shellwhat Documentation

Release 1.5.0

DataCamp

Feb 03, 2020

Glossary

For an introduction to SCTs and how to use shellwhat, visit the README.

This documentation features:

- A glossary with typical use-cases and corresponding SCT constructs.
- Reference documentation of all actively maintained shellwhat functions.
- Some articles that gradually expose of shellwhat's functionality and best practices.

If you are new to writing SCTs for Shell exercises, start with the tutorial. The glossary is good to get a quick overview of how all functions play together after you have a basic understanding. The reference docs become useful when you grasp all concepts and want to look up details on how to call certain functions and specify custom feedback messages.

CHAPTER 1

Glossary

This article lists some example solutions. For each of these solutions, an SCT is included, as well as some example student submissions that would pass and fail. In all of these, a submission that is identical to the solution will evidently pass.

Note: These SCT examples are not golden bullets that are perfect for your situation. Depending on the exercise, you may want to focus on certain parts of a statement, or be more accepting for different alternative answers.

All these examples come from the Intro to Shell for Data Science and Introduction to Git for Data Science courses. You can have a look at their respective GitHub sources here and here, respectively.

1.1 Checking the current directory

```
# solution command
cd test
```

```
# sct
Ex().has_cwd('/home/repl/test')
```

1.2 Checking the `ls` statement

```
# solution command
ls
```

```
# sct
Ex().check_correct(
    has_cwd('/home/repl')
```

(continues on next page)

(continued from previous page)

```
    has_expr_output()
)
```

1.3 Checking whether a directory exists

```
# solution command
mdkir /home/repl/test
```

```
# sct
Ex().has_dir('/home/repl/test')
```

1.4 Checking command output

```
# solution command
echo 'this is a printout!'
```

```
# sct
Ex().has_output(r'this\\s+is\\s+a\\s+print\\s*out')
```

```
# Submissions that would pass:
echo 'this   is a print out'
test='this is a printout!' && echo $test

# Submissions that would fail:
echo 'this is a wrong printout'
```

1.5 Checking contents of a file

```
# solution command
echo hello > test.txt
```

```
# sct
Ex().check_file('/home/repl/test.txt').multi(
    # check that file contains hello or hi
    has_code(r'hello|hi'),
    # check that file does not contain goodbye
    check_not(has_code('goodbye'),
              incorrect_msg="meaningful error message")
)
```

1.6 Git: check branch

```
# solution command (while in the test git repo)
git checkout make-change
```

```
Ex().multi(
    has_cwd('/home/repl/test'),
    has_expr_output(expr='git rev-parse --abbrev-ref HEAD | grep make-change',
                    output='make-change', strict=True,
                    incorrect_msg=meaningful message")
)
```

1.7 Git: check that file was staged

```
# solution command (while in the test git repo)
git add test.txt
```

```
# sct
Ex().multi(
    has_cwd('/home/repl/test')
    has_expr_output(expr="git diff --name-only --staged | grep test.txt",
                    output="test.txt", strict=True,
                    incorrect_msg="meaningful message")
)
```

CHAPTER 2

Checks

has_code(*state: shellwhat.State.State*, *text: str*, *incorrect_msg: str = 'The checker expected to find '{{text}}' in your command.'*, *fixed: bool = False*) → shellwhat.State.State

Check whether the student code contains text.

This function is a simpler override of the *has_code* function in protowhat, because `ast_node.get_text()` is not implemented in the OSH parser

Using `has_code()` should be a last resort. It is always better to look at the result of code or the side effects they had on the state of your program.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **text** – text that student code must contain. Can be a regex pattern or a simple string.
- **incorrect_msg** – if specified, this overrides the automatically generated feedback message in case `text` is not found in the student code.
- **fixed** – whether to match `text` exactly, rather than using regular expressions.

Example Suppose the solution requires you to do:

```
git push origin master
```

The following SCT can be written:

```
Ex().has_code(r'git\s+push\s+origin\s+master')
```

Submissions that would pass:

```
git push origin master
git   push    origin    master
```

Submissions that would fail:

```
git push --force origin master
```

has_output(*state: shellwhat.State.State*, *text: str*, *incorrect_msg: str = "The checker expected to find {{" if fixed else 'the pattern '}}'{{text}}' in the output of your command."*, *fixed: bool = False*, *strip_ansi: bool = True*) → shellwhat.State.State

Check whether student output contains specific text.

Before you use `has_output()`, have a look at `has_expr_output()` or `has_expr_error()`; they might be more fit for your use case.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with `Ex()`.
- **text** – text that student output must contain. Can be a regex pattern or a simple string.
- **incorrect_msg** – if specified, this overrides the automatically generated feedback message in case `text` is not found in the student output.
- **fixed** – whether to match `text` exactly, rather than using regular expressions.
- **strip_ansi** – whether to remove ANSI escape codes from output

Example Suppose the solution requires you to do:

```
echo 'this is a printout!'
```

The following SCT can be written:

```
Ex().has_output(r'this\s+is\s+a\s+print\s*out')
```

Submissions that would pass:

```
echo 'this   is a print out'
test='this is a printout!' && echo $test
```

Submissions that would fail:

```
echo 'this is a wrong printout'
```

has_cwd(*state: shellwhat.State.State*, *dir: str*, *incorrect_msg: str = 'Your current working directory should be '{{dir}}'. Use 'cd {{dir}}' to navigate there.'*) → shellwhat.State.State

Check whether the student is in the expected directory.

This check is typically used before using `has_expr_output()` to make sure the student didn't navigate somewhere else.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with `Ex()`.
- **dir** – Directory that the student should be in. Always use the absolute path.
- **incorrect_msg** – If specified, this overrides the automatically generated message in case the student is not in the expected directory.

Example If you want to be sure that the student is in `/home/repl/my_dir`:

```
Ex().has_cwd('/home/repl/my_dir')
```

has_expr_output(*state: shellwhat.State.State*, *expr: str = None*, ***, *incorrect_msg: Union[str, protowhat.Feedback.FeedbackComponent] = "The checker expected to find the result of '{{expr}}' in your output, but couldn't."*, *strict: bool = False*, *output: str = None*, *test='output'*, *strip_ansi: bool = True*) → shellwhat.State.State

Run a shell expression, and see if its result is in the output or in manually specified output.

By default, the result of the student's code is compared to the result of running `expr`. You can compare the result of running `expr` with an arbitrary output by specifing `output`.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **expr** – expression to run in the shell. If not specified, this defaults to the solution code.
- **msg** – feedback message if expression result is not in output.
- **strict** – whether result must be exactly equal to output, or (if False) contained therein.
- **output** – overrides the output that the expression result is compared to.
- **test** – whether to use stdout ("output") from the expression, or its exit code ("error").
- **strip_ansi** – whether to remove ANSI escape codes from result.

Example As a first example, suppose you expect the student to show the status of a git repository:

```
git status
```

The following SCT would check that:

```
Ex().has_expr_output()  # expr set to solution code
```

As a second example, suppose you want to verify that a student staged a the changes to the file *test.txt* in a git repo:

```
git add test.txt
```

The following SCT would check that this file is actually staged:

```
Ex().has_expr_output(expr="git diff --name-only --staged | grep test.
↪txt",
                     output="test.txt", strict=True,
                     incorrect_msg="meaningful message")
```

Notice how manually specifying `expr` and `output` allows you to probe virtually any property or state of your terminal without the student knowing.

has_expr_exit_code(*state: shellwhat.State.State*, *expr: str = None*, ***, *incorrect_msg: Union[str, protowhat.Feedback.FeedbackComponent] = "The checker expected to get the exit code '{{output}}' when executing '{{expr}}' in your output, but didn't."*, *strict: bool = True*, *output: str = None*, *test='exit_code'*, *strip_ansi: bool = True*) → shellwhat.State.State

Run a shell expression, and see if its exit code is in the output or in manually specified output.

By default, the result of the student's code is compared to the result of running `expr`. You can compare the result of running `expr` with an arbitrary output by specifing `output`.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().

- **`expr`** – expression to run in the shell. If not specified, this defaults to the solution code.
- **`msg`** – feedback message if expression result is not in output.
- **`strict`** – whether result must be exactly equal to output, or (if False) contained therein.
- **`output`** – overrides the output that the expression result is compared to.
- **`test`** – whether to use stdout ("output") from the expression, or its exit code ("error").
- **`strip_ansi`** – whether to remove ANSI escape codes from result.

CHAPTER 3

Files

check_file(*state: protowhat.State.State*, *path*, *missing_msg='Did you create the file '{}'?'*, *is_dir_msg='Want to check the file '{}', but found a directory.'*, *parse=True*, *solution_code=None*)

Test whether file exists, and make its contents the student code.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **path** – expected location of the file
- **missing_msg** – feedback message if no file is found in the expected location
- **is_dir_msg** – feedback message if the path is a directory instead of a file
- **parse** – If `True` (the default) the content of the file is interpreted as code in the main exercise technology. This enables more checks on the content of the file.
- **solution_code** – this argument can be used to pass the expected code for the file so it can be used by subsequent checks.

Note: This SCT fails if the file is a directory.

Example To check if a user created the file `my_output.txt` in the subdirectory `resources` of the directory where the exercise is run, use this SCT:

```
Ex().check_file("resources/my_output.txt", parse=False)
```

has_dir(*state: protowhat.State.State*, *path*, *msg='Did you create a directory '{}'?'*)

Test whether a directory exists.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **path** – expected location of the directory
- **msg** – feedback message if no directory is found in the expected location

Example To check if a user created the subdirectory `resources` in the directory where the exercise is run, use this SCT:

```
Ex().has_dir("resources")
```

CHAPTER 4

Bash history checks

`update_bash_history_info`(*bash_history_path=None*)

Store the current number of commands in the bash history

`get_bash_history` can use this info later to get only newer commands.

Depending on the wanted behaviour this function should be called at the start of the exercise or every time the exercise is submitted.

Import using `from protowhat.checks import update_bash_history_info`.

`get_bash_history`(*full_history=False*, *bash_history_path=None*)

Get the commands in the bash history

Parameters

- **`full_history`** (*`bool`*) – if true, returns all commands in the bash history, else only return the commands executed after the last bash history info update
- **`bash_history_path`** (*`str | Path`*) – path to the bash history file

Returns a list of commands (empty if the file is not found)

Import from `from protowhat.checks import get_bash_history`.

`has_command`(*state*, *pattern*, *msg*, *fixed=False*, *commands=None*)

Test whether the bash history has a command matching the pattern

Parameters

- **`state`** – State instance describing student and solution code. Can be omitted if used with Ex().
- **`pattern`** – text that command must contain (can be a regex pattern or a simple string)
- **`msg`** – feedback message if no matching command is found
- **`fixed`** – whether to match text exactly, rather than using regular expressions

- **commands** – the bash history commands to check against. By default this will be all commands since the last bash history info update. Otherwise pass a list of commands to search through, created by calling the helper function `get_bash_history()`.

Note: The helper function `update_bash_history_info(bash_history_path=None)` needs to be called in the pre-exercise code in exercise types that don't have built-in support for bash history features.

Note: If the bash history info is updated every time code is submitted (by using `update_bash_history_info()` in the pre-exercise code), it's advised to only use this function as the second part of a `check_correct()` to help students debug the command they haven't correctly run yet. Look at the examples to see what could go wrong.

If bash history info is only updated at the start of an exercise, this can be used everywhere as the (cumulative) commands from all submissions are known.

Example The goal of an exercise is to use `man`.

If the exercise doesn't have built-in support for bash history SCTs, update the bash history info in the pre-exercise code:

```
update_bash_history_info()
```

In the SCT, check whether a command with `man` was used:

```
Ex().has_command("$man\s", "Your command should start with ``man ...
↪``.")
```

Example The goal of an exercise is to use `touch` to create two files.

In the pre-exercise code, put:

```
update_bash_history_info()
```

This SCT can cause problems:

```
Ex().has_command("touch.*file1", "Use `touch` to create `file1`")
Ex().has_command("touch.*file2", "Use `touch` to create `file2`")
```

If a student submits after running `touch file0 && touch file1` in the console, they will get feedback to create `file2`. If they submit again after running `touch file2` in the console, they will get feedback to create `file1`, since the SCT only has access to commands after the last bash history info update (only the second command in this case). Only if they execute all required commands in a single submission the SCT will pass.

A better SCT in this situation checks the outcome first and checks the command to help the student achieve it:

```
Ex().check_correct(
    check_file('file1', parse=False),
    has_command("touch.*file1", "Use `touch` to create `file1`")
)
Ex().check_correct(
    check_file('file2', parse=False),
    has_command("touch.*file2", "Use `touch` to create `file2`")
)
```

prepare_validation(*state: protowhat.State.State, commands: List[str], bash_history_path: Optional[str] = None*) → protowhat.State.State

Let the exercise validation know what shell commands are required to complete the exercise

Import using `from protowhat.checks import prepare_validation`.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **commands** – List of strings that a student is expected to execute
- **bash_history_path** (*str | Path*) – path to the bash history file

Example The goal of an exercise is to run a build and check the output.

At the start of the SCT, put:

```
Ex().prepare_validation(["make", "cd build", "ls"])
```

Further down you can now use `has_command`.

CHAPTER 5

Logic

multi (*state*, **tests*)

Run multiple subtests. Return original state (for chaining).

This function could be thought as an AND statement, since all tests it runs must pass

Parameters

- **state** – State instance describing student and solution code, can be omitted if used with Ex()
- **tests** – one or more sub-SCTs to run.

Example The SCT below checks two has_code cases..

```
Ex().multi(has_code('SELECT'), has_code('WHERE'))
```

The SCT below uses `multi` to 'branch out' to check that the SELECT statement has both a WHERE and LIMIT clause..

```
Ex().check_node('SelectStmt', 0).multi(
    check_edge('where_clause'),
    check_edge('limit_clause')
)
```

check_not (*state*, **tests*, *msg*)

Run multiple subtests that should fail. If all subtests fail, returns original state (for chaining)

- This function is currently only tested in working with `has_code()` in the subtests.
- This function can be thought as a `NOT(x OR y OR ...)` statement, since all tests it runs must fail
- This function can be considered a direct counterpart of multi.

Parameters

- **state** – State instance describing student and solution code, can be omitted if used with Ex()

- ***tests** – one or more sub-SCTs to run
- **msg** – feedback message that is shown in case not all tests specified in `*tests` fail.

Example Thh SCT below runs two has_code cases..

```
Ex().check_not(
    has_code('INNER'),
    has_code('OUTER'),
    incorrect_msg="Don't use `INNER` or `OUTER`!"
)
```

If students use `INNER (JOIN)` or `OUTER (JOIN)` in their code, this test will fail.

check_or (*state*, **tests*)
Test whether at least one SCT passes.

Parameters

- **state** – State instance describing student and solution code, can be omitted if used with Ex()
- **tests** – one or more sub-SCTs to run

Example The SCT below tests that the student typed either 'SELECT' or 'WHERE' (or both)..

```
Ex().check_or(
    has_code('SELECT'),
    has_code('WHERE')
)
```

The SCT below checks that a SELECT statement has at least a WHERE c or LIMIT clause..

```
Ex().check_node('SelectStmt', 0).check_or(
    check_edge('where_clause'),
    check_edge('limit_clause')
)
```

check_correct (*state*, *check*, *diagnose*)
Allows feedback from a diagnostic SCT, only if a check SCT fails.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **check** – An sct chain that must succeed.
- **diagnose** – An sct chain to run if the check fails.

Example The SCT below tests whether students query result is correct, before running diagnostic SCTs..

```
Ex().check_correct(
    check_result(),
    check_node('SelectStmt')
)
```

disable_highlighting (*state*)
Disable highlighting in the remainder of the SCT chain.

Include this function if you want to avoid that pythonwhat marks which part of the student submission is incorrect.

fail(*state*, *msg='fail'*)

Always fails the SCT, with an optional msg.

This function takes a single argument, `msg`, that is the feedback given to the student. Note that this would be a terrible idea for grading submissions, but may be useful while writing SCTs. For example, failing a test will highlight the code as if the previous test/check had failed.

CHAPTER 6

Electives

has_chosen (*state*, *correct*, *msgs*)
Verify exercises of the type MultipleChoiceExercise

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **correct** – index of correct option, where 1 is the first option.
- **msgs** – list of feedback messages corresponding to each option.

Example The following SCT is for a multiple choice exercise with 2 options, the first of which is correct.:

```
Ex().has_chosen(1, ['Correct!', 'Incorrect. Try again!'])
```

allow_errors (*state*)
Allow running the student code to generate errors.

This has to be used only once for every time code is executed or a different xwhat library is used. In most exercises that means it should be used just once.

Example The following SCT allows the student code to generate errors:

```
Ex().allow_errors()
```

success_msg (*state*, *msg*)
Changes the success message to display if submission passes.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **msg** – feedback message if student and solution ASTs don't match

Example The following SCT changes the success message:

```
Ex().success_msg("You did it!")
```

CHAPTER 7

Tutorial

shellwhat uses the `.` to 'chain together' SCT functions. Every chain starts with the `Ex()` function call, which holds the exercise state. This exercise state contains all the information that is required to check if an exercise is correct, which are:

- the student submission and the solution as text, and their corresponding parse trees.
- the result of running the solution, as an ANSI-formatted string.
- the result of running the student's query, as an ANSI-formatted string.
- the errors that running the student's query generated, if any.

As SCT functions are chained together with `.`, the `Ex()` exercise state is copied and adapted into 'sub states' to zoom in on particular parts of the state. Before this theory blows your brains out, some examples will be included in this tutorial soon.

For details, questions and suggestions, contact us.

Python Module Index

Index