Pythonwhat Documentation

Release 1.2.0

Vincent Vankrunkelsven

May 22, 2016

CONTENTS

CHAPTER

ONE

TEST_EXPRESSION_OUTPUT

pythonwhat.test_expression_output.**test_expression_output**(*extra_env=None*, *context_vals=None*, *incorrect_msg=None*, *eq_condition='equal'*, *pre_code=None*, *keep_objs_in_env=None*)

Test output of expression.

The code of the student is ran in the active state and the output it generates is compared with the code of the solution. This can be used in nested pythonwhat calls like test_if_else. In these kind of calls, the code of the active state is set to the code in a part of the sub statement (e.g. the body of an if statement). It has various parameters to control the execution of the (sub)expression.

Parameters

- **extra_env** (*dict*) – set variables to the extra environment. They will update the student and solution environment in the active state before the student/solution code in the active state is ran. This argument should contain a dictionary with the keys the names of the variables you want to set, and the values are the values of these variables.
- **context_vals** (*list*) – set variables which are bound in a for loop to certain values. This argument is only useful if you use the function in a test_for_loop. It contains a list with the values of the bound variables.
- **incorrect_msg** (*str*) – feedback message if the output of the expression in the solution doesn't match the one of the student. This feedback message will be expanded if it is used in the context of another test function, like test_if_else.
- **eq_condition** (*str*) – the condition which is checked on the eval of the group. Can be "equal" – meaning that the operators have to evaluate to exactly the same value, or "equivalent" – which can be used when you expect an integer and the result can differ slightly. Defaults to "equal".
- **pre_code** (*str*) – the code in string form that should be executed before the expression is executed. This is the ideal place to set a random seed, for example.
- **keep_obj_in_env** (*list ()*) – a list of variable names that should be hold in the copied environment where the expression is evaluated. All primitive types are copied automatically, other objects have to be passed explicitely.

Examples

Student code

```
a = 12
if a > 3:
    print('test %d' % a)
```

Soltuion code

```
a = 4
if a > 3:
    print('test %d' % a)
```

SCT

```
test_if_else(1,
    body = lambda:  test_expression_output(extra_env = { 'a':  5 },
        incorrect_msg = "Print out the correct things"))
```

This SCT will pass as the subexpression will output ‘test 5’ in both student as solution environment, since the extra environment sets *a* to 5.

CHAPTER

TWO

TEST_EXPRESSION_RESULT

`pythonwhat.test_expression_result.`**`test_expression_result`**(*extra_env=None*, *context_vals=None*, *incorrect_msg=None*, *eq_condition='equal'*, *expr_code=None*, *pre_code=None*, *keep_objs_in_env=None*)

Test result of expression.

The code of the student is ran in the active state and the result of the evaluation is compared with the result of the solution. This can be used in nested pythonwhat calls like test_if_else. In these kind of calls, the code of the active state is set to the code in a part of the sub statement (e.g. the condition of an if statement). It has various parameters to control the execution of the (sub)expression.

Parameters

- **`extra_env`** (*`dict`*) – set variables to the extra environment. They will update the student and solution environment in the active state before the student/solution code in the active state is ran. This argument should contain a dictionary with the keys the names of the variables you want to set, and the values are the values of these variables.
- **`context_vals`** (*`list`*) – set variables which are bound in a for loop to certain values. This argument is only useful if you use the function in a test_for_loop. It contains a list with the values of the bound variables.
- **`incorrect_msg`** (*`str`*) – feedback message if the result of the expression in the solution doesn't match the one of the student. This feedback message will be expanded if it is used in the context of another test function, like test_if_else.
- **`eq_condition`** (*`str`*) – the condition which is checked on the eval of the group. Can be "equal" – meaning that the operators have to evaluate to exactly the same value, or "equivalent" – which can be used when you expect an integer and the result can differ slightly. Defaults to "equal".
- **`expr_code`** (*`str`*) – if this variable is not None, the expression in the studeont/solution code will not be ran. Instead, the given piece of code will be ran in the student as well as the solution environment and the result will be compared.
- **`pre_code`** (*`str`*) – the code in string form that should be executed before the expression is executed. This is the ideal place to set a random seed, for example.
- **`keep_obj_in_env`** (*`list ()`*) – a list of variable names that should be hold in the copied environment where the expression is evaluated. All primitive types are copied automatically, other objects have to be passed explicitely.

Examples

Student code

```
a = 12
if a > 3:
    print('test %d' % a)
```

Solution code

```
a = 4
b = 5
if (a + 1) > (b - 1):
    print('test %d' % a)
```

SCT

```
test_if_else(1,
    test = lambda:  test_expression_result(extra_env = { 'a':  3 }
        incorrect_msg = "Test if `a` > 3"))
```

This SCT will pass as the condition in the student's code (*a > 3*) will evaluate to the same value as the code in the solution code (*(a + 1) > (b - 1)*), with value of *a* set to *3*.

CHAPTER

THREE

TEST_FOR_LOOP

`pythonwhat.test_for_loop.`**`test_for_loop`**(*index=1*, *for_iter=None*, *body=None*, *orelse=None*, *expand_message=True*)

Test parts of the for loop.

This test function will allow you to extract parts of a specific for loop and perform a set of tests specifically on these parts. A for loop consists of two parts: the sequence, *for_iter*, which is the values over which are looped, and the *body*. A for loop can have a else part as well, *orelse*, but this is almost never used.

```
for i in range(10):
    print(i)
```

Has *range(10)* as the sequence and *print(i)* as the body.

Parameters

- **`index`** (*`int`*) – index of the function call to be checked. Defaults to 1.
- **`for_iter`** – this argument holds the part of code that will be ran to check the sequence of the for loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the sequence part of the for loop.
- **`body`** – this argument holds the part of code that will be ran to check the body of the for loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the for loop.
- **`orelse`** – this argument holds the part of code that will be ran to check the else part of the for loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the else part of the for loop.
- **`expand_message`** (*`bool`*) – if true, feedback messages will be expanded with *in the ___ of the for loop on line ___*. Defaults to True. If False, *test_for_loop()* will generate no extra feedback.

Examples

Student code

```
for i in range(10):
    print(i)
```

Solution code

```
for n in range(10):
    print(n)
```

SCT

```
test_for_loop(1,
    for_iter = lamdba:  test_function("range"),
    body = lambda:  test_expression_output(context_val = [5])
```

This SCT will evaluate to True as the function *“range”* is used in the sequence and the function *test_exression_output()* will pass on the body code.

CHAPTER

FOUR

TEST_FUNCTION

`pythonwhat.test_function.`**`test_function`**(*name*, *index=1*, *args=None*, *keywords=None*, *eq_condition='equal'*, *do_eval=True*, *not_called_msg=None*, *incorrect_msg=None*)

Test if function calls match.

This function compares a function call in the student's code with the corresponding one in the solution code. It will cause the reporter to fail if the corresponding calls do not match. The fail message that is returned will depend on the sort of fail.

Parameters

- **`name`** (*`str`*) – the name of the function to be tested.
- **`index`** (*`int`*) – index of the function call to be checked. Defaults to 1.
- **`args`** (*`list (int`*) – the indices of the positional arguments that have to be checked. If it is set to None, all positional arguments which are in the solution will be checked.
- **`keywords`** (*`list (str`*) – the indices of the keyword arguments that have to be checked. If it is set to None, all keyword arguments which are in the solution will be checked.
- **`eq_condition`** (*`str`*) – The condition which is checked on the eval of the group. Can be "equal" – meaning that the operators have to evaluate to exactly the same value, or "equivalent" – which can be used when you expect an integer and the result can differ slightly. Defaults to "equal".
- **`do_eval`** (*`bool`*) – Boolean representing whether the group should be evaluated and compared or not. Defaults to True.
- **`not_called_msg`** (*`str`*) – feedback message if the function is not called.
- **`incorret_msg`** (*`str`*) – feedback message if the arguments of the function in the solution doesn't match the one of the student.

Raises

- `NameError` – the eq_condition you passed is not "equal" or "equivalent".
- `NameError` – function is not called in the solution

Examples

Student code

```
import numpy as np
np.mean([1,2,3])
np.std([2,3,4])
```

Solution code

```
import numpy
numpy.mean([1,2,3], axis = 0)
numpy.std([4,5,6])
```

SCT

`test_function("numpy.mean", index = 1, keywords = [])`: pass.

`test_function("numpy.mean", index = 1)`: fail.

`test_function(index = 1, incorrect_op_msg = "Use the correct operators")`: fail.

`test_function(index = 1, used = [], incorrect_result_msg = "Incorrect result")`: fail.

CHAPTER

FIVE

TEST_IF_ELSE MODULE

pythonwhat.test_if_else.**test_if_else**(*index=1*, *test=None*, *body=None*, *orelse=None*, *expand_message=True*)

Test parts of the if statement.

This test function will allow you to extract parts of a specific if statement and perform a set of tests specifically on these parts. A for loop consists of three potential parts: the condition test, *test*, which specifies the condition of the if statement, the *body*, which is what's executed if the condition is True and a else part, *orelse*, which will be executed if the condition is not True.

```
if 5 == 3:
    print("success")
else:
    print("fail")
```

Has *5 == 3* as the condition test, *print("success")* as the body and *print("fail")* as the else part.

Parameters

- **index** (*int*) – index of the function call to be checked. Defaults to 1.
- **test** – this argument holds the part of code that will be ran to check the condition test of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the condition test of the if statement.
- **body** – this argument holds the part of code that will be ran to check the body of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the if statement.
- **orelse** – this argument holds the part of code that will be ran to check the else part of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the else part of the if statement.
- **expand_message** (*bool*) – if true, feedback messages will be expanded with *in the ___ of the if statement on line ___*. Defaults to True. If False, *test_if_else()* will generate no extra feedback.

Examples

Student code

```
a = 12
if a > 3:
    print('test %d' % a)
```

Solution code

```
a = 4
if a > 3:
    print('test %d' % a)
```

SCT

```
test_if_else(1,
    body = lambda:  test_expression_output(extra_env = { 'a':  5 }
        incorrect_msg = "Print out the correct things"))
```

This SCT will pass as *test_expression_output()* is ran on the body of the if statement and it will output the same thing in the solution as in the student code.

CHAPTER SIX

TEST_IMPORT

`pythonwhat.test_import.`**`test_import`**(*name*, *same_as=True*, *not_imported_msg=None*, *incorrect_as_msg=None*)

Test import.

Test whether an import statement is used the same in the student's environment as in the solution environment.

Parameters

- **`name`** (*`str`*) – the name of the package that has to be checked.
- **`same_as`** (*`bool`*) – if false, the alias of the package doesn't have to be the same. Defaults to True.
- **`not_imported_msg`** (*`str`*) – feedback message when the package is not imported.
- **`incorrect_as_msg`** (*`str`*) – feedback message if the alias is wrong.

Examples

Student code

```
import numpy as np
import pandas as pa
```

Solution code

```
import numpy as np
import pandas as pd
```

SCT

`test_import("numpy")`: pass.
`test_import("pandas")`: fail.
`test_import("pandas", same_as = False)`: pass.

CHAPTER

SEVEN

TEST_MC

`pythonwhat.test_mc.`**`test_mc`**(*correct*, *msgs*)

Test multiple choice exercise.

Test for a MultipleChoiceExercise. The correct answer (as an integer) and feedback messages are passed to this function.

Parameters

- **`correct`** (*`int`*) – the index of the correct answer (should be an instruction). Starts at 1.
- **`msgs`** (*`list (str`*) – a list containing all feedback messages belonging to each choice of the
- **`The list should have the same length as the number of instructions.`** (*`student.`*) –

CHAPTER EIGHT

TEST_OBJECT

`pythonwhat.test_object.`**`test_object`**(*name*, *eq_condition='equal'*, *do_eval=True*, *undefined_msg=None*, *incorrect_msg=None*)

Test object.

The value of an object in the ending environment is compared in the student's environment and the solution environment.

Parameters

- **`name`** (*`str`*) – the name of the object which value has to be checked.
- **`eq_condition`** (*`str`*) – the condition which is checked on the eval of the object. Can be "equal" – meaning that the operators have to evaluate to exactly the same value, or "equivalent" – which can be used when you expect an integer and the result can differ slightly. Defaults to "equal".
- **`do_eval`** (*`bool`*) – if False, the object will only be checked for existence. Defaults to True.
- **`undefined_msg`** (*`str`*) – feedback message when the object is not defined
- **`incorrect_msg`** (*`str`*) – feedback message if the value of the object in the solution environment doesn't match the one in the student environment.

Examples

Student code

```
a = 1
b = 5
```

Solution code

```
a = 1
b = 2
```

SCT

`test_object("a")`: pass.
`test_object("b")`: fail.

CHAPTER

NINE

TEST_OBJECT_AFTER_EXPRESSION

`pythonwhat.test_object_after_expression.`**`test_object_after_expression`**(*name*, *extra_env=None*, *context_vals=None*, *undefined_msg=None*, *incorrect_msg=None*, *eq_condition='equal'*, *pre_code=None*, *keep_objs_in_env=None*)

Test object after expression.

The code of the student is ran in the active state and the the value of the given object is compared with the value of that object in the solution. This can be used in nested pythonwhat calls like test_for_loop. In these kind of calls, the code of the active state is set to the code in a part of the sub statement (e.g. the body of a for loop). It has various parameters to control the execution of the (sub)expression. This test function is ideal to check if a value is updated correctly in the body of a for loop.

Parameters

- **`name`** (*`str`*) – the name of the object which value has to be checked after evaluation of the expression.
- **`extra_env`** (*`dict`*) – set variables to the extra environment. They will update the student and solution environment in the active state before the student/solution code in the active state is ran. This argument should contain a dictionary with the keys the names of the variables you want to set, and the values are the values of these variables.
- **`context_vals`** (*`list`*) – set variables which are bound in a for loop to certain values. This argument is only useful if you use the function in a test_for_loop. It contains a list with the values of the bound variables.
- **`incorrect_msg`** (*`str`*) – feedback message if the value of the object in the solution environment doesn't match the one in the student environment. This feedback message will be expanded if it is used in the context of another test function, like test_for_loop.
- **`eq_condition`** (*`str`*) – the condition which is checked on the eval of the object. Can be "equal" – meaning that the operators have to evaluate to exactly the same value, or "equivalent" – which can be used when you expect an integer and the result can differ slightly. Defaults to "equal".
- **`expr_code`** (*`str`*) – if this variable is not None, the expression in the studeont/solution code will not be ran. Instead, the given piece of code will be ran in the student as well as the

solution environment and the result will be compared.

- **pre_code** (*str*) – the code in string form that should be executed before the expression is executed. This is the ideal place to set a random seed, for example.
- **keep_obj_in_env** (*list ()*) – a list of variable names that should be hold in the copied environment where the expression is evaluated. All primitive types are copied automatically, other objects have to be passed explicitely.

Examples

Student code

```
count = 1
for i in range(100):
    count = count + i
```

Solution code

```
count = 15
for n in range(30):
    count = count + n
```

SCT

```
test_for_loop(1,
    body = lambda:  test_object_after_expression("count",
        extra_env = { 'count':  20 },
        contex_vals = [ 10 ])
```

This SCT will pass as the value of *count* is updated identically in the body of the for loop in the student code and solution code.

CHAPTER

TEN

TEST_OPERATOR

`pythonwhat.test_operator.`**`test_operator`**(*index=1*, *eq_condition='equal'*, *used=None*, *do_eval=True*, *not_found_msg=None*, *incorrect_op_msg=None*, *incorrect_result_msg=None*)

Test if operator groups match.

This function compares an operator group in the student's code with the corresponding one in the solution code. It will cause the reporter to fail if the corresponding operators do not match. The fail message that is returned will depend on the sort of fail. We say that one operator group correpsonds to a group of operators that is evaluated to one value (e.g. 3 + 5 * (1/3)).

Parameters

- **index** (*int*) – Index of the operator group to be checked. Defaults to 1.
- **eq_condition** (*str*) – The condition which is checked on the eval of the group. Can be "equal" – meaning that the operators have to evaluate to exactly the same value, or "equivalent" – which can be used when you expect an integer and the result can differ slightly. Defaults to "equal".
- **used** (*List[str]*) – A list of operators that have to be in the group. Valid operators are: "+", "-", "*", "/", "%", "**", "<<", ">>", "|", "^", "&" and "//". If the list is None, operators that are in the group in the solution have to be in the student code. Defaults to None.
- **do_eval** (*bool*) – Boolean representing whether the group should be evaluated and compared or not. Defaults to True.
- **not_found_msg** (*str*) – Feedback message if not enough operators groups are found in the student's code.
- **incorrect_op_msg** (*str*) – Feedback message if the wrong operators are used in the student's code.
- **incorrect_result_msg** (*str*) – Feedback message if the operator group evaluates to the wrong result in the student's code.

Raises

- `NameError` – the eq_condition you passed is not "equal" or "equivalent".
- `IndexError` – not enough operation groups in the solution environment.

Examples

Student code

```
1 + 5 * (3+5)
1 + 1 * 238
```

Solution code

```
3.1415 + 5
1 + 238
```

SCT

`test_operator(index = 2, used = ["+"])`: pass.

`test_operator(index = 2)`: fail.

`test_operator(index = 1, incorrect_op_msg = "Use the correct operators")`: fail.

`test_operator(index = 1, used = [], incorrect_result_msg = "Incorrect result")`: fail.

CHAPTER

ELEVEN

TEST_OUTPUT_CONTAINS

`pythonwhat.test_output_contains.`**`test_output_contains`**(*text*, *pattern=True*, *no_output_msg=None*)

Test the output.

Tests if the output contains a (pattern of) text.

Parameters

- **`text`** (*`str`*) – the text that is searched for
- **`pattern`** (*`bool`*) – if True, the text is treated as a pattern. If False, it is treated as plain text. Defaults to False.
- **`no_output_msg`** (*`str`*) – feedback message to be displayed if the output is not found.

TEST_STUDENT_TYPED

pythonwhat.test_student_typed.**test_student_typed**(*text*, *pattern=True*, *not_typed_msg=None*)

Test the student code.

Tests if the student typed a (pattern of) text.

Parameters

- **text** (*str*) – the text that is searched for
- **pattern** (*bool*) – if True, the text is treated as a pattern. If False, it is treated as plain text. Defaults to False.
- **not_typed_msg** (*str*) – feedback message to be displayed if the student did not type the text.

CHAPTER

THIRTEEN

TEST_WHILE_LOOP MODULE

`pythonwhat.test_while_loop.`**`test_while_loop`**`(`*index=1*, *test=None*, *body=None*, *orelse=None*, *expand_message=True*`)`

Test parts of the while loop.

This test function will allow you to extract parts of a specific while loop and perform a set of tests specifically on these parts. A while loop generally consists of two parts: the condition test, *test*, which is the condition that is tested each loop, and the *body*. A for while can have a else part as well, *orelse*, but this is almost never used.

```
a = 10
while a < 5:
    print(a)
    a -= 1
```

Has *a < 5* as the condition test and *print(i)* as the body.

Parameters

- **`index`** (*`int`*) – index of the function call to be checked. Defaults to 1.
- **`test`** – this argument holds the part of code that will be ran to check the condition test of the while loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the condition test of the while loop.
- **`body`** – this argument holds the part of code that will be ran to check the body of the while loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the while loop.
- **`orelse`** – this argument holds the part of code that will be ran to check the else part of the while loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the else part of the while loop.
- **`expand_message`** (*`bool`*) – if true, feedback messages will be expanded with *in the ___ of the while loop on line ___*. Defaults to True. If False, *test_for_loop()* will generate no extra feedback.

Examples

Student code

```
a = 10
```

```
while a < 5:
    print(a)
    a -= 1
```

Solution code

```
a = 20
while a < 5:
    print(a)
    a -= 1
```

SCT

```
test_while_loop(1,
    test = lamdba:  test_expression_result({"a":  5}),
    body = lambda:  test_expression_output({"a":  5}))
```

This SCT will evaluate to True as condition test will have thes same result in student and solution code and *test_exression_output()* will pass on the body code.

PYTHON MODULE INDEX