TidyPy Documentation

Release 0.8.0

Jason Simeone

Sep 15, 2021

Contents:

CHAPTER 1

TidyPy

Contents

1.1 Overview

TidyPy is a tool that encapsulates a number of other static analysis tools and makes it easy to configure, execute, and review their results.

1.2 Features

- It's a consolidated tool for performing static analysis on an entire Python project – not just your `*.py` source files. In addition to executing a number of different *tools* on your code, it can also check your YAML, JSON, PO, POT, and RST files.
- Rather than putting yet another specialized configuration file in your project, TidyPy uses the `pyproject.toml` file defined by PEP 518. All options for all the tools TidyPy uses are declared in one place, rather than requiring that you configure each tool in a different way.
- Honors the pseudo-standard `# noqa` comment in your Python source to easily ignore issues reported by any tool.
- Includes a number of *integrations* so you can use it with your favorite toolchain.
- Includes a variety of *reporters* that allow you to view or use the results of TidyPy's analysis in whatever way works best for you.
- Provides a simple API for you to implement your own tool or reporter and include it in the analysis of your project.
- Supports both Python 2 and 3, as well as PyPy. Even runs on Windows.

1.3 Usage

When TidyPy is installed (`pip install tidypy`), the `tidypy` command should become available in your environment:

```
$ tidypy --help
Usage: tidypy [OPTIONS] COMMAND [ARGS]...

  A tool that executes several static analysis tools upon a Python project
  and aggregates the results.

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  check               Executes the tools upon the project files.
  default-config      Outputs a default configuration that can be used to
                      bootstrap your own configuration file.
  extensions          Outputs a listing of all available TidyPy extensions.
  install-vcs         Installs TidyPy as a pre-commit hook into the specified
                      VCS.
  list-codes          Outputs a listing of all known issue codes that tools
                      may report.
  purge-config-cache  Deletes the cache of configurations retrieved from
                      outside the primary configuration.
```

(continues on next page)

(continued from previous page)

```
  remove-vcs          Removes the TidyPy pre-commit hook from the specified
                      VCS.
```

To have TidyPy analyze your project, use the `check` subcommand:

```
$ tidypy check --help
Usage: tidypy check [OPTIONS] [PATH]

  Executes the tools upon the project files.

  Accepts one argument, which is the path to the base of the Python project.
  If not specified, defaults to the current working directory.

Options:
  -x, --exclude REGEX             Specifies a regular expression matched
                                  against paths that you want to exclude from
                                  the examination. Can be specified multiple
                                  times. Overrides the expressions specified
                                  in the configuration file.
  -t, --tool␣
↪[bandit|dlint|eradicate|jsonlint|manifest|mccabe|polint|pycodestyle|pydiatra|pydocstyle|pyflakes|py
                                  Specifies the name of a tool to use during
                                  the examination. Can be specified multiple
                                  times. Overrides the configuration file.
  -r, --report [console,csv,custom,json,null,pycodestyle,pylint,pylint-parseable,toml,
↪yaml][:filename]
                                  Specifies the name of a report to execute
                                  after the examination. Can specify an
                                  optional output file name using the form -r
                                  report:filename. If filename is unset, the
                                  report will be written on stdout. Can be
                                  specified multiple times. Overrides the
                                  configuration file.
  -c, --config FILENAME           Specifies the path to the TidyPy
                                  configuration file to use instead of the
                                  configuration found in the project's
                                  pyproject.toml.
  --workers NUM_WORKERS           The number of workers to use to concurrently
                                  execute the tools. Overrides the
                                  configuration file.
  --disable-merge                 Disable the merging of issues from various
                                  tools when TidyPy considers them equivalent.
                                  Overrides the configuration file.
  --disable-progress              Disable the display of the progress bar.
  --disable-noqa                  Disable the ability to ignore issues using
                                  the "# noqa" comment in Python files.
  --disable-config-cache          Disable the use of the cache when retrieving
                                  configurations referenced by the "extends"
                                  option.
  --help                          Show this message and exit.
```

If you need to generate a skeleton configuration file with the default options, use the `default-config` subcommand:

```
$ tidypy default-config --help
Usage: tidypy default-config [OPTIONS]
```

(continues on next page)

(continued from previous page)

```
  Outputs a default configuration that can be used to bootstrap your own
  configuration file.

Options:
  --pyproject  Output the config so that it can be used in a pyproject.toml
               file.
  --help       Show this message and exit.
```

If you'd like to see a list of the possible issue codes that could be returned, use the `list-codes` subcommand:

```
$ tidypy list-codes --help
Usage: tidypy list-codes [OPTIONS]

  Outputs a listing of all known issue codes that tools may report.

Options:
  -t, --tool␣
↪[bandit|dlint|eradicate|jsonlint|manifest|mccabe|polint|pycodestyle|pydiatra|pydocstyle|pyflakes|py
                                  Specifies the name of a tool whose codes
                                  should be output. If not specified, defaults
                                  to all tools.
  -f, --format [toml|json|yaml|csv]
                                  Specifies the format in which the tools
                                  should be output. If not specified, defaults
                                  to TOML.
  --help                          Show this message and exit.
```

If you want to install or remove TidyPy as a pre-commit hook in your project's VCS, use the `install-vcs`/`remove-vcs` subcommands:

```
$ tidypy install-vcs --help
Usage: tidypy install-vcs [OPTIONS] VCS [PATH]

  Installs TidyPy as a pre-commit hook into the specified VCS.

  Accepts two arguments:

    VCS: The version control system to install the hook into. Choose from:
    git, hg

    PATH: The path to the base of the repository to install the hook into.
    If not specified, defaults to the current working directory.

Options:
  --strict  Whether or not the hook should prevent the commit if TidyPy finds
            issues.
  --help    Show this message and exit.

$ tidypy remove-vcs --help
Usage: tidypy remove-vcs [OPTIONS] VCS [PATH]

  Removes the TidyPy pre-commit hook from the specified VCS.

  Accepts two arguments:
```

(continues on next page)

(continued from previous page)

```
    VCS: The version control system to remove the hook from. Choose from:
    git, hg

    PATH: The path to the base of the repository to remove the hook from. If
    not specified, defaults to the current working directory.

Options:
  --help  Show this message and exit.
```

If you'd like to enable bash completion for TidyPy, run the following in your shell (or put it in your bash startup scripts):

```
$ eval "$(_TIDYPY_COMPLETE=source tidypy)"
```

1.4 Docker

If you don't want to install TidyPy locally on your system or in your virtualenv, you can use the published Docker image:

```
$ docker run --rm --tty --volume=`pwd`:/project tidypy/tidypy
```

The command above will run `tidypy check` on the contents of the current directory. If you want to run it on a different directory, then change the `` `pwd` `` to whatever path you need (the goal being to mount your project directory to the container's `/project` volume).

Running TidyPy in this manner has a few limitiations, mostly around the fact that since TidyPy is running in its own, isolated Python environment, tools like pylint won't be able to introspect the packages your project installed locally, so it may report false positives around "import-error", "no-name-in-module", "no-member", etc.

If you want to run a command other than `check`, just pass that along when you invoke docker:

```
$ docker run --rm --tty --volume=`pwd`:/project tidypy/tidypy tidypy list-codes
```

1.5 Configuration

TODO

1.6 Ignoring Issues

In addition to ignoring entire files, tools, or specific issue types from tools via your configuration file, you can also use comments in your Python source files to ignore issues on specific lines. Some tools have their own built-in support and notation for doing this:

- pylint will respect comments that look like: `# pylint`
- bandit will respect comments that look like: `# nosec`
- pycodestyle will respect comments that look like: `# noqa`
- pydocstyle will also respect comments that look like: `# noqa`
- detect-secrets will respect comments that look like: `# pragma: whitelist secret`

TidyPy goes beyond these tool-specific flags to implement `# noqa` on a global scale for Python source files. It will ignore issues for lines that have the `# noqa` comment, regardless of what tools raise the issues. If you only want to ignore a particular type of issue on a line, you can use syntax like the following:

```
# noqa: CODE1,CODE2
```

Or, if a particular code is used in multiple tools, you can specify the exact tool in the comment:

```
# noqa: pycodestyle:CODE1,pylint:CODE2
```

Or, if you want to ignore any issue a specific tool raises on a line, you can specify the tool:

```
# noqa: @pycodestyle,@pylint
```

You can, of course, mix and match all three notations in a single comment if you need to:

```
# noqa: CODE1,pylint:CODE2,@pycodestyle
```

You can disable TidyPy's NOQA behavior by specifying the `--disable-noqa` option on the command line, or by setting the `noqa` option to `false` in your configuration file. A caveat, though: currently pycodestyle and pydocstyle do not respect this option and will always honor any `# noqa` comments they find.

1.7 Included Tools

Out of the box, TidyPy includes support for a number of tools:

pylint Pylint is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells.

pycodestyle pycodestyle is a tool to check your Python code against some of the style conventions in PEP 8.

pydocstyle pydocstyle is a static analysis tool for checking compliance with Python docstring conventions (e.g., PEP 257).

pyroma Pyroma tests your project's packaging friendliness.

vulture Vulture finds unused code in Python programs.

bandit Bandit is a security linter for Python source code.

eradicate Eradicate finds commented-out code in Python files.

pyflakes Pyflakes is a simple program which checks Python source files for errors.

mccabe Ned Batchelder's script to check the McCabe cyclomatic complexity of Python code.

jsonlint A part of the demjson package, this tool validates your JSON documents for strict conformance to the JSON specification, and to detect potential data portability issues.

yamllint The yamllint tool, as its name implies, is a linter for YAML files.

rstlint The restructuredtext-lint tool, as its name implies, is a linter for reStructuredText files.

polint A part of the dennis package, this tool lints PO and POT files for problems.

manifest Uses the check-manifest script to detect discrepancies or problems with your project's MANIFEST.in file.

pydiatra pydiatra is yet another static checker for Python code.

secrets The detect-secrets tool attempts to find secrets (keys, passwords, etc) within a code base.

dlint Dlint is a tool for encouraging best coding practices and helping ensure we're writing secure Python code.

1.8 Included Reporters

TidyPy includes a number of different methods to present and/or export the results of the analysis of a project. Out of the box, it provides the following:

console The default reporter. Prints a colored report to the console that groups issues by the file they were found in.

pylint Prints a report to the console that is in the same format as Pylint's default output.

pylint-parseable Prints a report to the console that is in roughly the same format as Pylint's "parseable" output.

pycodestyle Prints a report to the console that is in the same format as pycodestyle's default output.

json Generates a JSON-serialized object that contains the results of the analysis.

yaml Generates a YAML-serialized object that contains the results of the analysis.

toml Generates a TOML-serialized object that contains the results of the analysis.

csv Generates a set of CSV records that contains the results of the analysis.

custom Prints ouput to the console that is in the format defined by a template string specified in the project configuration. The template string is expected to be one allowed by the str.format() Python method. It will receive the following arguments: `filename`, `line`, `character`, `tool`, `code`, `message`.

1.9 Included Integrations

TidyPy includes a handful of plugins/integrations that hook it into other tools.

pytest TidyPy can be run during execution of your pytest test suite. To enable it, you need to specify `--tidypy` on the command line when you run pytest, or include it as part of the `addopts` property in your pytest config.

nose TidyPy can be run during execution of your nose test suite. To enable it, you can either specify `--with-tidypy` on the command line when you run nose, or set the `with-tidypy` property to `1` in your `setup.cfg`.

pbbt TidyPy can be included in your PBBT scripts using the `tidypy` test. To enable it, you can either specify `--extend=tidypy.plugin.pbbt` on the command line when you run PBBT, or set the `extend` property in your `setup.cfg` or `pbbt.yaml` to `tidypy.plugin.pbbt`.

1.10 Extending TidyPy

A simple interface exists for extending TidyPy to include more and different tools and reporters. To add a tool, create a class that extends tidypy.Tool, and in your `setup.py`, declare an `entry_point` for `tidypy.tools` that points to your class:

```
entry_points={
    'tidypy.tools': [
        'mycooltool = path.to.model:MyCoolToolClassName',
    ],
}
```

To add a reporter, the process is nearly identical, except that you extend tidypy.Report and declare an `entry_point` for `tidypy.reports`.

1.11 FAQs

Aren't there already tools like this? Yup. There's prospector, pylama, flake8, and ciocheck just to name a few. But, as is customary in the world of software development, if the wheel isn't as round as you'd like it to be, you must spend countless hours to reinvent it. I've tried a number of these tools (and even contributed to some), but in the end, I always found something lacking or annoying. Thus, TidyPy was born.

How do I run TidyPy on a single file? The short answer is, you don't (at the moment, anyway). It wasn't designed with that use case in mind. TidyPy was built to analyze a whole project, and show you everything.

I tried TidyPy out on my project and it reported hundreds/thousands of issues. My ego is now bruised. Yea, that happens. The philosophy I chose to follow with this tool is that I didn't want it to hide anything from me. I wanted its default behavior to execute every tool in its suite using their most obnoxious setting. Then, when I can see the full scope of damage, I can then decide to disable specific tools or issues via a project-level configuration. I figured if someone took the time to implement a check for a particular issue, they must think it has some value. If my tooling hides that from me by default, then I won't be able to gain any benefits from it.

In general, I don't recommend starting to use linters or other sorts of static analyzers when you think you're "done". You should incorporate them into your workflow right at the beginning of a project – just as you would (or should) your unit tests. That way you find things early and learn from them (or disable them). It's much less daunting a task to deal with when you address them incrementally.

1.12 Contributing

Contributions are most welcome. Particularly if they're bug fixes! To hack on this code, simply clone it, and then run `make setup`. This will create a virtualenv with all the tools you'll need. The `Makefile` also has a `test` target for running the pytest suite, and a `lint` target for running TidyPy on itself.

1.13 License

CHAPTER 2

API Reference

tidypy.**execute_tools**(*config*, *path*, *progress=None*)

Executes the suite of TidyPy tools upon the project and returns the issues that are found.

Parameters

- **config** (*dict*) – the TidyPy configuration to use
- **path** (*str*) – that path to the project to analyze
- **progress** (tidypy.Progress) – the progress reporter object that will receive callbacks during the execution of the tool suite. If not specified, not progress notifications will occur.

Return type *tidypy.Collector*

tidypy.**execute_reports**(*config*, *path*, *collector*, *on_report_finish=None*, *output_file=None*)

Executes the configured suite of issue reports.

Parameters

- **config** (*dict*) – the TidyPy configuration to use
- **path** (*str*) – that path to the project that was analyzed
- **collector** (tidypy.Collector) – the issues to report

tidypy.**get_tools**()

Retrieves the TidyPy tools that are available in the current Python environment.

The returned dictionary has keys that are the tool names and values are the tool classes.

Return type dict

tidypy.**get_reports**()

Retrieves the TidyPy issue reports that are available in the current Python environment.

The returned dictionary has keys are the report names and values are the report classes.

Return type dict

tidypy.**get_extenders**()
Retrieves the TidyPy configuration extenders that are available in the current Python environment.

The returned dictionary has keys are the extender names and values are the extender classes.

Return type dict

tidypy.**get_default_config**()
Produces a stock/out-of-the-box TidyPy configuration.

Return type dict

tidypy.**get_user_config**(*project_path*, *use_cache=True*)
Produces a TidyPy configuration that incorporates the configuration files stored in the current user's home directory.

Parameters

- **project_path** (*str*) – the path to the project that is going to be analyzed
- **use_cache** (*bool*) – whether or not to use cached versions of any remote/referenced TidyPy configurations. If not specified, defaults to `True`.

Return type dict

tidypy.**get_local_config**(*project_path*, *use_cache=True*)
Produces a TidyPy configuration using the `pyproject.toml` in the project's directory.

Parameters

- **project_path** (*str*) – the path to the project that is going to be analyzed
- **use_cache** (*bool*) – whether or not to use cached versions of any remote/referenced TidyPy configurations. If not specified, defaults to `True`.

Return type dict

tidypy.**get_project_config**(*project_path*, *use_cache=True*)
Produces the Tidypy configuration to use for the specified project.

If a `pyproject.toml` exists, the configuration will be based on that. If not, the TidyPy configuration in the user's home directory will be used. If one does not exist, the default configuration will be used.

Parameters

- **project_path** (*str*) – the path to the project that is going to be analyzed
- **use_cache** (*bool*) – whether or not to use cached versions of any remote/referenced TidyPy configurations. If not specified, defaults to `True`.

Return type dict

tidypy.**purge_config_cache**(*location=None*)
Clears out the cache of TidyPy configurations that were retrieved from outside the normal locations.

class tidypy.**Tool**(*config*)
The base class for TidyPy tools.

Parameters **config** (*dict*) – the tool configuration to use during execution

classmethod can_be_used()
Indicates whether or not this tool can be executed now. Useful when you need to check for certain environmental conditions (e.g., Python version, dependency availability, etc).

Unless overridden, always returns `True`.

Return type bool

config = None
The tool's configuration to use during its execution.

execute(*finder*)
Analyzes the project and generates a list of issues found during that analysis.

Must be implemented by concrete classes.

Parameters **finder** (tidypy.Finder) – the Finder class that should be used to identify the files or directories that the tool will analyze.

Return type list(*tidypy.Issue*)

classmethod get_all_codes()
Produces a sequence of all the issue codes this tool is capable of generating. Elements in this sequence must all be 2-element tuples, where the first element is the code, and the second is a textual description of what the code means.

Must be implemented by concrete classes.

Returns tuple of tuples containing two strings each

classmethod get_default_config()
Produces a tool configuration stanza that acts as the base/default for this tool.

rtype: dict

class tidypy.**PythonTool**(*config*)
A convenience abstract class that automatically sets the `filters` in the tool configuration to target Python source files.

Parameters **config** (*dict*) – the tool configuration to use during execution

classmethod get_default_config()
Produces a tool configuration stanza that acts as the base/default for this tool.

rtype: dict

class tidypy.**Issue**(*code=None*, *message=None*, *filename=None*, *line=None*, *character=None*)
A class that encapsulates an issue found during the analysis of a project.

character = None
The character number within the line of the file where the issue was found (if known). The first column in a line is notated as 1 (not zero).

code = None
A string containing a code that identifies the type of issue found.

filename = None
A string containing the full path to the file where the issue was found.

line = None
The line number within the file where the issue was found (if known). The first line in a file is notated as 1 (not zero).

message = None
A string containing a description of the issue.

pylint_type = 'E'
A character indicating the comparable pylint category this issue would fall into: E=error, W=warning, R=refactor, C=convention

tool = None
A string containing name of the tool that found the issue.

class tidypy.**TidyPyIssue**(*code=None*, *message=None*, *filename=None*, *line=None*, *character=None*)
The base class for all TidyPy application issues that are produced.

class tidypy.**UnknownIssue**(*exc*, *filename*)
A completely unanticipated exception/problem was encountered during the execution of a tool.

class tidypy.**AccessIssue**(*exc*, *filename*)
An issue indicating that a file/directory cannot be accessed (typically due to permissions).

class tidypy.**ParseIssue**(*exc*, *filename*, *line=None*, *character=None*)
An issue indicating that a file could not be parsed as expected (e.g., a Python source file with invalid syntax).

class tidypy.**ToolIssue**(*message*, *project_path*, *details=None*, *failure=False*)
An issue indicating that a tool completely crashed/failed during its execution.

class tidypy.**Finder**(*base_path*, *config*)
A class that encapsulates the logic of finding files in a project that will be analyzed.

Parameters

- **base_path** (*str*) – the path to the base of the project
- **config** (*dict*) – the configuration to use when searching the project

directories(*filters=None*, *containing=None*)
A generator that produces a sequence of paths to directories in the project that matches the specified filters.

Parameters

- **filters** (*list(str)*) – the regular expressions to use when finding directories in the project. If not specified, all directories are returned.
- **containing** (*list(str)*) – if a directory passes through the specified filters, it is checked for the presence of a file that matches one of the regular expressions in this parameter.

files(*filters=None*)
A generator that produces a sequence of paths to files in the project that matches the specified filters.

Parameters **filters** (*list(str)*) – the regular expressions to use when finding files in the project. If not specified, all files are returned.

is_excluded(*path*)
Determines whether or not the specified file is excluded by the project's configuration.

Parameters **path** (*pathlib.Path*) – the path to check

Return type bool

is_excluded_dir(*path*)
Determines whether or not the specified directory is excluded by the project's configuration.

Parameters **path** (*pathlib.Path*) – the path to check

Return type bool

modules(*filters=None*)
A generator that produces a sequence of paths to files that look to be Python modules (e.g., `*.py`).

Parameters **filters** (*list(str)*) – the regular expressions to use when finding files in the project. If not specified, all files are returned.

packages(*filters=None*)
A generator that produces a sequence of paths to directories that look to be Python packages (e.g., they contain an `__init__.py`).

Parameters **filters** (*list(str)*) – the regular expressions to use when finding directories in the project. If not specified, all directories are returned.

project_path
The path to the project that this Finder is operating from.

read_file(*filepath*)
Retrieves the contents of the specified file.

This function performs simple caching so that the same file isn't read more than once per process.

Parameters **filepath** (*str*) – the file to read.

Return type str

relative_to_project(*filepath*)
Reformats a file path to be relative to this Finder's project path.

Parameters **filepath** (*str or pathlib.Path*) – the path to reformat

Return type str

sys_paths(*filters=None*)
Produces a list of paths that would be suitable to use in `sys.path` in order to access the Python modules/packages found in this project.

Parameters **filters** (*list(str)*) – the regular expressions to use when finding files in the project. If not specified, all files are returned.

class tidypy.**Collector**(*config*)
A class that contains all the issues found during an execution of the TidyPy tool suite.

Parameters **config** (*dict*) – the configuration used to during the analysis of the project

add_issues(*issues*)
Adds an issue to the collection.

Parameters **issues** (*tidypy.Issue or list(tidypy.Issue)*) – the issue(s) to add

get_grouped_issues(*keyfunc=None*, *sortby=None*)
Retrieves the issues in the collection grouped into buckets according to the key generated by the keyfunc.

Parameters

- **keyfunc** (*func*) – a function that will be used to generate the key that identifies the group that an issue will be assigned to. This function receives a single tidypy.Issue argument and must return a string. If not specified, the filename of the issue will be used.
- **sortby** (*list(str)*) – the properties to sort the issues by

Return type OrderedDict

get_issues(*sortby=None*)
Retrieves the issues in the collection.

Parameters **sortby** (*list(str)*) – the properties to sort the issues by

Return type list(*tidypy.Issue*)

issue_count(*include_unclean=False*)
Returns the number of issues in the collection.

Parameters **include_unclean** (*bool*) – whether or not to include issues that are being ignored due to being a duplicate, excluded, etc.

Return type int

class tidypy.**Report**(*config*, *base_path*, *output_file=None*)
The base class for TidyPy issue reporters.

Parameters

- **config** (*dict*) – the configuration used during the analysis of the project
- **base_path** (*str*) – the path to the project base directory

execute(*collector*)
Produces the contents of the report.

Must be implemented by concrete classes.

Parameters **collector** (tidypy.Collector) – the collection of issues to report on

output(*msg*, *newline=True*)
Writes the specified string to the output target of the report.

Parameters

- **msg** (*str*) – the message to output.
- **newline** (*str*) – whether or not to append a newline to the end of the message

relative_filename(*filename*)
Generates a path for the specified filename that is relative to the project path.

Parameters **filename** (*str*) – the filename to generate the path for

Return type str

class tidypy.**Extender**
The base class for TidyPy configuration extenders.

classmethod can_handle(*location*)
Indicates whether or not this Extender is capable of retrieving the specified location.

Parameters **location** (*str*) – a URI indicating where to retrieve the TidyPy configuration from

Return type bool

classmethod parse(*content*, *is_pyproject=False*)
A convenience method for parsing a TOML-serialized configuration.

Parameters

- **content** (*str*) – a TOML string containing a TidyPy configuration
- **is_pyproject** (*bool*) – whether or not the content is (or resembles) a `pyproject.toml` file, where the TidyPy configuration is located within a key named `tool`.

Return type dict

classmethod retrieve(*location*, *project_path*)
Retrieves a TidyPy configuration from the specified location.

Parameters

- **location** (*str*) – a URI indicating where to retrieve the TidyPy configuration from
- **project_path** (*str*) – the full path to the project's base

Return type dict

exception tidypy.**ExtenderError**
The base class for all exceptions raised by an Extender during its operation.

exception tidypy.**DoesNotExistError**
An exception indicating that the specified Extender does not exist in the current environment.

class tidypy.**Progress**
An interface for receiving events that occur during the execution of the TidyPy tool suite.

on_finish()
Called after all tools in the suite have completed.

on_start()
Called when the execution of the TidyPy tool suite begins.

on_tool_finish(*tool*)
Called when an individual tool completes execution.

Parameters **tool** (*str*) – the name of the tool that completed

on_tool_start(*tool*)
Called when an individual tool begins execution.

Parameters **tool** (*str*) – the name of the tool that is starting

class tidypy.**QuietProgress**
An implementation of tidypy.Progress that produces no output.

class tidypy.**ConsoleProgress**(*config*)
An implementation of tidypy.Progress that outputs a progress bar to the console.

on_finish()
Called after all tools in the suite have completed.

on_start()
Called when the execution of the TidyPy tool suite begins.

on_tool_finish(*tool*)
Called when an individual tool completes execution.

Parameters **tool** (*str*) – the name of the tool that completed

on_tool_start(*tool*)
Called when an individual tool begins execution.

Parameters **tool** (*str*) – the name of the tool that is starting

CHAPTER 3

TidyPy Change Log

Releases

3.1 0.21.1 (2021-09-14)

Fixes

- Fixed an installation failure due to an old version of `demjson` not working with `setuptools>=58`.

3.2 0.21.0 (2021-08-28)

Enhancements

- Upgraded the `pylint` tool.
- Upgraded the `pep8-naming` plugin of the `pycodestyle` tool.

3.3 0.20.0 (2021-03-19)

Enhancements

- Upgraded the `pylint`, `secrets`, `pyroma`, `pydocstyle`, `pycodestyle`, and `pyflakes` tools.

3.4 0.19.0 (2021-01-16)

Enhancements

- Upgraded the `manifest` tool.

Fixes

- Fixed a crash due to the latest version of `vulture`.

3.5 0.18.0 (2020-11-27)

Enhancements

- Upgraded the `manifest` and `dlint` tools.

Fixes

- Fixed an issue that caused crashes when specifying additional options to the `yamllint` tool.

3.6 0.17.0 (2020-10-10)

Enhancements

- Upgraded the `manifest` and `eradicate` tools.

3.7 0.16.0 (2020-09-12)

Enhancements

- Upgraded the `pylint` and `vulture` tools.
- Addd a `--config` option to the `check` command.

3.8 0.15.0 (2020-07-12)

Enhancements

- Upgraded the `secrets` and `manifest` tools.
- Upgraded the `pep8-naming` plugin of the `pycodestyle` tool.

Fixes

- Fixed an crash that occurred with v1.5 of `vulture`.

3.9 0.14.0 (2020-05-12)

Enhancements

- Upgraded the `pycodestyle`, `pylint`, and `pyflakes` tools.

3.10 0.13.0 (2020-04-10)

Enhancements

- Upgraded the `dlint` and `manifest` tools.
- Upgraded the `pep8-naming` plugin of the `pycodestyle` tool.

Fixes

- Fixed a dependency conflict with `pyflakes`.

3.11 0.12.0 (2020-01-05)

Enhancements

- Upgraded the `manifest`, `secrets`, `pydocstyle`, and `dlint` tools.
- Upgraded the `pep8-naming` plugin of the `pycodestyle` tool.

Changes

- Removed support for Python 2.
- Removed the `setuptools` plugin, as it was causing many problems, and was little-used, anyway.

3.12 0.11.0 (2019-09-29)

Enhancements

- Added the `dlint` tool.
- Upgraded the `pylint`, `pydocstyle`, and `manifest` tools.

Fixes

- Fixed an issue with the most recent version of the `vulture` tool crashing.

3.13 0.10.1 (2019-06-02)

Fixes

- Fixed an issue listing the codes from the most recent version of the `pyroma` tool.

3.14 0.10.0 (2019-05-18)

Enhancements

- Upgraded the `manifest` tool.
- Enabled the `eradicate` tool in PY3 environments.

3.15 0.9.0 (2019-03-16)

Enhancements

- Upgraded the `pylint` and `secrets` tools.
- Added a reporter named `pylint-parseable` that emulates `pylint`'s "parseable" output format.
- Added a reporter named `custom` that allows you to specify the output format of issues.
- Added support for the `vulture` options `ignore-names`, `ignore-decorators`, and `min-confidence` (thanks acaprari).

3.16 0.8.0 (2019-01-30)

Enhancements

- Added ability to specify a filename for reports on the command line (thanks douardda).
- Upgraded the `secrets`, `pylint`, `pycodestyle`, and `eradicate` tools.
- Upgraded the `pep8-naming` plugin of the `pycodestyle` tool.

3.17 0.7.0 (2018-10-24)

Enhancements

- Upgraded the `pycodestyle`, `pydocstyle`, `vulture`, and `pyflakes` tools.
- Added ability to distinguish and disable specific codes from the `secrets` tool.

3.18 0.6.0 (2018-09-30)

Enhancements

- Added the `secrets` tool.
- Enabled the `pydiatra` tool on windows (thanks jwilk).
- Upgraded the `pylint` and `vulture` tools.
- Upgraded the `pep8-naming` plugin of the `pycodestyle` tool.

Fixes

- Fixed an issue with `rstlint` crashing due to recent updates to Sphinx.

3.19 0.5.0 (2018-05-05)

Enhancements

- Added `manifest` and `pydiatra` tools.
- Upgraded the `pylint` tool.
- Upgraded the `pep8-naming` plugin of the `pycodestyle` tool.
- Added some convenience handling of the `License` vs `Licence` and `LicenceClassifier` vs `LicenseClassifier` codes reported by `pyroma`.
- Added the first draft of the project documentation.
- Added an `extensions` command that will output a listing of all the available tools, reports, and extenders that are available.

Fixes

- Fixed the character location reported in `pylint` issues being off-by-one.
- Fixed various issues with the `pyroma` tool leaking problems to stderr.

3.20 0.4.0 (2017-12-02)

Enhancements

- Added a `sphinx-extensions` option to the `rstlint` tool to enable the automatic recognition of Sphinx-specific extensions to ReST (Sphinx must be installed in the same environment as TidyPy for it to work).
- Added a `ignore-roles` option to the `rstlint` tool to help deal with non-standard ReST text roles.

- Changed tool execution from a multithreaded model to multiprocess. Larger projects should see an improvement in execution speed.

Changes

- The `--threads` option to the `check` command has been changed to `--workers`.

Fixes

- Fixed an issue that caused the `pylint` tool to crash when it encountered `duplicate-code` issues on files that are being excluded from analysis.

3.21 0.3.0 (2017-11-18)

Enhancements

- Added `ignore-directives` and `load-directives` options to the `rstlint` tool to help deal with non-standard ReST directives.
- Added support for the `extension-pkg-whitelist` option to the `pylint` tool.
- Added `install-vcs` and `remove-vcs` commands to install/remove pre-commit hooks into the VCS of a project that will execute TidyPy. Currently supports both Git and Mercurial.

Changes

- Changed the `merge_issues` and `ignore_missing_extends` options to `merge-issues` and `ignore-missing-extends` for naming consistency.
- Replaced the `radon` tool with the traditional `mccabe` tool.

Fixes

- Fixed issue that caused TidyPy to spin out of control if you used CTRL-C to kill it while it was executing tools.
- Fixed issue where `pylint`'s `duplicate-code` issue was reported only against one file, and it was usually the wrong file. TidyPy will now report an issue against each file identified with the duplicate code.
- Numerous fixes to support running TidyPy on Windows.

3.22 0.2.0 (2017-11-04)

Enhancements

- Added a `2to3` tool.
- All tools that report issues against Python source files can now use the `# noqa` comment to ignore issues for that specific line.
- Added support for the `ignore-nosec` option in the `bandit` tool.
- Added the ability for TidyPy configurations to extend from other configuration files via the `extends` property.
- Upgraded the `vulture` tool.
- Upgraded the `pyflakes` tool.

Changes

- Changed the `--no-merge` and `--no-progress` options to the `check` command to `--disable-merge` and `--disable-progress`.

- The `check` command will now return `1` to the shell if TidyPy finds issues.
- No longer overriding `pycodestyle`'s default max-line-length.

Fixes

- If any tools output directly to stdout or stderr, TidyPy will now capture it and report it as a `tidypy:tool` issue.
- Fixed crash/hang that occurred when using `--disable-progress`.

3.23 0.1.0 (2017-10-15)

- Initial public release.

CHAPTER 4

MIT License

CHAPTER 5

Indices and tables

- genindex
- search

Python Module Index

t

Index