Coverage.py

Release 4.1

April 28, 2017

Contents

1 Quick start 3

2 Using coverage.py 5

3 Getting help 7

4 More information 9

Python Module Index 53

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 4.1, released May 21st 2016. It is supported on:

- Python versions 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6.
- PyPy 4.0 and 5.1.
- PyPy3 2.4.

CHAPTER 1

Quick start

Getting started is easy:

1. Install coverage.py from the coverage.py page on the Python Package Index, or by using "pip install coverage". For a few more details, see *Installation*.

2. Use `coverage run` to run your program and gather data:

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

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

CHAPTER 2

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, nose has a cover plug-in.

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.

CHAPTER 3

Getting help

If the *FAQ* doesn't answer your question, you can discuss coverage.py or get help using it on the Testing In Python mailing list.

Bug reports are gladly accepted at the Bitbucket issue tracker. Bitbucket also hosts the code repository. There is a mirrored repo on GitHub.

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

CHAPTER 4

More information

Installation

Installing coverage.py is done in the usual ways. The simplest way is with pip:

```
$ pip install coverage
```

The alternate old-school technique is:

1. Install (or already have installed) setuptools or Distribute.
2. Download the appropriate kit from the coverage.py page on the Python Package Index.
3. Run `python setup.py install`.

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.

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
```

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

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

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

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

Installing on Windows

For Windows, kits are provided on the PyPI page for different versions of Python and different CPU architectures. These kits require that setuptools be installed as a pre-requisite, but otherwise are self-contained. They have the C extension pre-compiled so there's no need to worry about compilers.

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 4.1 with C extension
Documentation at https://coverage.readthedocs.io
```

You can also invoke coverage.py as a module:

```
$ python -m coverage --version
Coverage.py, version 4.1 with C extension
Documentation at https://coverage.readthedocs.io
```

Coverage.py command line usage

When you install coverage.py, a command-line script simply called `coverage` is placed in your Python scripts directory. To help with multi-version installs, it will also create either a `coverage2` or `coverage3` alias, and a `coverage-X.Y` alias, depending on the version of Python you're using. For example, when installing on Python 2.7, you will be able to use `coverage`, `coverage2`, or `coverage-2.7` on the command line.

Coverage.py has a number of commands which determine the action performed:

- **run** – Run a Python program and collect execution data.
- **report** – Report coverage results.
- **html** – Produce annotated HTML listings with coverage results.
- **xml** – Produce an XML report with coverage results.
- **annotate** – Annotate source files with coverage results.
- **erase** – Erase previously collected coverage data.
- **combine** – Combine together a number of data files.
- **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`.

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 files* for more details.

Execution

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
```

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
```

Coverage.py can measure multi-threaded programs by default. If you are using more exotic concurrency, with the multiprocessing, greenlet, eventlet, or gevent libraries, then coverage.py will get very 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*.

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. Projects that use DecoratorTools, including TurboGears, will need to use `--timid` to get correct results.

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* below.

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

- "Trace function changed, measurement is likely wrong: XXX"

  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 indicate 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"

  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"

  You asked coverage.py to measure module XXX, but it was never imported by your program.

- "No data was 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.

- "Module XXX was previously imported, but 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.

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 disjoint options), you can accumulate data across runs with the `-a` flag on the **run** command.

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

```
$ coverage erase
```

Combining data files

If you need to collect coverage data from different machines or processes, coverage.py can combine multiple files into one for reporting.

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 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 looks in directories for data files to combine, even the current directory, it only reads files with certain names. 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
```

The `run --parallel-mode` switch automatically creates separate data files for each run which can be combined later. The file names include the machine name, the process id, and a random number:

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

If the different machines run your code from different places in their file systems, coverage.py won't know how to combine the data. You can tell coverage.py how the different locations correlate with a `[paths]` section in your configuration file. See *[paths]* for details.

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

Reporting

Coverage.py provides a few styles of reporting, with the **report**, **html**, **annotate**, 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**.

Coverage summary

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.

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 leave out any file with 100% coverage, letting you focus on the files that still need attention.

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

HTML annotation

Coverage.py can annotate your source code for 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. The 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.

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]* 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.

Text annotation

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
```

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

XML reporting

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

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

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

Diagnostics

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
```

Two types of information are available: `sys` to show system configuration, and `data` to show a summary of the collected coverage data. The `--debug` option is 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.
- `pid`: annotate all debug output with the process id.
- `plugin`: print information about plugin operations.
- `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.

The debug output goes to stderr, unless the `COVERAGE_DEBUG_FILE` environment variable names a different file, which will be appended to.

Configuration files

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 configuration files 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 name for the configuration file can be specified with the `--rcfile=FILE` command line option.

Coverage.py will read settings from a `setup.cfg` file if no other configuration file is used. In this case, the section names have "coverage:" prefixed, so the `[run]` options described below will be found in the `[coverage:run]` section of `setup.cfg`.

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.

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 `$$`. Missing environment variables will result in empty strings with no error.

Many sections and values correspond roughly to commands and options in the *command-line interface*.

Here's a sample configuration file:

```
# .coveragerc to control coverage.py
[run]
branch = True

[report]
# Regexes for lines to exclude from consideration
exclude_lines =
    # Have to re-enable the standard pragma
    pragma: no cover

    # 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__.:

ignore_errors = True
```

```
[html]
directory = coverage_html_report
```

[run]

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

`branch` (boolean, default False): whether to measure *branch coverage* in addition to statement coverage.

`cover_pylib` (boolean, default False): whether to measure the Python standard library.

`concurrency` (string, default "thread"): the name of the concurrency library 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.

New in version 4.0.

`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.

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

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

`note` (string): an arbitrary string that will be written to the data file. You can use the *`CoverageData.run_infos()`* method to retrieve this string from a data file.

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

`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* for more information.

`plugins` (multi-string): a list of plugin package names. See *Plugins* for more information.

`source` (multi-string): a list of packages or directories, the source to measure during execution. See *Specifying source files* for details.

`timid` (boolean, default False): use a simpler but slower trace method. Try this if you get seemingly impossible results.

[paths]

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

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

The names of the entries are ignored, you may choose any name that you like. The value is a lists 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.

See *Combining data files* for more information.

[report]

Values common to many kinds of reporting.

`exclude_lines` (multi-string): a list of regular expressions. Any line of your source code that matches 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.

`fail_under` (integer): a target coverage percentage. If the total coverage measurement is under this value, then exit with a status code of 2.

`ignore_errors` (boolean, default False): ignore source code that can't be found.

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

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

`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.

`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%".

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

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

[html]

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

`directory` (string, default "htmlcov"): where to write the HTML report files.

`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.

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

[xml]

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

`output` (string, default "coverage.xml"): where to write the XML report.

`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.

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.

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 package names. If specified, only source inside these directories or packages will be measured. Specifying the source option also enables coverage.py to report on unexecuted 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, and files with unusual punctuation in their names will be skipped (they are assumed to be scratch files written by text editors).

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 typical shell syntax: `*` matches any number of characters and `?` matches a single character. Patterns that start with a wildcard character are used as-is, other patterns are interpreted relative to the current directory:

```
[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.

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`, `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.

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.

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.

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.

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:

```
[report]
exclude_lines = def __repr__
```

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

```
[report]
exclude_lines =
    pragma: no cover
    def __repr__
    if self.debug:
    if settings.DEBUG
    raise AssertionError
    raise NotImplementedError
    if 0:
    if __name__ == .__main__.:
```

Note that when using the `exclude_lines` option in a configuration file, you are taking control of the entire list of regexes, so you need to re-specify the default "pragma: no cover" match if you still want it to apply.

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

Excluding source files

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

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):        # line 1
2     if x:                    #      2
3         y = 10               #      3
4     return y                 #      4
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.

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 report produced by `coverage xml` also includes branch information, including separate statement and branch coverage percentages.

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!

Excluding code

If you have *excluded code*, 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.

Structurally partial branches

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

```
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.

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.

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.

Configuring Python for sub-process coverage

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.

Signal handlers and atexit

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 the atexit handler it registers.

For example if you send SIGTERM to end the sub-process, but your sub-process has never registered any SIGTERM handler, then a coverage file won't be written. See the atexit docs for details of when the handler isn't run.

Coverage.py API

The API to coverage.py is very simple, contained 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:

```
import coverage

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

# .. call your code ..

cov.stop()
cov.save()

cov.html_report()
```

The *coverage.CoverageData* class provides access to coverage data stored in coverage.py data files.

The Coverage class

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

Programmatic access to coverage.py.

To use:

```
from coverage import Coverage

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

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

data_file is the base name of the data file to use, defaulting to ".coverage". *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"). 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.

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. Valid strings are "greenlet", "eventlet", "gevent", "multiprocessing", or "thread" (the default).

New in version 4.0: The *concurrency* parameter.

analysis(*morf*)
Like *analysis2* but doesn't return excluded line numbers.

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.

annotate(*morfs=None*, *directory=None*, *ignore_errors=None*, *omit=None*, *include=None*)
Annotate a list of modules.

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.

clear_exclude(*which='exclude'*)
Clear the exclude list.

combine(*data_paths=None*)
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.

New in version 4.0: The *data_paths* parameter.

erase()
Erase previously-collected coverage data.

This removes the in-memory data collected in this session as well as discarding the data file.

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.

get_data()
Get the collected data and reset the collector.

Also warn about various problems collecting data.

Returns a *coverage.CoverageData*, the collected coverage data.

New in version 4.0.

get_exclude_list(*which='exclude'*)
Return a list of excluded regex patterns.

which indicates which list is desired. See *exclude()* for the lists that are available, and their meaning.

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.

New in version 4.0.

html_report(*morfs=None*, *directory=None*, *ignore_errors=None*, *omit=None*, *include=None*, *extra_css=None*, *title=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.

load()
Load previously-collected coverage data from the data file.

report(*morfs=None*, *show_missing=None*, *ignore_errors=None*, *file=None*, *omit=None*, *include=None*, *skip_covered=None*)
Write a summary report to *file*.

Each module in *morfs* is listed, with counts of statements, executed statements, missing statements, and a list of lines missed.

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.

Returns a float, the total percentage covered.

save()
Save the collected coverage data to the data file.

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 a Python value where appropriate. For example, use True for booleans, not the string `"True"`.

As an example, calling:

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

has the same effect as this configuration file:

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

New in version 4.0.

start()
Start measuring code coverage.

Coverage measurement actually occurs 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.

stop()
Stop measuring code coverage.

xml_report(*morfs=None*, *outfile=None*, *ignore_errors=None*, *omit=None*, *include=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.

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:

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

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

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

Returns the *Coverage* instance that was started, or None if it was not started by this call.

The CoverageData class

New in version 4.0.

class coverage.**CoverageData**(*debug=None*)
Manages collected coverage data, including file storage.

This class is the public supported API to the data 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 file format is not documented or guaranteed. It will change in the future, in possibly complicated ways. Do not read coverage.py data files directly. Use this API to avoid disruption.

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.
- **run information**: information about the program execution. This is written during "coverage run", and then accumulated during "coverage combine".

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.

To read a coverage.py data file, use *read_file()*, or *read_fileobj()* if you have an already-opened file. You can then access the line, arc, or file tracer data with *lines()*, *arcs()*, or *file_tracer()*. Run information is available with *run_infos()*.

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

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. The *add_run_info()* method adds key-value pairs to the run information.

To add a file without any measured data, use *touch_file()*.

You write to a named file with *write_file()*, or to an already opened file with *write_fileobj()*.

You can clear the data in memory with *erase()*. Two data collections can be combined by using *update()* on one *CoverageData*, passing it the other.

__init__(*debug=None*)
Create a CoverageData.

debug is a *DebugControl* object for writing debug messages.

add_arcs(*arc_data*)
Add measured arc data.

arc_data is a dictionary mapping file names to dictionaries:

```
{ filename: { (l1,l2): None, ... }, ...}
```

add_file_tracers(*file_tracers*)
Add per-file plugin information.

file_tracers is { filename: plugin_name, ... }

add_lines(*line_data*)
Add measured line data.

line_data is a dictionary mapping file names to dictionaries:

```
{ filename: { lineno: None, ... }, ...}
```

add_run_info(***kwargs*)
Add information about the run.

Keywords are arbitrary, and are stored in the run dictionary. Values must be JSON serializable. You may use this function more than once, but repeated keywords overwrite each other.

add_to_hash(*filename*, *hasher*)
Contribute *filename*'s data to the *hasher*.

hasher is a *coverage.misc.Hasher* instance to be updated with the file's data. It should only get the results data, not the run data.

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.

erase()
Erase the data in this object.

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.

has_arcs()
Does this data have arcs?

Arc data is only available if branch coverage was used during collection.

Returns a boolean.

line_counts(*fullpath=False*)
Return a dict summarizing the line coverage data.

Keys are based on the file names, and values are the number of executed lines. If *fullpath* is true, then the keys are the full pathnames of the files, otherwise they are the basenames of the files.

Returns a dict mapping file names to counts of lines.

lines(*filename*)
Get the list of lines executed for a 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.

measured_files()
A list of all files that had been measured.

read_file(*filename*)
Read the coverage data from *filename* into this object.

read_fileobj(*file_obj*)
Read the coverage data from the given file object.

Should only be used on an empty CoverageData object.

run_infos()
Return the list of dicts of run information.

For data collected during a single run, this will be a one-element list. If data has been combined, there will be one element for each original data file.

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

update(*other_data*, *aliases=None*)
Update this data with data from another *CoverageData*.

If *aliases* is provided, it's a *PathAliases* object that is used to re-map paths to match the local machine's.

write_file(*filename*)
Write the coverage data to *filename*.

write_fileobj(*file_obj*)
Write the coverage data to *file_obj*.

Plugin classes

New in version 4.0.

The CoveragePlugin class

class coverage.**CoveragePlugin**
Base class for coverage.py plugins.

To write a coverage.py plugin, 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.

Currently the only plugin type is a file tracer, for implementing measurement support for non-Python files. File tracer plugins implement the *file_tracer()* method to claim files and the *file_reporter()* method to report on those files.

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

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

To register your plugin, define a function called *coverage_init* in your module:

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

You use the *reg* parameter passed to your *coverage_init* function to register your plugin object. It has one method, *add_file_tracer*, which takes a newly created instance of your plugin.

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

file_tracer(*filename*)
Get a *FileTracer* object for a file.

Every Python source file is offered to the plugin to give it a chance to take responsibility for tracing the file. If your plugin can handle the file, then return a *FileTracer* object. Otherwise return None.

There is no way to register your plugin for particular files. Instead, this method is invoked for all files, and the plugin 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 plugin.

The file name will be a Python file being executed. There are two broad categories of behavior for a plugin, depending on the kind of files your plugin supports:

- Static file names: each of your original source files has been converted into a distinct Python file. Your plugin 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 plugin 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 plugin cannot trace this file.

file_reporter(*filename*)
Get the *FileReporter* class to use for a file.

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*.

sys_info()
Get a list of information useful for debugging.

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

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

The FileTracer class

class coverage.**FileTracer**
Support needed for files during the execution phase.

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 plugin 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.

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.

dynamic_source_filename(*filename*, *frame*)
Get a dynamically computed source file name.

Some plugins 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.

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.

The FileReporter class

class coverage.**FileReporter**(*filename*)
Support needed for files during the analysis and reporting phases.

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

FileReporter objects should only be created in the `CoveragePlugin.file_reporter()` method.

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

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.

source()
Get the source for the file.

Returns a Unicode string.

The base implementation simply reads the *self.filename* file and decodes it as UTF8. Override this method if your file isn't readable as a text file, or if you need other encoding support.

lines()
Get the executable lines in this file.

Your plugin 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.

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

Your plugin 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.

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 plugin 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.

arcs()
Get the executable arcs in this file.

To support branch coverage, your plugin 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.

no_branch_lines()
Get the lines excused from branch coverage in this file.

Your plugin 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.

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.

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 plugin...

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}".

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
- `'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'`.

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`.

Let's look at each phase in more detail.

Execution

At the heart of the execution phase is a Python 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.

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 slowdown. 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.

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.

At the end of execution, coverage.py writes the data it collected to a data file, usually named `.coverage`. This is a JSON-based file containing all of the recorded file names and line numbers executed.

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.

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, 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.

Plugins

Plugins interact with these phases.

Plugins

Coverage.py's behavior can be extended with third-party plugins. A plugin is a separately installed Python class that you register in your .coveragerc. Plugins can be used to implement coverage measurement for non-Python files.

Plugins are only supported with the *C extension*, which must be installed for plugins to work.

Information about using plugins is on this page. To write a plugin, see *Plugin classes*.

New in version 4.0.

Using plugins

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

1. Install the plugin's package as you would any other Python package:

```
pip install something
```

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

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

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

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

   Check the documentation for the plugin to see if it takes any options, and what they are.

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.

Available plugins

Some coverage.py plugins you might find useful:

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

Contributing to coverage.py

I welcome contributions to coverage.py. Over the years, dozens of people have provided patches of various sizes to add features or fix bugs. 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.

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.

Getting the code

The coverage.py code is hosted on a Mercurial repository at https://bitbucket.org/ned/coveragepy. To get a working environment, follow these steps:

1. (Optional, but recommended) Create a virtualenv to work in, and activate it.
2. Clone the repo:

```
$ hg clone https://bitbucket.org/ned/coveragepy
$ cd coveragepy
```

3. Install the requirements:

```
$ pip install -r requirements/dev.pip
```

4. Install a number of versions of Python. Coverage.py supports a wide range of Python versions. The more you can test with, the more easily your code can be used as-is. If you only have one version, that's OK too, but may mean more work integrating your contribution.

Running the tests

The tests are written as standard unittest-style tests, and are run with tox:

```
$ tox
py27 create: /Users/ned/coverage/trunk/.tox/py27
py27 installdeps: nose==1.3.7, mock==1.3.0, PyContracts==1.7.6, gevent==1.0.2, eventlet==0.17.4, gree
py27 develop-inst: /Users/ned/coverage/trunk
py27 installed: -f /Users/ned/Downloads/local_pypi,-e hg+ssh://hg@bitbucket.org/ned/coveragepy@22fe9a
py27 runtests: PYTHONHASHSEED='1294330776'
py27 runtests: commands[0] | python setup.py --quiet clean develop
py27 runtests: commands[1] | python igor.py zip_mods install_egg remove_extension
py27 runtests: commands[2] | python igor.py test_with_tracer py
=== CPython 2.7.10 with Python tracer (.tox/py27/bin/python) ===
.....................................................................(etc)
----------------------------------------------------------------------
Ran 592 tests in 65.524s

OK (SKIP=20)
py27 runtests: commands[3] | python setup.py --quiet build_ext --inplace
py27 runtests: commands[4] | python igor.py test_with_tracer c
=== CPython 2.7.10 with C tracer (.tox/py27/bin/python) ===
.....................................................................(etc)
----------------------------------------------------------------------
Ran 592 tests in 69.635s

OK (SKIP=4)
py33 create: /Users/ned/coverage/trunk/.tox/py33
py33 installdeps: nose==1.3.7, mock==1.3.0, PyContracts==1.7.6, greenlet==0.4.7
py33 develop-inst: /Users/ned/coverage/trunk
py33 installed: -f /Users/ned/Downloads/local_pypi,-e hg+ssh://hg@bitbucket.org/ned/coveragepy@22fe9a
```

```
py33 runtests: PYTHONHASHSEED='1294330776'
py33 runtests: commands[0] | python setup.py --quiet clean develop
py33 runtests: commands[1] | python igor.py zip_mods install_egg remove_extension
py33 runtests: commands[2] | python igor.py test_with_tracer py
=== CPython 3.3.6 with Python tracer (.tox/py33/bin/python) ===
.............................................S.............................(etc)
----------------------------------------------------------------------
Ran 592 tests in 73.007s

OK (SKIP=22)
py33 runtests: commands[3] | python setup.py --quiet build_ext --inplace
py33 runtests: commands[4] | python igor.py test_with_tracer c
=== CPython 3.3.6 with C tracer (.tox/py33/bin/python) ===
.............................................S.............................(etc)
----------------------------------------------------------------------
Ran 592 tests in 72.071s

OK (SKIP=5)
(and so on...)
```

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

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

```
$ tox -e py27,py33
```

To run just a few tests, you can use nose test selector syntax:

```
$ tox tests.test_misc:SetupPyTest.test_metadata
```

This looks in *tests/test_misc.py* to find the *SetupPyTest* class, and runs the *test_metadata* test method.

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

Lint, etc

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

```
$ make lint
pylint coverage setup.py tests
python -m tabnanny coverage setup.py tests
python igor.py check_eol
```

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.

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.

Contributing

When you are ready to contribute a change, any way you can get it to me is probably fine. A pull request on Bitbucket is great, but a simple diff or patch is great too.

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.

Things that don't work

There are a number of popular modules, packages, and libraries 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.

Things that require –timid

Some packages interfere with coverage measurement, but you might be able to make it work by using the `--timid` command-line switch, or the `[run] timid=True` configuration option.

- DecoratorTools, or any package which uses it, notably TurboGears. DecoratorTools fiddles with the trace function. You will need to use `--timid`.

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.

FAQ and other help

Frequently asked questions

Q: I use nose to run my tests, and its cover plugin doesn't let me create HTML or XML reports. What should I do?

First run your tests and collect coverage data with nose and its plugin. This will write coverage data into a .coverage file. Then run coverage.py from the *command line* to create the reports you need from that data.

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 you are using the `-x` command line action, it doesn't erase first by default. Switch to the `coverage run` command, or use the `-e` switch to erase all data before starting the next run.

Q: Why do the bodies of functions (or classes) show as executed, but the def lines do not?

This happens because coverage.py is 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.

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: 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 `tracer: CTracer` or `tracer: PyTracer`. If it says `PyTracer` 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: Does coverage.py work on Python 3.x?

Yes, Python 3 is fully supported.

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 Testing In Python mailing list.

Bug reports are gladly accepted at the Bitbucket issue tracker.

Announcements of new coverage.py releases are sent to the coveragepy-announce mailing list.

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

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.

Major change history for coverage.py

These are the major changes for coverage.py. For a more complete change history, see the CHANGES.rst file in the source tree.

Version 4.1b — 2016-05-21

- 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.
  - 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.
  - 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.
    * 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.
- 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.
- The HTML report has a few changes:
  - 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.
- The XML report now produces correct package names for modules found in directories specified with `source=`. Fixes issue 465.
- `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)
- 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.

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, Arcady 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.

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.

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.

Version 4.0 — 2015-09-20

Backward incompatibilities:

- Python versions supported are now:

  - CPython 2.6, 2.7, 3.3, 3.4 and 3.5
  - PyPy2 2.4, 2.6
  - PyPy3 2.4

- The original command line switches (`-x` to run a program, etc) are no longer supported.
- The `COVERAGE_OPTIONS` environment variable is no longer supported. It was a hack for `--timid` before configuration files were available.
- 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.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.
- Coverage.py is now licensed under the Apache 2.0 license. See NOTICE.txt for details.
- Coverage.py kits no longer include tests and docs. If you were using them, get in touch and let me know how.

Major new features:

- Plugins: third parties can write plugins to add file support for non-Python files, such as web application templating engines, or languages that compile down to Python. See *Plugins* for how to use plugins, and *Plugin classes* for details of how to write them. A plugin for measuring Django template coverage is available: django_coverage_plugin
- Gevent, eventlet, and greenlet are now supported. The `[run] concurrency` setting, or the `--concurrency` command line switch, 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.
- The data storage has been re-written, using JSON instead of pickle. The *`CoverageData`* class is a new supported API to the contents of the data file. Data files from older versions of coverage.py can be converted to the new format with `python -m coverage.pickle2json`.
- 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. Thanks, Eduardo Schettino. Currently, this does not work on Windows.

New features:

- 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.
- 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.
- A new option: `coverage report --skip-covered` (or `[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.
- You can now specify the `--fail-under` option in the `.coveragerc` file as the `[report] fail_under` option.
- The `report -m` command now shows missing branches when reporting on branch coverage. Thanks, Steve Leonard.
- The `coverage combine` command now accepts any number of directories or files as arguments, and will combine all the data from them. This means you don't have to copy the files to one directory before combining. Thanks, Christine Lytwynec.

- 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.
- 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.
- The COVERAGE_DEBUG environment variable can be used to set the `[run] debug` configuration option to control what internal operations are logged.
- A new version identifier is available, *coverage.version_info*, a plain tuple of values similar to sys.version_info.

Improvements:

- Coverage.py now always adds the current directory to sys.path, so that plugins can import files in the current directory.
- Coverage.py now accepts a directory name for `coverage run` and will run a `__main__.py` found there, just like Python will. Thanks, Dmitry Trofimov.
- The `--debug` switch can now be used on any command.
- 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.
- Missing branches in the HTML report now have a bit more information in the right-hand annotations. Hopefully this will make their meaning clearer.
- The XML report now contains a <source> element. Thanks Stan Hu.
- 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.
- The XML report now reports each directory as a package again. This was a bad regression, I apologize.
- In parallel mode, `coverage erase` will now delete all of the data files.
- A new warning is possible, if a desired file isn't measured because it was imported before coverage.py was started.
- 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.
- Unrecognized configuration options will now print an error message and stop coverage.py. This should help prevent configuration mistakes from passing silently.

API changes:

- The class defined in the coverage module is now called `Coverage` instead of `coverage`, though the old name still works, for backward compatibility.
- You can now programmatically adjust the configuration of coverage.py by calling *`Coverage.set_option()`* after construction. *`Coverage.get_option()`* reads the configuration values.
- 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.

Bug fixes:

- 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.

- The `fail-under` value is now rounded the same as reported results, preventing paradoxical results, fixing issue 284.
- Branch coverage couldn't properly handle certain extremely long files. This is now fixed, closing 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.
- Files with incorrect encoding declaration comments are no longer ignored by the reporting commands.
- Empty files are now reported as 100% covered in the XML report, not 0% covered.
- The XML report will now create the output directory if need be. Thanks, Chris Rose.
- HTML reports no longer raise UnicodeDecodeError if a Python file has undecodable characters.
- The annotate command will now annotate all files, not just ones relative to the current directory.

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.

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. For details, see *the run –debug options*.
- 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 unexecuted, 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 ZeroDivideError, 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.

Version 3.6 — 2013-01-05

Features:

- 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.

Packaging:

- The C extension is optionally compiled using a different more widely-used technique, taking another stab at fixing issue 80 once and for all.
- 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.

Docs:

- Added a page to the docs about contributing to coverage.py, closing issue 171.
- Added a page to the docs about troublesome situations, closing issue 226.
- 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 some info to the TODO file, closing issue 227.

Fixes:

- 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.
- Embarrassingly, the *[xml] output=* setting in the .coveragerc file simply didn't work. Now it does.
- 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.
- The XML report now consistently uses file names for the filename 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 unexecuted 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.
- 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.
- 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.

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.

Version 3.5.2 — 2012-05-04

- 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.
- 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.

Version 3.5.1 — 2011-09-23

- 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.

Version 3.5 — 2011-06-29

HTML reporting:

- 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.
- 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.

Running Python files

- 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 detail: the current directory inserted into `sys.path` was relative rather than absolute. This is now fixed.

- 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.
- 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.

Reporting

- 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 `--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.
- Syntax errors in supposed Python files can now be ignored during reporting with the `-i` switch just like other source errors. Closes issue 115.

Version 3.4 — 2010-09-19

Controlling source:

- 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. The run command also 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.
- 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.

Reporting:

- Completely unexecuted 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.
- 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.
- 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 XML report output now properly includes a percentage for branch coverage, fixing issue 65 and issue 81, and the report is sorted by package name, fixing issue 88.
- The XML report is now sorted by package name, fixing issue 88.
- The precision of reported coverage percentages can be set with the `[report] precision` config file setting. Completes issue 16.
- 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.

Execution and measurement:

- 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.
- Doctest text files are no longer recorded in the coverage data, since they can't be reported anyway. Fixes issue 52 and issue 61.
- 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.
- Programs that exited with `sys.exit()` with no argument weren't handled properly, producing a coverage.py stack trace. This is now fixed.
- 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.
- 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.
- Coverage.py can now be run as a module: `python -m coverage`. Thanks, Brett Cannon.

Version 3.3.1 — 2010-03-06

- Using `parallel=True` in a .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.

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 `--rcfile=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`.
- 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. Also, 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.
- Fixed issue 39 and issue 47.

Version 3.2 — 2009-12-05

- Branch coverage: coverage.py can tell you which branches didn't have both (or all) choices executed, even where the choice doesn't affect which lines were executed. See *Branch coverage measurement* for more details.
- The table of contents in the HTML report is now sortable: click the headers on any column. The sorting is persisted so that subsequent reports are sorted as you wish. Thanks, Chris Adams.
- XML reporting has file paths that let Cobertura find the source code, fixing issue 21.
- The `--omit` option now works much better than before, fixing issue 14 and issue 33. Thanks, Danek Duvall.
- Added a `--version` option on the command line.
- Program execution under coverage.py is 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.
- Fixed some problems syntax coloring sources with line continuations and source with tabs: issue 30 and issue 31.

Version 3.1 — 2009-10-04

- 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 now display syntax-colored Python source.
- Added a `coverage debug` command for getting diagnostic information about the coverage.py installation.
- Source code can now be read from eggs. Thanks, Ross Lawley. Fixes issue 25.

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 a bizarre problem involving pyexpat, whereby lines following XML parser invocations could be overlooked.
- 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` plugin.
- 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.

Version 3.0 — 2009-06-13

- Coverage.py is now a package rather than a module. Functionality has been split into classes.
- HTML reports and annotation of source files: use the new `-b` (browser) switch. Thanks to George Song for code, inspiration and guidance.
- 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.
- The minimum supported Python version is 2.3.
- 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.
- 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.
- API changes:
  - 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.
  - coverage.annotate_file is no longer available.
  - Removed the undocumented cache_file argument to coverage.usecache().

C

coverage, 16

Symbols

__init__() (coverage.Coverage method), 24
__init__() (coverage.CoverageData method), 29

A

add_arcs() (coverage.CoverageData method), 29
add_file_tracers() (coverage.CoverageData method), 29
add_lines() (coverage.CoverageData method), 29
add_run_info() (coverage.CoverageData method), 29
add_to_hash() (coverage.CoverageData method), 29
analysis() (coverage.Coverage method), 25
analysis2() (coverage.Coverage method), 25
annotate() (coverage.Coverage method), 25
arcs() (coverage.CoverageData method), 29
arcs() (coverage.FileReporter method), 33

C

clear_exclude() (coverage.Coverage method), 25
combine() (coverage.Coverage method), 25
Coverage (class in coverage), 24
coverage (module), 16, 24, 28, 30, 40
CoverageData (class in coverage), 28
CoveragePlugin (class in coverage), 31

D

dynamic_source_filename() (coverage.FileTracer method), 32

E

erase() (coverage.Coverage method), 26
erase() (coverage.CoverageData method), 29
exclude() (coverage.Coverage method), 26
excluded_lines() (coverage.FileReporter method), 33
exit_counts() (coverage.FileReporter method), 34

F

file_reporter() (coverage.CoveragePlugin method), 31
file_tracer() (coverage.CoverageData method), 29
file_tracer() (coverage.CoveragePlugin method), 31
FileReporter (class in coverage), 32
FileTracer (class in coverage), 32

G

get_data() (coverage.Coverage method), 26
get_exclude_list() (coverage.Coverage method), 26
get_option() (coverage.Coverage method), 26

H

has_arcs() (coverage.CoverageData method), 29
has_dynamic_source_filename() (coverage.FileTracer method), 32
html_report() (coverage.Coverage method), 26

L

line_counts() (coverage.CoverageData method), 30
line_number_range() (coverage.FileTracer method), 32
lines() (coverage.CoverageData method), 30
lines() (coverage.FileReporter method), 33
load() (coverage.Coverage method), 26

M

measured_files() (coverage.CoverageData method), 30
missing_arc_description() (coverage.FileReporter method), 34

N

no_branch_lines() (coverage.FileReporter method), 33

P

process_startup() (in module coverage), 27

R

read_file() (coverage.CoverageData method), 30
read_fileobj() (coverage.CoverageData method), 30
relative_filename() (coverage.FileReporter method), 33
report() (coverage.Coverage method), 26
run_infos() (coverage.CoverageData method), 30

S

save() (coverage.Coverage method), 27

set_option() (coverage.Coverage method), 27
source() (coverage.FileReporter method), 33
source_filename() (coverage.FileTracer method), 32
source_token_lines() (coverage.FileReporter method), 34
start() (coverage.Coverage method), 27
stop() (coverage.Coverage method), 27
sys_info() (coverage.CoveragePlugin method), 31

T

touch_file() (coverage.CoverageData method), 30
translate_arcs() (coverage.FileReporter method), 34
translate_lines() (coverage.FileReporter method), 33

U

update() (coverage.CoverageData method), 30

W

write_file() (coverage.CoverageData method), 30
write_fileobj() (coverage.CoverageData method), 30

X

xml_report() (coverage.Coverage method), 27