protowhat Documentation

Release 2.1.3

DataCamp

Dec 29, 2022

REFERENCE

1 Ex() 3

2 AST checks 5

3 Logic 9

4 Simple checks 11

5 File checks 13

6 Bash history checks 15

Python Module Index 19

Index 21

protowhat is a utility package required by

- sqlwhat to write SCTs for SQL exercises, and
- protowhat to write SCTs for Shell exercises.

protowhat contains functionality that is shared between these packages, including:

- SCT function chaining and syntactical sugar,
- State management,
- AST element selection, dispatching and message generation,
- Basic SCT functions such as `success_msg()` and `has_chosen()`.

All relevent documentation to write SCTs for SQL and Shell exercises, including functions that reside in `protowhat`, can be found in the sqlwhat and protowhat documentation.

CHAPTER

ONE

EX()

CHAPTER

TWO

AST CHECKS

check_edge(*state*, *name*, *index=0*, *missing_msg=None*)

Select an attribute from an abstract syntax tree (AST) node, using the attribute name.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **name** – the name of the attribute to select from current AST node.
- **index** – entry to get from a list field. If too few entires, will fail with missing_msg.
- **missing_msg** – feedback message if attribute is not in student AST.

Example If both the student and solution code are..

```
SELECT a FROM b; SELECT x FROM y;
```

then we can get the from_clause using

```
# approach 1: with manually created State instance -----
state = State(*args, **kwargs)
select = check_node(state, 'SelectStmt', 0)
clause = check_edge(select, 'from_clause')

# approach 2: with Ex and chaining ---------------------
select = Ex().check_node('SelectStmt', 0)           # get first select␣
↪statement
clause =  select.check_edge('from_clause', None)    # get from_clause␣
↪(a list)
clause2 = select.check_edge('from_clause', 0)       # get first entry␣
↪in from_clause
```

check_node(*state*, *name*, *index=0*, *missing_msg=None*, *priority=None*)

Select a node from abstract syntax tree (AST), using its name and index position.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **name** – the name of the abstract syntax tree node to find.
- **index** – the position of that node (see below for details).
- **missing_msg** – feedback message if node is not in student AST.

- **priority** – the priority level of the node being searched for. This determines whether to descend into other AST nodes during the search. Higher priority nodes descend into lower priority. Currently, the only important part of priority is that setting a very high priority (e.g. 99) will search every node.

Example If both the student and solution code are..

```
SELECT a FROM b; SELECT x FROM y;
```

then we can focus on the first select with:

```
# approach 1: with manually created State instance
state = State(*args, **kwargs)
new_state = check_node(state, 'SelectStmt', 0)

# approach 2:  with Ex and chaining
new_state = Ex().check_node('SelectStmt', 0)
```

has_code(*state*, *text*, *incorrect_msg='Check the {ast_path}. The checker expected to find {text}.'*, *fixed=False*)

Test whether the student code contains text.

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** – feedback message if text is not in student code.
- **fixed** – whether to match text exactly, rather than using regular expressions.

Note: Functions like `check_node` focus on certain parts of code. Using these functions followed by `has_code` will only look in the code being focused on.

Example If the student code is..

```
SELECT a FROM b WHERE id < 100
```

Then the first test below would (unfortunately) pass, but the second would fail..:

```
# contained in student code
Ex().has_code(text="id < 10")

# the $ means that you are matching the end of a line
Ex().has_code(text="id < 10$")
```

By setting `fixed = True`, you can search for fixed strings:

```
# without fixed = True, '*' matches any character
Ex().has_code(text="SELECT * FROM b")               # passes
Ex().has_code(text="SELECT \\* FROM b")             # fails
Ex().has_code(text="SELECT * FROM b", fixed=True)   # fails
```

You can check only the code corresponding to the WHERE clause, using

```
where = Ex().check_node('SelectStmt', 0).check_edge('where_clause')
where.has_code(text = "id < 10)
```

has_equal_ast(*state*, *incorrect_msg=None*, *sql=None*, *start='expression'*, *exact=None*, *should_append_msg=False*)

Test whether the student and solution code have identical AST representations

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **incorrect_msg** – feedback message if student and solution ASTs don't match
- **sql** – optional code to use instead of the solution ast that is zoomed in on.
- **start** – if `sql` arg is used, the parser rule to parse the sql code. One of 'expression' (the default), 'subquery', or 'sql_script'.
- **exact** – whether to require an exact match (True), or only that the student AST contains the solution AST. If not specified, this defaults to `True` if `sql` is not specified, and to `False` if `sql` is specified. You can always specify it manually.
- **should_append_msg** – prepend the auto generated incorrect_msg with the previous append_messages.

Example Example 1 - Suppose the solution code is

```
SELECT * FROM cities
```

and you want to verify whether the *FROM* part is correct:

```
Ex().check_node('SelectStmt').from_clause().has_equal_ast()
```

Example 2 - Suppose the solution code is

```
SELECT * FROM b WHERE id > 1 AND name = 'filip'
```

Then the following SCT makes sure `id > 1` was used somewhere in the WHERE clause.:

```
Ex().check_node('SelectStmt') \/
    .check_edge('where_clause') \/
    .has_equal_ast(sql = 'id > 1')
```

CHAPTER

THREE

LOGIC

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')
)
```

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')
)
```

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.

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')
)
```

CHAPTER

FOUR

SIMPLE CHECKS

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()
```

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!'])
```

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

FIVE

FILE CHECKS

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

SIX

BASH HISTORY CHECKS

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.

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.

PYTHON MODULE INDEX

p

protowhat.checks.check_bash_history, 15
protowhat.checks.check_files, 13
protowhat.checks.check_funcs, 5
protowhat.checks.check_logic, 9
protowhat.checks.check_simple, 11

INDEX

A

allow_errors() (*in module protowhat.checks.check_simple*), 11

C

check_correct() (*in module protowhat.checks.check_logic*), 9
check_edge() (*in module protowhat.checks.check_funcs*), 5
check_file() (*in module protowhat.checks.check_files*), 13
check_node() (*in module protowhat.checks.check_funcs*), 5
check_not() (*in module protowhat.checks.check_logic*), 9
check_or() (*in module protowhat.checks.check_logic*), 9

D

disable_highlighting() (*in module protowhat.checks.check_logic*), 10

F

fail() (*in module protowhat.checks.check_logic*), 10

G

get_bash_history() (*in module protowhat.checks.check_bash_history*), 15

H

has_chosen() (*in module protowhat.checks.check_simple*), 11
has_code() (*in module protowhat.checks.check_funcs*), 6
has_command() (*in module protowhat.checks.check_bash_history*), 15
has_dir() (*in module protowhat.checks.check_files*), 13
has_equal_ast() (*in module protowhat.checks.check_funcs*), 7

M

module
    protowhat.checks.check_bash_history, 15
    protowhat.checks.check_files, 13
    protowhat.checks.check_funcs, 5
    protowhat.checks.check_logic, 9
    protowhat.checks.check_simple, 11
multi() (*in module protowhat.checks.check_logic*), 10

P

prepare_validation() (*in module protowhat.checks.check_bash_history*), 16
protowhat.checks.check_bash_history
    module, 15
protowhat.checks.check_files
    module, 13
protowhat.checks.check_funcs
    module, 5
protowhat.checks.check_logic
    module, 9
protowhat.checks.check_simple
    module, 11

S

success_msg() (*in module protowhat.checks.check_simple*), 11

U

update_bash_history_info() (*in module protowhat.checks.check_bash_history*), 17