1 ShellSpec

ShellSpec is a **full-featured BDD unit testing framework** for dash, bash, ksh, zsh and **all POSIX shells** that provides first-class features such as code coverage, mocking, parameterized test, parallel execution and more. It was developed as a dev/test tool for **cross-platform shell scripts and shell script libraries**. ShellSpec is a new modern testing framework released in 2019, but it's already stable enough. With lots of practical CLI features and simple yet powerful syntax, it provides you with a fun shell script test environment.

Thank you for your interest in ShellSpec. Please visit the official website[1] to know the impressive features!

Let's have fun testing your shell scripts! (Try Online Demo[2] on your browser).

Latest Update.

See CHANGELOG.md

NOTE: This documentation contains unreleased features. Check them in the changelog.

---

2 Tutorial

Just create your project directory and run `shellspec --init` to setup your project

```
 Create your project directory, for example "hello".
$ mkdir hello
$ cd hello

 Initialize
$ shellspec --init
  create   .shellspec
  create   spec/spec_helper.sh

 Write your first specfile (of course you can use your favorite editor)
$ cat<<'HERE'>spec/hello_spec.sh
Describe 'hello.sh'
  Include lib/hello.sh
  It 'says hello'
    When call hello ShellSpec
    The output should equal 'Hello ShellSpec!'
  End
End
HERE

 Create lib/hello.sh
$ mkdir lib
$ touch lib/hello.sh

 It will fail because the hello function is not implemented.
$ shellspec

 Write hello function
$ cat<<'HERE'>lib/hello.sh
hello() {
  echo "Hello ${1}!"
}
HERE

 It will success!
$ shellspec
```

[1]https://shellspec.info/
[2]https://shellspec.info/demo

3 ShellSpec CLI

3.1 runs specfile using `/bin/sh` by default

ShellSpec CLI runs specfiles with the shell running `shellspec`. Usually it is `/bin/sh` that is the shebang of `shellspec`. If you run `bash shellspec`, it will be bash. `Include` files from specfile will be executed in the same shell as well.

The purpose of this specification is to allow ShellSpec to easily change multiple types of shells and enable the development of cross-platform shell scripts that support multiple shells and environments.

If you want to test with a specific shell, use the `-s` (`--shell`) option. You can specify the default shell in the `.shellspec` file.

NOTE: If you execute a **shell script file** (not a shell function) from within the specfile, its shebang will be respected. Because in that case, it will be run as an external command. The `-s` (`--shell`) option also has no effect. If you are testing a external shell script file, you can use `When run script` or `When run source`. These ignore the shebang of external shell script file and run in the same shell that runs specfile.

3.2 command options

NOTE: Since version 0.28.0, getoptions[3] is used to parse options, so all POSIX and GNU compatible option syntax can be used. For example, you can abbreviate a long option.

See more info: ShellSpec CLI[4]

```
$ shellspec -h
Usage: shellspec [ -c ] [-C <directory>] [options...] [files or directories...]

  Using + instead of - for short options causes reverses the meaning

    -s, --shell SHELL               Specify a path of shell [default: "auto" (the shell running shellspec)]
        --require MODULE            Require a MODULE (shell script file)
    -O, --options PATH              Specify the path to an additional options file
    -I, --load-path PATH            Specify PATH to add to $SHELLSPEC_LOAD_PATH (may be used more than once)
        --helperdir DIRECTORY       The directory to load helper files (spec_helper.sh, etc) [default: "spec"
        --path PATH                 Set PATH environment variable at startup
        --{no-}sandbox              Force the use of the mock instead of the actual command
        --sandbox-path PATH         Make PATH the sandbox path instead of empty [default: empty]
        --execdir @LOCATION[/DIR]    Specify the execution directory of each specfile | [default: @project]
    -e, --env NAME[=VALUE]           Set environment variable
        --env-from ENV-SCRIPT       Set environment variable from shell script file
    -w, --{no-}warning-as-failure   Treat warning as failure [default: enabled]
        --{no-}fail-fast[=COUNT]    Abort the run after first (or COUNT) of failures [default: disabled]
        --{no-}fail-no-examples     Fail if no examples found [default: disabled]
        --{no-}fail-low-coverage    Fail on low coverage [default: disabled]
        --failure-exit-code CODE    Override the exit code used when there are failing specs [default: 101]
        --error-exit-code CODE      Override the exit code used when there are fatal errors [default: 102]
    -p, --{no-}profile              Enable profiling and list the slowest examples [default: disabled]
        --profile-limit N           List the top N slowest examples [default: 10]
        --{no-}boost                Increase the CPU frequency to boost up testing speed [default: disabled]
        --log-file LOGFILE          Log file for %logger directive and trace [default: "/dev/tty"]
        --tmpdir TMPDIR             Specify temporary directory [default: $TMPDIR, $TMP or "/tmp"]
        --keep-tmpdir               Do not cleanup temporary directory [default: disabled]

  The following options must be specified before other options and cannot be specified in the options file

    -c, --chdir                     Change the current directory to the first path of arguments at the start
    -C, --directory DIRECTORY       Change the current directory at the start

  **** Execution ****

    -q, --{no-}quick                Run not-passed examples if it exists, otherwise run all [default: disable
    -r, --repair, --only-failures   Run failure examples only (Depends on quick mode)
    -n, --next-failure              Run failure examples and abort on first failure (Depends on quick mode)
    -j, --jobs JOBS                 Number of parallel jobs to run [default: 0 (disabled)]
        --random TYPE[:SEED]         Run examples by the specified random type | <[none]> [specfiles] [examp
    -x, --xtrace                    Run examples with trace output of evaluation enabled [default: disabled]
```

[3] https://github.com/ko1nksm/getoptions
[4] docs/cli.md

```
    -X, --xtrace-only                 Run examples with trace output only enabled [default: disabled]
        --dry-run                     Print the formatter output without running any examples [default: disable

  **** Output ****

        --{no-}banner                 Show banner if exist "<HELPERDIR>/banner[.md]" [default: enabled]
        --reportdir DIRECTORY         Output directory of the report [default: "report"]
    -f, --format FORMATTER            Choose a formatter for display | <[p]> [d] [t] [j] [f] [null] [debug]
    -o, --output FORMATTER            Choose a generator(s) to generate a report file(s) [default: none]
        --{no-}color                  Enable or disable color [default: enabled if the output is a TTY]
        --skip-message VERBOSITY      Mute skip message | <[verbose]> [moderate] [quiet]
        --pending-message VERBOSITY Mute pending message | <[verbose]> [quiet]
        --quiet                       Equivalent of --skip-message quiet --pending-message quiet
        --(show|hide)-deprecations    Show or hide deprecations details [default: show]

  **** Ranges / Filters / Focus ****

    You can run selected examples by specified the line numbers or ids

      shellspec path/to/a_spec.sh:10   # Run the groups or examples that includes lines 10
      shellspec path/to/a_spec.sh:@1-5 # Run the 5th groups/examples defined in the 1st group
      shellspec a_spec.sh:10:@1:20:@2  # You can mixing multiple line numbers and ids with join by ":"

    -F, --focus                       Run focused groups / examples only
    -P, --pattern PATTERN             Load files matching pattern [default: "*_spec.sh"]
    -E, --example PATTERN             Run examples whose names include PATTERN
    -T, --tag TAG[:VALUE]              Run examples with the specified TAG
        --default-path PATH           Set the default path where looks for examples [default: "spec"]

    You can specify the path recursively by prefixing it with the pattern "*/" or "**/"
      (This is not glob patterns and requires quotes. It is also available with --default-path)

      shellspec "*/spec"               # The pattern "*/" matches 1 directory
      shellspec "**/spec"              # The pattern "**/" matches 0 and more directories
      shellspec "*/*/**/test_spec.sh"  # These patterns can be specified multiple times

    -L, --dereference                 Dereference all symlinks in in the above pattern [default: disabled]

  **** Coverage ****

        --covdir DIRECTORY            Output directory of the Coverage Report [default: coverage]
        --{no-}kcov                   Enable coverage using kcov [default: disabled]
        --kcov-path PATH              Specify kcov path [default: kcov]
        --kcov-options OPTIONS        Additional Kcov options (coverage limits, coveralls id, etc)

  **** Utility ****

        --init [TEMPLATE...]           Initialize your project with ShellSpec | [spec] [git] [hg] [svn]
        --gen-bin [@COMMAND...]        Generate test support commands in "<HELPERDIR>/support/bin"
        --count                       Count the number of specfiles and examples
        --list LIST                   List the specfiles/examples | [specfiles] [examples(:id|:lineno)]
        --syntax-check                Syntax check of the specfiles without running any examples
        --translate                   Output translated specfile
        --task [TASK]                  Run the TASK or Show the task list if TASK is not specified
        --docker DOCKER-IMAGE         Run tests in specified docker image (EXPERIMENTAL)
    -v, --version                     Display the version
    -h, --help                        -h: short help, --help: long help
```

4 Project directory

All specfiles for ShellSpec must be under the project directory. The root of the project directory must have a `.shellspec` file. This file is that specify the default options to be used in the project, but an empty file is required even if the project has no options.

NOTE: The `.shellspec` file was described in the documentation as a required file for some time, but ShellSpec worked without it. Starting with version 0.28.0, this file is checked and will be required in future versions.

You can easily create the necessary files by executing the `shellspec --init` command in an existing directory.

4.1 Typical directory structure

This is the typical directory structure. Version 0.28.0 allows many of these to be changed by specifying options, supporting a more flexible directory structure[5].

```
<PROJECT-ROOT> directory
+- .shellspec                           [mandatory]
+- .shellspec-local                     [optional] Ignore from version control
+- .shellspec-quick.log                 [optional] Ignore from version control
+- report/                              [optional] Ignore from version control
+- coverage/                            [optional] Ignore from version control
|
+- bin/
|   +- your_script1.sh
|              :
+- lib/
|   +- your_library1.sh
|              :
|
+- spec/ (also <HELPERDIR>)
|   +- spec_helper.sh                   [recommended]
|   +- banner[.md]                      [optional]
|   +- support/                         [optional]
|   |
|   +- bin/
|   |   +- your_script1_spec.sh
|   |              :
|   +- lib/
|   |   +- your_library1_spec.sh
```

4.2 Options file

To change the default options for the `shellspec` command, create options file(s). Files are read in the order shown below, options defined last take precedence.

1. `$XDG_CONFIG_HOME/shellspec/options`
2. `$HOME/.shellspec-options` (version >= 0.28.0) or `$HOME/.shellspec` (deprecated)
3. `<PROJECT-ROOT>/.shellspec`
4. `<PROJECT-ROOT>/.shellspec-local` (Do not store in VCS such as git)

Specify your default options with `$XDG_CONFIG_HOME/shellspec/options` or `$HOME/.shellspec-options`. Specify default project options with `.shellspec` and overwrite to your favorites with `.shellspec-local`.

4.3 `.shellspec` - project options file

Specifies the default options to use for the project.

4.4 `.shellspec-local` - user custom options file

Override the default options used by the project with your favorites.

4.5 `.shellspec-basedir` - specfile execution base directory

Used to specify the directory in which the specfile will be run. See directory structure[6] or `--execdir` option for details.

[5]docs/directory_structure.md
[6]docs/directory_structure.md

4.6 `.shellspec-quick.log` - quick execution log

If this file is present, Quick mode will be enabled and the log of Quick execution will be recorded. It created automatically when `--quick` option is specified. If you want to turn off Quick mode, delete it.

4.7 `report/` - report file directory

The output location for reports generated by the `--output` or `--profile` options. This can be changed with the `--reportdir` option.

4.8 `coverage/` - coverage reports directory

The output location for coverage reports. This can be changed with the `--covdir` option.

4.9 `spec/` - (default) specfiles directory

By default, it is assumed that all specfiles are store under the `spec` directory, but it is possible to create multiple directories with different names.

NOTE: In Version <= 0.27.x, the `spec` directory was the only directory that contained the specfiles.

4.10 <HELPERDIR> (default: `spec/`)

The directory to store `spec_helper.sh` and other files. By default, the `spec` directory also serves as `HELPERDIR` directory, but you can change it to another directory with the `--helperdir` option.

4.10.1 `spec_helper.sh` - (default) helper file for specfile

The `spec_helper.sh` is loaded to specfile by the `--require spec_helper` option. This file is used to define global functions, initial setting for examples, custom matchers, etc.

4.10.2 `banner[.md]` - banner file displayed at test execution

If the file `<HELPERDIR>/banner` or `<HELPERDIR>/banner.md` exists, Display a banner when the `shellspec` command is executed. It can be used to display information about the tests. The `--no-banner` option can be used to disable this behavior.

4.10.3 `support/` - directory for support files

This directory can be used to store files such as custom matchers and tasks.

4.10.3.1 `bin` - directory for support commands This directory is used to store support commands.

5 Specfile (test file)

In ShellSpec, you write your tests in a specfile. By default, specfile is a file ending with `_spec.sh` under the `spec` directory.

The specfile is executed using the `shellspec` command, but it can also be executed directly. See self-executable specfile for details.

5.1 Example

```
Describe 'lib.sh' # example group
  Describe 'bc command'
    add() { echo "$1 + $2" | bc; }

    It 'performs addition' # example
      When call add 2 3 # evaluation
      The output should eq 5  # expectation
    End
  End
End
```

The best place to learn how to write a specfile is the examples/spec directory. You should take a look at it ! *(Those examples include failure examples on purpose.)*

5.2 About DSL

ShellSpec has its own DSL to write tests. It may seem like a distinctive code because DSL starts with a capital letter, but the syntax is compatible with shell scripts, and you can embed shell functions and use ShellCheck[7] to check the syntax.

You may feel rejected by this DSL, but It starts with a capital letter to avoid confusion with the command, and it does a lot more than you think, such as realizing scopes, getting shell-independent line numbers, and workarounds for bugs in some shells.

5.3 Execution directory

Since version 0.28.0, the current directory when run a specfile is the project root directory by default. Even if you run a specfile from a any subdirectory in the project directory, It is the project root directory. Before 0.27.x, it was the current directory when the `shellspec` is executed.

You can change this directory (location) by using the `--execdir @LOCATION[/DIR]` option. You can choose from the following locations and specify a path relative to the location if necessary. However, you cannot specify a directory outside the project directory.

- @project Where the ".shellspec" file is located (project root) [default]
- @basedir Where the ".shellspec" or ".shellspec-basedir" file is located
- @specfile Where the specfile is located

If basedir is specified, the parent directory is searched from the directory containing the specfile to be run, and the first directory where `.shellspec-basedir` or `.shellspec` is found is used as the execution directory. This is useful if you want to have a separate directory for each utilities (command) you want to test.

NOTE: You will need to change under the project directory or use the `-c` (`--chdir`) or `-C` (`--directory`) option before running specfile.

5.4 Embedded shell scripts

You can embed shell function (or shell script code) in the specfile. This shell function can be used for test preparation and complex testing.

Note that the specfile implements the scope using subshell. Shell functions defined in the specfile can only be used within blocks (e.g. `Describe`, `It`, etc).

If you want to use a global function, you can define it in `spec_helper.sh`.

5.5 Translation process

The specfile will not be executed directly by the shell, but will be translated into a regular shell script and output to a temporary directory (default: `/tmp`) before being executed.

The translation process is simple in that it only replaces forward-matched words (DSLs), with a few exceptions. If you are interested in the translated code, you can see with `shellspec --translate`.

[7]https://github.com/koalaman/shellcheck

6 DSL syntax

6.1 Basic structure

6.1.1 `Describe`, `Context`, `ExampleGroup` - example group block

`ExampleGroup` is a block for grouping example groups or examples. `Describe` and `Context` are alias for `ExampleGroup`. It can be nested and they can contain example groups or examples.

```
Describe 'is example group'
  Describe 'is nestable'
    ...
  End

  Context 'is used to facilitate understanding depending on the context'
    ...
  End
End
```

The example groups can be optionally tagged. See Tagging for details.

```
Describe 'is example group' tag1:value1 tag2:value2 ...
```

6.1.2 `It`, `Specify`, `Example` - example block

`Example` is a block for writing evaluation and expectations. `It` and `Specify` are alias for `Example`.

An example is composed by up to one evaluation and multiple expectations.

```
add() { echo "$1 + $2" | bc; }

It 'performs addition'          # example
  When call add 2 3             # evaluation
  The output should eq 5        # expectation
  The status should be success  # another expectation
End
```

The examples can be optionally tagged. See Tagging for details.

```
It 'performs addition' tag1:value1 tag2:value2 ...
```

6.1.3 `Todo` - one liner empty example

`Todo` is the same as the empty example and is treated as pending example.

```
Todo 'will be used later when we write a test'

It 'is an empty example, the same as Todo'
End
```

6.1.4 `When` - evaluation

Evaluation executes shell function or command for verification. Only one evaluation can be defined for each example and also can be omitted.[8]

6.1.4.1 `call` - call a shell function (without subshell)

It calls a function without subshell. Practically, it can also run commands.

```
When call add 1 2 # call `add` shell function with two arguments.
```

6.1.4.2 `run` - run a command (in a subshell)

It runs a command within subshell. Practically, it can also call a shell function. The command does not have to be a shell script. (NOTE: This does not support coverage measurement.)

```
When run touch /tmp/foo # run `touch` command.
```

Some commands below are specially handled by ShellSpec. The execution of a command in a `When [call|run]` evaluation clause can be further refined by adding one of the keywords, `[command|script|source]`:

`command` - runs an external command It runs a command, respecting shebang. It can not call shell function. The command does not have to be a shell script. (NOTE: This does not support coverage measurement.)

```
When run command touch /tmp/foo # run `touch` command.
```

`script` - runs a shell script It runs a shell script, ignoring shebang. The script has to be a shell script. It will be executed in another instance of the same shell as the current shell.

```
When run script my.sh # run `my.sh` script.
```

`source` - runs a script by `.` (dot) command It sources a shell script, ignoring its shebang. The script has to be a shell script. It is similar to `run script`, but with some differences. Unlike `run script`, function-based mock is available.

```
When run source my.sh # source `my.sh` script.
```

6.1.4.3 About executing aliases

If you want to execute aliases, you need a workaround using `eval`.

```
alias alias-name='echo this is alias'
When call alias-name # alias-name: not found

 eval is required
When call eval alias-name

 When using embedded shell scripts
foo() { eval alias-name; }
When call foo
```

6.1.5 `The` - Expectation

Expectation begins with `The` which does the verification. The basic syntax is as follows:

```
The output should equal 4
```

Use `should not` for the opposite verification.

```
The output should not equal 4
```

6.1.5.1 Subjects

The subject is the target of the verification.

```
The output should equal 4
      |
      +-- subject
```

There are `output` (`stdout`), `error` (`stdout`), `status`, `variable`, `path`, etc. subjects.

Please refer to the Subjects[9] for more details.

[8] See `docs/references.md` for more details of Evaluation.

[9] docs/references.md#subjects

6.1.5.2 Modifiers

The modifier concretizes the target of the verification (subject).

```
The line 2 of output should equal 4
      |
      +-- modifier
```

The modifiers are chainable.

```
The word 1 of line 2 of output should equal 4
```

If the modifier argument is a number, you can use an ordinal numeral instead of a number.

```
The first word of second line of output should equal 4
```

There are `line`, `word`, `length`, `contents`, `result`, etc. modifiers. The `result` modifier is useful for making the result of a user-defined function the subject.

Please refer to the Modifiers[10] for more details.

6.1.5.3 Matchers

The matcher is the verification.

```
The output should equal 4
                   |
                   +-- matcher
```

There are many matchers such as string matcher, status matcher, variable matchers and stat matchers. The `satisfy` matcher is useful for verification with user-defined function.

Please refer to the Matchers[11] for more details.

6.1.5.4 Language chains

ShellSpec supports *language chains* like chai.js[12]. It only improves readability, does not affect the expectation: `a`, `an`, `as`, `the`.

The following two sentences have the same meaning:

```
The first word of second line of output should valid number

The first word of the second line of output should valid as a number
```

6.1.6 `Assert` - expectation for custom assertion

The `Assert` is yet another expectation to verify with a user-defined function. It is designed for verification of side effects, not the result of the evaluation.

```
still_alive() {
  ping -c1 "$1" >/dev/null
}

Describe "example.com"
  It "responses"
    Assert still_alive "example.com"
  End
End
```

[10] docs/references.md#modifiers
[11] docs/references.md#matchers
[12] https://www.chaijs.com/

6.2 Pending, skip and focus

6.2.1 `Pending` - pending example

`Pending` is similar to `Skip`, but the test passes if the verification fails, and the test fails if the verification succeeds. This is useful if you want to specify that you will implement something later.

```
Describe 'Pending'
  Pending "not implemented"

  hello() { :; }

  It 'will success when test fails'
    When call hello world
    The output should "Hello world"
  End
End
```

6.2.2 `Skip` - skip example

Use `Skip` to skip executing the example.

```
Describe 'Skip'
  Skip "not exists bc"

  It 'is always skip'
    ...
  End
End
```

6.2.2.1 `if` - conditional skip

Use `Skip if` if you want to skip conditionally.

```
Describe 'Conditional skip'
  not_exists_bc() { ! type bc >/dev/null 2>&1; }
  Skip if "not exists bc" not_exists_bc

  add() { echo "$1 + $2" | bc; }

  It 'performs addition'
    When call add 2 3
    The output should eq 5
  End
End
```

6.2.3 'x' prefix for example group and example

6.2.3.1 `xDescribe`, `xContext`, `xExampleGroup` - skipped example group

`xDescribe`, `xContext`, `xExampleGroup` are skipped example group blocks. Execution of examples contained in these blocks is skipped.

```
Describe 'is example group'
  xDescribe 'is skipped example group'
    ...
  End
End
```

6.2.3.2 `xIt`, `xSpecify`, `xExample` - skipped example

`xIt`, `xSpecify`, `xExample` are skipped example blocks. Execution of the example is skipped.

```
xIt 'is skipped example'
  ...
End
```

6.2.4 'f' prefix for example group and example

6.2.4.1 `fDescribe`, `fContext`, `fExampleGroup` - focused example group

`fDescribe`, `fContext`, `fExampleGroup` are focused example group blocks. Only the examples included in these will be executed when the `--focus` option is specified.

```
Describe 'is example group'
  fDescribe 'is focues example group'
    ...
  End
End
```

6.2.4.2 `fIt`, `fSpecify`, `fExample` - focused example

`fIt`, `fSpecify`, `fExample` are focused example blocks. Only these examples will be executed when the `--focus` option is specified.

```
fIt 'is focused example'
  ...
End
```

6.2.5 About temporary pending and skip

The pending and skip without message is "temporary pending" and "temporary skip". "x"-prefixed example groups and examples are treated as a temporary skip.

The non-temporary pending and skip (with message) is used when it takes a long time to resolve. It may be committed to a version control system. The temporary pending and skip is used during the current work. We do not recommend committing it to a version control system.

These two types differ in the display of the report. Refer to `--skip-message` and `--pending-message` options.

```
 Temporary pending and skip
Pending
Skip
Skip # this comment will be displayed in the report
Todo
xIt
  ...
End

 Non-temporary pending and skip
Pending "reason"
Skip "reason"
Skip if "reason" condition
Todo "It will be implemented"
```

6.3 Hooks

6.3.1 `BeforeEach` (`Before`), `AfterEach` (`After`) - example hook

You can specify commands to be executed before / after each example by `BeforeEach` (`Before`), `AfterEach` (`After`).

NOTE: `BeforeEach` and `AfterEach` are supported in version 0.28.0 and later. Previous versions should use `Before` and `After` instead.

NOTE: `AfterEach` is for cleanup and not for assertions.

```
Describe 'example hook'
  setup() { :; }
  cleanup() { :; }
  BeforeEach 'setup'
  AfterEach 'cleanup'

  It 'is called before and after each example'
    ...
  End

  It 'is called before and after each example'
    ...
  End
End
```

6.3.2 `BeforeAll`, `AfterAll` - example group hook

You can specify commands to be executed before / after all examples by `BeforeAll` and `AfterAll`

```
Describe 'example all hook'
  setup() { :; }
  cleanup() { :; }
  BeforeAll 'setup'
  AfterAll 'cleanup'

  It 'is called before/after all example'
    ...
  End

  It 'is called before/after all example'
    ...
  End
End
```

6.3.3 `BeforeCall`, `AfterCall` - call evaluation hook

You can specify commands to be executed before / after call evaluation by `BeforeCall` and `AfterCall`

NOTE: These hooks were originally created to test ShellSpec itself. Please use the `BeforeEach` / `AfterEach` hooks whenever possible.

```
Describe 'call evaluation hook'
  setup() { :; }
  cleanup() { :; }
  BeforeCall 'setup'
  AfterCall 'cleanup'

  It 'is called before/after call evaluation'
    When call hello world
    ...
  End
End
```

6.3.4 `BeforeRun`, `AfterRun` - run evaluation hook

You can specify commands to be executed before / after run evaluation (`run`, `run command`, `run script` and `run source`) by `BeforeRun` and `AfterRun`

These hooks are executed in the same subshell as the "run evaluation". Therefore, you can access the variables after executing the evaluation.

NOTE: These hooks were originally created to test ShellSpec itself. Please use the `BeforeEach` / `AfterEach` hooks whenever possible.

```
Describe 'run evaluation hook'
  setup() { :; }
  cleanup() { :; }
  BeforeRun 'setup'
  AfterRun 'cleanup'

  It 'is called before/after run evaluation'
    When run hello world
    ...
  End
End
```

6.4 Helpers

6.4.1 `Dump` - dump stdout, stderr and status for debugging

Dump stdout, stderr and status of the evaluation. It is useful for debugging.

```
When call echo hello world
Dump # stdout, stderr and status
```

6.4.2 `Include` - include a script file

Include a shell script to test.

```
Describe 'lib.sh'
  Include lib.sh # hello function defined

  Describe 'hello()'
    It 'says hello'
      When call hello ShellSpec
      The output should equal 'Hello ShellSpec!'
    End
  End
End
```

6.4.3 `Set` - set shell option

Set shell option before executing each example. The shell option name is the long name of `set` or the name of `shopt`:

NOTE: Use `Set` instead of the `set` command because the `set` command may not work as expected in some shells.

```
Describe 'Set helper'
  Set 'errexit:off' 'noglob:on'

  It 'sets shell options before executiong the example'
    When call foo
  End
End
```

6.4.4 `Path`, `File`, `Dir` - path alias

`Path` is used to define a short pathname alias. `File` and `Dir` are aliases for `Path`.

```
Describe 'Path helper'
  Path hosts-file="/etc/hosts"

  It 'defines short alias for long path'
    The path hosts-file should be exists
  End
End
```

6.4.5 `Data` - pass data as stdin to evaluation

You can use the Data Helper which inputs data from stdin for evaluation. The input data is specified after `#|` in the `Data` or `Data:expand` block.

```
Describe 'Data helper'
  It 'provides with Data helper block style'
    Data # Use Data:expand instead if you want expand variables.
      #|item1 123
      #|item2 456
      #|item3 789
    End
    When call awk '{total+=$2} END{print total}'
    The output should eq 1368
  End
End
```

You can also use a file, function or string as data sources.

See more details of Data[13]

[13] docs/references.md#data

6.4.6 `Parameters` - parameterized example

Parameterized test (aka Data Driven Test) is used to run the same test with different parameters. `Parameters` defines its parameters.

```
Describe 'example'
  Parameters
    "#1" 1 2 3
    "#2" 1 2 3
  End

  Example "example $1"
    When call echo "$(($2 + $3))"
    The output should eq "$4"
  End
End
```

In addition to the default `Parameters`, three styles are supported: `Parameters:value`, `Parameters:matrix` and `Parameters:dynamic`.

See more details of Parameters[14]

NOTE: You can also cooperate the `Parameters` and `Data:expand` helpers.

6.4.7 `Mock` - create a command-based mock

See Command-based mock

6.4.8 `Intercept` - create an intercept point

See Intercept

[14] docs/references.md#parameters

7 Directives

Directives are instructions that can be used in embedded shell scripts. It is used to solve small problems of shell scripts in testing.

This is like a shell function, but not a shell function. Therefore, the supported grammar is limited and can only be used at the beginning of a function definition or at the beginning of a line.

```
foo() { %puts "foo"; } # supported

bar() {
  %puts "bar" # supported
}

baz() {
  any command; %puts "baz" # not supported
}
```

7.1 `%const` (`%`) - constant definition

`%const` (`%` is short hand) directive defines a constant value. The characters which can be used for variable names are uppercase letters `[A-Z]`, digits `[0-9]` and underscore `_` only. It can not be defined inside an example group nor an example.

The value is evaluated during the specfile translation process. So you can access ShellSpec variables, but you can not access variable or function in the specfile.

This feature assumes use with conditional skip. The conditional skip may run outside of the examples. As a result, sometimes you may need variables defined outside of the examples.

7.2 `%text` - embedded text

You can use the `%text` directive instead of a hard-to-use heredoc with indented code. The input data is specified after `#|`.

```
Describe '%text directive'
  It 'outputs texts'
    output() {
      echo "start" # you can write code here
      %text
      #|aaa
      #|bbb
      #|ccc
      echo "end" # you can write code here
    }

    result() { %text
      #|start
      #|aaa
      #|bbb
      #|ccc
      #|end
    }

    When call output
    The output should eq "$(result)"
    The line 3 of output should eq 'bbb'
  End
End
```

7.3 `%puts` (`%-`), `%putsn` (`%=`) - output a string (with newline)

`%puts` (put string) and `%putsn` (put string with newline) can be used instead of (not portable) echo. Unlike echo, it does not interpret escape sequences regardless of the shell. `%-` is an alias of `%puts`, `%=` is an alias of `%putsn`.

7.4 `%printf` - alias for printf

This is the same as `printf`, but it can be used in the sandbox mode because the path has been resolved.

7.5 `%sleep` - alias for sleep

This is the same as `sleep`, but it can be used in the sandbox mode because the path has been resolved.

7.6 `%preserve` - preserve variables

Use `%preserve` directive to preserve the variables in subshells and external shell script.

In the following cases, `%preserve` is required because variables are not preserved.

- `When run` evaluation - It runs in a subshell.
- Command-based mock (`Mock`) - It is an external shell script.
- Function-based Mock called by command substitution

```
Describe '%preserve directive'
  It 'preserves variables'
    func() { foo=1; bar=2; baz=3; }
    preserve() { %preserve bar baz:BAZ; }
    AfterRun preserve

    When run func
    The variable foo should eq 1 # This will be failure
    The variable bar should eq 2 # This will be success
    The variable BAZ should eq 3 # Preserved to different variable (baz:BAZ)
  End
End
```

7.7 `%logger` - debug output

Output log messages to the log file (default: `/dev/tty`) for debugging.

7.8 `%data` - define parameter

See `Parameters`.

8 Mocking

There are two ways to create a mock, (shell) function-based mock and (external) command-based mock. The function-based mock is usually recommended for performance reasons. Both can be overwritten with an internal block and will be restored when the block ends.

8.1 Function-based mock

The (shell) function-based mock is simply (re)defined with shell function.

```
Describe 'function-based mock'
  get_next_day() { echo $(($(date +%s) + 86400)); }

  date() {
    echo 1546268400
  }

  It 'calls the date function'
    When call get_next_day
    The stdout should eq 1546354800
  End
End
```

8.2 Command-based mock

The (external) command-based mock creates a temporary mock shell script and runs as an external command. This is slow, but there are some advantages over the function-based mock.

- Can be use invalid characters as the shell function name.
  - e.g `docker-compose` (`-` cannot be used as a function name in POSIX)
- Can be invoke a mocked command from an external command (not limited to shell script).

A command-based mock creates an external shell script with the contents of a `Mock` block, so there are some restrictions.

- It is not possible to mock shell functions or shell built-in functions.
- It is not possible to call shell functions outside the `Mock` block.
  - Exception: Can be called exported (`export -f`) functions. (bash only)
- To reference variables outside the `Mock` block, they must be exported.
- To return a variable from a Mock block, you need to use the `%preserve` directive.

```
Describe 'command-based mock'
  get_next_day() { echo $(($(date +%s) + 86400)); }

  Mock date
    echo 1546268400
  End

  It 'runs the mocked date command'
    When call get_next_day
    The stdout should eq 1546354800
  End
End
```

NOTE: To achieve this feature, a directory for mock commands is included at the beginning of the `PATH`.

9 Support commands

9.1 Execute the actual command within a mock function

Support commands are helper commands that can be used in the specfile. For example, it can be used in a mock function to execute the actual command. It is recommended that the support command name be the actual command name prefixed with `@`.

```
Describe "Support commands example"
  touch() {
    @touch "$@" # @touch executes actual touch command
    echo "$1 was touched"
  }

  It "touch a file"
    When run touch "file"
    The output should eq "file was touched"
    The file "file" should be exist
  End
End
```

Support commands are generated in the `spec/support/bin` directory by the `--gen-bin` option. For example run `shellspec --gen-bin @touch` to generate the `@touch` command.

This is the main purpose, but support commands are just shell scripts, so they can also be used for other purposes. You can freely edit the support command script.

9.2 Make mock not mandatory in sandbox mode

The sandbox mode forces the use of mocks. However, you may not want to require mocks for some commands. For example, `printf` is a built-in command in many shells and does not require a mock in the sandbox mode for these shells. But there are shells where it is an external command and then it requires to be mocked.

To allow `printf` to be called without mocking in certain cases, create a support command named `printf` (`shellspec --gen-bin printf`).

9.3 Resolve command incompatibilities

Some commands have different options between BSD and GNU. If you handle the difference in the specfile, the test will be hard to read. You can solve it with the support command.

```
!/bin/sh -e
 Command name: @sed
. "$SHELLSPEC_SUPPORT_BIN"
case $OSTYPE in
  *darwin*) invoke gsed "$@" ;;
  *) invoke sed "$@" ;;
esac
```

10 Tagging

The example groups or examples can be tagged, and the `--tag` option can be used to filter the examples to be run. The tag name and tag value are separated by `:`, and the tag value is optional. You can use any character if quoted.

```
Describe "Checking something" someTag:someVal
  It "does foo" tagA:val1
    ...
  It "does bar" tagA:val2
    ...
  It "does baz" tagA
    ...
End
```

1. Everything nested inside a selected element is selected in parent elements. e.g. `--tag someTag` will select everything above.
2. Specifying a tag but no value selects everything with that tag whether or not it has a value, e.g. `--tag tagA` will select everything above.
3. Specifying multiple tags will select the union of everything tagged, e.g. `--tag tagA:val1,tagA:val2` will select `does foo` and `does bar`.
4. Tests included multiple times are not a problem, e.g. `--tag someTag,tagA,tagA:val1` just selects everything.
5. If no tag matches, nothing will be run, e.g. `--tag tagA:` runs nothing (it does not match baz above, as empty values are not the same as no value).
6. The –tag option can be used multiple times, e.g. `--tag tagA:val1 --tag tagA:val2` works the same as `--tag tagA:val1,tagA:val2`

---

11 About testing external commands

ShellSpec is a testing framework for shell scripts, but it can be used to test anything that can be executed as an external command, even if it is written in another language. Even shell scripts can be tested as external commands.

If you are testing a shell script as an external command, please note the following.

- It will be executed in the shell specified by the shebang not the shell running the specfile.
- The coverage of the shell script will not be measured.
- Cannot refer to variables inside the shell script.
- Shell built-in commands cannot be mocked.
- Functions defined inside the shell script cannot be mocked.
- Only command-based mock can be used (if the script is calling an external command).
- Interceptor is not available.

To get around these limitations, use `run script` or `run source`. See How to test a single file shell script.

12 How to test a single file shell script

If the shell script consists of a single file, unit testing becomes difficult. However, there are many such shell scripts.

ShellSpec has the ability to testing in such cases with only few modifications to the shell script.

12.1 Using `run script`

Unlike the case of executing as an external command, it has the following features.

- It will run in the same shell (but another process) that is running specfile.
- The coverage of the shell script will be measured.

There are limitations as follows.

- Cannot refer to variables inside the shell script.
- Shell built-in commands cannot be mocked.
- Functions defined inside the shell script cannot be mocked.
- Only command-based mock can be used (if the script is calling an external command).
- Interceptor is not available.

12.2 Using `run source`

It is even less limitations than `run script` and has the following features.

- It will run in the same shell and same process that is running specfile.
- The coverage of the shell script will be measured.
- Can be refer to variables inside the shell script.
- Function-based mock and command-based mock are available.
- Interceptor is available.
- Shell built-in commands can be mocked.
- Functions defined inside the shell script can be mocked using interceptor.

However, since it is simulated using the `.` command, there are some differences in behavior. For example, the value of `$0` is different.

NOTE: Mocking of shell built-in commands can be done before `run source`. However, if you are using interceptor, mocking of the `test` command must be done in the `__<name>__` function.

12.3 Testing shell functions

12.3.1 `__SOURCED__`

This is the way to test shell functions defined in a shell script.

Loading a script with `Include` defines a `__SOURCED__` variable available in the sourced script. If the variable `__SOURCED__` is defined, please return from the shell script.

```
!/bin/sh
 hello.sh

hello() { echo "Hello $1"; }

# This is the writing style presented by ShellSpec, which is short but unfamiliar.
# Note that it returns the current exit status (could be non-zero).
${__SOURCED__:+return}

# The above means the same as below.
 ${__SOURCED__:+x} && return $?

# If you don't like the coding style, you can use the general writing style.
 if [ "${__SOURCED__:+x}" ]; then
   return 0
 fi

hello "$1"
```

With a test file:

```
Describe "hello.sh"
  Include "./hello.sh"
  Describe "hello()"
    It "says hello"
      When call hello world
      The output should eq "Hello world"
    End
  End
End
```

12.4 Intercepting

Interceptor is a feature that allows you to intercept your shell script in the middle of its execution. This makes it possible to mock functions that cannot be mocked in advance at arbitrary timing, and to make assertions by retrieving the state of during script execution.

It is a powerful feature, but avoid using it as possible, because it requires you to modify your code and may reduce readability. Normally, it is not a good idea to modify the code just for testing, but in some cases, there is no choice but to use this.

```
!/bin/sh
 ./today.sh

 When run directly without testing, the "__()" function does nothing.
test || __() { :; }

 the "now()" function is defined here, so it can't be mocked in advance.
now() { date +"%Y-%m-%d %H:%M:%S"; }

 The function you want to test
today() {
  now=$(now)
  echo "${now% *}"
}

 I want to mock the "now()" function here.
__ begin __

today=$(today)
echo "Today is $today"

__ end __
```

```
Describe "today.sh"
  Intercept begin
  __begin__() {
    now() { echo "2021-01-01 01:02:03"; }
  }
  __end__() {
    # The "run source" is run in a subshell, so you need to use "%preserve"
    # to preserve variables
    %preserve today
  }

  It "gets today's date"
    When run source ./today.sh
    The output should eq "Today is 2021-01-01"
    The variable today should eq "2021-01-01"
  End
End
```

12.4.1 Intercept

Usage: `Intercept [<name>...]`

Specify the name(s) to intercept.

NOTE: I will change `Intercept` to `Interceptors` to make it a declarative DSL.

12.4.2 `test || __() { :; }`

Define the `__` function that does nothing except when run as a test (via ShellSpec). This allows you to run it as a production without changing the code.

The `test` command is the shell built-in `test` command. This command returns false (non-zero) when called with no arguments. This will allow who are not familiar with ShellSpec to will understand what the result will be, even if they don't know what the code is for. Of course, it is good practice to comment on what the code is for

When run via ShellSpec, the `test` command is redefined and returns true "only once" when called with no arguments. After that, it will return to its original behavior. This means that this code needs to be executed only once, at the start of the shell script.

12.4.3 `__`

Usage: `__ <name> [arguments...] __`

This is where the process is intercepted. You can define more than one. If the name matches the name specified in `Intercept`, the `__<name>__` function will be called.

Note that if the name is not specified in `Intercept`, nothing will be done, but the exit status will be changed to 0.

13 spec_helper

The `spec_helper` can be used to set shell options for all specfiles, define global functions,check the execution shell, load custom matchers, etc.

The `spec_helper` is the default module name. It can be changed to any other name, and multiple modules can be used. Only characters accepted by POSIX as identifiers can be used in module names. The file name of the module must be the module name with the extension `.sh` appended. It is loaded from `SHELLSPEC_LOAD_PATH` using the `--require` option.

The following is a typical `spec_helper`. The following three callback functions are available.

```
 Filename: spec/spec_helper.sh

set -eu

spec_helper_precheck() {
  minimum_version "0.28.0"
  if [ "$SHELL_TYPE" != "bash" ]; then
    abort "Only bash is supported."
  fi
}

spec_helper_loaded() {
  : # In most cases, you won't use it.
}

spec_helper_configure() {
  import 'support/custom_matcher'
  before_each "global_before_each_hook"
}

 User-defined global function
global_before_each_hook() {
  :
}

 In version <= 0.27.x, only shellspec_spec_helper_configure was available.
 This callback function is still supported but deprecated in the future.
 Please rename it to spec_helper_configure.
 shellspec_spec_helper_configure() {
  :
 }
```

The `spec_helper` will be loaded at least twice. The first time is at precheck phase, which is executed in a separate process before the specfile execution. The second time will be load at the beginning of the specfile execution. If you are using parallel execution, it will be loaded every specfile.

Within each callback function, there are several helper functions available. These functions are not available outside of the callback function. Also, these callback functions will be removed automatically when `spec_helper` is finished loading. (User-defined functions will not be removed.)

13.1 `<module>_precheck`

This callback function will be invoked only once before loading specfiles. Exit with `exit` or `abort`, or `return` non-zero to exit without executing specfiles. Inside this function, `set -eu` is executed, so an explicit return on error is not necessary.

Since it is invoked in a separate process from specfiles, changes made in this function will not be affected in specfiles.

13.1.1 `minimum_version`

- Usage: `minimum_version <version>`

Specifies the minimum version of ShellSpec that the specfile supports. The version format is semantic version[15]. Pre-release versions have a lower precedence than the associated normal version, but comparison between pre-release versions is not supported. The build metadata will simply be ignored.

NOTE: Since `<module>_precheck` is only available in 0.28.0 or later, it can be executed with earlier ShellSpecs even if minimum_version is specified. To avoid this, you can implement a workaround using `--env-from`.

```
 spec/env.sh
 Add `--env-from spec/env.sh` to `.shellspec`
major_minor=${SHELLSPEC_VERSION%".${SHELLSPEC_VERSION#*.*.}"}
if [ "${major_minor%.*}" -eq 0 ] && [ "${major_minor#*.}" -lt 28 ]; then
  echo "ShellSpec version 0.28.0 or higher is required." >&2
  exit 1
fi
```

13.1.2 `error`, `warn`, `info`

- Usage: `error [messages...]`
- Usage: `warn [messages...]`
- Usage: `info [messages...]`

Outputs a message according to the type. You can also use `echo` or `printf`.

13.1.3 `abort`

- Usage: `abort [messages...]`
- Usage: `abort <exit status> [messages...]`

Display an error message and `exit`. If the exit status is omitted, it is `1`. You can also exit with exit. `exit 0` will exit normally without executing the specfiles.

13.1.4 `setenv`, `unsetenv`

- Usage: `setenv [name=value...]`
- Usage: `unset [name...]`

You can use `setenv` or `unsetenv` to pass or remove environment variables from precheck to specfiles.

13.1.5 environment variables

The following environment variables are defined.

- `VERSION` - ShellSpec Version
- `SHELL_TYPE` - Currently running shell type (e.g. `bash`)
- `SHELL_VERSION` - Currently running shell version (e.g. `4.4.20(1)-release`)

NOTE: Be careful not to confuse `SHELL_TYPE` with the environment variable `SHELL`. The environment variable `SHELL` is the user login shell, not the currently running shell. It is a variable set by the system, and which unrelated to ShellSpec.

[15] https://semver.org/

13.2 `<module>_loaded`

It is called after loading the shellspec's general internal functions, but before loading the core modules (subject, modifire, matcher, etc). If parallel execution is enabled, it may be called multiple times in isolated processes. Internal functions starting with `shellspec_` can also be used, but be aware that they may change.

This was created to perform workarounds[16] for specific shells in order to test ShellSpec itself. Other than that, I have not come up with a case where this is absolutely necessary, but if you have one, please let me know.

13.3 `<module>_configure`

This callback function will be called after core modules (subject, modifire, matcher, etc) has been loaded. If parallel execution is enabled, it may be called multiple times in isolated processes. Internal functions starting with `shellspec_` can also be used, but be aware that they may change. It can be used to set global hooks, load custom matchers, etc., and override core module functions.

13.3.1 `import`

- Usage: `import <module> [arguments...]`

Import a custom module from `SHELLSPEC_LOAD_PATH`.

13.3.2 `before_each`, `after_each`

- Usage: `before_each [hooks...]`
- Usage: `after_each [hooks...]`

Register hooks to be executed before and after every example. It is the same as executing `BeforeEach`/`AfterEach` at the top of all specfiles.

13.3.3 `before_all`, `after_all`

- Usage: `before_all [hooks...]`
- Usage: `after_all [hooks...]`

Register hooks to be executed before and after all example. It is the same as executing `BeforeAll`/`AfterAll` at the top of all specfiles.

NOTE: This is a hook that is called before and after each specfile, not before and after all specfiles.

[16] helper/ksh_workaround.sh

14 Self-executable specfile

Add `eval "$(shellspec - -c) exit 1"` to the top of the specfile and give execute permission to the specfile. You can use `/bin/sh`, `/usr/bin/env bash`, etc. for shebang. The specfile will be run in the shell written in shebang.

```
!/bin/sh

eval "$(shellspec - -c) exit 1"

 Use the following if version <= 0.27.x
 eval "$(shellspec -)"

Describe "bc command"
  bc() { echo "$@" | command bc; }

  It "performs addition"
    When call bc "2+3"
    The output should eq 5
  End
End
```

The `-c` option is available since 0.28.0, and you can also pass other options. If you run the specfile directly, `--pattern` will be automatically set to `*`. These options will be ignored if run via `shellspec` command.

The use of `shellspec` as shebang is deprecated because it is not portable.

```
!/usr/bin/env shellspec -c
Linux does not allow passing options

!/usr/bin/env -S shellspec -c
The -S option requires GNU Core Utilities 8.30 (2018-07-01) or later.
```

15 Use with Docker

You can run ShellSpec without installation using Docker. ShellSpec and specfiles run in a Docker container.

See How to use ShellSpec with Docker[17].

16 Extension

16.1 Custom subject, modifier and matcher

You can create custom subject, custom modifier and custom matcher.

See examples/spec/support/custom_matcher.sh for custom matcher.

NOTE: If you want to verify using shell function, you can use result[18] modifier or satisfy[19] matcher. You don't need to create a custom matcher, etc.

[17] docs/docker.md
[18] docs/references.md#result
[19] docs/references.md#satisfy