FileCheck.py

Release 0.0.24

Stanislav Pankevich

Jul 10, 2024

CONTENTS

CHAPTER

ONE

WHAT IS FILECHECK.PY

FileCheck.py is a Python port of **LLVM's FileCheck**, "flexible pattern matching file verifier" [*1*].

LLVM's FileCheck is a command-line tool written in C++ which is developed and maintained as part of LLVM source code [*2*].

FileCheck is most often used in a combination with another tool called **LIT (LLVM Integrated Tester)** [*3*]. LIT is a test runner that runs commands from the test files, FileCheck is used as a test matcher tool that checks output of the commands run by LIT.

1.1 2024-07-10 STATUS UPDATE - PROJECT DISCONTINUED

The project has been discontinued in favor of another project: antonlydike/filecheck, where the developers are aiming to achieve greater compatibility with the upstream LLVM FileCheck and add more features that this project was lacking.

The filecheck PyPI package has also been transferred to the owner of antonlydike/filecheck.

1.2 Why Python port?

There are software projects that would benefit from having a suite of LIT-based integration tests. Mull mutation testing system is one example [*4*].

The problem is that you have to build FileCheck from LLVM sources which is not a trivial task for 1) people who are not familiar with the LLVM infrastructure and 2) Python-based projects which would prefer to not depend on anything C or C++-related including building dependencies from LLVM sources.

The option of having pre-compiled binaries is a workaround, but it is not always possible to keep third-party binary artifacts in source code (see https://github.com/doorstop-dev/doorstop/pull/431#issuecomment-549237579).

Note: FileCheck.py is not intended to be a replacement for LLVM's FileCheck in any way. See *Roadmap*.

1.3 What's next?

If you are new to FileCheck and integration testing using LIT, we recommend you to read the tutorials: *Tutorial: Hello World* and *Tutorial: LIT and FileCheck*.

If you know how FileCheck and LIT work, you can check out the status of the port on the *Roadmap* page.

1.4 Links

[1] FileCheck - Flexible pattern matching file verifier

[2] llvm/utils/FileCheck/FileCheck.cpp

[3] lit - LLVM Integrated Tester

[4] Mull mutation testing system

CHAPTER

TWO

INSTALLATION

FileCheck.py is a pip package:

```
pip install filecheck
```

LIT is also a pip package:

```
pip install lit
```

2.1 2024-07-10 STATUS UPDATE - PROJECT DISCONTINUED

The project has been discontinued in favor of another project: antonlydike/filecheck, where the developers are aiming to achieve greater compatibility with the upstream LLVM FileCheck and add more features that this project was lacking.

The filecheck PyPI package has also been transferred to the owner of antonlydike/filecheck.

CHAPTER

THREE

TUTORIAL: HELLO WORLD

This tutorial assumes that you have `filecheck` installed and have it available in your PATH:

```
$ filecheck
/usr/local/bin/filecheck
<check-file> not specified
```

FileCheck can be seen as an improved version of `grep` that makes automated testing of the command-line tools easier. FileCheck reads an input and scans it against a number of checks which can be substring or regex matches. If all check matches are found, `FileCheck` exits with an exit code `0`. When a check without a match found, the FileCheck prints an error message and exits with an exit code 1.

The FileCheck program expects a path to a check file and a number of optional option arguments:

```
$ filecheck <check-file> [<options>]
```

3.1 Matching strings

Create new file `hello-world.check` with the following contents:

```
CHECK: Hello world
```

Now we can provide a valid input to `filecheck` which will match it against the check file:

```
$ echo "Hello world" | filecheck hello-world.check
/usr/local/bin/filecheck
$ echo $?
0
```

Note: By convention, original LLVM's `FileCheck` always prints a full path to its executable and `FileCheck.py` follows this convention.

If we provide an invalid output we will see an error message:

```
$ echo "What is FileCheck" | filecheck hello-world.check
/usr/local/bin/filecheck
examples/hello-world.check:1:8: error: CHECK: expected string not found in input
CHECK: Hello world
       ^
<stdin>:1:1: note: scanning from here
What is FileCheck
```

(continues on next page)

(continued from previous page)

```
^
$ echo $?
1
```

It is as easy as that!

3.2 Matching regular expressions

`FileCheck` also supports regex matching using special `{{ }}` syntax:

Create a new file `hello-world-regex.check` with the following contents:

```
CHECK: {{^Hello world$}}
```

Let's run it with a valid input:

```
echo "Hello world" | filecheck examples/hello-world-regex.check
/usr/local/bin/filecheck
$ echo $?
0
```

With invalid input:

```
$ echo "Hello world Hello world" | filecheck examples/hello-world-regex.check
/usr/local/bin/filecheck
examples/hello-world-regex.check:1:8: error: CHECK: expected string not found in input
CHECK: {{^Hello world$}}
       ^
<stdin>:1:1: note: scanning from here
Hello world Hello world
^
$ echo $?
1
```

3.3 What's next?

FileCheck is rarely used alone. The main use case for FileCheck is to serve as an string matching tool when it is used in a combination with LIT (LLVM Integrated Tester) and this is what our next tutorial is about. Don't stop here and check it out right away: *Tutorial: LIT and FileCheck*.

CHAPTER

FOUR

TUTORIAL: LIT AND FILECHECK

This tutorial assumes that you have installed `lit` and `filecheck` and have them available in your PATH:

```
$ filecheck
/usr/local/bin/filecheck
<check-file> not specified
```

```
$ lit
...
lit: error: No inputs specified
```

4.1 Minimal example

When writing LIT/FileCheck tests it is common, but not required, to combine LIT's `RUN` commands and FileCheck's `CHECK` assertions in a single file.

Let's create a file `minimal.itest` with the following contents

```
RUN: echo "Hello world" | filecheck %s
CHECK: Hello world
```

LIT expects a config file in a directory from which it is run. Let's create a minimal one called `lit.cfg`:

```
import lit.formats
config.test_format = lit.formats.ShTest("0")
```

Now we can run `lit`:

```
lit minimal.itest
```

```
-- Testing: 1 tests, single process --
PASS: <unnamed> :: test.itest (1 of 1)
Testing Time: 0.10s
  Expected Passes    : 1
```

4.2 Example: testing output of C program

4.2.1 Passing test

Test file `01-pass.c`:

```
/**
RUN: clang %s -o %S/hello-world && %S/hello-world | filecheck %s
CHECK: Hello world
 */

#include <stdio.h>
int main() {
  printf("Hello world\n");
  return 0;
}
```

Command:

```
$ lit 01-pass.c
```

Output:

```
-- Testing: 1 tests, single process --
PASS: <unnamed> :: 01-pass.c (1 of 1)
Testing Time: 0.10s
  Expected Passes    : 1
```

4.2.2 Failing test

Test file `02-fail.c`:

```
/**
RUN: clang %s -o %S/hello-world && %S/hello-world | filecheck %s
CHECK: Wrong line
 */

#include <stdio.h>
int main() {
  printf("Hello world\n");
  return 0;
}
```

Command:

```
$ lit 02-fail.c
```

Output:

```
-- Testing: 1 tests, single process --
FAIL: <unnamed> :: 02-fail.c (1 of 1)
Testing Time: 0.11s
```

(continues on next page)

(continued from previous page)

```
********************
Failing Tests (1):
    <unnamed> :: 02-fail.c

  Unexpected Failures: 1
```

The verbose version:

```
$ lit -v 02-fail.c
```

Produces more output:

```
-- Testing: 1 tests, single process --
FAIL: <unnamed> :: 02-fail.c (1 of 1)
******************** TEST '<unnamed> :: 02-fail.c' FAILED ********************
02-fail.c:3:8: error: CHECK: expected string not found in input
CHECK: Wrong line
       ^
<stdin>:1:1: note: scanning from here
Hello world
...
********************
Testing Time: 0.11s
********************
Failing Tests (1):
    <unnamed> :: 02-fail.c

  Unexpected Failures: 1
```

CHAPTER

FIVE

CHECK COMMANDS

If you are new to FileCheck, please make sure you have read the tutorials: *Tutorial: Hello World* and *Tutorial: LIT and FileCheck* because they show how FileCheck is used: standalone and in combination with LIT.

For all of the examples below, please note:

- `.check` extension is chosen arbitrarily. FileCheck can work with any file names.

5.1 CHECK

`CHECK` command means that a given string or a regular expression must be present in input provided to FileCheck.

Create a new file `CHECK.check` with the following contents:

```
CHECK: String1
CHECK: String2
CHECK: String3
```

Valid input results in the exit code `0` that indicates success:

```
printf "String1\nString2\nString3" | filecheck CHECK.check
/Users/Stanislaw/.pyenv/versions/3.5.0/bin/filecheck
```

Invalid input results in the exit code `1` and error message:

```
echo "String1" | filecheck CHECK.check
/Users/Stanislaw/.pyenv/versions/3.5.0/bin/filecheck
CHECK.check:2:8: error: CHECK: expected string not found in input
CHECK: String2
       ^
<stdin>:1:8: note: scanning from here
String1
       ^
```

5.1.1 Order of matching

CHECK commands are checked one after another. If a CHECK string is not found in output, FileCheck exits with error immediately.

Create a new file `order-of-matching.check` with the following contents:

```
CHECK: String1
CHECK: String2
CHECK: String3
```

And run with invalid input:

```
echo "String1" | filecheck order-of-matching.check
...
order-of-matching.check:2:8: error: CHECK: expected string not found in input
CHECK: String2
       ^
<stdin>:1:8: note: scanning from here
String1
       ^
```

5.2 CHECK-NOT

`CHECK-NOT` is the opposite of `CHECK`: a given string or a regular expression must not be present in input provided to FileCheck.

5.2.1 Example

`CHECK-NOT.check` file:

```
CHECK-NOT: String1
CHECK-NOT: String2
CHECK-NOT: String3
```

```
$ echo "String3" | filecheck CHECK-NOT.check
filecheck
CHECK-NOT.check:3:12: error: CHECK-NOT: excluded string found in input
CHECK-NOT: String3
           ^
<stdin>:1:1: note: found here
String3
^~~~~~
```

5.3 CHECK-NEXT

`CHECK-NEXT` command means that a given string or a regular expression must be present in input provided to FileCheck. Additionally, there must be another check right before `CHECK-NEXT`, that has passed on the input line just before the current input line. `CHECK-NEXT` cannot be the first check in the check file.

Check file `CHECK-NEXT.check`:

```
CHECK: String1
CHECK-NEXT: String2
```

```
$ printf "String1\nString2" | filecheck CHECK-NEXT.check
...filecheck
$ echo $?
0
```

```
$ printf "String1\nfoo\nString2" | filecheck CHECK-NEXT.check
...filecheck
CHECK-NEXT.check:2:13: error: CHECK-NEXT: is not on the line after the previous match
CHECK-NEXT: String2
            ^
<stdin>:3:1: note: 'next' match was here
String2
^
<stdin>:1:8: note: previous match ended here
String1
       ^
<stdin>:2:1: note: non-matching line after previous match is here
foo
^
```

5.4 CHECK-EMPTY

`CHECK-EMPTY` command is used to match empty lines.

Consider the following check file:

```
CHECK: String1
CHECK-EMPTY:
CHECK: String2
```

In the following example, there is an empty line so the test will pass:

```
$ printf "String1\n\nString2" | filecheck CHECK-EMPTY.check
...filecheck
$ echo $?
0
```

If the empty line is removed, the test will fail:

```
$ printf "String1\nString2" | filecheck CHECK-EMPTY.check
...filecheck
```

(continues on next page)

(continued from previous page)

```
...CHECK-EMPTY.check:2:13: error: CHECK-EMPTY: expected string not found in input
CHECK-EMPTY:
            ^
<stdin>:2:1: note: scanning from here
String2
^
$ echo $?
1
```

CHAPTER

SIX

OPTIONS

6.1 strict-whitespace

When the `--strict-whitespace` option is not provided, FileCheck ignores differences between spaces and tabs. Additionally multiple spaces are ignored and treated as one space.

6.1.1 Example

The following check file `without--strict-whitespace.check`:

```
CHECK: String1   String2          String3
```

will pass on any of the following inputs:

```
printf "String1 String2 String3" | filecheck without--strict-whitespace.check
printf "String1  String2         String3" | filecheck without--strict-whitespace.check
printf "   String1\tString2\t\t\tString3 " | filecheck without--strict-whitespace.check
```

Adding `--strict-whitespace` disables this behavior.

6.2 match-full-lines

When the `--match-full-lines` option is not provided, FileCheck does not match full lines.

6.2.1 Example

The following check file `without--match-full-lines.check`:

```
CHECK: tring1
CHECK: ring2
CHECK: String3
```

will pass on the following input:

```
printf "String1\nString2\nString3" | filecheck without--match-full-lines.check
```

The `--match-full-lines` disables this behavior.

6.2.2 Strict mode

Additionally, when the `--strict-whitespace` option is also provided, FileCheck does not allow leading and trailing whitespaces.

The following input: `printf "String1\nString2\nString3"` will only be matched with the following check file:

```
CHECK:String1
CHECK:String2
CHECK:String3
```

Notice absence of spaces between `CHECK:` and the lines.

```
$ printf "String1\nString2\nString3" | filecheck strict-mode.check --strict-whitespace --
↪match-full-lines
...filecheck
$ echo $?
0
```

6.3 check-prefix

The `--check-prefix` option allows changing a default match keyword *CHECK* to an arbitrary keyword. This is useful when you want to test different behavior in the same file:

```
RUN: printf "String1" | %FILECHECK_EXEC %s --check-prefix STRING1
RUN: printf "String2" | %FILECHECK_EXEC %s --check-prefix STRING2
STRING1: String1
STRING2: String2
```

One usual case is testing of how a program behaves when it is run with or without a specific option.

6.4 implicit-check-not

The `--implicit-check-not` option adds implicit *CHECK-NOT* check that works on every input line.

FileCheck.py follows LLVM FileCheck in the following implementation details:

- The implicit checks are substring-matched i.e. their are `in` checks, not == checks.
- The implicit checks are case sensitive, so `error` check will not match `ERROR` in the input.
- The implicit check has lower priority than the positive *CHECK** checks, but it has higher priority than negative *CHECK-NOT* checks.
- To provide multiple implicit checks, duplicate the argument `--implicit-check-not <your check>` multiple times.

6.4.1 Example

A very useful application of this option is to add implicit `--implicit-check-not error --implicit-check-not warning` checks to make sure that the input never has lines that contain `error` or `warning` in them.

CHAPTER

SEVEN

EXTRA FEATURES FROM LLVM FILECHECK

7.1 Line number expression

It is often useful to check for a specific line number in your regular expression, relative to its location in the file. Hard-coding that number can make the test fragile – rearranging, adding, or deleting lines requires changing the expression. To solve this, FileCheck supports a variable for the current line number, `[[# @LINE ]]`, as well as simple offsets from this variable, e.g. `[[# @LINE + 4 ]]` or `[[# @LINE - 2 ]]`.

Example:

```
/**
RUN: gcc "%s" -o %S/line && %S/line | filecheck %s
*/

#include <stdio.h>
int main() {
  // CHECK: Hello from line [[# @LINE + 1 ]]
  printf("Hello from line %d\n", __LINE__);
  return 0;
}
```

See also LLVM FileCheck documentation for this feature.

CHAPTER

EIGHT

ROADMAP

8.1 2024-07-10 STATUS UPDATE - PROJECT DISCONTINUED

The project has been discontinued in favor of another project: antonlydike/filecheck, where the developers are aiming to achieve greater compatibility with the upstream LLVM FileCheck and add more features that this project was lacking.

The filecheck PyPI package has also been transferred to the owner of antonlydike/filecheck.

As described in *What is FileCheck.py*, FileCheck.py is only a Python port for LLVM's FileCheck. It is not intended to be a replacement for LLVM's FileCheck.

FileCheck.py is being tested against Mull mutation testing system. The first goal is to make FileCheck.py pass on Mull's current integration tests suite that uses only a very limited subset of FileCheck's features which are as follows:

- Commands (both substring and regex matching):
  - `CHECK`
  - `CHECK-NEXT`
  - `CHECK-NOT`
  - `CHECK-EMPTY`
- Options:
  - `--strict-whitespace`
  - `--match-full-lines`
  - `--check-prefix`

When the above features are implemented and considered stable, it might be a good idea to implement full FileCheck's contract and be 100% compatible with the C++ version.

There is a file in FileCheck.py's repository: FileCheck.pdf which has some rough implementation coverage: green color means something is implemented, yellow color means that the text is not relevant to implementation, no highlighting means "still to be implemented".

8.2 Notes on implementation

The implementation is done against the latest FileCheck - Flexible pattern matching file verifier document. The plan is to skip looking into the LLVM's implementation of FileCheck, unless some very advanced implementation details are encountered.

It a nice programming exercise of implementing something from a spec but we also want to make FileCheck.py conform to the same contract this is why FileCheck.py is tested using LIT and LLVM's FileCheck:

- First, the LLVM FileCheck is run against a FileCheck.py test suite based on LIT/FileCheck.
- Second, FileCheck.py is run on the same tests to make sure that it has the same behavior as the LLVM FileCheck.

CHAPTER

NINE

KNOWN ISSUES

9.1 Unintuitive behavior of CHECK-NOT

9.1.1 A failing CHECK has a higher precedence than a failing CHECK-NOT

A failing *CHECK* has a higher precedence than a failing *CHECK-NOT* even if *CHECK-NOT* appears first in the check file. If this happens, the output is related to the failing *CHECK*, not the failing *CHECK-NOT*.

Input:

```
String1
Stringggg2
```

Check file:

```
CHECK-NOT:String1
CHECK:String2
```

Result:

```
/Users/Stanislaw/.pyenv/shims/filecheck
filecheck.check:2:7: error: CHECK: expected string not found in input
CHECK:String2
      ^
<stdin>:1:1: note: scanning from here
String1
^
```