behavior can be extended with third-party plug-ins. A plug-in is a separately installed Python class that you register in your .coveragerc. Plugins can alter a number of aspects of coverage.py's behavior, including implementing coverage measurement for non-Python files.

Information about using plug-ins is on this page. To write a plug-in, see *Plug-in classes*.

New in version 4.0.

6.11.1 Using plug-ins

To use a coverage.py plug-in, you install it and configure it. For this example, let's say there's a Python package called `something` that provides a coverage.py plug-in called `something.plugin`.

1. Install the plug-in's package as you would any other Python package:

```
$ python3 -m pip install something
```

2. Configure coverage.py to use the plug-in. You do this by editing (or creating) your .coveragerc file, as described in *Configuration reference*. The `plugins` setting indicates your plug-in. It's a list of importable module names of plug-ins:

Listing 22: .coveragerc

```
[run]
plugins =
    something.plugin
```

Listing 23: pyproject.toml

```
[tool.coverage.run]
plugins = [ "something.plugin" ]
```

Listing 24: setup.cfg, tox.ini

```
[coverage:run]
plugins =
    something.plugin
```

3. If the plug-in needs its own configuration, you can add those settings in the .coveragerc file in a section named for the plug-in:

Listing 25: .coveragerc

```
[something.plugin]
option1 = True
option2 = abc.foo
```

Listing 26: pyproject.toml

```
[tool.coverage.something.plugin]
option1 = true
option2 = "abc.foo"
```

Listing 27: setup.cfg, tox.ini

```
[coverage:something.plugin]
option1 = True
option2 = abc.foo
```

Check the documentation for the plug-in for details on the options it takes.

4. Run your tests with coverage.py as you usually would. If you get a message like "Plugin file tracers (something.plugin) aren't supported with PyTracer," then you don't have the *C extension* installed. The C extension is needed for certain plug-ins.

6.11.2 Available plug-ins

Some coverage.py plug-ins you might find useful:

- Django template coverage.py plug-in: for measuring coverage in Django templates.

- Conditional coverage plug-in: for measuring coverage based on any rules you define! Can exclude different lines of code that are only executed on different platforms, python versions, and with different dependencies installed.

- Mako template coverage plug-in: for measuring coverage in Mako templates. Doesn't work yet, probably needs some changes in Mako itself.

6.12 Contributing to coverage.py

I welcome contributions to coverage.py. Over the years, hundreds of people have provided contributions of various sizes to add features, fix bugs, or just help diagnose thorny issues. This page should have all the information you need to make a contribution.

One source of history or ideas are the bug reports against coverage.py. There you can find ideas for requested features, or the remains of rejected ideas.

6.12.1 Before you begin

If you have an idea for coverage.py, run it by me before you begin writing code. This way, I can get you going in the right direction, or point you to previous work in the area. Things are not always as straightforward as they seem, and having the benefit of lessons learned by those before you can save you frustration.

6.12.2 Getting the code

The coverage.py code is hosted on a GitHub repository at https://github.com/nedbat/coveragepy. To get a working environment, follow these steps:

1. Fork the repo into your own GitHub account. The coverage.py code will then be copied into a GitHub repository at `https://github.com/GITHUB_USER/coveragepy` where GITHUB_USER is your GitHub username.

2. (Optional) Create a virtualenv to work in, and activate it. There are a number of ways to do this. Use the method you are comfortable with.

3. Clone the repository:

```
$ git clone https://github.com/GITHUB_USER/coveragepy
$ cd coveragepy
```

4. Install the requirements:

```
$ python3 -m pip install -r requirements/dev.in
```

Note: You may need to upgrade pip to install the requirements.

6.12.3 Running the tests

The tests are written mostly as standard unittest-style tests, and are run with pytest running under tox:

```
$ python3 -m tox -e py38
ROOT: tox-gh won't override envlist because tox is not running in GitHub Actions
py38: wheel-0.40.0-py3-none-any.whl already present in /Users/nedbatchelder/Library/
→Application Support/virtualenv/wheel/3.8/embed/3/wheel.json
py38: pip-23.1.2-py3-none-any.whl already present in /Users/nedbatchelder/Library/
→Application Support/virtualenv/wheel/3.8/embed/3/pip.json
py38: setuptools-67.8.0-py3-none-any.whl already present in /Users/nedbatchelder/Library/
→Application Support/virtualenv/wheel/3.8/embed/3/setuptools.json
py38: install_deps> python -m pip install -U -r requirements/pip.pip -r requirements/
→pytest.pip -r requirements/light-threads.pip
.pkg: install_requires> python -I -m pip install setuptools
.pkg: _optional_hooks> python /usr/local/virtualenvs/coverage/lib/python3.8/site-
→packages/pyproject_api/_backend.py True setuptools.build_meta
.pkg: get_requires_for_build_editable> python /usr/local/virtualenvs/coverage/lib/
→python3.8/site-packages/pyproject_api/_backend.py True setuptools.build_meta
.pkg: install_requires_for_build_editable> python -I -m pip install wheel
.pkg: build_editable> python /usr/local/virtualenvs/coverage/lib/python3.8/site-packages/
→pyproject_api/_backend.py True setuptools.build_meta
py38: install_package_deps> python -m pip install -U 'tomli; python_full_version <= "3.
→11.0a6"'
py38: install_package> python -m pip install -U --force-reinstall --no-deps .tox/.tmp/
```

```
↪package/1/coverage-7.2.8a0.dev1-0.editable-cp38-cp38-macosx_13_0_x86_64.whl
py38: commands[0]> python igor.py zip_mods
py38: commands[1]> python setup.py --quiet build_ext --inplace
py38: commands[2]> python -m pip install -q -e .
py38: commands[3]> python igor.py test_with_tracer c
=== CPython 3.8.17 with C tracer (.tox/py38/bin/python) ===
bringing up nodes...
.......................................................................................
↪... [  6%]
.........................................x.....x...............s..s..s.s.................
↪... [ 13%]
.......................................................................................
↪... [ 20%]
.......................................................................................
↪... [ 27%]
.......................................................................................
↪... [ 34%]
.......................................................................................
↪... [ 41%]
.......................................................................................
↪... [ 47%]
.......................................................................................
↪... [ 54%]
....................................................s...........s.......................
↪... [ 61%]
.......................................................................................
↪... [ 68%]
..........s............................s...........s...................................
↪... [ 75%]
................s...................s...........................................s...
↪... [ 82%]
..............................s........................................................
↪... [ 88%]
.......................................................................................
↪... [ 95%]
............................s............................           ␣
↪    [100%]
1332 passed, 14 skipped, 2 xfailed in 60.54s (0:01:00)
py38: commands[4]> python igor.py remove_extension
py38: commands[5]> python igor.py test_with_tracer py
=== CPython 3.8.17 with Python tracer (.tox/py38/bin/python) ===
bringing up nodes...
.......................................................................................
↪... [  6%]
...................................x...................................................
↪... [ 13%]
..ss.......................................x...................................ss.........
↪... [ 20%]
...........s...........................................ss...................s.............
↪sss.. [ 27%]
.ss.....s.............................................................................
↪... [ 34%]
```

```
.......................................................................................
→... [ 41%]
...............................................................s.........................
→... [ 47%]
....................................................s..s.ss..............
→... [ 54%]
...ss.sss.................................s.........s.........sss....
→... [ 61%]
.ss...............s.s...................s.............s.s...........................
→... [ 68%]
...................................................................................
→..s [ 75%]
.........................................s..........................s........
→... [ 82%]
...................ss.s.................ssss...............................
→... [ 88%]
................................................................................
→... [ 95%]
..............................s..............ss.........                              ␣
→    [100%]
1297 passed, 49 skipped, 2 xfailed in 44.59s
.pkg: _exit> python /usr/local/virtualenvs/coverage/lib/python3.8/site-packages/
→pyproject_api/_backend.py True setuptools.build_meta
  py38: OK (143.82=setup[23.23]+cmd[0.29,1.60,8.43,61.64,0.34,48.28] seconds)
  congratulations :) (144.93 seconds)
```

Tox runs the complete test suite twice for each version of Python you have installed. The first run uses the C implementation of the trace function, the second uses the Python implementation.

To limit tox to just a few versions of Python, use the `-e` switch:

```
$ python3 -m tox -e py38,py39
```

On the tox command line, options after `--` are passed to pytest. To run just a few tests, you can use pytest test selectors:

```
$ python3 -m tox -- tests/test_misc.py
$ python3 -m tox -- tests/test_misc.py::HasherTest
$ python3 -m tox -- tests/test_misc.py::HasherTest::test_string_hashing
```

These commands run the tests in one file, one class, and just one test, respectively. The pytest `-k` option selects tests based on a word in their name, which can be very convenient for ad-hoc test selection. Of course you can combine tox and pytest options:

```
$ python3 -m tox -q -e py310 -- -n 0 -vv -k hash
=== CPython 3.10.12 with C tracer (.tox/py310/bin/python) ===
=================================== test session starts␣
→========================================
platform darwin -- Python 3.10.12, pytest-7.3.2, pluggy-1.0.0 -- /Users/nedbatchelder/
→coverage/trunk/.tox/py310/bin/python
cachedir: .tox/py310/.pytest_cache
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/Users/
→nedbatchelder/coverage/trunk/.hypothesis/examples')
rootdir: /Users/nedbatchelder/coverage/trunk
```

```
configfile: pyproject.toml
plugins: hypothesis-6.78.3, flaky-3.7.0, xdist-3.3.1
collected 1348 items / 1338 deselected / 10 selected
run-last-failure: no previously failed tests, not deselecting items.

tests/test_data.py::CoverageDataTest::test_add_to_hash_with_lines PASSED          ␣
↪     [ 10%]
tests/test_data.py::CoverageDataTest::test_add_to_hash_with_arcs PASSED           ␣
↪     [ 20%]
tests/test_data.py::CoverageDataTest::test_add_to_lines_hash_with_missing_file PASSED  ␣
↪     [ 30%]
tests/test_data.py::CoverageDataTest::test_add_to_arcs_hash_with_missing_file PASSED   ␣
↪     [ 40%]
tests/test_execfile.py::RunPycFileTest::test_running_hashed_pyc PASSED            ␣
↪     [ 50%]
tests/test_misc.py::HasherTest::test_string_hashing PASSED                        ␣
↪     [ 60%]
tests/test_misc.py::HasherTest::test_bytes_hashing PASSED                         ␣
↪     [ 70%]
tests/test_misc.py::HasherTest::test_unicode_hashing PASSED                       ␣
↪     [ 80%]
tests/test_misc.py::HasherTest::test_dict_hashing PASSED                          ␣
↪     [ 90%]
tests/test_misc.py::HasherTest::test_dict_collision PASSED                        ␣
↪     [100%]

============================== 10 passed, 1338 deselected in 2.24s␣
↪===============================
Skipping tests with Python tracer: Only one tracer: no Python tracer for CPython
  py310: OK (17.99 seconds)
  congratulations :) (19.09 seconds)
```

You can also affect the test runs with environment variables. Define any of these as 1 to use them:

- `COVERAGE_NO_PYTRACER=1` disables the Python tracer if you only want to run the CTracer tests.

- `COVERAGE_NO_CTRACER=1` disables the C tracer if you only want to run the PyTracer tests.

- `COVERAGE_ONE_TRACER=1` will use only one tracer for each Python version. This will use the C tracer if it is available, or the Python tracer if not.

- `COVERAGE_AST_DUMP=1` will dump the AST tree as it is being used during code parsing.

There are other environment variables that affect tests. I use set_env.py as a simple terminal interface to see and set them.

Of course, run all the tests on every version of Python you have, before submitting a change.

6.12.4 Lint, etc

I try to keep the coverage.py source as clean as possible. I use pylint to alert me to possible problems:

```
$ make lint
```

The source is pylint-clean, even if it's because there are pragmas quieting some warnings. Please try to keep it that way, but don't let pylint warnings keep you from sending patches. I can clean them up.

Lines should be kept to a 100-character maximum length. I recommend an editorconfig.org plugin for your editor of choice, which will also help with indentation, line endings and so on.

Other style questions are best answered by looking at the existing code. Formatting of docstrings, comments, long lines, and so on, should match the code that already exists.

Many people love black, but I would prefer not to run it on coverage.py.

6.12.5 Continuous integration

When you make a pull request, GitHub actions will run all of the tests and quality checks on your changes. If any fail, either fix them or ask for help.

6.12.6 Dependencies

Coverage.py has no direct runtime dependencies, and I would like to keep it that way.

It has many development dependencies. These are specified generically in the `requirements/*.in` files. The .in files should have no versions specified in them. The specific versions to use are pinned in `requirements/*.pip` files. These are created by running `make upgrade`.

It's important to use Python 3.8 to run `make upgrade` so that the pinned versions will work on all of the Python versions currently supported by coverage.py.

If for some reason we need to constrain a version of a dependency, the constraint should be specified in the `requirements/pins.pip` file, with a detailed reason for the pin.

6.12.7 Coverage testing coverage.py

Coverage.py can measure itself, but it's complicated. The process has been packaged up to make it easier:

```
$ make metacov metahtml
```

Then look at htmlcov/index.html. Note that due to the recursive nature of coverage.py measuring itself, there are some parts of the code that will never appear as covered, even though they are executed.

6.12.8 Contributing

When you are ready to contribute a change, any way you can get it to me is probably fine. A pull request on GitHub is great, but a simple diff or patch works too.

All contributions are expected to include tests for new functionality and fixes. If you need help writing tests, please ask.

6.13 Things that cause trouble

Coverage.py works well, and I want it to properly measure any Python program, but there are some situations it can't cope with. This page details some known problems, with possible courses of action, and links to coverage.py bug reports with more information.

I would love to *hear from you* if you have information about any of these problems, even just to explain to me why you want them to start working properly.

If your problem isn't discussed here, you can of course search the coverage.py bug tracker directly to see if there is some mention of it.

6.13.1 Things that don't work

There are a few modules or functions that prevent coverage.py from working properly:

- execv, or one of its variants. These end the current program and replace it with a new one. This doesn't save the collected coverage data, so your program that calls execv will not be fully measured. A patch for coverage.py is in issue 43.

- thread, in the Python standard library, is the low-level threading interface. Threads created with this module will not be traced. Use the higher-level threading module instead.

- sys.settrace is the Python feature that coverage.py uses to see what's happening in your program. If another part of your program is using sys.settrace, then it will conflict with coverage.py, and it won't be measured properly.

- sys.setprofile calls your code, but while running your code, does not fire trace events. This means that coverage.py can't see what's happening in that code.

6.13.2 Still having trouble?

If your problem isn't mentioned here, and isn't already reported in the coverage.py bug tracker, please *get in touch with me*, we'll figure out a solution.

6.14 FAQ and other help

6.14.1 Frequently asked questions

Q: Why are some of my files not measured?

Coverage.py has a number of mechanisms for deciding which files to measure and which to skip. If your files aren't being measured, use the `--debug=trace` *option*, also settable as `[run] debug=trace` in the *settings file*, or as `COVERAGE_DEBUG=trace` in an environment variable.

This will write a line for each file considered, indicating whether it is traced or not, and if not, why not. Be careful though: the output might be swallowed by your test runner. If so, a `COVERAGE_DEBUG_FILE=/tmp/cov.out` environment variable can direct the output to a file instead to ensure you see everything.

Q: Why do unexecutable lines show up as executed?

Usually this is because you've updated your code and run coverage.py on it again without erasing the old data. Coverage.py records line numbers executed, so the old data may have recorded a line number which has since moved, causing coverage.py to claim a line has been executed which cannot be.

If old data is persisting, you can use an explicit `coverage erase` command to clean out the old data.

Q: Why are my function definitions marked as run when I haven't tested them?

The `def` and `class` lines in your Python file are executed when the file is imported. Those are the lines that define your functions and classes. They run even if you never call the functions. It's the body of the functions that will be marked as not executed if you don't test them, not the `def` lines.

This can mean that your code has a moderate coverage total even if no tests have been written or run. This might seem surprising, but it is accurate: the `def` lines have actually been run.

Q: Why do the bodies of functions show as executed, but the def lines do not?

If this happens, it's because coverage.py has started after the functions are defined. The definition lines are executed without coverage measurement, then coverage.py is started, then the function is called. This means the body is measured, but the definition of the function itself is not.

The same thing can happen with the bodies of classes.

To fix this, start coverage.py earlier. If you use the *command line* to run your program with coverage.py, then your entire program will be monitored. If you are using the *API*, you need to call coverage.start() before importing the modules that define your functions.

Q: My decorator lines are marked as covered, but the "def" line is not. Why?

Different versions of Python report execution on different lines. Coverage.py adapts its behavior to the version of Python being used. In Python 3.7 and earlier, a decorated function definition only reported the decorator as executed. In Python 3.8 and later, both the decorator and the "def" are reported. If you collect execution data on Python 3.7, and then run coverage reports on Python 3.8, there will be a discrepancy.

Q: Can I find out which tests ran which lines?

Yes! Coverage.py has a feature called *Dynamic contexts* which can collect this information. Add this to your .coveragerc file:

```
[run]
dynamic_context = test_function
```

and then use the `--contexts` option when generating an HTML report.

Q: How is the total percentage calculated?

Coverage.py counts the total number of possible executions. This is the number of executable statements minus the number of excluded statements. It then counts the number of those possibilities that were actually executed. The total percentage is the actual executions divided by the possible executions.

As an example, a coverage report with 1514 statements and 901 missed statements would calculate a total percentage of (1514-901)/1514, or 40.49%.

Branch coverage extends the calculation to include the total number of possible branch exits, and the number of those taken. In this case the specific numbers shown in coverage reports don't calculate out to the percentage shown, because the number of missing branch exits isn't reported explicitly. A branch line that wasn't executed at all is counted once as a missing statement in the report, instead of as two missing branches. Reports show the number of partial branches, which is the lines that were executed but did not execute all of their exits.

Q: Coverage.py is much slower than I remember, what's going on?

Make sure you are using the C trace function. Coverage.py provides two implementations of the trace function. The C implementation runs much faster. To see what you are running, use `coverage debug sys`. The output contains details of the environment, including a line that says either `CTracer: available` or `CTracer: unavailable`. If it says unavailable, then you are using the slow Python implementation.

Try re-installing coverage.py to see what happened and if you get the CTracer as you should.

Q: Isn't coverage testing the best thing ever?

It's good, but it isn't perfect.

Q: Where can I get more help with coverage.py?

You can discuss coverage.py or get help using it on the Python discussion forums. If you ping me (@nedbat), there's a higher chance I'll see the post.

Bug reports are gladly accepted at the GitHub issue tracker.

I can be reached in a number of ways, I'm happy to answer questions about using coverage.py.

6.14.2 History

Coverage.py was originally written by Gareth Rees. Since 2004, Ned Batchelder has extended and maintained it with the help of many others. The *change history* has all the details.

6.15 Change history for coverage.py

These changes are listed in decreasing version number order. Note this can be different from a strict chronological order when there are two branches in development at the same time, such as 4.5.x and 5.0.

6.15.1 Version 7.3.1 — 2023-09-06

- The semantics of stars in file patterns has been clarified in the docs. A leading or trailing star matches any number of path components, like a double star would. This is different than the behavior of a star in the middle of a pattern. This discrepancy was identified by Sviatoslav Sydorenko, who provided patient detailed diagnosis and graciously agreed to a pragmatic resolution.

- The API docs were missing from the last version. They are now restored.

6.15.2 Version 7.3.0 — 2023-08-12

- Added a `Coverage.collect()` context manager to start and stop coverage data collection.

- Dropped support for Python 3.7.

- Fix: in unusual circumstances, SQLite cannot be set to asynchronous mode. Coverage.py would fail with the error `Safety level may not be changed inside a transaction.` This is now avoided, closing issue 1646. Thanks to Michael Bell for the detailed bug report.

- Docs: examples of configuration files now include separate examples for the different syntaxes: .coveragerc, pyproject.toml, setup.cfg, and tox.ini.

- Fix: added `nosemgrep` comments to our JavaScript code so that semgrep-based SAST security checks won't raise false alarms about security problems that aren't problems.

- Added a CITATION.cff file, thanks to Ken Schackart.

6.15.3 Version 7.2.7 — 2023-05-29

- Fix: reverted a change from 6.4.3 that helped Cython, but also increased the size of data files when using dynamic contexts, as described in the now-fixed issue 1586. The problem is now avoided due to a recent change (issue 1538). Thanks to Anders Kaseorg and David Szotten for persisting with problem reports and detailed diagnoses.

- Wheels are now provided for CPython 3.12.

6.15.4 Version 7.2.6 — 2023-05-23

- Fix: the `lcov` command could raise an IndexError exception if a file is translated to Python but then executed under its own name. Jinja2 does this when rendering templates. Fixes issue 1553.

- Python 3.12 beta 1 now inlines comprehensions. Previously they were compiled as invisible functions and coverage.py would warn you if they weren't completely executed. This no longer happens under Python 3.12.

- Fix: the `coverage debug sys` command includes some environment variables in its output. This could have included sensitive data. Those values are now hidden with asterisks, closing issue 1628.

6.15.5 Version 7.2.5 — 2023-04-30

- Fix: `html_report()` could fail with an AttributeError on `isatty` if run in an unusual environment where sys.stdout had been replaced. This is now fixed.

6.15.6 Version 7.2.4 — 2023-04-28

PyCon 2023 sprint fixes!

- Fix: with `relative_files = true`, specifying a specific file to include or omit wouldn't work correctly (issue 1604). This is now fixed, with testing help by Marc Gibbons.

- Fix: the XML report would have an incorrect `<source>` element when using relative files and the source option ended with a slash (issue 1541). This is now fixed, thanks to Kevin Brown-Silva.

- When the HTML report location is printed to the terminal, it's now a terminal-compatible URL, so that you can click the location to open the HTML file in your browser. Finishes issue 1523 thanks to Ricardo Newbery.

- Docs: a new *Migrating page* with details about how to migrate between major versions of coverage.py. It currently covers the wildcard changes in 7.x. Thanks, Brian Grohe.

6.15.7 Version 7.2.3 — 2023-04-06

- Fix: the *[run] sigterm* setting was meant to capture data if a process was terminated with a SIGTERM signal, but it didn't always. This was fixed thanks to Lewis Gaul, closing issue 1599.

- Performance: HTML reports with context information are now much more compact. File sizes are typically as small as one-third the previous size, but can be dramatically smaller. This closes issue 1584 thanks to Oleh Krehel.

- Development dependencies no longer use hashed pins, closing issue 1592.

6.15.8 Version 7.2.2 — 2023-03-16

- Fix: if a virtualenv was created inside a source directory, and a sourced package was installed inside the virtualenv, then all of the third-party packages inside the virtualenv would be measured. This was incorrect, but has now been fixed: only the specified packages will be measured, thanks to Manuel Jacob.

- Fix: the `coverage lcov` command could create a .lcov file with incorrect LF (lines found) and LH (lines hit) totals. This is now fixed, thanks to Ian Moore.

- Fix: the `coverage xml` command on Windows could create a .xml file with duplicate `<package>` elements. This is now fixed, thanks to Benjamin Parzella, closing issue 1573.

6.15.9 Version 7.2.1 — 2023-02-26

- Fix: the PyPI page had broken links to documentation pages, but no longer does, closing issue 1566.

- Fix: public members of the coverage module are now properly indicated so that mypy will find them, fixing issue 1564.

6.15.10 Version 7.2.0 — 2023-02-22

- Added a new setting `[report] exclude_also` to let you add more exclusions without overwriting the defaults. Thanks, Alpha Chen, closing issue 1391.

- Added a `CoverageData.purge_files()` method to remove recorded data for a particular file. Contributed by Stephan Deibel.

- Fix: when reporting commands fail, they will no longer congratulate themselves with messages like "Wrote XML report to file.xml" before spewing a traceback about their failure.

- Fix: arguments in the public API that name file paths now accept pathlib.Path objects. This includes the `data_file` and `config_file` arguments to the Coverage constructor and the `basename` argument to CoverageData. Closes issue 1552.

- Fix: In some embedded environments, an IndexError could occur on stop() when the originating thread exits before completion. This is now fixed, thanks to Russell Keith-Magee, closing issue 1542.

- Added a `py.typed` file to announce our type-hintedness. Thanks, KotlinIsland.

6.15.11 Version 7.1.0 — 2023-01-24

- Added: the debug output file can now be specified with `[run] debug_file` in the configuration file. Closes issue 1319.

- Performance: fixed a slowdown with dynamic contexts that's been around since 6.4.3. The fix closes issue 1538. Thankfully this doesn't break the Cython change that fixed issue 972. Thanks to Mathieu Kniewallner for the deep investigative work and comprehensive issue report.

- Typing: all product and test code has type annotations.

6.15.12 Version 7.0.5 — 2023-01-10

- Fix: On Python 3.7, a file with type annotations but no `from __future__ import annotations` would be missing statements in the coverage report. This is now fixed, closing issue 1524.

6.15.13 Version 7.0.4 — 2023-01-07

- Performance: an internal cache of file names was accidentally disabled, resulting in sometimes drastic reductions in performance. This is now fixed, closing issue 1527. Thanks to Ivan Ciuvalschii for the reproducible test case.

6.15.14 Version 7.0.3 — 2023-01-03

- Fix: when using pytest-cov or pytest-xdist, or perhaps both, the combining step could fail with `assert row is not None` using 7.0.2. This was due to a race condition that has always been possible and is still possible. In 7.0.1 and before, the error was silently swallowed by the combining code. Now it will produce a message "Couldn't combine data file" and ignore the data file as it used to do before 7.0.2. Closes issue 1522.

6.15.15 Version 7.0.2 — 2023-01-02

- Fix: when using the `[run] relative_files = True` setting, a relative `[paths]` pattern was still being made absolute. This is now fixed, closing issue 1519.

- Fix: if Python doesn't provide tomllib, then TOML configuration files can only be read if coverage.py is installed with the `[toml]` extra. Coverage.py will raise an error if TOML support is not installed when it sees your settings are in a .toml file. But it didn't understand that `[tools.coverage]` was a valid section header, so the error wasn't reported if you used that header, and settings were silently ignored. This is now fixed, closing issue 1516.

- Fix: adjusted how decorators are traced on PyPy 7.3.10, fixing issue 1515.

- Fix: the `coverage lcov` report did not properly implement the `--fail-under=MIN` option. This has been fixed.

- Refactor: added many type annotations, including a number of refactorings. This should not affect outward behavior, but they were a bit invasive in some places, so keep your eyes peeled for oddities.

- Refactor: removed the vestigial and long untested support for Jython and IronPython.

6.15.16 Version 7.0.1 — 2022-12-23

- When checking if a file mapping resolved to a file that exists, we weren't considering files in .whl files. This is now fixed, closing issue 1511.

- File pattern rules were too strict, forbidding plus signs and curly braces in directory and file names. This is now fixed, closing issue 1513.

- Unusual Unicode or control characters in source files could prevent reporting. This is now fixed, closing issue 1512.

- The PyPy wheel now installs on PyPy 3.7, 3.8, and 3.9, closing issue 1510.

6.15.17 Version 7.0.0 — 2022-12-18

Nothing new beyond 7.0.0b1.

6.15.18 Version 7.0.0b1 — 2022-12-03

A number of changes have been made to file path handling, including pattern matching and path remapping with the `[paths]` setting (see *[paths]*). These changes might affect you, and require you to update your settings.

(This release includes the changes from *6.6.0b1*, since 6.6.0 was never released.)

- Changes to file pattern matching, which might require updating your configuration:

  - Previously, * would incorrectly match directory separators, making precise matching difficult. This is now fixed, closing issue 1407.

  - Now ** matches any number of nested directories, including none.

- Improvements to combining data files when using the *[run] relative_files* setting, which might require updating your configuration:

  - During `coverage combine`, relative file paths are implicitly combined without needing a `[paths]` configuration setting. This also fixed issue 991.

- A [paths] setting like */foo will now match foo/bar.py so that relative file paths can be combined more easily.

- The *[run] relative_files* setting is properly interpreted in more places, fixing issue 1280.

- When remapping file paths with [paths], a path will be remapped only if the resulting path exists. The documentation has long said the prefix had to exist, but it was never enforced. This fixes issue 608, improves issue 649, and closes issue 757.

- Reporting operations now implicitly use the [paths] setting to remap file paths within a single data file. Combining multiple files still requires the coverage combine step, but this simplifies some single-file situations. Closes issue 1212 and issue 713.

- The coverage report command now has a --format= option. The original style is now --format=text, and is the default.

  - Using --format=markdown will write the table in Markdown format, thanks to Steve Oswald, closing issue 1418.

  - Using --format=total will write a single total number to the output. This can be useful for making badges or writing status updates.

- Combining data files with coverage combine now hashes the data files to skip files that add no new information. This can reduce the time needed. Many details affect the speed-up, but for coverage.py's own test suite, combining is about 40% faster. Closes issue 1483.

- When searching for completely un-executed files, coverage.py uses the presence of __init__.py files to determine which directories have source that could have been imported. However, implicit namespace packages don't require __init__.py. A new setting [report] include_namespace_packages tells coverage.py to consider these directories during reporting. Thanks to Felix Horvat for the contribution. Closes issue 1383 and issue 1024.

- Fixed environment variable expansion in pyproject.toml files. It was overly broad, causing errors outside of coverage.py settings, as described in issue 1481 and issue 1345. This is now fixed, but in rare cases will require changing your pyproject.toml to quote non-string values that use environment substitution.

- An empty file has a coverage total of 100%, but used to fail with --fail-under. This has been fixed, closing issue 1470.

- The text report table no longer writes out two separator lines if there are no files listed in the table. One is plenty.

- Fixed a mis-measurement of a strange use of wildcard alternatives in match/case statements, closing issue 1421.

- Fixed internal logic that prevented coverage.py from running on implementations other than CPython or PyPy (issue 1474).

- The deprecated [run] note setting has been completely removed.

6.15.19 Version 6.6.0b1 — 2022-10-31

(Note: 6.6.0 final was never released. These changes are part of *7.0.0b1*.)

- Changes to file pattern matching, which might require updating your configuration:

  - Previously, * would incorrectly match directory separators, making precise matching difficult. This is now fixed, closing issue 1407.

  - Now ** matches any number of nested directories, including none.

- Improvements to combining data files when using the *[run] relative_files* setting:

  - During coverage combine, relative file paths are implicitly combined without needing a [paths] configuration setting. This also fixed issue 991.

- A [paths] setting like */foo will now match foo/bar.py so that relative file paths can be combined more easily.

- The setting is properly interpreted in more places, fixing issue 1280.

- Fixed environment variable expansion in pyproject.toml files. It was overly broad, causing errors outside of coverage.py settings, as described in issue 1481 and issue 1345. This is now fixed, but in rare cases will require changing your pyproject.toml to quote non-string values that use environment substitution.

- Fixed internal logic that prevented coverage.py from running on implementations other than CPython or PyPy (issue 1474).

6.15.20 Version 6.5.0 — 2022-09-29

- The JSON report now includes details of which branches were taken, and which are missing for each file. Thanks, Christoph Blessing. Closes issue 1425.

- Starting with coverage.py 6.2, class statements were marked as a branch. This wasn't right, and has been reverted, fixing issue 1449. Note this will very slightly reduce your coverage total if you are measuring branch coverage.

- Packaging is now compliant with PEP 517, closing issue 1395.

- A new debug option --debug=pathmap shows details of the remapping of paths that happens during combine due to the [paths] setting.

- Fix an internal problem with caching of invalid Python parsing. Found by OSS-Fuzz, fixing their bug 50381.

6.15.21 Version 6.4.4 — 2022-08-16

- Wheels are now provided for Python 3.11.

6.15.22 Version 6.4.3 — 2022-08-06

- Fix a failure when combining data files if the file names contained glob-like patterns. Thanks, Michael Krebs and Benjamin Schubert.

- Fix a messaging failure when combining Windows data files on a different drive than the current directory, closing issue 1428. Thanks, Lorenzo Micò.

- Fix path calculations when running in the root directory, as you might do in a Docker container. Thanks Arthur Rio.

- Filtering in the HTML report wouldn't work when reloading the index page. This is now fixed. Thanks, Marc Legendre.

- Fix a problem with Cython code measurement, closing issue 972. Thanks, Matus Valo.

6.15.23 Version 6.4.2 — 2022-07-12

- Updated for a small change in Python 3.11.0 beta 4: modules now start with a line with line number 0, which is ignored. This line cannot be executed, so coverage totals were thrown off. This line is now ignored by coverage.py, but this also means that truly empty modules (like `__init__.py`) have no lines in them, rather than one phantom line. Fixes issue 1419.

- Internal debugging data added to sys.modules is now an actual module, to avoid confusing code that examines everything in sys.modules. Thanks, Yilei Yang.

6.15.24 Version 6.4.1 — 2022-06-02

- Greatly improved performance on PyPy, and other environments that need the pure Python trace function. Thanks, Carl Friedrich Bolz-Tereick (pull 1381 and pull 1388). Slightly improved performance when using the C trace function, as most environments do. Closes issue 1339.

- The conditions for using tomllib from the standard library have been made more precise, so that 3.11 alphas will continue to work. Closes issue 1390.

6.15.25 Version 6.4 — 2022-05-22

- A new setting, *[run] sigterm*, controls whether a SIGTERM signal handler is used. In 6.3, the signal handler was always installed, to capture data at unusual process ends. Unfortunately, this introduced other problems (see issue 1310). Now the signal handler is only used if you opt-in by setting `[run] sigterm = true`.

- Small changes to the HTML report:

    - Added links to next and previous file, and more keyboard shortcuts: `[` and `]` for next file and previous file; `u` for up to the index; and `?` to open/close the help panel. Thanks, J. M. F. Tsang.

    - The time stamp and version are displayed at the top of the report. Thanks, Ammar Askar. Closes issue 1351.

- A new debug option `debug=sqldata` adds more detail to `debug=sql`, logging all the data being written to the database.

- Previously, running `coverage report` (or any of the reporting commands) in an empty directory would create a .coverage data file. Now they do not, fixing issue 1328.

- On Python 3.11, the `[toml]` extra no longer installs tomli, instead using tomllib from the standard library. Thanks Shantanu.

- In-memory CoverageData objects now properly update(), closing issue 1323.

6.15.26 Version 6.3.3 — 2022-05-12

- Fix: Coverage.py now builds successfully on CPython 3.11 (3.11.0b1) again. Closes issue 1367. Some results for generators may have changed.

6.15.27 Version 6.3.2 — 2022-02-20

- Fix: adapt to pypy3.9's decorator tracing behavior. It now traces function decorators like CPython 3.8: both the @-line and the def-line are traced. Fixes issue 1326.

- Debug: added `pybehave` to the list of *coverage debug* and *--debug* options.

- Fix: show an intelligible error message if `--concurrency=multiprocessing` is used without a configuration file. Closes issue 1320.

6.15.28 Version 6.3.1 — 2022-02-01

- Fix: deadlocks could occur when terminating processes. Some of these deadlocks (described in issue 1310) are now fixed.

- Fix: a signal handler was being set from multiple threads, causing an error: "ValueError: signal only works in main thread". This is now fixed, closing issue 1312.

- Fix: `--precision` on the command-line was being ignored while considering `--fail-under`. This is now fixed, thanks to Marcelo Trylesinski.

- Fix: releases no longer provide 3.11.0-alpha wheels. Coverage.py uses CPython internal fields which are moving during the alpha phase. Fixes issue 1316.

6.15.29 Version 6.3 — 2022-01-25

- Feature: Added the `lcov` command to generate reports in LCOV format. Thanks, Bradley Burns. Closes issues 587 and 626.

- Feature: the coverage data file can now be specified on the command line with the `--data-file` option in any command that reads or writes data. This is in addition to the existing `COVERAGE_FILE` environment variable. Closes issue 624. Thanks, Nikita Bloshchanevich.

- Feature: coverage measurement data will now be written when a SIGTERM signal is received by the process. This includes `Process.terminate`, and other ways to terminate a process. Currently this is only on Linux and Mac; Windows is not supported. Fixes issue 1307.

- Dropped support for Python 3.6, which reached end-of-life on 2021-12-23.

- Updated Python 3.11 support to 3.11.0a4, fixing issue 1294.

- Fix: the coverage data file is now created in a more robust way, to avoid problems when multiple processes are trying to write data at once. Fixes issues 1303 and 883.

- Fix: a .gitignore file will only be written into the HTML report output directory if the directory is empty. This should prevent certain unfortunate accidents of writing the file where it is not wanted.

- Releases now have MacOS arm64 wheels for Apple Silicon, fixing issue 1288.

6.15.30 Version 6.2 — 2021-11-26

- Feature: Now the `--concurrency` setting can now have a list of values, so that threads and another lightweight threading package can be measured together, such as `--concurrency=gevent,thread`. Closes issue 1012 and issue 1082.

- Fix: A module specified as the `source` setting is imported during startup, before the user program imports it. This could cause problems if the rest of the program isn't ready yet. For example, issue 1203 describes a Django setting that is accessed before settings have been configured. Now the early import is wrapped in a try/except so errors then don't stop execution.

- Fix: A colon in a decorator expression would cause an exclusion to end too early, preventing the exclusion of the decorated function. This is now fixed.

- Fix: The HTML report now will not overwrite a .gitignore file that already exists in the HTML output directory (follow-on for issue 1244).

- API: The exceptions raised by Coverage.py have been specialized, to provide finer-grained catching of exceptions by third-party code.

- API: Using `suffix=False` when constructing a Coverage object with multiprocessing wouldn't suppress the data file suffix (issue 989). This is now fixed.

- Debug: The `coverage debug data` command will now sniff out combinable data files, and report on all of them.

- Debug: The `coverage debug` command used to accept a number of topics at a time, and show all of them, though this was never documented. This no longer works, to allow for command-line options in the future.

6.15.31 Version 6.1.2 — 2021-11-10

- Python 3.11 is supported (tested with 3.11.0a2). One still-open issue has to do with exits through with-statements.

- Fix: When remapping file paths through the `[paths]` setting while combining, the `[run] relative_files` setting was ignored, resulting in absolute paths for remapped file names (issue 1147). This is now fixed.

- Fix: Complex conditionals over excluded lines could have incorrectly reported a missing branch (issue 1271). This is now fixed.

- Fix: More exceptions are now handled when trying to parse source files for reporting. Problems that used to terminate coverage.py can now be handled with `[report] ignore_errors`. This helps with plugins failing to read files (django_coverage_plugin issue 78).

- Fix: Removed another vestige of jQuery from the source tarball (issue 840).

- Fix: Added a default value for a new-to-6.x argument of an internal class. This unsupported class is being used by coveralls (issue 1273). Although I'd rather not "fix" unsupported interfaces, it's actually nicer with a default value.

6.15.32 Version 6.1.1 — 2021-10-31

- Fix: The sticky header on the HTML report didn't work unless you had branch coverage enabled. This is now fixed: the sticky header works for everyone. (Do people still use coverage without branch measurement!? j/k)

- Fix: When using explicitly declared namespace packages, the "already imported a file that will be measured" warning would be issued (issue 888). This is now fixed.

6.15.33 Version 6.1 — 2021-10-30

- Deprecated: The `annotate` command and the `Coverage.annotate` function will be removed in a future version, unless people let me know that they are using it. Instead, the `html` command gives better-looking (and more accurate) output, and the `report -m` command will tell you line numbers of missing lines. Please get in touch if you have a reason to use `annotate` over those better options: ned@nedbatchelder.com.

- Feature: Coverage now sets an environment variable, `COVERAGE_RUN` when running your code with the `coverage run` command. The value is not important, and may change in the future. Closes issue 553.

- Feature: The HTML report pages for Python source files now have a sticky header so the file name and controls are always visible.

- Feature: The `xml` and `json` commands now describe what they wrote where.

- Feature: The `html`, `combine`, `xml`, and `json` commands all accept a `-q/--quiet` option to suppress the messages they write to stdout about what they are doing (issue 1254).

- Feature: The `html` command writes a `.gitignore` file into the HTML output directory, to prevent the report from being committed to git. If you want to commit it, you will need to delete that file. Closes issue 1244.

- Feature: Added support for PyPy 3.8.

- Fix: More generated code is now excluded from measurement. Code such as attrs boilerplate, or doctest code, was being measured though the synthetic line numbers meant they were never reported. Once Cython was involved though, the generated .so files were parsed as Python, raising syntax errors, as reported in issue 1160. This is now fixed.

- Fix: When sorting human-readable names, numeric components are sorted correctly: file10.py will appear after file9.py. This applies to file names, module names, environment variables, and test contexts.

- Performance: Branch coverage measurement is faster, though you might only notice on code that is executed many times, such as long-running loops.

- Build: jQuery is no longer used or vendored (issue 840 and issue 1118). Huge thanks to Nils Kattenbeck (septatrix) for the conversion to vanilla JavaScript in pull request 1248.

6.15.34 Version 6.0.2 — 2021-10-11

- Namespace packages being measured weren't properly handled by the new code that ignores third-party packages. If the namespace package was installed, it was ignored as a third-party package. That problem (issue 1231) is now fixed.

- Packages named as "source packages" (with `source`, or `source_pkgs`, or pytest-cov's `--cov`) might have been only partially measured. Their top-level statements could be marked as un-executed, because they were imported by coverage.py before measurement began (issue 1232). This is now fixed, but the package will be imported twice, once by coverage.py, then again by your test suite. This could cause problems if importing the package has side effects.

- The `CoverageData.contexts_by_lineno()` method was documented to return a dict, but was returning a defaultdict. Now it returns a plain dict. It also no longer returns negative numbered keys.

6.15.35 Version 6.0.1 — 2021-10-06

- In 6.0, the coverage.py exceptions moved from coverage.misc to coverage.exceptions. These exceptions are not part of the public supported API, CoverageException is. But a number of other third-party packages were importing the exceptions from coverage.misc, so they are now available from there again (issue 1226).

- Changed an internal detail of how tomli is imported, so that tomli can use coverage.py for their own test suite (issue 1228).

- Defend against an obscure possibility under code obfuscation, where a function can have an argument called "self", but no local named "self" (pull request 1210). Thanks, Ben Carlsson.

6.15.36 Version 6.0 — 2021-10-03

- The `coverage html` command now prints a message indicating where the HTML report was written. Fixes issue 1195.

- The `coverage combine` command now prints messages indicating each data file being combined. Fixes issue 1105.

- The HTML report now includes a sentence about skipped files due to `skip_covered` or `skip_empty` settings. Fixes issue 1163.

- Unrecognized options in the configuration file are no longer errors. They are now warnings, to ease the use of coverage across versions. Fixes issue 1035.

- Fix handling of exceptions through context managers in Python 3.10. A missing exception is no longer considered a missing branch from the with statement. Fixes issue 1205.

- Fix another rarer instance of "Error binding parameter 0 - probably unsupported type." (issue 1010).

- Creating a directory for the coverage data file now is safer against conflicts when two coverage runs happen simultaneously (pull 1220). Thanks, Clément Pit-Claudel.

6.15.37 Version 6.0b1 — 2021-07-18

- Dropped support for Python 2.7, PyPy 2, and Python 3.5.

- Added support for the Python 3.10 `match/case` syntax.

- Data collection is now thread-safe. There may have been rare instances of exceptions raised in multi-threaded programs.

- Plugins (like the Django coverage plugin) were generating "Already imported a file that will be measured" warnings about Django itself. These have been fixed, closing issue 1150.

- Warnings generated by coverage.py are now real Python warnings.

- Using `--fail-under=100` with coverage near 100% could result in the self-contradictory message `total of 100 is less than fail-under=100`. This bug (issue 1168) is now fixed.

- The `COVERAGE_DEBUG_FILE` environment variable now accepts `stdout` and `stderr` to write to those destinations.

- TOML parsing now uses the tomli library.

- Some minor changes to usually invisible details of the HTML report:

  - Use a modern hash algorithm when fingerprinting, for high-security environments (issue 1189). When generating the HTML report, we save the hash of the data, to avoid regenerating an unchanged HTML

page. We used to use MD5 to generate the hash, and now use SHA-3-256. This was never a security concern, but security scanners would notice the MD5 algorithm and raise a false alarm.

- Change how report file names are generated, to avoid leading underscores (issue 1167), to avoid rare file name collisions (issue 584), and to avoid file names becoming too long (issue 580).

6.15.38 Version 5.6b1 — 2021-04-13

Note: 5.6 final was never released. These changes are part of 6.0.

- Third-party packages are now ignored in coverage reporting. This solves a few problems:
  - Coverage will no longer report about other people's code (issue 876). This is true even when using `--source=.` with a venv in the current directory.
  - Coverage will no longer generate "Already imported a file that will be measured" warnings about coverage itself (issue 905).

- The HTML report uses j/k to move up and down among the highlighted chunks of code. They used to highlight the current chunk, but 5.0 broke that behavior. Now the highlighting is working again.

- The JSON report now includes `percent_covered_display`, a string with the total percentage, rounded to the same number of decimal places as the other reports' totals.

6.15.39 Version 5.5 — 2021-02-28

- `coverage combine` has a new option, `--keep` to keep the original data files after combining them. The default is still to delete the files after they have been combined. This was requested in issue 1108 and implemented in pull request 1110. Thanks, Éric Larivière.

- When reporting missing branches in `coverage report`, branches aren't reported that jump to missing lines. This adds to the long-standing behavior of not reporting branches from missing lines. Now branches are only reported if both the source and destination lines are executed. Closes both issue 1065 and issue 955.

- Minor improvements to the HTML report:
  - The state of the line visibility selector buttons is saved in local storage so you don't have to fiddle with them so often, fixing issue 1123.
  - It has a little more room for line numbers so that 4-digit numbers work well, fixing issue 1124.

- Improved the error message when combining line and branch data, so that users will be more likely to understand what's happening, closing issue 803.

6.15.40 Version 5.4 — 2021-01-24

- The text report produced by `coverage report` now always outputs a TOTAL line, even if only one Python file is reported. This makes regex parsing of the output easier. Thanks, Judson Neer. This had been requested a number of times (issue 1086, issue 922, issue 732).

- The `skip_covered` and `skip_empty` settings in the configuration file can now be specified in the `[html]` section, so that text reports and HTML reports can use separate settings. The HTML report will still use the `[report]` settings if there isn't a value in the `[html]` section. Closes issue 1090.

- Combining files on Windows across drives now works properly, fixing issue 577. Thanks, Valentin Lab.

- Fix an obscure warning from deep in the _decimal module, as reported in issue 1084.

- Update to support Python 3.10 alphas in progress, including PEP 626: Precise line numbers for debugging and other tools.

6.15.41 Version 5.3.1 — 2020-12-19

- When using `--source` on a large source tree, v5.x was slower than previous versions. This performance regression is now fixed, closing issue 1037.

- Mysterious SQLite errors can happen on PyPy, as reported in issue 1010. An immediate retry seems to fix the problem, although it is an unsatisfying solution.

- The HTML report now saves the sort order in a more widely supported way, fixing issue 986. Thanks, Sebastián Ramírez (pull request 1066).

- The HTML report pages now have a *Sleepy Snake* favicon.

- Wheels are now provided for manylinux2010, and for PyPy3 (pp36 and pp37).

- Continuous integration has moved from Travis and AppVeyor to GitHub Actions.

6.15.42 Version 5.3 — 2020-09-13

- The `source` setting has always been interpreted as either a file path or a module, depending on which existed. If both interpretations were valid, it was assumed to be a file path. The new `source_pkgs` setting can be used to name a package to disambiguate this case. Thanks, Thomas Grainger. Fixes issue 268.

- If a plugin was disabled due to an exception, we used to still try to record its information, causing an exception, as reported in issue 1011. This is now fixed.

6.15.43 Version 5.2.1 — 2020-07-23

- The dark mode HTML report still used light colors for the context listing, making them unreadable (issue 1009). This is now fixed.

- The time stamp on the HTML report now includes the time zone. Thanks, Xie Yanbo (pull request 960).

6.15.44 Version 5.2 — 2020-07-05

- The HTML report has been redesigned by Vince Salvino. There is now a dark mode, the code text is larger, and system sans serif fonts are used, in addition to other small changes (issue 858 and pull request 931).

- The `coverage report` and `coverage html` commands now accept a `--precision` option to control the number of decimal points displayed. Thanks, Teake Nutma (pull request 982).

- The `coverage report` and `coverage html` commands now accept a `--no-skip-covered` option to negate `--skip-covered`. Thanks, Anthony Sottile (issue 779 and pull request 932).

- The `--skip-empty` option is now available for the XML report, closing issue 976.

- The `coverage report` command now accepts a `--sort` option to specify how to sort the results. Thanks, Jerin Peter George (pull request 1005).

- If coverage fails due to the coverage total not reaching the `--fail-under` value, it will now print a message making the condition clear. Thanks, Naveen Yadav (pull request 977).

- TOML configuration files with non-ASCII characters would cause errors on Windows (issue 990). This is now fixed.

- The output of `--debug=trace` now includes information about how the `--source` option is being interpreted, and the module names being considered.

6.15.45 Version 5.1 — 2020-04-12

- The JSON report now includes counts of covered and missing branches. Thanks, Salvatore Zagaria.

- On Python 3.8, try-finally-return reported wrong branch coverage with decorated async functions (issue 964). This is now fixed. Thanks, Kjell Braden.

- The `get_option()` and `set_option()` methods can now manipulate the `[paths]` configuration setting. Thanks to Bernát Gábor for the fix for issue 967.

6.15.46 Version 5.0.4 — 2020-03-16

- If using the `[run] relative_files` setting, the XML report will use relative files in the `<source>` elements indicating the location of source code. Closes issue 948.

- The textual summary report could report missing lines with negative line numbers on PyPy3 7.1 (issue 943). This is now fixed.

- Windows wheels for Python 3.8 were incorrectly built, but are now fixed. (issue 949)

- Updated Python 3.9 support to 3.9a4.

- HTML reports couldn't be sorted if localStorage wasn't available. This is now fixed: sorting works even though the sorting setting isn't retained. (issue 944 and pull request 945). Thanks, Abdeali Kothari.

6.15.47 Version 5.0.3 — 2020-01-12

- A performance improvement in 5.0.2 didn't work for test suites that changed directory before combining data, causing "Couldn't use data file: no such table: meta" errors (issue 916). This is now fixed.

- Coverage could fail to run your program with some form of "ModuleNotFound" or "ImportError" trying to import from the current directory. This would happen if coverage had been packaged into a zip file (for example, on Windows), or was found indirectly (for example, by pyenv-virtualenv). A number of different scenarios were described in issue 862 which is now fixed. Huge thanks to Agbonze O. Jeremiah for reporting it, and Alexander Waters and George-Cristian Bîrzan for protracted debugging sessions.

- Added the "premain" debug option.

- Added SQLite compile-time options to the "debug sys" output.

6.15.48 Version 5.0.2 — 2020-01-05

- Programs that used multiprocessing and changed directories would fail under coverage. This is now fixed (issue 890). A side effect is that debug information about the config files read now shows absolute paths to the files.

- When running programs as modules (`coverage run -m`) with `--source`, some measured modules were imported before coverage starts. This resulted in unwanted warnings ("Already imported a file that will be measured") and a reduction in coverage totals (issue 909). This is now fixed.

- If no data was collected, an exception about "No data to report" could happen instead of a 0% report being created (issue 884). This is now fixed.

- The handling of source files with non-encodable file names has changed. Previously, if a file name could not be encoded as UTF-8, an error occurred, as described in issue 891. Now, those files will not be measured, since their data would not be recordable.

- A new warning ("dynamic-conflict") is issued if two mechanisms are trying to change the dynamic context. Closes issue 901.

- `coverage run --debug=sys` would fail with an AttributeError. This is now fixed (issue 907).

6.15.49 Version 5.0.1 — 2019-12-22

- If a 4.x data file is the cause of a "file is not a database" error, then use a more specific error message, "Looks like a coverage 4.x data file, are you mixing versions of coverage?" Helps diagnose the problems described in issue 886.

- Measurement contexts and relative file names didn't work together, as reported in issue 899 and issue 900. This is now fixed, thanks to David Szotten.

- When using `coverage run --concurrency=multiprocessing`, all data files should be named with parallel-ready suffixes. 5.0 mistakenly named the main process' file with no suffix when using `--append`. This is now fixed, closing issue 880.

- Fixed a problem on Windows when the current directory is changed to a different drive (issue 895). Thanks, Olivier Grisel.

- Updated Python 3.9 support to 3.9a2.

6.15.50 Version 5.0 — 2019-12-14

Nothing new beyond 5.0b2.

A summary of major changes in 5.0 since 4.5.x is in see whatsnew5x.

6.15.51 Version 5.0b2 — 2019-12-08

- An experimental `[run] relative_files` setting tells coverage to store relative file names in the data file. This makes it easier to run tests in one (or many) environments, and then report in another. It has not had much real-world testing, so it may change in incompatible ways in the future.

- When constructing a `coverage.Coverage` object, *data_file* can be specified as None to prevent writing any data file at all. In previous versions, an explicit *data_file=None* argument would use the default of ".coverage". Fixes issue 871.

- Python files run with `-m` now have `__spec__` defined properly. This fixes issue 745 (about not being able to run unittest tests that spawn subprocesses), and issue 838, which described the problem directly.

- The `[paths]` configuration section is now ordered. If you specify more than one list of patterns, the first one that matches will be used. Fixes issue 649.

- The `coverage.numbits.register_sqlite_functions()` function now also registers *numbits_to_nums* for use in SQLite queries. Thanks, Simon Willison.

- Python 3.9a1 is supported.

- Coverage.py has a mascot: *Sleepy Snake*.

6.15.52 Version 5.0b1 — 2019-11-11

- The HTML and textual reports now have a `--skip-empty` option that skips files with no statements, notably `__init__.py` files. Thanks, Reya B.

- Configuration can now be read from TOML files. This requires installing coverage.py with the `[toml]` extra. The standard "pyproject.toml" file will be read automatically if no other configuration file is found, with settings in the `[tool.coverage.]` namespace. Thanks to Frazer McLean for implementation and persistence. Finishes issue 664.

- The `[run] note` setting has been deprecated. Using it will result in a warning, and the note will not be written to the data file. The corresponding *CoverageData* methods have been removed.

- The HTML report has been reimplemented (no more table around the source code). This allowed for a better presentation of the context information, hopefully resolving issue 855.

- Added sqlite3 module version information to `coverage debug sys` output.

- Asking the HTML report to show contexts (`[html]` `show_contexts=True` or `coverage html --show-contexts`) will issue a warning if there were no contexts measured (issue 851).

6.15.53 Version 5.0a8 — 2019-10-02

- The *CoverageData* API has changed how queries are limited to specific contexts. Now you use *CoverageData.set_query_context()* to set a single exact-match string, or *CoverageData.set_query_contexts()* to set a list of regular expressions to match contexts. This changes the command-line `--contexts` option to use regular expressions instead of filename-style wildcards.

6.15.54 Version 5.0a7 — 2019-09-21

- Data can now be "reported" in JSON format, for programmatic use, as requested in issue 720. The new `coverage json` command writes raw and summarized data to a JSON file. Thanks, Matt Bachmann.

- Dynamic contexts are now supported in the Python tracer, which is important for PyPy users. Closes issue 846.

- The compact line number representation introduced in 5.0a6 is called a "numbits." The *coverage.numbits* module provides functions for working with them.

- The reporting methods used to permanently apply their arguments to the configuration of the Coverage object. Now they no longer do. The arguments affect the operation of the method, but do not persist.

- A class named "test_something" no longer confuses the `test_function` dynamic context setting. Fixes issue 829.

- Fixed an unusual tokenizing issue with backslashes in comments. Fixes issue 822.

- `debug=plugin` didn't properly support configuration or dynamic context plugins, but now it does, closing issue 834.

6.15.55 Version 5.0a6 — 2019-07-16

- Reporting on contexts. Big thanks to Stephan Richter and Albertas Agejevas for the contribution.

    - The `--contexts` option is available on the `report` and `html` commands. It's a comma-separated list of shell-style wildcards, selecting the contexts to report on. Only contexts matching one of the wildcards will be included in the report.

    - The `--show-contexts` option for the `html` command adds context information to each covered line. Hovering over the "ctx" marker at the end of the line reveals a list of the contexts that covered the line.

- Database changes:

    - Line numbers are now stored in a much more compact way. For each file and context, a single binary string is stored with a bit per line number. This greatly improves memory use, but makes ad-hoc use difficult.

    - Dynamic contexts with no data are no longer written to the database.

    - SQLite data storage is now faster. There's no longer a reason to keep the JSON data file code, so it has been removed.

- Changes to the *CoverageData* interface:

    - The new *CoverageData.dumps()* method serializes the data to a string, and a corresponding *CoverageData.loads()* method reconstitutes this data. The format of the data string is subject to change at any time, and so should only be used between two installations of the same version of coverage.py.

    - The *CoverageData constructor* has a new argument, *no_disk* (default: False). Setting it to True prevents writing any data to the disk. This is useful for transient data objects.

- Added the class method *Coverage.current()* to get the latest started Coverage instance.

- Multiprocessing support in Python 3.8 was broken, but is now fixed. Closes issue 828.

- Error handling during reporting has changed slightly. All reporting methods now behave the same. The `--ignore-errors` option keeps errors from stopping the reporting, but files that couldn't parse as Python will always be reported as warnings. As with other warnings, you can suppress them with the `[run]` `disable_warnings` configuration setting.

- Coverage.py no longer fails if the user program deletes its current directory. Fixes issue 806. Thanks, Dan Hemberger.

- The scrollbar markers in the HTML report now accurately show the highlighted lines, regardless of what categories of line are highlighted.

- The hack to accommodate ShiningPanda looking for an obsolete internal data file has been removed, since ShiningPanda 0.22 fixed it four years ago.

- The deprecated *Reporter.file_reporters* property has been removed.

6.15.56 Version 5.0a5 — 2019-05-07

- Drop support for Python 3.4

- Dynamic contexts can now be set two new ways, both thanks to Justas Sadzevičius.

    - A plugin can implement a `dynamic_context` method to check frames for whether a new context should be started. See *Dynamic Context Switchers* for more details.

    - Another tool (such as a test runner) can use the new *Coverage.switch_context()* method to explicitly change the context.

- The `dynamic_context = test_function` setting now works with Python 2 old-style classes, though it only reports the method name, not the class it was defined on. Closes issue 797.

- `fail_under` values more than 100 are reported as errors. Thanks to Mike Fiedler for closing issue 746.

- The "missing" values in the text output are now sorted by line number, so that missing branches are reported near the other lines they affect. The values used to show all missing lines, and then all missing branches.

- Access to the SQLite database used for data storage is now thread-safe. Thanks, Stephan Richter. This closes issue 702.

- Combining data stored in SQLite is now about twice as fast, fixing issue 761. Thanks, Stephan Richter.

- The `filename` attribute on `CoverageData` objects has been made private. You can use the `data_filename` method to get the actual file name being used to store data, and the `base_filename` method to get the original filename before parallelizing suffixes were added. This is part of fixing issue 708.

- Line numbers in the HTML report now align properly with source lines, even when Chrome's minimum font size is set, fixing issue 748. Thanks Wen Ye.

6.15.57 Version 5.0a4 — 2018-11-25

- You can specify the command line to run your program with the `[run] command_line` configuration setting, as requested in issue 695.

- Coverage will create directories as needed for the data file if they don't exist, closing issue 721.

- The `coverage run` command has always adjusted the first entry in sys.path, to properly emulate how Python runs your program. Now this adjustment is skipped if sys.path[0] is already different than Python's default. This fixes issue 715.

- Improvements to context support:

  - The "no such table: meta" error is fixed.: issue 716.

  - Combining data files is now much faster.

- Python 3.8 (as of today!) passes all tests.

6.15.58 Version 5.0a3 — 2018-10-06

- Context support: static contexts let you specify a label for a coverage run, which is recorded in the data, and retained when you combine files. See *Measurement contexts* for more information.

- Dynamic contexts: specifying `[run] dynamic_context = test_function` in the config file will record the test function name as a dynamic context during execution. This is the core of "Who Tests What" (issue 170). Things to note:

  - There is no reporting support yet. Use SQLite to query the .coverage file for information. Ideas are welcome about how reporting could be extended to use this data.

  - There's a noticeable slow-down before any test is run.

  - Data files will now be roughly N times larger, where N is the number of tests you have. Combining data files is therefore also N times slower.

  - No other values for `dynamic_context` are recognized yet. Let me know what else would be useful. I'd like to use a pytest plugin to get better information directly from pytest, for example.

- Environment variable substitution in configuration files now supports two syntaxes for controlling the behavior of undefined variables: if `VARNAME` is not defined, `${VARNAME?}` will raise an error, and `${VARNAME-default value}` will use "default value".

- Partial support for Python 3.8, which has not yet released an alpha. Fixes issue 707 and issue 714.

6.15.59 Version 5.0a2 — 2018-09-03

- Coverage's data storage has changed. In version 4.x, .coverage files were basically JSON. Now, they are SQLite databases. This means the data file can be created earlier than it used to. A large amount of code was refactored to support this change.

  - Because the data file is created differently than previous releases, you may need `parallel=true` where you didn't before.

  - The old data format is still available (for now) by setting the environment variable COVERAGE_STORAGE=json. Please tell me if you think you need to keep the JSON format.

  - The database schema is guaranteed to change in the future, to support new features. I'm looking for opinions about making the schema part of the public API to coverage.py or not.

- Development moved from Bitbucket to GitHub.

- HTML files no longer have trailing and extra white space.

- The sort order in the HTML report is stored in local storage rather than cookies, closing issue 611. Thanks, Federico Bond.

- pickle2json, for converting v3 data files to v4 data files, has been removed.

6.15.60 Version 5.0a1 — 2018-06-05

- Coverage.py no longer supports Python 2.6 or 3.3.

- The location of the configuration file can now be specified with a `COVERAGE_RCFILE` environment variable, as requested in issue 650.

- Namespace packages are supported on Python 3.7, where they used to cause TypeErrors about path being None. Fixes issue 700.

- A new warning (`already-imported`) is issued if measurable files have already been imported before coverage.py started measurement. See *Warnings* for more information.

- Running coverage many times for small runs in a single process should be faster, closing issue 625. Thanks, David MacIver.

- Large HTML report pages load faster. Thanks, Pankaj Pandey.

6.15.61 Version 4.5.4 — 2019-07-29

- Multiprocessing support in Python 3.8 was broken, but is now fixed. Closes issue 828.

6.15.62 Version 4.5.3 — 2019-03-09

- Only packaging metadata changes.

6.15.63 Version 4.5.2 — 2018-11-12

- Namespace packages are supported on Python 3.7, where they used to cause TypeErrors about path being None. Fixes issue 700.

- Python 3.8 (as of today!) passes all tests. Fixes issue 707 and issue 714.

- Development moved from Bitbucket to GitHub.

6.15.64 Version 4.5.1 — 2018-02-10

- Now that 4.5 properly separated the `[run] omit` and `[report] omit` settings, an old bug has become apparent. If you specified a package name for `[run] source`, then omit patterns weren't matched inside that package. This bug (issue 638) is now fixed.

- On Python 3.7, reporting about a decorated function with no body other than a docstring would crash coverage.py with an IndexError (issue 640). This is now fixed.

- Configurer plugins are now reported in the output of `--debug=sys`.

6.15.65 Version 4.5 — 2018-02-03

- A new kind of plugin is supported: configurers are invoked at start-up to allow more complex configuration than the .coveragerc file can easily do. See *Plug-in classes* for details. This solves the complex configuration problem described in issue 563.

- The `fail_under` option can now be a float. Note that you must specify the `[report] precision` configuration option for the fractional part to be used. Thanks to Lars Hupfeldt Nielsen for help with the implementation. Fixes issue 631.

- The `include` and `omit` options can be specified for both the `[run]` and `[report]` phases of execution. 4.4.2 introduced some incorrect interactions between those phases, where the options for one were confused for the other. This is now corrected, fixing issue 621 and issue 622. Thanks to Daniel Hahler for seeing more clearly than I could.

- The `coverage combine` command used to always overwrite the data file, even when no data had been read from apparently combinable files. Now, an error is raised if we thought there were files to combine, but in fact none of them could be used. Fixes issue 629.

- The `coverage combine` command could get confused about path separators when combining data collected on Windows with data collected on Linux, as described in issue 618. This is now fixed: the result path always uses the path separator specified in the `[paths]` result.

- On Windows, the HTML report could fail when source trees are deeply nested, due to attempting to create HTML filenames longer than the 250-character maximum. Now filenames will never get much larger than 200 characters, fixing issue 627. Thanks to Alex Sandro for helping with the fix.

6.15.66 Version 4.4.2 — 2017-11-05

- Support for Python 3.7. In some cases, class and module docstrings are no longer counted in statement totals, which could slightly change your total results.

- Specifying both `--source` and `--include` no longer silently ignores the include setting, instead it displays a warning. Thanks, Loïc Dachary. Closes issue 265 and issue 101.

- Fixed a race condition when saving data and multiple threads are tracing (issue 581). It could produce a "dictionary changed size during iteration" RuntimeError. I believe this mostly but not entirely fixes the race condition. A true fix would likely be too expensive. Thanks, Peter Baughman for the debugging, and Olivier Grisel for the fix with tests.

- Configuration values which are file paths will now apply tilde-expansion, closing issue 589.

- Now secondary config files like tox.ini and setup.cfg can be specified explicitly, and prefixed sections like *[coverage:run]* will be read. Fixes issue 588.

- Be more flexible about the command name displayed by help, fixing issue 600. Thanks, Ben Finney.

6.15.67 Version 4.4.1 — 2017-05-14

- No code changes: just corrected packaging for Python 2.7 Linux wheels.

6.15.68 Version 4.4 — 2017-05-07

- Reports could produce the wrong file names for packages, reporting `pkg.py` instead of the correct `pkg/__init__.py`. This is now fixed. Thanks, Dirk Thomas.

- XML reports could produce `<source>` and `<class>` lines that together didn't specify a valid source file path. This is now fixed. (issue 526)

- Namespace packages are no longer warned as having no code. (issue 572)

- Code that uses `sys.settrace(sys.gettrace())` in a file that wasn't being coverage-measured would prevent correct coverage measurement in following code. An example of this was running doctests programmatically. This is now fixed. (issue 575)

- Errors printed by the `coverage` command now go to stderr instead of stdout.

- Running `coverage xml` in a directory named with non-ASCII characters would fail under Python 2. This is now fixed. (issue 573)

6.15.69 Version 4.4b1 — 2017-04-04

- Some warnings can now be individually disabled. Warnings that can be disabled have a short name appended. The `[run] disable_warnings` setting takes a list of these warning names to disable. Closes both issue 96 and issue 355.

- The XML report now includes attributes from version 4 of the Cobertura XML format, fixing issue 570.

- In previous versions, calling a method that used collected data would prevent further collection. For example, *save()*, *report()*, *html_report()*, and others would all stop collection. An explicit *start()* was needed to get it going again. This is no longer true. Now you can use the collected data and also continue measurement. Both issue 79 and issue 448 described this problem, and have been fixed.

- Plugins can now find un-executed files if they choose, by implementing the *find_executable_files* method. Thanks, Emil Madsen.

- Minimal IronPython support. You should be able to run IronPython programs under `coverage run`, though you will still have to do the reporting phase with CPython.

- Coverage.py has long had a special hack to support CPython's need to measure the coverage of the standard library tests. This code was not installed by kitted versions of coverage.py. Now it is.

6.15.70 Version 4.3.4 — 2017-01-17

- Fixing 2.6 in version 4.3.3 broke other things, because the too-tricky exception wasn't properly derived from Exception, described in issue 556. A newb mistake; it hasn't been a good few days.

6.15.71 Version 4.3.3 — 2017-01-17

- Python 2.6 support was broken due to a testing exception imported for the benefit of the coverage.py test suite. Properly conditionalizing it fixed issue 554 so that Python 2.6 works again.

6.15.72 Version 4.3.2 — 2017-01-16

- Using the `--skip-covered` option on an HTML report with 100% coverage would cause a "No data to report" error, as reported in issue 549. This is now fixed; thanks, Loïc Dachary.

- If-statements can be optimized away during compilation, for example, *if 0:* or *if __debug__:*. Coverage.py had problems properly understanding these statements which existed in the source, but not in the compiled bytecode. This problem, reported in issue 522, is now fixed.

- If you specified `--source` as a directory, then coverage.py would look for importable Python files in that directory, and could identify ones that had never been executed at all. But if you specified it as a package name, that detection wasn't performed. Now it is, closing issue 426. Thanks to Loïc Dachary for the fix.

- If you started and stopped coverage measurement thousands of times in your process, you could crash Python with a "Fatal Python error: deallocating None" error. This is now fixed. Thanks to Alex Groce for the bug report.

- On PyPy, measuring coverage in subprocesses could produce a warning: "Trace function changed, measurement is likely wrong: None". This was spurious, and has been suppressed.

- Previously, coverage.py couldn't start on Jython, due to that implementation missing the multiprocessing module (issue 551). This problem has now been fixed. Also, issue 322 about not being able to invoke coverage conveniently, seems much better: `jython -m coverage run myprog.py` works properly.

- Let's say you ran the HTML report over and over again in the same output directory, with `--skip-covered`. And imagine due to your heroic test-writing efforts, a file just achieved the goal of 100% coverage. With coverage.py 4.3, the old HTML file with the less-than-100% coverage would be left behind. This file is now properly deleted.

6.15.73 Version 4.3.1 — 2016-12-28

- Some environments couldn't install 4.3, as described in issue 540. This is now fixed.

- The check for conflicting `--source` and `--include` was too simple in a few different ways, breaking a few perfectly reasonable use cases, described in issue 541. The check has been reverted while we re-think the fix for issue 265.

6.15.74 Version 4.3 — 2016-12-27

Special thanks to **Loïc Dachary**, who took an extraordinary interest in coverage.py and contributed a number of improvements in this release.

- Subprocesses that are measured with automatic subprocess measurement used to read in any pre-existing data file. This meant data would be incorrectly carried forward from run to run. Now those files are not read, so each subprocess only writes its own data. Fixes issue 510.

- The `coverage combine` command will now fail if there are no data files to combine. The combine changes in 4.2 meant that multiple combines could lose data, leaving you with an empty .coverage data file. Fixes issue 525, issue 412, issue 516, and probably issue 511.

- Coverage.py wouldn't execute sys.excepthook when an exception happened in your program. Now it does, thanks to Andrew Hoos. Closes issue 535.

- Branch coverage fixes:

  - Branch coverage could misunderstand a finally clause on a try block that never continued on to the following statement, as described in issue 493. This is now fixed. Thanks to Joe Doherty for the report and Loïc Dachary for the fix.

  - A while loop with a constant condition (while True) and a continue statement would be mis-analyzed, as described in issue 496. This is now fixed, thanks to a bug report by Eli Skeggs and a fix by Loïc Dachary.

  - While loops with constant conditions that were never executed could result in a non-zero coverage report. Artem Dayneko reported this in issue 502, and Loïc Dachary provided the fix.

- The HTML report now supports a `--skip-covered` option like the other reporting commands. Thanks, Loïc Dachary for the implementation, closing issue 433.

- Options can now be read from a tox.ini file, if any. Like setup.cfg, sections are prefixed with "coverage:", so `[run]` options will be read from the `[coverage:run]` section of tox.ini. Implements part of issue 519. Thanks, Stephen Finucane.

- Specifying both `--source` and `--include` no longer silently ignores the include setting, instead it fails with a message. Thanks, Nathan Land and Loïc Dachary. Closes issue 265.

- The `Coverage.combine` method has a new parameter, `strict=False`, to support failing if there are no data files to combine.

- When forking subprocesses, the coverage data files would have the same random number appended to the file name. This didn't cause problems, because the file names had the process id also, making collisions (nearly) impossible. But it was disconcerting. This is now fixed.

- The text report now properly sizes headers when skipping some files, fixing issue 524. Thanks, Anthony Sottile and Loïc Dachary.

- Coverage.py can now search .pex files for source, just as it can .zip and .egg. Thanks, Peter Ebden.

- Data files are now about 15% smaller.

- Improvements in the `[run] debug` setting:

  - The "dataio" debug setting now also logs when data files are deleted during combining or erasing.

  - A new debug option, "multiproc", for logging the behavior of `concurrency=multiprocessing`.

  - If you used the debug options "config" and "callers" together, you'd get a call stack printed for every line in the multi-line config output. This is now fixed.

- Fixed an unusual bug involving multiple coding declarations affecting code containing code in multi-line strings: issue 529.

- Coverage.py will no longer be misled into thinking that a plain file is a package when interpreting `--source` options. Thanks, Cosimo Lupo.

- If you try to run a non-Python file with coverage.py, you will now get a more useful error message. Issue 514.

- The default pragma regex changed slightly, but this will only matter to you if you are deranged and use mixed-case pragmas.

- Deal properly with non-ASCII file names in an ASCII-only world, issue 533.

- Programs that set Unicode configuration values could cause UnicodeErrors when generating HTML reports. Pytest-cov is one example. This is now fixed.

- Prevented deprecation warnings from configparser that happened in some circumstances, closing issue 530.

- Corrected the name of the jquery.ba-throttle-debounce.js library. Thanks, Ben Finney. Closes issue 505.

- Testing against PyPy 5.6 and PyPy3 5.5.

- Switched to pytest from nose for running the coverage.py tests.

- Renamed AUTHORS.txt to CONTRIBUTORS.txt, since there are other ways to contribute than by writing code. Also put the count of contributors into the author string in setup.py, though this might be too cute.

6.15.75 Version 4.2 — 2016-07-26

- Since `concurrency=multiprocessing` uses subprocesses, options specified on the coverage.py command line will not be communicated down to them. Only options in the configuration file will apply to the subprocesses. Previously, the options didn't apply to the subprocesses, but there was no indication. Now it is an error to use `--concurrency=multiprocessing` and other run-affecting options on the command line. This prevents failures like those reported in issue 495.

- Filtering the HTML report is now faster, thanks to Ville Skyttä.

6.15.76 Version 4.2b1 — 2016-07-04

Work from the PyCon 2016 Sprints!

- BACKWARD INCOMPATIBILITY: the `coverage combine` command now ignores an existing `.coverage` data file. It used to include that file in its combining. This caused confusing results, and extra tox "clean" steps. If you want the old behavior, use the new `coverage combine --append` option.

- The `concurrency` option can now take multiple values, to support programs using multiprocessing and another library such as eventlet. This is only possible in the configuration file, not from the command line. The configuration file is the only way for sub-processes to all run with the same options. Fixes issue 484. Thanks to Josh Williams for prototyping.

- Using a `concurrency` setting of `multiprocessing` now implies `--parallel` so that the main program is measured similarly to the sub-processes.

- When using automatic subprocess measurement, running coverage commands would create spurious data files. This is now fixed, thanks to diagnosis and testing by Dan Riti. Closes issue 492.

- A new configuration option, `report:sort`, controls what column of the text report is used to sort the rows. Thanks to Dan Wandschneider, this closes issue 199.

- The HTML report has a more-visible indicator for which column is being sorted. Closes issue 298, thanks to Josh Williams.

- If the HTML report cannot find the source for a file, the message now suggests using the `-i` flag to allow the report to continue. Closes issue 231, thanks, Nathan Land.

- When reports are ignoring errors, there's now a warning if a file cannot be parsed, rather than being silently ignored. Closes issue 396. Thanks, Matthew Boehm.

- A new option for `coverage debug` is available: `coverage debug config` shows the current configuration. Closes issue 454, thanks to Matthew Boehm.

- Running coverage as a module (`python -m coverage`) no longer shows the program name as `__main__.py`. Fixes issue 478. Thanks, Scott Belden.

- The *test_helpers* module has been moved into a separate pip-installable package: unittest-mixins.

6.15.77 Version 4.1 — 2016-05-21

- The internal attribute *Reporter.file_reporters* was removed in 4.1b3. It should have come has no surprise that there were third-party tools out there using that attribute. It has been restored, but with a deprecation warning.

6.15.78 Version 4.1b3 — 2016-05-10

- When running your program, execution can jump from an `except X:` line to some other line when an exception other than `X` happens. This jump is no longer considered a branch when measuring branch coverage.

- When measuring branch coverage, `yield` statements that were never resumed were incorrectly marked as missing, as reported in issue 440. This is now fixed.

- During branch coverage of single-line callables like lambdas and generator expressions, coverage.py can now distinguish between them never being called, or being called but not completed. Fixes issue 90, issue 460 and issue 475.

- The HTML report now has a map of the file along the rightmost edge of the page, giving an overview of where the missed lines are. Thanks, Dmitry Shishov.

- The HTML report now uses different monospaced fonts, favoring Consolas over Courier. Along the way, issue 472 about not properly handling one-space indents was fixed. The index page also has slightly different styling, to try to make the clickable detail pages more apparent.

- Missing branches reported with `coverage report -m` will now say `->exit` for missed branches to the exit of a function, rather than a negative number. Fixes issue 469.

- `coverage --help` and `coverage --version` now mention which tracer is installed, to help diagnose problems. The docs mention which features need the C extension. (issue 479)

- Officially support PyPy 5.1, which required no changes, just updates to the docs.

- The *Coverage.report* function had two parameters with non-None defaults, which have been changed. *show_missing* used to default to True, but now defaults to None. If you had been calling *Coverage.report* without specifying *show_missing*, you'll need to explicitly set it to True to keep the same behavior. *skip_covered* used to default to False. It is now None, which doesn't change the behavior. This fixes issue 485.

- It's never been possible to pass a namespace module to one of the analysis functions, but now at least we raise a more specific error message, rather than getting confused. (issue 456)

- The *coverage.process_startup* function now returns the *Coverage* instance it creates, as suggested in issue 481.

- Make a small tweak to how we compare threads, to avoid buggy custom comparison code in thread classes. (issue 245)

6.15.79 Version 4.1b2 — 2016-01-23

- Problems with the new branch measurement in 4.1 beta 1 were fixed:

  - Class docstrings were considered executable. Now they no longer are.

  - `yield from` and `await` were considered returns from functions, since they could transfer control to the caller. This produced unhelpful "missing branch" reports in a number of circumstances. Now they no longer are considered returns.

  - In unusual situations, a missing branch to a negative number was reported. This has been fixed, closing issue 466.

- The XML report now produces correct package names for modules found in directories specified with `source=`. Fixes issue 465.

- `coverage report` won't produce trailing white space.

6.15.80 Version 4.1b1 — 2016-01-10

- Branch analysis has been rewritten: it used to be based on bytecode, but now uses AST analysis. This has changed a number of things:

  - More code paths are now considered runnable, especially in `try/except` structures. This may mean that coverage.py will identify more code paths as uncovered. This could either raise or lower your overall coverage number.

  - Python 3.5's `async` and `await` keywords are properly supported, fixing issue 434.

  - Some long-standing branch coverage bugs were fixed:

    * issue 129: functions with only a docstring for a body would incorrectly report a missing branch on the `def` line.

    * issue 212: code in an `except` block could be incorrectly marked as a missing branch.

    * issue 146: context managers (`with` statements) in a loop or `try` block could confuse the branch measurement, reporting incorrect partial branches.

    * issue 422: in Python 3.5, an actual partial branch could be marked as complete.

- Pragmas to disable coverage measurement can now be used on decorator lines, and they will apply to the entire function or class being decorated. This implements the feature requested in issue 131.

- Multiprocessing support is now available on Windows. Thanks, Rodrigue Cloutier.

- Files with two encoding declarations are properly supported, fixing issue 453. Thanks, Max Linke.

- Non-ascii characters in regexes in the configuration file worked in 3.7, but stopped working in 4.0. Now they work again, closing issue 455.

- Form-feed characters would prevent accurate determination of the beginning of statements in the rest of the file. This is now fixed, closing issue 461.

6.15.81 Version 4.0.3 — 2015-11-24

- Fixed a mysterious problem that manifested in different ways: sometimes hanging the process (issue 420), sometimes making database connections fail (issue 445).

- The XML report now has correct <source> elements when using a --source= option somewhere besides the current directory. This fixes issue 439. Thanks, Arcadiy Ivanov.

- Fixed an unusual edge case of detecting source encodings, described in issue 443.

- Help messages that mention the command to use now properly use the actual command name, which might be different than "coverage". Thanks to Ben Finney, this closes issue 438.

6.15.82 Version 4.0.2 — 2015-11-04

- More work on supporting unusually encoded source. Fixed issue 431.

- Files or directories with non-ASCII characters are now handled properly, fixing issue 432.

- Setting a trace function with sys.settrace was broken by a change in 4.0.1, as reported in issue 436. This is now fixed.

- Officially support PyPy 4.0, which required no changes, just updates to the docs.

6.15.83 Version 4.0.1 — 2015-10-13

- When combining data files, unreadable files will now generate a warning instead of failing the command. This is more in line with the older coverage.py v3.7.1 behavior, which silently ignored unreadable files. Prompted by issue 418.

- The –skip-covered option would skip reporting on 100% covered files, but also skipped them when calculating total coverage. This was wrong, it should only remove lines from the report, not change the final answer. This is now fixed, closing issue 423.

- In 4.0, the data file recorded a summary of the system on which it was run. Combined data files would keep all of those summaries. This could lead to enormous data files consisting of mostly repetitive useless information. That summary is now gone, fixing issue 415. If you want summary information, get in touch, and we'll figure out a better way to do it.

- Test suites that mocked os.path.exists would experience strange failures, due to coverage.py using their mock inadvertently. This is now fixed, closing issue 416.

- Importing a __init__ module explicitly would lead to an error: `AttributeError: 'module' object has no attribute '__path__'`, as reported in issue 410. This is now fixed.

- Code that uses `sys.settrace(sys.gettrace())` used to incur a more than 2x speed penalty. Now there's no penalty at all. Fixes issue 397.

- Pyexpat C code will no longer be recorded as a source file, fixing issue 419.

- The source kit now contains all of the files needed to have a complete source tree, re-fixing issue 137 and closing issue 281.

6.15.84 Version 4.0 — 2015-09-20

No changes from 4.0b3

6.15.85 Version 4.0b3 — 2015-09-07

- Reporting on an unmeasured file would fail with a traceback. This is now fixed, closing issue 403.

- The Jenkins ShiningPanda plugin looks for an obsolete file name to find the HTML reports to publish, so it was failing under coverage.py 4.0. Now we create that file if we are running under Jenkins, to keep things working smoothly. issue 404.

- Kits used to include tests and docs, but didn't install them anywhere, or provide all of the supporting tools to make them useful. Kits no longer include tests and docs. If you were using them from the older packages, get in touch and help me understand how.

6.15.86 Version 4.0b2 — 2015-08-22

- 4.0b1 broke `--append` creating new data files. This is now fixed, closing issue 392.

- `py.test --cov` can write empty data, then touch files due to `--source`, which made coverage.py mistakenly force the data file to record lines instead of arcs. This would lead to a "Can't combine line data with arc data" error message. This is now fixed, and changed some method names in the CoverageData interface. Fixes issue 399.

- *CoverageData.read_fileobj* and *CoverageData.write_fileobj* replace the *.read* and *.write* methods, and are now properly inverses of each other.

- When using `report --skip-covered`, a message will now be included in the report output indicating how many files were skipped, and if all files are skipped, coverage.py won't accidentally scold you for having no data to report. Thanks, Krystian Kichewko.

- A new conversion utility has been added: `python -m coverage.pickle2json` will convert v3.x pickle data files to v4.x JSON data files. Thanks, Alexander Todorov. Closes issue 395.

- A new version identifier is available, *coverage.version_info*, a plain tuple of values similar to sys.version_info.

6.15.87 Version 4.0b1 — 2015-08-02

- Coverage.py is now licensed under the Apache 2.0 license. See NOTICE.txt for details. Closes issue 313.

- The data storage has been completely revamped. The data file is now JSON-based instead of a pickle, closing issue 236. The *CoverageData* class is now a public supported documented API to the data file.

- A new configuration option, `[run] note`, lets you set a note that will be stored in the *runs* section of the data file. You can use this to annotate the data file with any information you like.

- Unrecognized configuration options will now print an error message and stop coverage.py. This should help prevent configuration mistakes from passing silently. Finishes issue 386.

- In parallel mode, `coverage erase` will now delete all of the data files, fixing issue 262.

- Coverage.py now accepts a directory name for `coverage run` and will run a `__main__.py` found there, just like Python will. Fixes issue 252. Thanks, Dmitry Trofimov.

- The XML report now includes a `missing-branches` attribute. Thanks, Steve Peak. This is not a part of the Cobertura DTD, so the XML report no longer references the DTD.

- Missing branches in the HTML report now have a bit more information in the right-hand annotations. Hopefully this will make their meaning clearer.

- All the reporting functions now behave the same if no data had been collected, exiting with a status code of 1. Fixed `fail_under` to be applied even when the report is empty. Thanks, Ionel Cristian Mărieș.

- Plugins are now initialized differently. Instead of looking for a class called `Plugin`, coverage.py looks for a function called `coverage_init`.

- A file-tracing plugin can now ask to have built-in Python reporting by returning *"python"* from its *file_reporter()* method.

- Code that was executed with *exec* would be mis-attributed to the file that called it. This is now fixed, closing issue 380.

- The ability to use item access on *Coverage.config* (introduced in 4.0a2) has been changed to a more explicit *Coverage.get_option* and *Coverage.set_option* API.

- The `Coverage.use_cache` method is no longer supported.

- The private method `Coverage._harvest_data` is now called `Coverage.get_data`, and returns the `CoverageData` containing the collected data.

- The project is consistently referred to as "coverage.py" throughout the code and the documentation, closing issue 275.

- Combining data files with an explicit configuration file was broken in 4.0a6, but now works again, closing issue 385.

- `coverage combine` now accepts files as well as directories.

- The speed is back to 3.7.1 levels, after having slowed down due to plugin support, finishing up issue 387.

6.15.88 Version 4.0a6 — 2015-06-21

- Python 3.5b2 and PyPy 2.6.0 are supported.

- The original module-level function interface to coverage.py is no longer supported. You must now create a `coverage.Coverage` object, and use methods on it.

- The `coverage combine` command now accepts any number of directories as arguments, and will combine all the data files from those directories. This means you don't have to copy the files to one directory before combining. Thanks, Christine Lytwynec. Finishes issue 354.

- Branch coverage couldn't properly handle certain extremely long files. This is now fixed (issue 359).

- Branch coverage didn't understand yield statements properly. Mickie Betz persisted in pursuing this despite Ned's pessimism. Fixes issue 308 and issue 324.

- The COVERAGE_DEBUG environment variable can be used to set the `[run] debug` configuration option to control what internal operations are logged.

- HTML reports were truncated at formfeed characters. This is now fixed (issue 360). It's always fun when the problem is due to a bug in the Python standard library.

- Files with incorrect encoding declaration comments are no longer ignored by the reporting commands, fixing issue 351.

- HTML reports now include a time stamp in the footer, closing issue 299. Thanks, Conrad Ho.

- HTML reports now begrudgingly use double-quotes rather than single quotes, because there are "software engineers" out there writing tools that read HTML and somehow have no idea that single quotes exist. Capitulates to the absurd issue 361. Thanks, Jon Chappell.

- The `coverage annotate` command now handles non-ASCII characters properly, closing issue 363. Thanks, Leonardo Pistone.

- Drive letters on Windows were not normalized correctly, now they are. Thanks, Ionel Cristian Mărieș.

- Plugin support had some bugs fixed, closing issue 374 and issue 375. Thanks, Stefan Behnel.

6.15.89 Version 4.0a5 — 2015-02-16

- Plugin support is now implemented in the C tracer instead of the Python tracer. This greatly improves the speed of tracing projects using plugins.

- Coverage.py now always adds the current directory to sys.path, so that plugins can import files in the current directory (issue 358).

- If the *config_file* argument to the Coverage constructor is specified as ".coveragerc", it is treated as if it were True. This means setup.cfg is also examined, and a missing file is not considered an error (issue 357).

- Wildly experimental: support for measuring processes started by the multiprocessing module. To use, set `--concurrency=multiprocessing`, either on the command line or in the .coveragerc file (issue 117). Thanks, Eduardo Schettino. Currently, this does not work on Windows.

- A new warning is possible, if a desired file isn't measured because it was imported before coverage.py was started (issue 353).

- The *coverage.process_startup* function now will start coverage measurement only once, no matter how many times it is called. This fixes problems due to unusual virtualenv configurations (issue 340).

- Added 3.5.0a1 to the list of supported CPython versions.

6.15.90 Version 4.0a4 — 2015-01-25

- Plugins can now provide sys_info for debugging output.

- Started plugins documentation.

- Prepared to move the docs to readthedocs.org.

6.15.91 Version 4.0a3 — 2015-01-20

- Reports now use file names with extensions. Previously, a report would describe a/b/c.py as "a/b/c". Now it is shown as "a/b/c.py". This allows for better support of non-Python files, and also fixed issue 69.

- The XML report now reports each directory as a package again. This was a bad regression, I apologize. This was reported in issue 235, which is now fixed.

- A new configuration option for the XML report: `[xml]` `package_depth` controls which directories are identified as packages in the report. Directories deeper than this depth are not reported as packages. The default is that all directories are reported as packages. Thanks, Lex Berezhny.

- When looking for the source for a frame, check if the file exists. On Windows, .pyw files are no longer recorded as .py files. Along the way, this fixed issue 290.

- Empty files are now reported as 100% covered in the XML report, not 0% covered (issue 345).

- Regexes in the configuration file are now compiled as soon as they are read, to provide error messages earlier (issue 349).

6.15.92 Version 4.0a2 — 2015-01-14

- Officially support PyPy 2.4, and PyPy3 2.4. Drop support for CPython 3.2 and older versions of PyPy. The code won't work on CPython 3.2. It will probably still work on older versions of PyPy, but I'm not testing against them.

- Plugins!

- The original command line switches (*-x* to run a program, etc) are no longer supported.

- A new option: *coverage report –skip-covered* will reduce the number of files reported by skipping files with 100% coverage. Thanks, Krystian Kichewko. This means that empty *__init__.py* files will be skipped, since they are 100% covered, closing issue 315.

- You can now specify the `--fail-under` option in the `.coveragerc` file as the `[report]` `fail_under` option. This closes issue 314.

- The `COVERAGE_OPTIONS` environment variable is no longer supported. It was a hack for `--timid` before configuration files were available.

- The HTML report now has filtering. Type text into the Filter box on the index page, and only modules with that text in the name will be shown. Thanks, Danny Allen.

- The textual report and the HTML report used to report partial branches differently for no good reason. Now the text report's "missing branches" column is a "partial branches" column so that both reports show the same numbers. This closes issue 342.

- If you specify a `--rcfile` that cannot be read, you will get an error message. Fixes issue 343.

- The `--debug` switch can now be used on any command.

- You can now programmatically adjust the configuration of coverage.py by setting items on *Coverage.config* after construction.

- A module run with `-m` can be used as the argument to `--source`, fixing issue 328. Thanks, Buck Evan.

- The regex for matching exclusion pragmas has been fixed to allow more kinds of white space, fixing issue 334.

- Made some PyPy-specific tweaks to improve speed under PyPy. Thanks, Alex Gaynor.

- In some cases, with a source file missing a final newline, coverage.py would count statements incorrectly. This is now fixed, closing issue 293.

- The status.dat file that HTML reports use to avoid re-creating files that haven't changed is now a JSON file instead of a pickle file. This obviates issue 287 and issue 237.

6.15.93 Version 4.0a1 — 2014-09-27

- Python versions supported are now CPython 2.6, 2.7, 3.2, 3.3, and 3.4, and PyPy 2.2.

- Gevent, eventlet, and greenlet are now supported, closing issue 149. The `concurrency` setting specifies the concurrency library in use. Huge thanks to Peter Portante for initial implementation, and to Joe Jevnik for the final insight that completed the work.

- Options are now also read from a setup.cfg file, if any. Sections are prefixed with "coverage:", so the `[run]` options will be read from the `[coverage:run]` section of setup.cfg. Finishes issue 304.

- The `report -m` command can now show missing branches when reporting on branch coverage. Thanks, Steve Leonard. Closes issue 230.

- The XML report now contains a <source> element, fixing issue 94. Thanks Stan Hu.

- The class defined in the coverage module is now called `Coverage` instead of `coverage`, though the old name still works, for backward compatibility.

- The `fail-under` value is now rounded the same as reported results, preventing paradoxical results, fixing issue 284.

- The XML report will now create the output directory if need be, fixing issue 285. Thanks, Chris Rose.

- HTML reports no longer raise UnicodeDecodeError if a Python file has un-decodable characters, fixing issue 303 and issue 331.

- The annotate command will now annotate all files, not just ones relative to the current directory, fixing issue 57.

- The coverage module no longer causes deprecation warnings on Python 3.4 by importing the imp module, fixing issue 305.

- Encoding declarations in source files are only considered if they are truly comments. Thanks, Anthony Sottile.

6.15.94 Version 3.7.1 — 2013-12-13

- Improved the speed of HTML report generation by about 20%.

- Fixed the mechanism for finding OS-installed static files for the HTML report so that it will actually find OS-installed static files.

6.15.95 Version 3.7 — 2013-10-06

- Added the `--debug` switch to `coverage run`. It accepts a list of options indicating the type of internal activity to log to stderr.

- Improved the branch coverage facility, fixing issue 92 and issue 175.

- Running code with `coverage run -m` now behaves more like Python does, setting sys.path properly, which fixes issue 207 and issue 242.

- Coverage.py can now run .pyc files directly, closing issue 264.

- Coverage.py properly supports .pyw files, fixing issue 261.

- Omitting files within a tree specified with the `source` option would cause them to be incorrectly marked as un-executed, as described in issue 218. This is now fixed.

- When specifying paths to alias together during data combining, you can now specify relative paths, fixing issue 267.

- Most file paths can now be specified with username expansion (`~/src`, or `~build/src`, for example), and with environment variable expansion (`build/$BUILDNUM/src`).

- Trying to create an XML report with no files to report on, would cause a ZeroDivisionError, but no longer does, fixing issue 250.

- When running a threaded program under the Python tracer, coverage.py no longer issues a spurious warning about the trace function changing: "Trace function changed, measurement is likely wrong: None." This fixes issue 164.

- Static files necessary for HTML reports are found in system-installed places, to ease OS-level packaging of coverage.py. Closes issue 259.

- Source files with encoding declarations, but a blank first line, were not decoded properly. Now they are. Thanks, Roger Hu.

- The source kit now includes the `__main__.py` file in the root coverage directory, fixing issue 255.

6.15.96 Version 3.6 — 2013-01-05

- Added a page to the docs about troublesome situations, closing issue 226, and added some info to the TODO file, closing issue 227.

6.15.97 Version 3.6b3 — 2012-12-29

- Beta 2 broke the nose plugin. It's fixed again, closing issue 224.

6.15.98 Version 3.6b2 — 2012-12-23

- Coverage.py runs on Python 2.3 and 2.4 again. It was broken in 3.6b1.

- The C extension is optionally compiled using a different more widely-used technique, taking another stab at fixing issue 80 once and for all.

- Combining data files would create entries for phantom files if used with `source` and path aliases. It no longer does.

- `debug sys` now shows the configuration file path that was read.

- If an oddly-behaved package claims that code came from an empty-string file name, coverage.py no longer associates it with the directory name, fixing issue 221.

6.15.99 Version 3.6b1 — 2012-11-28

- Wildcards in `include=` and `omit=` arguments were not handled properly in reporting functions, though they were when running. Now they are handled uniformly, closing issue 143 and issue 163. **NOTE**: it is possible that your configurations may now be incorrect. If you use `include` or `omit` during reporting, whether on the command line, through the API, or in a configuration file, please check carefully that you were not relying on the old broken behavior.

- The **report**, **html**, and **xml** commands now accept a `--fail-under` switch that indicates in the exit status whether the coverage percentage was less than a particular value. Closes issue 139.

- The reporting functions coverage.report(), coverage.html_report(), and coverage.xml_report() now all return a float, the total percentage covered measurement.

- The HTML report's title can now be set in the configuration file, with the `--title` switch on the command line, or via the API.

- Configuration files now support substitution of environment variables, using syntax like `${WORD}`. Closes issue 97.

- Embarrassingly, the `[xml] output=` setting in the .coveragerc file simply didn't work. Now it does.

- The XML report now consistently uses file names for the file name attribute, rather than sometimes using module names. Fixes issue 67. Thanks, Marcus Cobden.

- Coverage percentage metrics are now computed slightly differently under branch coverage. This means that completely un-executed files will now correctly have 0% coverage, fixing issue 156. This also means that your total coverage numbers will generally now be lower if you are measuring branch coverage.

- When installing, now in addition to creating a "coverage" command, two new aliases are also installed. A "coverage2" or "coverage3" command will be created, depending on whether you are installing in Python 2.x or 3.x. A "coverage-X.Y" command will also be created corresponding to your specific version of Python. Closes issue 111.

- The coverage.py installer no longer tries to bootstrap setuptools or Distribute. You must have one of them installed first, as issue 202 recommended.

- The coverage.py kit now includes docs (closing issue 137) and tests.

- On Windows, files are now reported in their correct case, fixing issue 89 and issue 203.

- If a file is missing during reporting, the path shown in the error message is now correct, rather than an incorrect path in the current directory. Fixes issue 60.

- Running an HTML report in Python 3 in the same directory as an old Python 2 HTML report would fail with a UnicodeDecodeError. This issue (issue 193) is now fixed.

- Fixed yet another error trying to parse non-Python files as Python, this time an IndentationError, closing issue 82 for the fourth time...

- If *coverage xml* fails because there is no data to report, it used to create a zero-length XML file. Now it doesn't, fixing issue 210.

- Jython files now work with the `--source` option, fixing issue 100.

- Running coverage.py under a debugger is unlikely to work, but it shouldn't fail with "TypeError: 'NoneType' object is not iterable". Fixes issue 201.

- On some Linux distributions, when installed with the OS package manager, coverage.py would report its own code as part of the results. Now it won't, fixing issue 214, though this will take some time to be repackaged by the operating systems.

- Docstrings for the legacy singleton methods are more helpful. Thanks Marius Gedminas. Closes issue 205.

- The pydoc tool can now show documentation for the class *coverage.coverage*. Closes issue 206.

- Added a page to the docs about contributing to coverage.py, closing issue 171.

- When coverage.py ended unsuccessfully, it may have reported odd errors like `'NoneType' object has no attribute 'isabs'`. It no longer does, so kiss issue 153 goodbye.

6.15.100 Version 3.5.3 — 2012-09-29

- Line numbers in the HTML report line up better with the source lines, fixing issue 197, thanks Marius Gedminas.

- When specifying a directory as the source= option, the directory itself no longer needs to have a `__init__.py` file, though its sub-directories do, to be considered as source files.

- Files encoded as UTF-8 with a BOM are now properly handled, fixing issue 179. Thanks, Pablo Carballo.

- Fixed more cases of non-Python files being reported as Python source, and then not being able to parse them as Python. Closes issue 82 (again). Thanks, Julian Berman.

- Fixed memory leaks under Python 3, thanks, Brett Cannon. Closes issue 147.

- Optimized .pyo files may not have been handled correctly, issue 195. Thanks, Marius Gedminas.

- Certain unusually named file paths could have been mangled during reporting, issue 194. Thanks, Marius Gedminas.

- Try to do a better job of the impossible task of detecting when we can't build the C extension, fixing issue 183.

- Testing is now done with tox, thanks, Marc Abramowitz.

6.15.101 Version 3.5.2 — 2012-05-04

No changes since 3.5.2.b1

6.15.102 Version 3.5.2b1 — 2012-04-29

- The HTML report has slightly tweaked controls: the buttons at the top of the page are color-coded to the source lines they affect.

- Custom CSS can be applied to the HTML report by specifying a CSS file as the `extra_css` configuration value in the `[html]` section.

- Source files with custom encodings declared in a comment at the top are now properly handled during reporting on Python 2. Python 3 always handled them properly. This fixes issue 157.

- Backup files left behind by editors are no longer collected by the source= option, fixing issue 168.

- If a file doesn't parse properly as Python, we don't report it as an error if the file name seems like maybe it wasn't meant to be Python. This is a pragmatic fix for issue 82.

- The `-m` switch on `coverage report`, which includes missing line numbers in the summary report, can now be specified as `show_missing` in the config file. Closes issue 173.

- When running a module with `coverage run -m <modulename>`, certain details of the execution environment weren't the same as for `python -m <modulename>`. This had the unfortunate side-effect of making `coverage run -m unittest discover` not work if you had tests in a directory named "test". This fixes issue 155 and issue 142.

- Now the exit status of your product code is properly used as the process status when running `python -m coverage run ...`. Thanks, JT Olds.

- When installing into PyPy, we no longer attempt (and fail) to compile the C tracer function, closing issue 166.

6.15.103 Version 3.5.1 — 2011-09-23

- The `[paths]` feature unfortunately didn't work in real world situations where you wanted to, you know, report on the combined data. Now all paths stored in the combined file are canonicalized properly.

6.15.104 Version 3.5.1b1 — 2011-08-28

- When combining data files from parallel runs, you can now instruct coverage.py about which directories are equivalent on different machines. A `[paths]` section in the configuration file lists paths that are to be considered equivalent. Finishes issue 17.

- for-else constructs are understood better, and don't cause erroneous partial branch warnings. Fixes issue 122.

- Branch coverage for `with` statements is improved, fixing issue 128.

- The number of partial branches reported on the HTML summary page was different than the number reported on the individual file pages. This is now fixed.

- An explicit include directive to measure files in the Python installation wouldn't work because of the standard library exclusion. Now the include directive takes precedence, and the files will be measured. Fixes issue 138.

- The HTML report now handles Unicode characters in Python source files properly. This fixes issue 124 and issue 144. Thanks, Devin Jeanpierre.

- In order to help the core developers measure the test coverage of the standard library, Brandon Rhodes devised an aggressive hack to trick Python into running some coverage.py code before anything else in the process. See the coverage/fullcoverage directory if you are interested.

6.15.105 Version 3.5 — 2011-06-29

- The HTML report hotkeys now behave slightly differently when the current chunk isn't visible at all: a chunk on the screen will be selected, instead of the old behavior of jumping to the literal next chunk. The hotkeys now work in Google Chrome. Thanks, Guido van Rossum.

6.15.106 Version 3.5b1 — 2011-06-05

- The HTML report now has hotkeys. Try `n`, `s`, `m`, `x`, `b`, `p`, and `c` on the overview page to change the column sorting. On a file page, `r`, `m`, `x`, and `p` toggle the run, missing, excluded, and partial line markings. You can navigate the highlighted sections of code by using the `j` and `k` keys for next and previous. The `1` (one) key jumps to the first highlighted section in the file, and `0` (zero) scrolls to the top of the file.

- The `--omit` and `--include` switches now interpret their values more usefully. If the value starts with a wildcard character, it is used as-is. If it does not, it is interpreted relative to the current directory. Closes issue 121.

- Partial branch warnings can now be pragma'd away. The configuration option `partial_branches` is a list of regular expressions. Lines matching any of those expressions will never be marked as a partial branch. In addition, there's a built-in list of regular expressions marking statements which should never be marked as partial. This list includes `while True:`, `while 1:`, `if 1:`, and `if 0:`.

- The `coverage()` constructor accepts single strings for the `omit=` and `include=` arguments, adapting to a common error in programmatic use.

- Modules can now be run directly using `coverage run -m modulename`, to mirror Python's -m flag. Closes issue 95, thanks, Brandon Rhodes.

- `coverage run` didn't emulate Python accurately in one small detail: the current directory inserted into `sys.path` was relative rather than absolute. This is now fixed.

- HTML reporting is now incremental: a record is kept of the data that produced the HTML reports, and only files whose data has changed will be generated. This should make most HTML reporting faster.

- Pathological code execution could disable the trace function behind our backs, leading to incorrect code measurement. Now if this happens, coverage.py will issue a warning, at least alerting you to the problem. Closes issue 93. Thanks to Marius Gedminas for the idea.

- The C-based trace function now behaves properly when saved and restored with `sys.gettrace()` and `sys.settrace()`. This fixes issue 125 and issue 123. Thanks, Devin Jeanpierre.

- Source files are now opened with Python 3.2's `tokenize.open()` where possible, to get the best handling of Python source files with encodings. Closes issue 107, thanks, Brett Cannon.

- Syntax errors in supposed Python files can now be ignored during reporting with the `-i` switch just like other source errors. Closes issue 115.

- Installation from source now succeeds on machines without a C compiler, closing issue 80.

- Coverage.py can now be run directly from a working tree by specifying the directory name to python: `python coverage_py_working_dir run ...`. Thanks, Brett Cannon.

- A little bit of Jython support: *coverage run* can now measure Jython execution by adapting when $py.class files are traced. Thanks, Adi Roiban. Jython still doesn't provide the Python libraries needed to make coverage reporting work, unfortunately.

- Internally, files are now closed explicitly, fixing issue 104. Thanks, Brett Cannon.

6.15.107 Version 3.4 — 2010-09-19

- The XML report is now sorted by package name, fixing issue 88.

- Programs that exited with `sys.exit()` with no argument weren't handled properly, producing a coverage.py stack trace. That is now fixed.

6.15.108 Version 3.4b2 — 2010-09-06

- Completely un-executed files can now be included in coverage results, reported as 0% covered. This only happens if the –source option is specified, since coverage.py needs guidance about where to look for source files.

- The XML report output now properly includes a percentage for branch coverage, fixing issue 65 and issue 81.

- Coverage percentages are now displayed uniformly across reporting methods. Previously, different reports could round percentages differently. Also, percentages are only reported as 0% or 100% if they are truly 0 or 100, and are rounded otherwise. Fixes issue 41 and issue 70.

- The precision of reported coverage percentages can be set with the `[report]` `precision` config file setting. Completes issue 16.

- Threads derived from `threading.Thread` with an overridden *run* method would report no coverage for the *run* method. This is now fixed, closing issue 85.

6.15.109 Version 3.4b1 — 2010-08-21

- BACKWARD INCOMPATIBILITY: the `--omit` and `--include` switches now take file patterns rather than file prefixes, closing issue 34 and issue 36.

- BACKWARD INCOMPATIBILITY: the *omit_prefixes* argument is gone throughout coverage.py, replaced with *omit*, a list of file name patterns suitable for *fnmatch*. A parallel argument *include* controls what files are included.

- The run command now has a `--source` switch, a list of directories or module names. If provided, coverage.py will only measure execution in those source files.

- Various warnings are printed to stderr for problems encountered during data measurement: if a `--source` module has no Python source to measure, or is never encountered at all, or if no data is collected.

- The reporting commands (report, annotate, html, and xml) now have an `--include` switch to restrict reporting to modules matching those file patterns, similar to the existing `--omit` switch. Thanks, Zooko.

- The run command now supports `--include` and `--omit` to control what modules it measures. This can speed execution and reduce the amount of data during reporting. Thanks Zooko.

- Since coverage.py 3.1, using the Python trace function has been slower than it needs to be. A cache of tracing decisions was broken, but has now been fixed.

- Python 2.7 and 3.2 have introduced new opcodes that are now supported.

- Python files with no statements, for example, empty `__init__.py` files, are now reported as having zero statements instead of one. Fixes issue 1.

- Reports now have a column of missed line counts rather than executed line counts, since developers should focus on reducing the missed lines to zero, rather than increasing the executed lines to varying targets. Once suggested, this seemed blindingly obvious.

- Line numbers in HTML source pages are clickable, linking directly to that line, which is highlighted on arrival. Added a link back to the index page at the bottom of each HTML page.

- Programs that call `os.fork` will properly collect data from both the child and parent processes. Use `coverage run -p` to get two data files that can be combined with `coverage combine`. Fixes issue 56.

- Coverage.py is now runnable as a module: `python -m coverage`. Thanks, Brett Cannon.

- When measuring code running in a virtualenv, most of the system library was being measured when it shouldn't have been. This is now fixed.

- Doctest text files are no longer recorded in the coverage data, since they can't be reported anyway. Fixes issue 52 and issue 61.

- Jinja HTML templates compile into Python code using the HTML file name, which confused coverage.py. Now these files are no longer traced, fixing issue 82.

- Source files can have more than one dot in them (foo.test.py), and will be treated properly while reporting. Fixes issue 46.

- Source files with DOS line endings are now properly tokenized for syntax coloring on non-DOS machines. Fixes issue 53.

- Unusual code structure that confused exits from methods with exits from classes is now properly analyzed. See issue 62.

- Asking for an HTML report with no files now shows a nice error message rather than a cryptic failure ('int' object is unsubscriptable). Fixes issue 59.

6.15.110 Version 3.3.1 — 2010-03-06

- Using *parallel=True* in .coveragerc file prevented reporting, but now does not, fixing issue 49.

- When running your code with "coverage run", if you call *sys.exit()*, coverage.py will exit with that status code, fixing issue 50.

6.15.111 Version 3.3 — 2010-02-24

- Settings are now read from a .coveragerc file. A specific file can be specified on the command line with –rc-file=FILE. The name of the file can be programmatically set with the *config_file* argument to the coverage() constructor, or reading a config file can be disabled with *config_file=False*.

- Fixed a problem with nested loops having their branch possibilities mis-characterized: issue 39.

- Added coverage.process_start to enable coverage measurement when Python starts.

- Parallel data file names now have a random number appended to them in addition to the machine name and process id.

- Parallel data files combined with "coverage combine" are deleted after they're combined, to clean up unneeded files. Fixes issue 40.

- Exceptions thrown from product code run with "coverage run" are now displayed without internal coverage.py frames, so the output is the same as when the code is run without coverage.py.

- The *data_suffix* argument to the coverage constructor is now appended with an added dot rather than simply appended, so that .coveragerc files will not be confused for data files.

- Python source files that don't end with a newline can now be executed, fixing issue 47.

- Added an AUTHORS.txt file.

6.15.112 Version 3.2 — 2009-12-05

- Added a `--version` option on the command line.

6.15.113 Version 3.2b4 — 2009-12-01

- Branch coverage improvements:

    - The XML report now includes branch information.

- Click-to-sort HTML report columns are now persisted in a cookie. Viewing a report will sort it first the way you last had a coverage report sorted. Thanks, Chris Adams.

- On Python 3.x, setuptools has been replaced by Distribute.

6.15.114 Version 3.2b3 — 2009-11-23

- Fixed a memory leak in the C tracer that was introduced in 3.2b1.

- Branch coverage improvements:

    - Branches to excluded code are ignored.

- The table of contents in the HTML report is now sortable: click the headers on any column. Thanks, Chris Adams.

6.15.115 Version 3.2b2 — 2009-11-19

- Branch coverage improvements:

    - Classes are no longer incorrectly marked as branches: issue 32.

    - "except" clauses with types are no longer incorrectly marked as branches: issue 35.

- Fixed some problems syntax coloring sources with line continuations and source with tabs: issue 30 and issue 31.

- The –omit option now works much better than before, fixing issue 14 and issue 33. Thanks, Danek Duvall.

6.15.116 Version 3.2b1 — 2009-11-10

- Branch coverage!

- XML reporting has file paths that let Cobertura find the source code.

- The tracer code has changed, it's a few percent faster.

- Some exceptions reported by the command line interface have been cleaned up so that tracebacks inside coverage.py aren't shown. Fixes issue 23.

6.15.117 Version 3.1 — 2009-10-04

- Source code can now be read from eggs. Thanks, Ross Lawley. Fixes issue 25.

6.15.118 Version 3.1b1 — 2009-09-27

- Python 3.1 is now supported.

- Coverage.py has a new command line syntax with sub-commands. This expands the possibilities for adding features and options in the future. The old syntax is still supported. Try "coverage help" to see the new commands. Thanks to Ben Finney for early help.

- Added an experimental "coverage xml" command for producing coverage reports in a Cobertura-compatible XML format. Thanks, Bill Hart.

- Added the –timid option to enable a simpler slower trace function that works for DecoratorTools projects, including TurboGears. Fixed issue 12 and issue 13.

- HTML reports show modules from other directories. Fixed issue 11.

- HTML reports now display syntax-colored Python source.

- Programs that change directory will still write .coverage files in the directory where execution started. Fixed issue 24.

- Added a "coverage debug" command for getting diagnostic information about the coverage.py installation.

6.15.119 Version 3.0.1 — 2009-07-07

- Removed the recursion limit in the tracer function. Previously, code that ran more than 500 frames deep would crash. Fixed issue 9.

- Fixed a bizarre problem involving pyexpat, whereby lines following XML parser invocations could be overlooked. Fixed issue 10.

- On Python 2.3, coverage.py could mis-measure code with exceptions being raised. This is now fixed.

- The coverage.py code itself will now not be measured by coverage.py, and no coverage.py modules will be mentioned in the nose –with-cover plug-in. Fixed issue 8.

- When running source files, coverage.py now opens them in universal newline mode just like Python does. This lets it run Windows files on Mac, for example.

6.15.120 Version 3.0 — 2009-06-13

- Fixed the way the Python library was ignored. Too much code was being excluded the old way.

- Tabs are now properly converted in HTML reports. Previously indentation was lost. Fixed issue 6.

- Nested modules now get a proper flat_rootname. Thanks, Christian Heimes.

6.15.121 Version 3.0b3 — 2009-05-16

- Added parameters to coverage.__init__ for options that had been set on the coverage object itself.

- Added clear_exclude() and get_exclude_list() methods for programmatic manipulation of the exclude regexes.

- Added coverage.load() to read previously-saved data from the data file.

- Improved the finding of code files. For example, .pyc files that have been installed after compiling are now located correctly. Thanks, Detlev Offenbach.

- When using the object API (that is, constructing a coverage() object), data is no longer saved automatically on process exit. You can re-enable it with the auto_data=True parameter on the coverage() constructor. The module-level interface still uses automatic saving.

6.15.122 Version 3.0b — 2009-04-30

HTML reporting, and continued refactoring.

- HTML reports and annotation of source files: use the new -b (browser) switch. Thanks to George Song for code, inspiration and guidance.

- Code in the Python standard library is not measured by default. If you need to measure standard library code, use the -L command-line switch during execution, or the cover_pylib=True argument to the coverage() constructor.

- Source annotation into a directory (-a -d) behaves differently. The annotated files are named with their hierarchy flattened so that same-named files from different directories no longer collide. Also, only files in the current tree are included.

- coverage.annotate_file is no longer available.

- Programs executed with -x now behave more as they should, for example, __file__ has the correct value.

- .coverage data files have a new pickle-based format designed for better extensibility.

- Removed the undocumented cache_file argument to coverage.usecache().

6.15.123 Version 3.0b1 — 2009-03-07

Major overhaul.

- Coverage.py is now a package rather than a module. Functionality has been split into classes.

- The trace function is implemented in C for speed. Coverage.py runs are now much faster. Thanks to David Christian for productive micro-sprints and other encouragement.

- Executable lines are identified by reading the line number tables in the compiled code, removing a great deal of complicated analysis code.

- Precisely which lines are considered executable has changed in some cases. Therefore, your coverage stats may also change slightly.

- The singleton coverage object is only created if the module-level functions are used. This maintains the old interface while allowing better programmatic use of coverage.py.

- The minimum supported Python version is 2.3.

6.15.124 Version 2.85 — 2008-09-14

- Add support for finding source files in eggs. Don't check for morf's being instances of ModuleType, instead use duck typing so that pseudo-modules can participate. Thanks, Imri Goldberg.

- Use os.realpath as part of the fixing of file names so that symlinks won't confuse things. Thanks, Patrick Mezard.

6.15.125 Version 2.80 — 2008-05-25

- Open files in rU mode to avoid line ending craziness. Thanks, Edward Loper.

6.15.126 Version 2.78 — 2007-09-30

- Don't try to predict whether a file is Python source based on the extension. Extension-less files are often Pythons scripts. Instead, simply parse the file and catch the syntax errors. Hat tip to Ben Finney.

6.15.127 Version 2.77 — 2007-07-29

- Better packaging.

6.15.128 Version 2.76 — 2007-07-23

- Now Python 2.5 is *really* fully supported: the body of the new with statement is counted as executable.

6.15.129 Version 2.75 — 2007-07-22

- Python 2.5 now fully supported. The method of dealing with multi-line statements is now less sensitive to the exact line that Python reports during execution. Pass statements are handled specially so that their disappearance during execution won't throw off the measurement.

6.15.130 Version 2.7 — 2007-07-21

- "#pragma: nocover" is excluded by default.

- Properly ignore docstrings and other constant expressions that appear in the middle of a function, a problem reported by Tim Leslie.

- coverage.erase() shouldn't clobber the exclude regex. Change how parallel mode is invoked, and fix erase() so that it erases the cache when called programmatically.

- In reports, ignore code executed from strings, since we can't do anything useful with it anyway.

- Better file handling on Linux, thanks Guillaume Chazarain.

- Better shell support on Windows, thanks Noel O'Boyle.

- Python 2.2 support maintained, thanks Catherine Proulx.

- Minor changes to avoid lint warnings.

6.15.131 Version 2.6 — 2006-08-23

- Applied Joseph Tate's patch for function decorators.

- Applied Sigve Tjora and Mark van der Wal's fixes for argument handling.

- Applied Geoff Bache's parallel mode patch.

- Refactorings to improve testability. Fixes to command-line logic for parallel mode and collect.

6.15.132 Version 2.5 — 2005-12-04

- Call threading.settrace so that all threads are measured. Thanks Martin Fuzzey.

- Add a file argument to report so that reports can be captured to a different destination.

- Coverage.py can now measure itself.

- Adapted Greg Rogers' patch for using relative file names, and sorting and omitting files to report on.

6.15.133 Version 2.2 — 2004-12-31

- Allow for keyword arguments in the module global functions. Thanks, Allen.

6.15.134 Version 2.1 — 2004-12-14

- Return 'analysis' to its original behavior and add 'analysis2'. Add a global for 'annotate', and factor it, adding 'annotate_file'.

6.15.135 Version 2.0 — 2004-12-12

Significant code changes.

- Finding executable statements has been rewritten so that docstrings and other quirks of Python execution aren't mistakenly identified as missing lines.

- Lines can be excluded from consideration, even entire suites of lines.

- The file system cache of covered lines can be disabled programmatically.

- Modernized the code.

6.15.136 Earlier History

2001-12-04 GDR Created.

2001-12-06 GDR Added command-line interface and source code annotation.

2001-12-09 GDR Moved design and interface to separate documents.

2001-12-10 GDR Open cache file as binary on Windows. Allow simultaneous -e and -x, or -a and -r.

2001-12-12 GDR Added command-line help. Cache analysis so that it only needs to be done once when you specify -a and -r.

2001-12-13 GDR Improved speed while recording. Portable between Python 1.5.2 and 2.1.1.

2002-01-03 GDR Module-level functions work correctly.

2002-01-07 GDR Update sys.path when running a file with the -x option, so that it matches the value the program would get if it were run on its own.

6.16 Migrating between versions

New versions of coverage.py or Python might require you to adjust your settings, options, or other aspects how you use coverage.py. This page details those changes.

6.16.1 Migrating to coverage.py 7.x

Consider these changes when migrating to coverage.py 7.x:

- The way that wildcards when specifying file paths work in certain cases has changed in 7.x:

    - Previously, * would incorrectly match directory separators, making precise matching difficult. Patterns such as `*tests/*` will need to be changed to `*/tests/*`.

    - `**` now matches any number of nested directories. If you wish to retain the behavior of `**/tests/*` in previous versions then `*/**/tests/*` can be used instead.

- When remapping file paths with `[paths]`, a path will be remapped only if the resulting path exists. Ensure that remapped `[paths]` exist when upgrading as this is now being enforced.

- The *[report] exclude_also* setting is new in 7.2.0. It adds exclusion regexes while keeping the default built-in set. It's better than the older *[report] exclude_lines* setting, which overwrote the entire list. Newer versions of coverage.py will be adding to the default set of exclusions. Using `exclude_also` will let you benefit from those updates.

6.16.2 Migrating to Python 3.12

Keep these things in mind when running under Python 3.12:

- Python 3.12 now inlines list, dict, and set comprehensions. Previously, they were compiled as functions that were called internally. Coverage.py would warn you if comprehensions weren't fully completed, but this no longer happens with Python 3.12.

6.17 Sleepy Snake

Coverage.py's mascot is Sleepy Snake, drawn by Ben Batchelder. Ben's art can be found on Instagram and at artof-batch.com. Some details of Sleepy's creation are on Ned's blog.

PYTHON MODULE INDEX

C