

Programming Python3 Autograders

This overview assumes you know how to add an autograder, what an autograder fundamentally is, etc. This is focusing on how to program the autograder.

[Boiler Plate Autograder](#)

[The Parameters](#)

[Things to keep in mind:](#)

[Tests That Can Be Run](#)

[Summary of Expectation Functions](#)

[Test Options](#)

[Test Options Examples](#)

[Tests With Input](#)


[Example](#)

[Some General Hacks and Tips](#)

Boiler Plate Autograder

This is what you see when you add the autograder. It's prefilled with examples of how to write tests.

```
class Suite(PythonTestSuite): Each test case needs to be in its own class

    # Any values that should be passed to any call to `input`
    inputs = []  These values will be given as user input to the student's program.
    It's most reliable to make this a list of strings

    # Write any tests that should run before the code is evaluated
    def before_run(self, student_code, solution_code):
        # expect(student_code).to_contain('def my_function(param1):')
        # expect(student_code).to_be('answer')
        pass These tests are run before the student's program is run

    # Write any tests that should run after the code is evaluated
    def after_run(self, student_code, solution_code, student_output, solution_output):
        # expect(student_output).to_contain('Hello World!')
        pass These tests are run after the student's program is run

suite() This executes the test case. If you add more classes, you need to remember to
call them, too!
```

The Parameters

- All parameters of these functions are Strings (other than self, which is the calling object)
- *student_code*: the text of the student's program
- *solution_code*: the text of the solution program
- *student_output*: the text of the student's output given the input (if there is any)
 - **VERY IMPORTANT**: this does not contain any text that was printed using *input*
- *solution_output*: the text of the solution program's output given the input (if there is any)

Things to keep in mind:

- If you want to pass more than one set of inputs to the program, you need to create another class. If you don't (or there is no input), then you can put all of your tests in the default class.
- To access the input list in the functions, use ***self.inputs***
- While inputs can hold any data type, it's best to make it a list of strings. There are errors sometimes when students print numerical data given by inputs without casting it to a string (for instance, by doing *print(some_input)*)
- I usually put any code analysis tests in *before_run*, and then use *after_run* to check the student's output.
- Don't forget to delete the *pass* statement if you implement the function

Tests That Can Be Run

You can see the full test options here:

<https://github.com/krodgers/codehs/blob/master/codehs/editor/static/js/autograder/components/autograder-python/templates/yerba.txt>

To create a test, use ***expect(...)*** and then call one of the following test methods.

Let *res* be the student's output/code. Your options are:

- ***expect(res).to_contain(expected)***
 - Checks that the string *expected* is contained in *res*
 - Example that checks the student has a while loop:
`expect(student_code).to_contain("while")`

- **expect(res).not_to_contain(expected)**
 - Checks that the string *expected* is NOT contained in *res*
 - Example that checks the student removed a placeholder pass statement:
`expect(student_code).not_to_contain("pass")`
- **expect(res).to_be(expected)**
 - Checks that *res* is the same as the object *expected*
 - Note this is not the same as *res == expected*; this is *res is expected*
 - This works for strings because of the way Python creates and stores strings
- **expect(res).to_be_greater_than(expected)**
 - Checks `res > expected`
 - Example that checks the student printed at least 4 lines
 - `lines = student_output.split_lines() # creates a list of lines`
`expect(len(lines)).to_be_greater_than(3)`
- **expect(res).to_be_greater_than_or_equal_to(expected)**
 - Checks `res >= expected`
 - Example that checks the student printed at least 4 lines
 - `lines = student_output.split_lines() # creates a list of lines`
`expect(len(lines)).to_be_greater_than_or_equal_to(4)`
- **expect(res).to_be_less_than(expected)**
 - Checks `res < expected`
 - Example that checks the student used no more than 4 for loops
 - `num_fors = student_code.count("for")`
`expect(num_fors).to_be_less_than(5)`
- **expect(res).to_be_less_than_or_equal_to(expected)**
 - Checks `res <= expected`
 - Example that checks the student used no more than 4 for loops
 - `num_fors = student_code.count("for")`
`expect(num_fors).to_be_less_than_or_equal_to(4)`

- **expect(res).to_equal(expected)**
 - Checks `res == expected`
 - You usually want to use this one and not `to_be(...)`
 - Example that checks the student printed the correct first line
 - `lines = student_output.split_lines()`
`expect(lines[0]).to_equal("My Fun Program")`
- **expect(res).not_to_equal(expected)**
 - Checks `res != expected`
 - Example that checks the student changed the last line of output
 - `lines = student_output.split_lines()`
`expect(lines[-1]).not_to_equal("Change this output")`
- **expect(res).to_be_truthy()**
 - Checks `res == True`
 - Takes no parameters; useful for custom tests or checking multiple things
 - Example that checks the student used a for loop and did not use a while loop
 - `used_for = "for" in student_code`
`not_use_while = "while" not in student_code`
`loops_right = used_for and not_use_while`
`expect(loops_right).to_be_truthy()`
- **expect(res).to_be_falsey()**
 - Checks `res == False`
 - Takes no parameters; useful for custom tests or checking multiple things
 - Example that checks the student changed the starter code
 - `code_changed = student_code == "# Put code here"`
`expect(code_changed).to_be_falsey()`

Summary of Expectation Functions

- `to_contain(expected)`
- `not_to_contain(expected)`
- `to_be_greater_than(value)`
- `to_be_greater_than_or_equal_to(va)`
- `to_be_less_than(value)`
- `to_be_less_than_or_equal_to(value)`
- `to_equal(value)`
- `not_to_equal(value)`
- `to_be_truthy()`
- `to_be_falsey()`
- `to_be(some_obj)`
- `not_to_be(some_obj)`

Test Options

With each of the previous methods, you can then call `with_options()` that will customize the following:

- **test_name**: the test name displayed to the student
 - defaults to a string representation of the test
 - for example, something like “Expected “print(“hello world”)” to contain “print””
- **message_pass**: Message displayed if the test passed
 - defaults to nothing
- **message_fail**: Message displayed if the test failed
 - defaults to nothing
- **student_output**: What is shown as the student output (labeled “your result”)
 - defaults to what’s passed to *expect*
- **solution_output**: What is shown as the solution output
 - defaults to empty
- **show_diff**: Shows the difference between the student’s output and the solution output
 - defaults to False
 - Noe: unless the program only passes with very specific formatting, students usually find this more confusing than helpful

Test Options Examples

- Example that checks the student used a for loop and did not use a while loop
 - `used_for = "for" in student_code`
`not_use_while = "while" not in student_code`
`loops_right = used_for and not_use_while`
`expect(loops_right).to_be_truthy().with_options(
 test_name = "You should use a for loop for this program",
 message_pass = "Great!",
 message_fail = "You should not use a while loop!",
 student_output = student_code)`
 - Note: it's a good idea to set the *student_output* here; otherwise, *student_output* would be the value of *loops_right*
- Example that checks the student changed the last line of output
 - `lines = student_output.split_lines()`
`expect(lines[-1]).not_to_equal("Change this output").with_options(
 test_name = "You should customize the last line of output",
 message_pass = "Great!",
 message_fail = "Check your last line!")`
 - Note: In this case, *student_output* will be the last line that the student printed

Tests With Input

To set the input to a program, put the values in the *input* list.

The program will be run **once** with the given input. If you want to test a different set of input, you need to write another test class (see examples below).

The input should be given as strings. This minimizes the number of weird errors that occur for the student. To use the *inputs* list in the *before_run* or *after_run* functions, use *self.inputs*.

Example

Problem: The student is supposed to ask the user for a number of feet and number of inches, then print out the number of inches.

Test Cases: 3 ft, 4 inches → 40 inches

0 ft, 6 inches → 6 inches

12 ft, 1 inches → 145 inches

The autograder:


```

1 class Suite(PythonTestSuite):
2     # Any values that should be passed to any call to `input`
3     inputs = ["3", "4"]
4
5     # Write any tests that should run before the code is evaluated
6     def before_run(self, student_code, solution_code):
7         pass
8
9     # Write any tests that should run after the code is evaluated
10    def after_run(self, student_code, solution_code, student_output, solution_output):
11        expected_inches = str(int(self.inputs[0]) * 12 + int(self.inputs[1]))
12
13        expect(student_output).to_contain(expected_inches).with_options(
14            test_name = "{} feet, {} inches is equal to {} inches"\
15                .format(self.inputs[0], self.inputs[1], expected_inches),
16            message_pass="Great!",
17            message_fail="Check your calculations")
18

```

```

19 class Suite01(PythonTestSuite):
20     # Any values that should be passed to any call to `input`
21     inputs = ["0", "6"]
22
23     # Write any tests that should run before the code is evaluated
24     def before_run(self, student_code, solution_code):
25         pass
26
27     # Write any tests that should run after the code is evaluated
28     def after_run(self, student_code, solution_code, student_output, solution_output):
29        expected_inches = str(int(self.inputs[0]) * 12 + int(self.inputs[1]))
30
31        expect(student_output).to_contain(expected_inches).with_options(
32            test_name = "{} feet, {} inches is equal to {} inches"\
33                .format(self.inputs[0], self.inputs[1], expected_inches),
34            message_pass="Great!",
35            message_fail="Check your calculations")
36
37

```

```

38 class Suite02(PythonTestSuite):
39     # Any values that should be passed to any call to `input`
40     inputs = ["12", "1"]
41
42     # Write any tests that should run before the code is evaluated
43     def before_run(self, student_code, solution_code):
44         pass
45
46     # Write any tests that should run after the code is evaluated
47     def after_run(self, student_code, solution_code, student_output, solution_output):
48        expected_inches = str(int(self.inputs[0]) * 12 + int(self.inputs[1]))
49
50        expect(student_output).to_contain(expected_inches).with_options(
51            test_name = "{} feet, {} inches is equal to {} inches"\
52                .format(self.inputs[0], self.inputs[1], expected_inches),
53            message_pass="Great!",
54            message_fail="Check your calculations")
55
56
57 suite()
58 Suite01()
59 Suite02()
60

```

Some things to Note:

- inputs are lists of strings
- Each set of input needs its own class (Suite, Suite01, Suite02)
 - You can name these classes anything you want; just be sure to inherit from PythonTestSuite
- Each class needs to be created at the bottom (lines 57 - 59)
- The tests are written in such a way that once the first one works, you can copy and paste the entire class -- the only thing you have to change is the inputs list and the class name!
- self.inputs refers to that class's inputs list
- I used format with the strings because I think it looks cleaner. String concatenation works just as well.
- student_output isn't set in the test options since it will default to student_output
 - since student_output is the parameter for *expect*
- Since solution_output isn't set in the test options, the expected output won't be shown to students

Some General Hacks and Tips

- You very, very rarely want to use `expect(student_output).to_equal(solution_output)`
 - It's better to just look for key pieces
- To debug your tests, set `student_output` or `solution_output` to see the values of your variables
 - Remember these have to be string values
- String comparisons are case sensitive and whitespace sensitive
 - use `str.lower()` to make `str` all lowercase
 - use `str.replace(' ', '')` to remove spaces (but not all whitespace)
 - first parameter is a single space, second is an empty string
 - use `"".join(str.split())` to remove ALL whitespace
 - "" is an empty string
- You can use `strip_comments(str)` to get rid of any comments from the code
 - helpful for when you're checking that they didn't use `for` loops for instance
 - (background story: there was a program that was failing because it said the `for` loop was incorrect. Turns out, the autograder was looking at a `'for'` it had found in a comment, and missed the one in the code)