27.6. `trace` — Trace or track Python statement execution

Source code: Lib/trace.py

The `trace` module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

See also:

Coverage.py
A popular third-party coverage tool that provides HTML output along with advanced features such as branch coverage.

27.6.1. Command-Line Usage

The `trace` module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

`--help`
Display usage and exit.

`--version`
Display the version of the module and exit.

27.6.1.1. Main options

At least one of the following options must be specified when invoking `trace`. The `--listfuncs` option is mutually exclusive with the `--trace` and `--count` options. When `--listfuncs` is provided, neither `--count` nor `--trace` are accepted, and vice versa.

`-c`, `--count`

Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also `--coverdir`, `--file` and `--no-report` below.

`-t`, `--trace`
Display lines as they are executed.

`-l`, `--listfuncs`
Display the functions executed by running the program.

`-r`, `--report`
Produce an annotated list from an earlier program run that used the `--count` and `--file` option. This does not execute any code.

`-T`, `--trackcalls`
Display the calling relationships exposed by running the program.

27.6.1.2. Modifiers

`-f`, `--file``=<file>`
Name of a file to accumulate counts over several tracing runs. Should be used with the `--count` option.

`-C`, `--coverdir``=<dir>`
Directory where the report files go. The coverage report for `package.module` is written to file *`dir/package/module`*`.cover`.

`-m`, `--missing`
When generating annotated listings, mark lines which were not executed with `>>>>>>`.

`-s`, `--summary`
When using `--count` or `--report`, write a brief summary to stdout for each file processed.

`-R`, `--no-report`
Do not generate annotated listings. This is useful if you intend to make several runs with `--count`, and then produce a single set of annotated listings at the end.

`-g`, `--timing`
Prefix each line with the time since the program started. Only used while tracing.

27.6.1.3. Filters

These options may be repeated multiple times.

`--ignore-module``=<mod>`
: Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

`--ignore-dir``=<dir>`
: Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by `os.pathsep`.

27.6.2. Programmatic Interface

class `trace.`**`Trace`**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)
: Create an object to trace execution of a single statement or expression. All parameters are optional. *count* enables counting of line numbers. *trace* enables line execution tracing. *countfuncs* enables listing of the functions called during the run. *countcallers* enables call relationship tracking. *ignoremods* is a list of modules or packages to ignore. *ignoredirs* is a list of directories whose modules or packages should be ignored. *infile* is the name of the file from which to read stored count information. *outfile* is the name of the file in which to write updated count information. *timing* enables a timestamp relative to when tracing was started to be displayed.

`run`(*cmd*)
: Execute the command and gather statistics from the execution with the current tracing parameters. *cmd* must be a string or code object, suitable for passing into `exec()`.

`runctx`(*cmd*, *globals=None*, *locals=None*)
: Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, *globals* and *locals* default to empty dictionaries.

`runfunc`(*func*, **args*, ***kwds*)
: Call *func* with the given arguments under control of the `Trace` object with the current tracing parameters.

`results`()

Return a `CoverageResults` object that contains the cumulative results of all previous calls to `run`, `runctx` and `runfunc` for the given `Trace` instance. Does not reset the accumulated trace results.

class `trace.`**CoverageResults**

A container for coverage results, created by `Trace.results()`. Should not be created directly by the user.

update(*other*)

Merge in data from another `CoverageResults` object.

write_results(*show_missing=True*, *summary=False*, *coverdir=None*)

Write coverage results. Set *show_missing* to show lines that had no hits. Set *summary* to include in the output the coverage summary per module. *coverdir* specifies the directory into which the coverage result files will be output. If `None`, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```