tangentsky Documentation

Release 0.1.0

Josh Bialkowski

May 20, 2019

Contents:

A highly-compatible “build” system for linting python files.

CHAPTER 1

Installation

1.1 Install with pip

The easiest way to install `makelint` is from pypi.org using pip. For example:

```
pip install makelint
```

If you're on a linux-type system (such as ubuntu) the above command might not work if it would install into a system-wide location. If that's what you really want you might need to use `sudo`, e.g.:

```
sudo pip install makelint
```

In general though I wouldn't really recommend doing that though since things can get pretty messy between your system python distributions and your `pip` managed directories. Alternatively you can install it for your user with:

```
pip install --user makelint
```

which I would probably recommend for most users.

1.2 Install from source

You can also install from source with pip. You can download a release package from github and then install it directly with pip. For example:

```
pip install v0.1.0.tar.gz
```

Note that the release packages are automatically generated from git tags which are the same commit used to generate the corresponding version package on `pypi.org`. So whether you install a particular version from github or pypi shouldn't matter.

Pip can also install directly from github. For example:

```
pip install git+https://github.com/cheshirekow/makelint.git
```

If you wish to test a pre-release or dev package from a branch called `foobar` you can install it with:

```
pip install "git+https://github.com/cheshirekow/makelint.git@v0.1.0"
```

CHAPTER 2

Design

2.1 Discovery/Indexing

The first phase is discovery and indexing. This is done at build time, rather than configure-time, because, let's face it, your build system already suffers from enough configure time bloat. Also, as mentioned above, this supports an opt-out system.

The discovery step performs a filesystem walk in order to build up an index of files to be checked. You can use a configuration file or command line options to setup inclusion and exclusion filters for the discovery process. In general, though, each directory that is scanned produces a list of files to lint. If the timestamp of a tracked directory changes, it is rescanned for new files, or new directories.

The output of the discovery phase is a manifest file per-directory tracked. The creation of this manifest depends on the modification time of the directory it corresponds to and will be re-built if the directory is changed. If a new subdirectory is added, the system will recursively index that new directory. If a directory is removed, it will recursively purge that directory from the manifest index.

2.2 Content Digest

The second phase is content summary and digest creation. The sha1 of each tracked file is computed and stored in a digest file (one per source file). The digest file depends on the modification time of the source file.

2.3 Dependency Inference

The third phase is dependency inference. During this phase each tracked source file is indexed to get a complete dependency footprint. Note that this is done by importing each module file in a clean interpreter process, and then inspecting the `__file__` attribute of all modules loaded by interpreter. Note that this has a couple of implications:

- Dynamically loaded modules may not be discovered as dependencies
- Any import work will increase the runtime of this phase

The outputs for this phase is a dependency manifest: one per source file. The manifest contains a list of files that are dependencies. The dependencies of this manifest are the modification times of the digest sidecar file for each of the source file itself, as well as all of it's dependencies. If any of these digest files are modified, the manifest is rebuilt. There is a fastpath, however, in that if none of the digests themselves have changed the manifest modification time is updated but the dependency scan is skipped.

2.4 Executing tools

Once the depency footprints are updated we can finally start executing the actual tools. There are two outputs of a tool execution : a stampfile (one per source file) and a logfile. The stampfile is skipped on failure and the logfile is removed on success.

CHAPTER 3

Usage

```
usage:
pymakelint [-h] [-v] [-l {debug,info,warning,error}] [--dump-config]
           [-c CONFIG_FILE] [<config-overrides> [...]]

Incremental execution system for python code analysis (linting).

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -l {debug,info,warning,error}, --log-level {debug,info,warning,error}
  --dump-config         If specified, print the default configuration to
                        stdout and exit
  -c CONFIG_FILE, --config-file CONFIG_FILE
                        path to configuration file

Configuration:
  Override configfile options

  --include-patterns [INCLUDE_PATTERNS [INCLUDE_PATTERNS ...]]
                        A list of python regular expression patterns which are
                        used to include files during the directory walk. They
                        are matched against relative paths of files (relative
                        to the root of the search). They are not matched
                        against directories. The default is `[".*\.py"]`.
  --exclude-patterns [EXCLUDE_PATTERNS [EXCLUDE_PATTERNS ...]]
                        A list of python regular expression patterns which are
                        used to exclude files during the directory walk. They
                        are matched against relative paths of files (relative
                        to the root of the search). If the pattern matches a
                        directory the whole directory is skipped. If it
                        matches an individual file then that file is skipped.
  --source-tree SOURCE_TREE
                        The root of the search tree for inclusion.
  --target-tree TARGET_TREE
```

(continues on next page)

(continued from previous page)

```
                        The root of the tree where the outputs are written.
  --tools [TOOLS [TOOLS ...]]
                        A list of tools to execute. The default is ["pylint",
                        "flake8"]. This can either be a string (a simple
                        command which takes one argument), or it can be an
                        object with a get_stamp() and an execute() method. See
                        SimpleTool for ane example.
  --fail-fast [FAIL_FAST]
                        If true, exit on the first failure, don't keep going.
                        Useful if you want a speedy CI gate.
  --merged-log MERGED_LOG
                        If specified, output logs for failed jobs will be
                        merged into a single file at this location. Useful if
                        you have a large number of issues to del with.
  --quiet [QUIET]       Don't print fancy progress bars to stdout.
  --jobs JOBS           Number of parallel jobs to execute.
```

3.1 Configuration

Most command line options can also be specified in a configuration file. Configuration files are python files. If not specified on the command line, the tool will automatically look for and load the configuration file at `<source_tree>/.makelint.py`.

You can use `--dump-config` to dump the default configuration to a file and use that as a starting point. The default config is also pasted below.

```
# A list of python regular expression patterns which are used to include files
# during the directory walk. They are matched against relative paths of files
# (relative to the root of the search). They are not matched against
# directories. The default is `[".*\.py"]`.
include_patterns = ['.*\\.py']

# A list of python regular expression patterns which are used to exclude files
# during the directory walk. They are matched against relative paths of files
# (relative to the root of the search). If the pattern matches a directory the
# whole directory is skipped. If it matches an individual file then that file is
# skipped.
exclude_patterns = []

# The root of the search tree for inclusion.
source_tree = None

# The root of the tree where the outputs are written.
target_tree = None

# A list of tools to execute. The default is ["pylint", "flake8"]. This can
# either be a string (a simple command which takes one argument), or it can be
# an object with a get_stamp() and an execute() method. See SimpleTool for ane
# example.
tools = ['flake8', 'pylint']

# A dictionary specifying the environment to use for the tools. Add your
# virtualenv configurations here.
env = {
```

(continues on next page)

(continued from previous page)

```
  "LANG": "en_US.UTF-8",
  "LANGUAGE": "en_US",
  "PATH": [
    "/usr/local/sbin",
    "/usr/local/bin",
    "/usr/sbin",
    "/usr/bin",
    "/sbin",
    "/bin"
  ]
}

# If true, exit on the first failure, don't keep going. Useful if you want a
# speedy CI gate.
fail_fast = False

# If specified, output logs for failed jobs will be merged into a single file
# at this location. Useful if you have a large number of issues to del with.
merged_log = None

# Don't print fancy progress bars to stdout.
quiet = False

# Number of parallel jobs to execute.
jobs = 12  # multiprocessing.cpu_count()
```

3.2 Example

For example, executing on this project itself:

```
$ time makelint --source-tree . --target-tree /tmp/makelint --jobs 1
real        0m10.221s
user        0m9.736s
sys 0m0.510s
$ time makelint --source-tree . --target-tree /tmp/makelint
real        0m0.097s
user        0m0.077s
sys 0m0.020s
```

CHAPTER 4

Release Notes

Details of changes can be found in the changelog, but this file will contain some high level notes and highlights from each release.

4.1 v0.1 series

4.1.1 v0.1.0

Initial release!

CHAPTER 5

Changelog

5.1 v0.1 series

5.1.1 v0.1.0

Initial release

- Working proof-of-concept implementation with filesystem storage
- Support for configuration with a file
- Support for parallel execution
- Support for running additional checker tools (default pylint and flake8)

CHAPTER 6

TODO

6.1 Single-file Manifest

Think about whether or not the file manifest can be a single file.

- If any directory changes we need to run the job.
- But we wont know if a directory has changed without reading it's timestamp…
- Reading the timestamp means reading the directory contents of the parent, so we are essentially walking the tree just to check timestamps.
- If we are walking the tree anyway, is there any reason to just write the one big file?
- We might not want to write a new file on every invocation (seems kind of dirty from a build system perspective) but we can probaby whole the entire manifest in memory and then write it out. If we can't hold it in memory we could write it to a temporary location and then copy it to the goal location.

6.2 Latch Environment

- Add environmental information to the target tree, so that we know when it is invalidated by a change in configuration
- We probably want to include the entire config object. We need to rescan wheneverer exclude patterns or include patterns change.. source tree, environment options (PYTHONPATH, etc).
- Implement `--add-source-tree-to-python-path` config/command line option. We don't want the user to have to do this in the config necessarily because then they can't store the config in the repo. They could "configure" the config but it would be nice for that not to be a requirement.

6.3 Makefile Jobserver Client

Implement GNU make jobserver client protocol

- see: this page on job slots
- see also: this page detaling the posix jobserver implementation

It's pretty easy actually. Just read the `MAKEFLAGS` environment variable. If it contains `--jobserver-auth=<R>,<W>` then `<R>`, `<W>` are integer filedescriptors of the read and write ends of a pipe. The read end contains one character for each job that we are allowed. For each byte we read out of the pipe we must write one byte back to the pipe when we are done.

- See the notes on that page about what to do in various cases for some make invocations. In particular, you should catch SIGINT and return the jobs back to the server.
- Note that you always get one implicit job (that you do not return to the server)
- There is ongoing discussion about implementing the makefile jobserver client protocol (and server protocol) in ninja. Cmake already appears to distribute a version of ninja that supports it. https://github.com/ninja-build/ninja/pull/1140
- This guy has an implementation that seems to work https://github.com/stefanb2/ninja
- Of course, ninja has it's `pool` implementation, so you can probably skip concerns about this

6.4 Other

- Implement sqlite database backend (versus filesystem)
- Change the name of this package/project
- Add a `--whitelist` command-line/config argument. Rather than secifying a large list of exact filenames in the exclusion patterns, this can be a set that we efficiently check for inclusion in.
- Implement `dlsym` checking to get a list of python modules that are loaded
- Implement a `--merge-env` option to merge the configured environment into the runtime environment.

CHAPTER 7

makelint package

7.1 Module contents

class makelint.**DependencyItem**(*path*, *digest*, *name*)

Bases: `tuple`

digest

Alias for field number 1

name

Alias for field number 2

path

Alias for field number 0

class makelint.**NullProgressReport**

Bases: `object`

No-op for quiet mode

class makelint.**ProgressReporter**

Bases: `object`

Prints a status message

do_print()

get_istep()

Return the index of our current step

get_nsteps()

Return the total number of steps to completion

get_progress()

Return current progress as a percentage

makelint.**cat_log**(*logfile_path*, *header*, *merged_log*)

Copy the content from logfile_path into merged_log

`makelint.`**`chunk_iter_file`**`(`*infile*, *chunk_size=4096*`)`
Read a file chunk by chunk

`makelint.`**`depmap_is_uptodate`**`(`*target_tree*, *relpath_file*`)`
Given a dictionary of dependency data, return true if all of the files listed are unchanged since we last ran the scan.f

`makelint.`**`digest_file`**`(`*source_path*, *digest_path*`)`
Compute a message digest of the file content, write the digest (in hexadecimal ascii encoding) to the output file

`makelint.`**`digest_sourcetree_content`**`(`*source_tree*, *target_tree*, *progress*, *njobs*`)`
The sha1 of each tracked file is computed and stored in a digest file (one per source file). The digest file depends on the modification time of the source file. If the sourcefile hasn't changed, the digest file doesn't need to be updated.

`makelint.`**`discover_sourcetree`**`(`*source_tree*, *target_tree*, *exclude_patterns*, *include_patterns*, *progress*`)`
The discovery step performs a filesystem walk in order to build up an index of files to be checked. You can use configuration files to setup inclusion and exclusion filters for the discovery process. In general, though, each directory that is scanned produces a list of files to lint. If the timestamp of a tracked directory changes, it is rescanned for new files, or new directories.

The output of the discovery phase is a manifest file per-directory tracked. The creation of this manifest depends on the modification time of the directory it corresponds to and will be re-built if the directory is changed. If a new subdirectory is added, the system will recursively index that new directory. If a directory is removed, it will recursively purge that directory from the manifest index.

`makelint.`**`execute_tool_ontree`**`(`*source_tree*, *target_tree*, *tool*, *env*, *fail_fast*, *merged_log*, *progress*, *njobs*`)`
Execute the given tool

`makelint.`**`get_progress_bar`**`(`*numchars*, *fraction=None*, *percent=None*`)`
Return a high resolution unicode progress bar

`makelint.`**`map_dependencies`**`(`*source_tree*, *target_tree*, *source_relpath*`)`
Get a dependency list from the sourcefile. Writeout the dependency file and it's sha1 digest.

`makelint.`**`map_sourcetree_dependencies`**`(`*source_tree*, *target_tree*, *progress*, *njobs*`)`
During this phase each tracked source file is indexed to get a complete dependency footprint. Note that this is done by importing each module file in a clean interpreter process, and then inspecting the *__file__* attribute of all modules loaded by interpreter.

`makelint.`**`toolstamp_is_uptodate`**`(`*toolstamp_path*, *depmap_path*`)`
Return true if the toolstamp is up to date with respect to the dependency map

`makelint.`**`waitforsize`**`(`*pidset*, *njobs*`)`
Given a set() of pids, wait until it has at most njobs alive children

7.2 Submodules

7.3 makelint.configuration module

`class` `makelint.configuration.`**`ConfigObject`**
Bases: `object`

Provides simple serialization to a dictionary based on the assumption that all args in the __init__() function are fields of this object.

as_dict()
Return a dictionary mapping field names to their values only for fields specified in the constructor

classmethod get_field_names()
Return a list of field names, extracted from kwargs to __init__(). The order of fields in the tuple representation is the same as the order of the fields in the __init__ function

class makelint.configuration.**Configuration**(*include_patterns=None*, *exclude_patterns=None*, *source_tree=None*, *target_tree=None*, *tools=None*, *env=None*, *fail_fast=False*, *merge_log=None*, *quiet=False*, *jobs=None*, ***extra*)
Bases: makelint.configuration.ConfigObject

Encapsulates various configuration options/parameters

clone()
Return a copy of self.

class makelint.configuration.**SimpleTool**(*name*)
Bases: object

Simple implementation of the tool API that works for commands which just take the name of the file as an argument.

as_dict()

execute(*source_tree*, *source_relpath*, *env*, *outfile*)

get_stamp(*target_cwd*, *filename*)

makelint.configuration.**get_default**(*value*, *default*)
return `value` if it is not None, else default

makelint.configuration.**parse_bool**(*string*)
Evaluate the truthiness of a string

makelint.configuration.**serialize**(*obj*)
Return a serializable representation of the object. If the object has an *as_dict* method, then it will call and return the output of that method. Otherwise return the object itself.

7.4 makelint.get_dependencies module

Helper module to get dependencies. exec() a python file and then inspect `sys.modules` and record everything that was read in.

makelint.get_dependencies.**main**()

7.5 makelint.__main__ module

Incremental execution system for python code analysis (linting).

makelint.__main__.**add_config_options**(*optgroup*)
Add configuration options as flags to the argument parser

makelint.__main__.**dump_config**(*args*, *config_dict*, *outfile*)
Dump the default configuration to stdout

`makelint.__main__.`**`load_config`**`(`*configfile_path*`)`

Read a configuration file and return as a configuration object

`makelint.__main__.`**`main`**`()`

Parse arguments, open files, start work.

`makelint.__main__.`**`setup_argparser`**`(`*parser*`)`

Add argparse options to the parser.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index