linesieve

Release 1.0

lemon24

Jul 09, 2023

CONTENTS

This is my text munging tool. There are many like it, but this one is mine.

linesieve is an unholy blend of grep, sed, awk, and Python, born out of spite.

CHAPTER

ONE

FEATURES

`linesieve` allows you to:

- split text input into sections
- apply filters to specific sections
- search and highlight success/failure markers
- match/sub/split with the full power of Python's re
- shorten paths, links and module names
- chain filters into pipelines
- color output!

CHAPTER

TWO

INSTALLING

Install and update using pip:

```
$ pip install --upgrade linesieve
```

CHAPTER

THREE

A SIMPLE EXAMPLE

```
$ ls -1 /* | linesieve -s '.*:' show bin match ^d head -n2
.....
/bin:
dash
date
......
/sbin:
disklabel
dmesg
...
```

This prints the first two files starting with `d` from each directory whose name contains `bin` (skipped directories are marked with a dot on stderr).

CHAPTER

FOUR

LINKS

- PyPI Releases: https://pypi.org/project/linesieve/
- Documentation: https://linesieve.readthedocs.io/
- Issue Tracker: https://github.com/lemon24/linesieve/issues
- Source Code: https://github.com/lemon24/linesieve

CHAPTER

FIVE

USER GUIDE

5.1 Reference

5.1.1 linesieve

linesieve is a tool for splitting text input into sections and applying filters to the lines in each section.

Example:

```
$ ls -1 /* | linesieve -s '.*:' show bin match ^d head -n2
.....
/bin:
dash
date
......
/sbin:
disklabel
dmesg
...
```

You can specify a section marker regex with `-s`/`--section`, as well as `--success` and `--failure` markers which cause linesieve to exit early. To show only specific sections, use the `show` subcommand; skipped sections are marked with a dot on stderr.

```
$ ls -1 /* | linesieve -s '.*:' --failure ^cat show bin
.....
/bin:
[
bash
cat
```

All patterns use the Python regular expression syntax.

You can use subcommands to filter the lines in each section. To restrict a filter to specific sections, use their `-s`/`--section` option; you can also temporarily restrict all filters using the `push` and `pop` subcommands.

```
$ ls -1 /* | linesieve -s '.*:' show bin \
> match -s /bin ^b match -s /sbin ^d head -n1
.....
/bin:
bash
......
```

(continues on next page)

(continued from previous page)

```
/sbin:
disklabel
...
```

You can also operate on multi-line records instead of individual lines:

```
$ linesieve --record-start '^\d+:\d+' match two << EOF
00:01 one
00:02 ...
two
00:03 three
EOF
00:02 ...
two
```

By default, linesieve reads from the standard input, but it can also read from a file or a command with the `read` and `read-cmd` subcommands.

```
$ linesieve read-cmd echo bonjour
bonjour
linesieve: echo exited with status code 0
```

On output, runs of blank lines are collapsed into a single line.

```
linesieve [OPTIONS] [COMMAND [ARGS]...]...
```

Options

-s, --section <section>

Consider matching lines the start of a new section. The section name is one of: the named group `name`, the first captured group, the entire match.

--success <success>

If matched, exit with a status code indicating success.

--failure <failure>

If matched, exit with a status code indicating failure. Before exiting, output the last section if it wasn't already.

--record-start, --rs <record_start>

Operate on multi-line records instead of individual lines. Records begin with lines matching `--record-start`, and end with the line *before* the next record start, or with the line matching `--record-end`, if provided. Lines outside record markers are also grouped into records. `--section` always applies to individual lines, regardless of `--record-start` (input is first split into sections, then into records). In patterns that apply to records, `.` matches any character, including newlines.

--record-end <record_end>

Consider matching lines the end of the current record. Requires `--record-start`.

--version

Show the version and exit.

head

Print the first COUNT lines.

Roughly equivalent to: `head -n COUNT`

```
$ echo -e 'a\nb\nc' | linesieve head -n2
a
b
```

```
linesieve head [OPTIONS]
```

Options

-n <COUNT>

Print the first COUNT lines. With a leading `-`, print all but the last COUNT lines.

Default
`10`

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

help

Show detailed help.

```
linesieve help [OPTIONS]
```

Options

--all

Show help for all commands, man-style.

hide

Do not output sections matching PATTERN.

`hide` patterns take priority over `show` patterns.

`^$` matches the lines before the first section. `$none` matches no section.

```
$ ls -1 /* | linesieve -s '.*:' show bin hide /bin head -n2
............
/sbin:
apfs_hfs_convert
disklabel
...
```

```
linesieve hide [OPTIONS] PATTERN
```

Options

-F, --fixed-strings
: Interpret the pattern as a fixed string.

-i, --ignore-case
: Perform case-insensitive matching.

-X, --verbose
: Ignore whitespace and comments in the pattern.

Arguments

PATTERN
: Required argument

match

Output only lines matching PATTERN.

Roughly equivalent to: `grep PATTERN`

Works like re.search() in Python.

```
$ seq 10 | linesieve match 1
1
10
$ echo a1b2c3 | linesieve match -o '\d+'
1
2
3
```

For parsing structured data, see `linesieve parse`.

```
linesieve match [OPTIONS] PATTERN
```

Options

-F, --fixed-strings
: Interpret the pattern as a fixed string.

-i, --ignore-case
: Perform case-insensitive matching.

-X, --verbose
: Ignore whitespace and comments in the pattern.

-o, --only-matching
: Output only the matching part of the line, one match per line. Works like re.findall() in Python: if there are no groups, output the entire match; if there is one group, output the group; if there are multiple groups, output all of them (tab-separated).

-v, --invert-match

Output only lines *not* matching the pattern.

--color

Color matches.

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

Arguments

PATTERN

Required argument

parse

Parse lines into structured data.

If the -e PATTERN uses named groups, output the groups as name-value pairs (unnamed groups are ignored):

```
$ echo +1a+ | linesieve parse -e '(?P<one>\d)(?P<two>\w)'
one     1       two     a
```

If there are only groups without names, output all the groups:

```
$ echo +1a+ | linesieve parse -e '(\d)(\w)'
1       a
```

If there are no groups, output the entire match.

If mutiple patterns are specified, they are tried in order, and the first matching one is used. If no pattern matched, the line is output as-is.

By default, both names and values are tab-separated. You can output JSON using the `--json` option:

```
$ echo +1a+ | linesieve parse --json -e '(?P<one>\d)(?P<two>\w)'
{"one": "1", "two": "a"}
$ echo +1a+ | linesieve parse --json -e '(\d)(\w)'
["1", "a"]
```

For matching groups with names of the form `<name>__<new_value>`, `name` is used as name and `new_value` as value, ignoring the actual group value. This can be used to tell which of multiple -e patterns matched, or to assign symbolic values to groups:

```
$ linesieve parse \
> -e '(?P<event__get>)got (?P<count>\d+) thing' \
> -e '(?x) (?P<event__send>) (
>   (?P<status__ok>sent) |
>   (?P<status__fail>could \s+ not \s+ send)
> )' \
> --json << EOF
got 10 things
sent the things
```

(continues on next page)

(continued from previous page)

```
could not send the things
EOF
{"event": "get", "count": "10"}
{"event": "send", "status": "ok"}
{"event": "send", "status": "fail"}
```

```
linesieve parse [OPTIONS]
```

Options

-e, --regexp <patterns>

Required Use PATTERN as the pattern; multiple patterns are tried in order.

-F, --fixed-strings

Interpret the pattern as a fixed string.

-i, --ignore-case

Perform case-insensitive matching.

-X, --verbose

Ignore whitespace and comments in the pattern.

-o, --only-matching

Output only matching lines.

--json

Output groups as JSON instead of tab-separated values. Output one JSON value per line, either a JSON object (if named groups are used) or a JSON array (otherwise).

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

pipe

Pipe lines to COMMAND and replace them with the output.

COMMAND is executed once per section.

Command output is not parsed back into multi-line records, even if `linesieve --record-start` was used.

```
$ echo a-b | linesieve pipe 'tr -d -'
ab
```

```
linesieve pipe [OPTIONS] COMMAND
```

Options

-s, --section <SECTION>
: Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

Arguments

COMMAND
: Required argument

pop

Pop patterns off the section stack. See 'push' for details.

With no arguments, removes the top pattern from the stack.

```
linesieve pop [OPTIONS]
```

Options

-a, --all
: Remove all the patterns from the stack.

push

Push a pattern onto the section stack.

When there are patterns on the section stack, filters apply only to the sections that match any of the patterns in the stack.

`filter --section PATTERN` is equivalent to `push PATTERN filter pop`.

```
$ ls -1 /* | linesieve -s '.*:' \
> show bin \
> push /bin \
>   head -n1 \
> pop \
> head -n2
.....
/bin:
[
......
/sbin:
apfs_hfs_convert
disklabel
...
```

```
linesieve push [OPTIONS] PATTERN
```

Options

-F, --fixed-strings
Interpret the pattern as a fixed string.

-i, --ignore-case
Perform case-insensitive matching.

-X, --verbose
Ignore whitespace and comments in the pattern.

Arguments

PATTERN
Required argument

read

Read input from FILE instead of standard input.

Roughly equivalent to: `linesieve < FILE`

```
$ linesieve read file.txt
hello
```

```
linesieve read [OPTIONS] FILE
```

Arguments

FILE
Required argument

read-cmd

Execute COMMAND and use its output as input.

Roughly equivalent to: `COMMAND | linesieve`

If the command finishes, exit with its status code.

```
$ linesieve read-cmd echo bonjour
bonjour
linesieve: echo exited with status code 0
```

```
linesieve read-cmd [OPTIONS] COMMAND [ARGUMENT]...
```

Arguments

COMMAND
Required argument

ARGUMENT
Optional argument(s)

show

Output only sections matching PATTERN.

`hide` patterns take priority over `show` patterns.

`^$` matches the lines before the first section. `$none` matches no section.

```
$ ls -1 /* | linesieve -s '.*:' show /bin match ash
.....
/bin:
bash
dash
.........
```

```
linesieve show [OPTIONS] PATTERN
```

Options

-F, --fixed-strings
Interpret the pattern as a fixed string.

-i, --ignore-case
Perform case-insensitive matching.

-X, --verbose
Ignore whitespace and comments in the pattern.

Arguments

PATTERN
Required argument

span

Output only lines between those matching `--start` and `--end`.

Roughly equivalent to: `grep START -A9999 | grep END -B9999 | head -n-1`

```
$ seq 20 | linesieve span --start ^2$ --end ^5$ --repl ...
...
2
3
```

(continues on next page)

(continued from previous page)

```
4
...
```

```
linesieve span [OPTIONS]
```

Options

--start, --start-with <start>

Span start (inclusive).

--end, --end-before <end>

Span end (exclusive).

-F, --fixed-strings

Interpret the pattern as a fixed string.

-i, --ignore-case

Perform case-insensitive matching.

-X, --verbose

Ignore whitespace and comments in the pattern.

-v, --invert-match

Output only lines *not* between those matching `--start` and `--end`.

--repl, --replacement <repl>

Replace non-matching line spans with TEXT. With `--invert-match`, backreferences to captures in `--start` are expanded; without `--invert-match`, only escapes are expanded.

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

split

Print selected parts of lines.

Roughly equivalent to:

```
awk '{ print ... }'     (no --delimiter)
cut -d delim            (--fixed-strings --delimiter delim)
```

Python equivalents:

```
line.split()            (no --delimiter)
line.split(delim)       (--fixed-strings --delimiter delim)
re.split(delim, line)   (--delimiter delim)
```

`--fields` takes a comma-separated list of ranges, each range one of:

```
N     Nth field, counted from 1
N-    from Nth field to end of line
N-M   from Nth to Mth (included) field
 -M   from first to Mth (included) field
```

This is the same as the cut command. Unlike cut, selected fields are printed in the order from the list, and more than once, if repeated.

```
$ echo -e 'a-b\nc-d' | linesieve split -d- -f2
b
d
```

```
linesieve split [OPTIONS]
```

Options

-d, --delimiter <PATTERN>

Use as field delimiter (consecutive delimiters delimit empty strings). If not given, use runs of whitespace as a delimiter (with leading/trailing whitespace stripped first).

-F, --fixed-strings

Interpret the pattern as a fixed string.

-i, --ignore-case

Perform case-insensitive matching.

-X, --verbose

Ignore whitespace and comments in the pattern.

-n, --max-split <INTEGER>

Maximum number of splits to do. The default is no limit.

-f, --fields <LIST>

Select only these fields.

-D, --output-delimiter <output_delimiter>

Use as the output field delimiter. If not given, and `--delimiter` and `--fixed-strings` are given, use the input delimiter. Otherwise, use one tab character.

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

sub

Replace PATTERN matches with REPL.

Roughly equivalent to: `sed 's/PATTERN/REPL/g'`

Works like re.sub() in Python.

```
$ echo a1b2c3 | linesieve sub '\d+' x
axbxcx
```

```
linesieve sub [OPTIONS] PATTERN REPL
```

Options

-F, --fixed-strings

Interpret the pattern as a fixed string.

-i, --ignore-case

Perform case-insensitive matching.

-X, --verbose

Ignore whitespace and comments in the pattern.

-o, --only-matching

Output only matching lines.

--color

Color replacements.

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

Arguments

PATTERN

Required argument

REPL

Required argument

sub-cwd

Make absolute paths in the working directory relative.

Roughly equivalent to: `sub $( pwd ) ''`

```
$ echo "hello from $( pwd )/src" | linesieve sub-cwd
hello from src
```

```
linesieve sub-cwd [OPTIONS]
```

Options

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

sub-link

Replace the target of symlink LINK with LINK.

Roughly equivalent to: `sub $( realpath LINK ) LINK`

```
linesieve sub-link [OPTIONS] LINK
```

Options

-s, --section <SECTION>

Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

Arguments

LINK

Required argument

sub-paths

Replace paths of existing files with shorter versions.

The replacement paths are still unique.

For example, given these files are selected:

```
src/one/mod1.py
src/one/two/mod2.py
tests/test.py
```

Their paths will be replaced with:

```
.../mod1.py
.../mod2.py
.../test.py
```

Dotted module names derived from the selected files can also be shortened. For example, with `--modules-skip 1 --modules-recursive`, these modules:

```
one.mod1
one.two.mod2
one.two
```

Will be replaced with:

```
..mod1
..mod2
..two
```

```
linesieve sub-paths [OPTIONS]
```

Options

--include <GLOB>
: Replace the paths of existing files matching this pattern. Both recursive globs and brace expansion are supported, e.g. `{src,tests}/**/*.py`.

--modules
: Also replace dotted module names.

--modules-skip <INTEGER>
: Path levels to skip to obtain module names from paths. Implies `--modules`.

--modules-recursive
: Consider the parent directories of selected files to be modules too. Implies `--modules`.

-s, --section <SECTION>
: Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

tail

Print the last COUNT lines.

Roughly equivalent to: `tail -n COUNT`

```
$ echo -e 'a\nb\nc' | linesieve tail -n2
b
c
```

```
linesieve tail [OPTIONS]
```

Options

-n <COUNT>
: Print the last COUNT lines. With a leading +, print lines starting with line COUNT.

  Default
  10

-s, --section <SECTION>
: Apply only to matching sections. If there are patterns on the section stack, push the pattern (that is, apply *also* to matching sections).

5.2 Examples

This section contains more complicated, real-world examples.

For shorter examples applying to specific subcommands, see the *Reference*.

5.2.1 Java tracebacks

Assume you're writing some Java tests with JUnit, on a project that looks like this:

```
.
├── src
│   └── com
│       └── example
│           └── someproject
│               └── somepackage
│                   └── ThingDoer.java
└── tst
    └── com
        └── example
            └── someproject
                └── somepackage
                    └── ThingDoerTest.java
```

This command:

```
linesieve \
span -v -X \
    --start '^ (\s+) at \s ( org\.junit\. | \S+ \. reflect\.\S+\.invoke )' \
    --end '^ (?! \s+ at \s )' \
    --repl '\1...' \
match -v '^\s+at \S+\.(rethrowAs|translateTo)IOException' \
sub-paths --include '{src,tst}/**/*.java' --modules-skip 1 \
sub -X '^( \s+ at \s+ (?! .+ \.\. | com\.example\. ) .*? ) \( .*' '\1' \
sub -X '^( \s+ at \s+ com\.example\. .*? ) \ ~\[ .*' '\1' \
sub -X '
    (?P<pre> \s+ at \s .*)
    (?P<cls> \w+ )
    (?P<mid> .* \( )
    (?P=cls) \.java
    (?P<suf> : .* )
    ' \
    '\g<pre>\g<cls>\g<mid>\g<suf>'
```

… shortens this 76 line traceback:

```
12:34:56.789 [main] ERROR com.example.someproject.somepackage.ThingDoer - exception␣
↪while notifying done listener
java.lang.RuntimeException: listener failed
    at com.example.someproject.somepackage.ThingDoerTest$DummyListener.
↪onThingDone(ThingDoerTest.java:420) ~[tests/:?]
    at com.example.someproject.somepackage.ThingDoer.doThing(ThingDoer.java:69) ~
↪[library/:?]
    at com.example.otherproject.Framework.doAllTheThings(Framework.java:1066) ~[example-
↪otherproject-2.0.jar:2.0]
    at com.example.someproject.somepackage.ThingDoerTest.listenerException(ThingDoerTest.
↪java:666) ~[tests/:?]
    at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[?:?]
    at jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
↪java:62) ~[?:?]
```

(continues on next page)

(continued from previous page)

```
    ...
    ... 60+ more lines of JUnit stuff we don't really care about ...
    ...
12:34:56.999 [main] INFO done
```

... to just:

```
12:34:56.789 [main] ERROR ..ThingDoer - exception while notifying done listener
java.lang.RuntimeException: listener failed
    at ..ThingDoerTest$DummyListener.onThingDone(:420) ~[tests/:?]
    at ..ThingDoer.doThing(:69) ~[library/:?]
    at com.example.otherproject.Framework.doAllTheThings(:1066)
    at ..ThingDoerTest.listenerException(:666) ~[tests/:?]
    ...
12:34:56.999 [main] INFO done
```

Let's break that `linesieve` command down a bit:

- The `span` gets rid of all the traceback lines coming from JUnit.
- The `match -v` skips some usually useless lines from stack traces.
- The `sub-paths` shortens and highlights the names of classes in the current project; `com.example.someproject.somepackage.ThingDoer` becomes `..ThingDoer` (presumably that's enough info to open the file).
- The first `sub` gets rid of line numbers and JAR names for everything that is not either in the current project or in another `com.example.` package.
- The second `sub` gets rid of JAR names for things in other `com.example.` packages.
- The third `sub` gets rid of the source file name; `..ThingDoer.doThing(ThingDoer.java:69)` becomes `..ThingDoer.doThing(:69)` (the file name matches the class name).

5.2.2 Apache Ant output

Let's look at why `linesieve` was born in the first place – cleaning up Apache Ant output.

We'll use Ant's own test output as an example, since it builds itself.

Running a single test with:

```
ant junit-single-test -Djunit.testcase=org.apache.tools.ant.ProjectTest
```

... produces 77 lines of output:

```
Buildfile: /Users/lemon/code/ant/build.xml

check-optional-packages:

prepare:

compile:

compile-jdk9+:

```

(continues on next page)

(continued from previous page)

```
build:
[delete] Deleting directory /Users/lemon/code/ant/build/classes/org/apache/tools/ant/
↪taskdefs/optional/junitlauncher/confined
        ... more lines

... more targets, until we get to the one that we care about

junit-single-test-only:
    [junit] WARNING: multiple versions of ant detected in path for junit
    [junit]          file:/Users/lemon/code/ant/build/classes/org/apache/tools/ant/
↪Project.class
    [junit]      and jar:file:/usr/local/Cellar/ant/1.10.12/libexec/lib/ant.jar!/org/
↪apache/tools/ant/Project.class
    [junit] Testsuite: org.apache.tools.ant.ProjectTest
    [junit] Tests run: 12, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 5.635 sec
        ... more lines

junit-single-test:

BUILD SUCCESSFUL
Total time: 12 seconds
```

If this doesn't look all that bad, try imagining what it looks like for a Serious Enterprise Project™.

Lots of output is indeed very helpful – if you're waiting minutes for the entire test suite to run, you want all the details in there, so you can debug failures without having to run it again.

However, it's not very helpful during development, when you only care about the thing you're working on *right now*. And it's doubly not helpful if you want to re-run the tests on each file update with something like entr.

This is where a `linesieve` wrapper script can help:

```
#!/bin/sh
linesieve \
    --section '^(\S+):$' \
    --success 'BUILD SUCCESSFUL' \
    --failure 'BUILD FAILED' \
show junit-batch \
show junit-single-test-only \
sub-cwd \
sub-paths --include 'src/main/**/*.java' --modules-skip 2 \
sub-paths --include 'src/tests/junit/**/*.java' --modules-skip 3 \
sub -s compile '^\s+\[javac?] ' '' \
push compile \
    match -v '^Compiling \d source file' \
    match -v '^Ignoring source, target' \
pop \
push junit \
    sub '^\s+\[junit] ?' '' \
    span -v \
        --start '^WARNING: multiple versions of ant' \
        --end '^Testsuite:' \
    match -v '^\s+at java\.\S+\.reflect\.' \
    match -v '^\s+at org.junit.Assert' \
```

(continues on next page)

(continued from previous page)

```
    span -v \
        --start '^\s+at org.junit.(runners|rules|internal)' \
        --end '^(?!\s+at )' \
pop \
sub -X '^( \s+ at \s+ (?! .+ \.\. ) .*? ) \( .*' '\1' \
sub -X '
    (?P<pre> \s+ at \s .*)
    (?P<cls> \w+ )
    (?P<mid> .* \( )
    (?P=cls) \.java
    (?P<suf> : .* )
    ' \
    '\g<pre>\g<cls>\g<mid>\g<suf>' \
sub --color -X '^( \w+ (\.\w+)+ (?= :\s ))' '\1' \
sub --color -X '(FAILED)' '\1' \
read-cmd ant "$@"
```

You can then call this instead of `ant`: `ant-wrapper.sh junit-single-test ...`.

Successful output looks like this (28 lines):

```
............
junit-single-test-only
Testsuite: ..ProjectTest
Tests run: 12, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 5.635 sec
------------- Standard Output ---------------
bar
------------- ---------------- ---------------
------------- Standard Error -----------------
bar
------------- ---------------- ---------------

Testcase: testResolveFileWithDriveLetter took 0.034 sec
    SKIPPED: Not DOS or Netware
Testcase: testResolveFileWithDriveLetter took 0.036 sec
Testcase: testInputHandler took 0.007 sec
Testcase: testAddTaskDefinition took 0.179 sec
Testcase: testTaskDefinitionContainsKey took 0.002 sec
Testcase: testDuplicateTargets took 0.05 sec
Testcase: testResolveRelativeFile took 0.002 sec
Testcase: testOutputDuringMessageLoggedIsSwallowed took 0.002 sec
Testcase: testDataTypes took 0.154 sec
Testcase: testDuplicateTargetsImport took 0.086 sec
Testcase: testNullThrowableMessageLog took 0.002 sec
Testcase: testTaskDefinitionContainsValue took 0.002 sec
Testcase: testResolveFile took 0.001 sec

.
BUILD SUCCESSFUL
```

… “failure” output looks like this (34 lines):

```
............
```

(continues on next page)

(continued from previous page)

```
junit-single-test-only
Testsuite: ..ProjectTest
Tests run: 12, Failures: 1, Errors: 0, Skipped: 1, Time elapsed: 5.638 sec
------------- Standard Output ---------------
bar
------------- ---------------- ---------------
------------- Standard Error -----------------
bar
------------- ---------------- ---------------

Testcase: testResolveFileWithDriveLetter took 0.033 sec
    SKIPPED: Not DOS or Netware
Testcase: testResolveFileWithDriveLetter took 0.035 sec
Testcase: testInputHandler took 0.005 sec
    FAILED
expected null, but was:<..DefaultInputHandler@61dc03ce>
junit.framework.AssertionFailedError: expected null, but was:<..
↪DefaultInputHandler@61dc03ce>
    at ..ProjectTest.testInputHandler(:254)

Testcase: testAddTaskDefinition took 0.182 sec
Testcase: testTaskDefinitionContainsKey took 0.003 sec
Testcase: testDuplicateTargets took 0.043 sec
Testcase: testResolveRelativeFile took 0.001 sec
Testcase: testOutputDuringMessageLoggedIsSwallowed took 0.003 sec
Testcase: testDataTypes took 0.161 sec
Testcase: testDuplicateTargetsImport took 0.088 sec
Testcase: testNullThrowableMessageLog took 0.001 sec
Testcase: testTaskDefinitionContainsValue took 0.001 sec
Testcase: testResolveFile took 0.001 sec
Test ..ProjectTest FAILED

.
BUILD SUCCESSFUL
```

… and true failure due to a compile error looks like this (12 lines):

```
...
compile
.../Project.java:65: error: cannot find symbol
public class Project implements xResourceFactory {
                                ^
symbol: class xResourceFactory
.../Project.java:2483: error: method does not override or implement a method from a␣
↪supertype
    @Override
    ^
2 errors

BUILD FAILED
```

Breaking down the `linesieve` command (skipping the parts from the traceback example):

- `--section '^(S+):$'` tells `linesieve` sections start with a word followed by a colon.
- The `show`s hide all sections except specific ones.
- `--success` and `--failure` tell `linesieve` to exit when encountering one of these patterns. Note that the failing section is shown regardless of `show`.
- `sub-cwd` makes absolute paths in the working directory relative.
- The `-s compile` option passed to `sub` applies it only to sections matching `compile`.
- `push compile` applies all the following filters, until `pop`, only to sections matching `compile`.
- The last two `sub --color ... '1'` color dotted words followed by a colon at the beginning of the line (e.g. `junit.framework.AssertionFailedError:`), and `FAILED` anywhere in the input.
- Finally, `read-cmd` executes a command and uses its output as input.

5.3 Changelog

5.3.1 Version 1.1

Unreleased

- Support operating on multi-line records (`linesieve --record-start`). (#1)
- Allow parsing structured data (`linesieve parse`). (#2)
- Allow piping lines to an arbitrary command (`linesieve pipe`).
- Allow hiding sections (`linesieve hide`).

5.3.2 Version 1.0

Released 2023-04-17

- First stable release.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

C

F

L

P

R