Writing Autograder Tests

Aug 15, 2020

Contents:

CHAPTER 1

Doctests

Most Python autograders run doctests against a global environment from the execution of a student's submission. These doctests are text formatted as if from a Python interpreter that tell the executor what code to run and what output to expect. An important nuance of doctests is that these test are based on *string comparison*, so the outputs `1` and `1.0` are *not* equal.

1.1 Doctest Format

Python doctests are formatted as input and output from a Python interpeter, e.g.

```
>>> import numpy as np
>>> np.random.seed(42)
>>> np.random.choice([1, 2, 3])
3
```

Note that Python structures that require multiple lines have `...` prompts, not >>>.

```
>>> def foo(x):
...     return x
>>> if foo(1) % 2 == 0:
...     print("even")
... else:
...     print("odd")
```

Lines that have an ellipsis prompt include `elif`, `else`, `except` and `finally` clauses, function bodies, continuations of escaped lines (lines after those ending with \), and other indented lines (those that begin with whitespace). Note that in a normal interpreter there would be an additional empty line after an ellipsis block with an ellipsis prompt, but this can be omitted when writing doctests.

Sometimes, writing autograder tests may require the use of special characters whose meanings are changed when Python compiles the string, e.g. \. If your autograder test involves these characters, when Python compiles the string, the doctest's meaning will be changed. For this reason, when writing doctests, it is always best practice to put autograder tests in raw strings, denoted by the letter `r` before the opening quote:

```
# this is an example test in the OK format
test = {
    "suites": [
        {
            "cases": [
                {
                    "code": r"""
                    >>> if 4 % 2 == 0:
                    ...     print("even")
                    even
                    """     # a raw string
                }
            ]
        }
    ]
}
```

Note also that the doctest library ignores leading whitespace before prompts and outputs, e.g. as with `textwrap.dedent`, so even though the spaces before each line in the string above are captured, they are ignored when the doctest is executed.

1.2 Running Doctests

Autograders abstract away executing doctests beyond the step of actually writing them. Once you understand the doctest format, you can write your own autograder tests and allow the autograder to take care of executing them and parsing the output. Most Python autogaders make use of Python's doctest library to perform these tests, and return results based on whether or not these pass.

CHAPTER 2

Comparisons

A fundamental aspect of doctests is that the pass/fail behavior is determined by **string comparisons**. When a doctest is run, the code provided in the Python interpreter format is executed line-by-line and then the output of that line is expected to equal the output shown in the doctest.

2.1 Return Value Types

2.1.1 None

Statements that return the Python value `None` have no output. If you put `None` into a Python interpeter, there is no output shown:

```
>>> None
>>> # the next interpreter prompt
```

Some statements that return the value `None` include assignment statements, import statements, calls to functions like `print` and `exec`, and any functions with no `return` statement or a return statement that returns `None` itself.

```
>>> def foo(x):
...     return x
>>> def bar(y):
...     y = "some string"
>>> def baz(z):
...     return None
>>> foo(1)
1
>>> bar(2)
>>> baz(3)
>>> foo(None)
>>> print("some other string")
some other string
```

2.1.2 `bool`

Boolean values in Python are by far the easiest to check with doctests because they have a static string representation and only two possible values. The string representation of `bool``s is the same as their variable names: ``True` and `False`.

```
>>> True
True
>>> False
False
>>> def is_even(x):
...     return x % 2 == 0
>>> is_even(1)
False
>>> is_even(4)
True
```

2.1.3 `int`, `float`, and other numeric types

Numeric types are the most difficult to test for two reasons: there are several different types of numeric values (e.g. `int`, `float`, and all of the NumPy types) and rounding errors can occur based on how and when students choose to round off calculations. For these reasons, unless you're working with integers, it is usually easiest to use functions that return boolean values to compare numeric values.

When working with integers, almost all data types that represent them have the same string representation. For this reason, it is a relatively easy thing to write doctests that compare integer values with each other:

```
>>> from math import factorial
>>> factorial(4)
24
>>> def square(x):
...     return x**2
>>> square(25)
625
```

If, however, it is possible for the numbers to be floating point values, other methods of comparison are better-suited to writing doctests. One of the best examples is NumPy's `isclose` function, which compares two values to each other within an adjustable tolerance (which defaults to `1e-8`). Because NumPy supports both single and double precision floating point values, rounding errors can occur even when performing the same operation on values represented in different precision data types. This is why using functions that perform comparisons and return boolean values is much more robust to all of the ways that students can format their answers.

```
>>> def divide(a, b):
...     return a / b
>>> divide(divide(5, 3), 3)       # solution (a)
0.5555555555555556
>>> divide(5, 3)                  # solution (b)
1.6666666666666667
>>> divide(1.66666667, 3)         # solution (b) cont.
0.5555555566666667
```

Note that while solutions (a) and (b) above are both substantially correct, the rounding in solution (b) cause the otputs to be different, so if a test using solution (a) check a student's response solution (b), the student would fail the test. Using a function like `np.isclose`, this is avoided:

```
>>> np.iclose(
...     divide(divide(5, 3), 3),     # solution (a)
...     divide(1.66666667, 3)        # solution (b)
... )
True
```

Because booleans have easy-to-compare string representations, this test is much more robust to all of the possible sooutions to this question, and demonstrates the best practice for comparing numeric values. (Note that NumPy also provides `np.allclose` for element-wise comparison of values in iterables.)

2.1.4 `str`

String comparisons are relatively easy and the most straightfoward because doctests are based on string comparison. The main concern is to be careful of leading and trailing whitespace and to note that unless the `'` character appears **in** the string, Python's default string delimeters are apostrophes. If both appear, then apostrophes are used and the apostrophe in the string is escaped:

```
>>> 'some string'
'some string'
>>> "some'string"
"some'string"
>>> "some string"
'some string'
>>> """some string"""
'some string'
>>> '''some string\n'''
'some string\n'
>>> '''some string '"\n'''
'some string \'"\n'
```

2.1.5 other data types

Other data types don't have very many complexities surrounding them. For custom objects, note what their `__repr__` function returns and use that. When creating and testing custom classes, always use a custom `__repr__` function, otherwise the representation will contain the pointer to the object in memory, which changes between sessions.

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
>>> Point(1, 2)          # this has no __repr__, so it will have the object id
<__main__.Point object at 0x102cb3ac8>
>>> class OtherPoint:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __repr__(self):
...         return f"OtherPoint(x={self.x}, y={self.y})"
>>> OtherPoint(1, 2)    # this has a __repr__, so it will be printed without the id
OtherPoint(x=1, y=2)
```

Always test your tests in a Python interpeter if you're unsure about the string representation of an object. Don't use a Jupyter Notebook or IPython, because they don't necessary have the same output and they have different prompts.

CHAPTER 3

Seeding

An important aspect of writing code in the data science environment is the ability to simulate random processes. This presents a unique challenge to autograding assignments because the random state of the students' environments will be completely different from the grading environment. Some autograding solutions account for this issue by incorporating calls to seed random libraries during execution; Otter-Grader does this very easily using grading configurations, and this is by far the most preferable solution if you're using Otter.

When working with other autograding solutions, seeds must be incorporated into the test files themselves and assignments must be written in such a way as to make the seeding *before* execution possible.

3.1 Solution 1: Functions

The first and easiest solution to this problem is to have students wrap their code involving randomness into a function. In this way, the tests can make a seeding call before calling the function directly, ensuring that the seed set before the code is executed. Consider the following example question and the following test:

```
import numpy as np

# Question 1: Define roll_die which rolls a 6-sided die using NumPy.
def roll_die():
    return np.random.choice(np.arange(1, 7))
```

```
>>> np.random.seed(42)
>>> roll_die()
4
```

This approach, while easy-to-implement and effective, is generally at odds with the general autograding paradigm of testing global variables directly rather than encapsulting logic in needless functions. For this reason, there is another solution that is perhaps more elegant.

3.2 Solution 2: Seeding the Students' Code

In this solution, the random libraries are seeded directly before the student writes their code in a manner visible to the students.

```
import numpy as np

np.random.seed(42)

# Question 1: Assign roll_value to the roll of a six-sided die.
roll_value = np.random.choice(np.arange(1, 7))
```

```
>>> np.random.seed(42)
>>> roll_die()
4
```

This method is significantly less elegant and is susceptible to students changing the seed, which could throw off the results of the test and fail students incorrectly. It is also important to note that if you're working in a notebook format, you need to include seeds im *every* randomness cell because repeated runs of cells will change the seed value for subsequent questions, which could result in students failing tests.

CHAPTER 4

OK Test Format

Most autograders developed at UC Berkeley, including Otter-Grader, rely on the OK test format, originally developed for OkPy. This test format relies on *doctests* to check that students' code works correctly.

The OK format is very simple: a test is defined by a single Python file that contains a global variable called `test` which is assigned to a dictionary containing test configurations. The structure of this dictionary is:

- `test["name"]` is the name of the test case, and should be a valid Python variable name
- `test["points"]` is the point value of the test case; points are assigned all-or-nothing and all test cases (more below) must pass to be awarded the points
- `test["suites"]` is a list of dictionaries that correspond to test suites, groups of test cases; a `suite` consists of:
  - `suite["cases"]` is a list of dictionaries that correspond to test cases (individual doctests); a `case` consists of:
    - `case["code"]` is a Python interpreter-formatted string of code to be executed
    - `case["hidden"]` is a boolean that indicates whether the test case is hidden from students
    - `case["locked"]` is a boolean that indicates whether the test case is locked; this configuration is generally only relevant to OkPy
  - `suite["score"]` is a boolean that indicates whether the suite is part of the score
  - `suite["setup"]` is a code string that is run before the individual test cases are run
  - `suite["teardown"]` is a code string that is run after the individual test cases are run
  - `suite["type"]` is a string indicating the type of test; this is almost always going to be `"doctest"` unless you're using OkPy

Note that graders like Otter-Grader only allow test files with a single test suite.

```
test = {
    "name": "q1",
    "points": 1,
```

(continues on next page)

(continued from previous page)

```
    "suites": [
        {
            "cases": [
                {
                    "code": r"""
                    >>> foo()
                    'bar'
                    """,
                    "hidden": False,
                    "locked": False
                }
            ],
            "scored": True,
            "setup": "",
            "teardown": "",
            "type": "doctest"
        }
    ]
```

CHAPTER 5

Master Notebook Tools

Most UC Berkeley autograders include tools to abstract away the generation of test files by allowing users to define the tests along with questions and solutions in a simple notebook format. Tools like this include jAssign for OkPy and Otter Assign for Otter-Grader.

Most of these tools rely on the *output* of comment-delimited code cells to generate the doctest- formatted test cases used in test files and parse these notebooks out into two versions: a directory containing the notebook with solutions and hidden tests, and a directory with the notebook stripped of solutions and only public tests.

In general, it is highly recommended that you use one of these tools as they ensure the gradeability of assignments and make the process of creating and distributing assignments much easier. Otter Assign also integrates nicely with its other tools and makes the process of using Otter with 3rd part services much easier.

This guide considers best-practices for writing Python autograder tests. While this guide is focused mainly on OK-formatted tests (those used in Berkeley's OkPy and Otter autograders), some of the ideas discussed are applicable to all Pythonic autograders and the tests written for them. The topics discussed herein include test formatting, ok-formatted test files, structuring tests, and working with randomness.