INF200_H21_Ju03

May 31, 2022

1 INF200 Lecture No Ju03

1.0.1 Hans Ekkehard Plesser / NMBU

1.0.2 1 June 2022

1.1 Today's topics

- Keeping your code tidy
- More on testing
  - Levels of testing
  - File layout
  - Suggestions for test design
  - Approximate comparisons
  - Test parameterization
  - Test classes with setup and teardown features
  - Mocking
  - Tests involving randomness

---

2 Keeping your code tidy

- Run `Code > Inspect code` regularly on your code
- Fix weaknesses reported
- Also keep an eye on typos
- In-class example: `examples/biolab_project`

---

3 Levels of testing

- *unit tests* are tests of small parts of code
  - test individual methods
- *integration tests* test that the parts of a larger project work together
  - test that class instances behave as expected
  - expect that a class, e.g., representing a landscape cell, properly manages animals
- *acceptance tests* test that the software as a whole
  - `check_sim.py`
  - `test_biosim_interface.py`

  - similar simulations, e.g., with parameter modifications
    * different islands and initial populations
    * parameter choices preventing birth, death, eating, movement, …
- *regression tests* are added when a bug is discovered
  - the test reproduces the bug
  - when the bug is fixed, the test passes
  - we keep the test, in case we should re-introduce the bug by a later change (regression)

---

4 File layout

- You should write different test modules (files) to keep everything neat and organized
- Rule of thumb: One test module for each module in your package
  - `animals.py` —> `test_animals.py`
  - `landscape.py` —> `test_landscape.py`
  - …
- Each individual test should have a descriptive name
  - When a test fails, the first thing you read is the name
    * Should describe what was tested and failed
  - Should write a docstring to further explain the test

4.1 Placement of tests

- Two alternatives, no definite "best" solution
- See course repository `examples`
- Both variants can be run in the same way from PyCharm by adding a suitable PyTest configuration
- **We will use variant 1**

4.1.1 Variant 1: tests parallel to code directory

- Based on recommendations by the Python Packaging Project

```
chutes_project/
   src/
       chutes/
          __init__.py
          board.py
          ...
   examples/
   tests/
      test_board.py
   setup.py
```

- `tests` is a directory "parallel" to `chutes` code directory
- `tests` is *not* a package
- Test files use absolute imports

```
from chutes.board import Board
```

- PyTest configuration in PyCharm should cover `tests` directory

4.1.2 Variant 2: tests in code directory

```
chutes_project_alt/
    src/
        chutes/
          __init__.py
          board.py
          ...
          tests/
             __init__.py
             test_board.py
   examples/
   setup.py
```

- `tests` is subdirectory of `chutes` code directory
- `tests` is a package (contains `__init__.py`)
- Test files use relative imports

```
from ..board import Board
```

- PyTest configuration in PyCharm should cover `chutes/tests` directory

---

5 Suggestions for test design

- Test code should be simple: if you cannot understand a test, it is not worth much
- Have only a single `assert` in each test: the test fails on the first failing assert, all checks in later asserts will not be performed
- If you use "magic values", document how you obtained them or best, compute them explicitly (but do not copy-paste code!)
- Use variables for input values instead of literal numbers—improved reliability

5.1 Poor example

```python
def test_growing():
    a = Baby()
    for _ in range(10):
        a.grow()
    assert a.age == 10
    assert a.height == 55
```

5.2 Good example

```python
def test_age_increase():
    num_days = 10
    baby = Baby()
    for _ in range(num_days):
```

```python
        baby.grow()
    assert baby.age == num_days

def test_height_increase():
    num_days = 10
    baby = Baby()
    for _ in range(num_days):
        baby.grow()
    assert baby.height == baby.birth_height + num_days * baby.growth_rate
```

---

6 Approximate comparisons

[1]:
```python
import numpy as np
```

[2]:
```python
from pytest import approx
```

Check if two numbers are equal to within a relative error of 10^{-6}

[3]:
```python
3.001 == approx(3)
```

[3]: False

[4]:
```python
3.0000001 == approx(3)
```

[4]: True

Comparing to zero uses absolute error of 10^{-12}

[5]:
```python
0.0001 == approx(0)
```

[5]: False

[6]:
```python
0.0000000000001 == approx(0)
```

[6]: True

Approximate comparisons also work for composite data types:

[7]:
```python
[1.000001, 3] == approx([1.000001, 3])
```

[7]: True

[8]:
```python
{'a': 1.000001, 'b': 3} == approx({'a': 1.000001, 'b': 3})
```

[8]: True

```python
[9]: np.array([1.000001, 3]) == approx(np.array([1.000001, 3]))
```

```
[9]: True
```

See https://docs.pytest.org/en/latest/reference.html#pytest-approx for details.

---

7 Test parameterization

- Parameterize tests: run one test several times with different values
- For more information, see http://pytest.readthedocs.io/en/latest/parametrize.html#parametrize

7.0.1 Poor example

```python
def test_default_board_adjustments():
    """Some tests on default board."""

    brd = Board()
    assert brd.position_adjustment(1) == 39
    assert brd.position_adjustment(2) == 0
    assert brd.position_adjustment(33) == -30
```

7.0.2 Better solution with parameterization

```python
@pytest.mark.parametrize("from_pos, to_pos",
                         [[1, 40],
                          [2, 2],
                          [33, 3]])
def test_default_board_adjustments(from_pos, to_pos):
    """Test chutes and ladders on default board."""

    brd = Board()
    assert from_pos + brd.position_adjustment(from_pos) == to_pos
```

---

8 Test classes with setup and teardown fixtures

- We can combine tests that are related into a class
- The class name must begin with `Test`
- Each method with a name beginning with `test_` will be run as a test
- Methods with other names can be used as helpers
- Most important helpers: setup and teardown fixtures
  - http://pytest.readthedocs.io/en/latest/fixture.html#fixture
  - PyTest-related material at http://pythontesting.net/start-here/
- How it works
  - Create method that does preparation for tests or cleanup after tests
  - Mark method as PyTest fixture with `@pytest.fixture` decorator

    - Fixtures with `autouse=True` will be applied to every test in the class
    - Other fixtures will only be used if passed to a test method
    - Code before `yield` is run before the test (setup)
    - Code after `yield` is run after the test (teardown), independent of whether the test fails or not
    - If there is no `yield`, the method only performs setup
- See `january_block/examples/biolab_project` for examples
- Note: fixtures can also be defined at the module level, but then it is difficult to share objects created during setup with the tests

```python
class TestDeathDivision:

    @pytest.fixture(autouse=True)
    def create_dish(self):
        self.n_a = 10
        self.n_b = 20
        self.dish = Dish(self.n_a, self.n_b)

    @pytest.fixture
    def reset_bacteria_defaults(self):
        # no setup
        yield

        # reset class parameters to default values after each test
        Bacteria.set_params(Bacteria.default_params)

    def test_death(self):
        n_a_old = self.dish.get_num_a()
        n_b_old = self.dish.get_num_b()

        for _ in range(10):
            self.dish.death()
            n_a = self.dish.get_num_a()
            n_b = self.dish.get_num_b()
            # n_a and n_b must never increase
            assert n_a <= n_a_old and n_b <= n_b_old
            n_a_old, n_b_old = n_a, n_b

    def test_division(self):
        n_a_old = self.dish.get_num_a()
        n_b_old = self.dish.get_num_b()

        for _ in range(10):
            self.dish.division()
            n_a = self.dish.get_num_a()
            n_b = self.dish.get_num_b()
            # n_a and n_b must never decrease
            assert n_a >= n_a_old and n_b >= n_b_old
```

```python
        n_a_old, n_b_old = n_a, n_b

def test_all_die(self, reset_bacteria_defaults):
    Bacteria.set_params({'p_death': 1.0})
    self.dish.death()
    assert self.dish.get_num_a() == 0 and self.dish.get_num_b() == 0

@pytest.mark.parametrize("n_a, n_b, p_death",
                         [[100, 200, 0.1],
                          [100, 200, 0.9],
                          [10, 20, 0.5]])
def test_death(self, reset_bacteria_defaults, n_a, n_b, p_death):

    Bacteria.set_params({'p_death': p_death})
    dish = Dish(n_a, n_b)
    dish.death()
    died_a = n_a - dish.get_num_a()
    died_b = n_b - dish.get_num_b()

    pass_a = binom_test(died_a, n_a, p_death) > ALPHA
    pass_b = binom_test(died_b, n_b, p_death) > ALPHA

    assert pass_a and pass_b
```

---

9 Mocking

- Temporarily replace a Python object with a different one, typically replacing a class or method
- Supported by Python `unittest.mock`
  - Relatively complex
  - We will not use it directly
  - For documentation, see
    * https://docs.python.org/3/library/unittest.mock-examples.html
    * https://docs.python.org/3/library/unittest.mock.html#the-mock-class
- For convenient mocking with py.test, we need a py.test extension `pytest-mock`
  - For documentation, see https://github.com/pytest-dev/pytest-mock/

9.1 Example: Replacing random generator with fixed value

- See also `chutes_project/tests/test_player.py`
- In the test below, `random.randint` is replaced by a function that always returns `1`. The modification is in force only in that test.

```python
def test_single_step_one(mocker):
    mocker.patch('random.randint', return_value=1)
    b = Board(chutes=[], ladders=[])
    pl = Player(b)
```

```python
    pl.move()
    assert pl.position == 1
```

- `mocker` is automatically provided by py.test if the `pytest-mock` extension is installed, no imports required

9.1.1 Example: Counting the number of calls to a method

- See `examples/biolab_project/biolab/tests/test_dish.py`

```python
class TestAgingCalls:
    def test_dish_ages(self, mocker):
        mocker.spy(Bacteria, 'ages')

        n_a, n_b = 10, 20
        d = Dish(n_a, n_b)
        d.aging()

        assert Bacteria.ages.call_count == n_a + n_b
```

- `mocker.spy()` wraps `Bacteria.ages` so we can extract information later
- `Bacteria.ages.call_count` gives the number of times `Bacteria.ages` has been called
- The "spy" has an effect only inside this test

---

10 Tests involving randomness

- Test methods that depend on random numbers
- Exact results will depend on precise sequence of random numbers generated, i.e., on the random generator used and the random seed

10.1 Brute-force approaches

10.1.1 Fixed seed

By seeding the random number generator with a fixed value, we can ensure that we always get the same sequence of random numbers; particularly important while debugging.

- Requires that we know which random number generator is used by methods tested
- Adding more tests or changing tests or code can change the way in which random numbers are consumed

10.1.2 Mocking

Mock the random number function to return a fixed value.

- Allows us to check that the code using the random numbers works as expected
- Does not test whether the result has the expected distribution
- Requires that we know exactly how the code draws random numbers (white box testing)

10.2 Statistical tests

- The principal approach is based on statistical testing of hypothesis
  - Formulate a hypothesis (expectation), e.g., "value x is a sample of random variable X which has a normal (Gaussian) distribution of given mean μ and variance σ"
  - Find the p-value of x, i.e., the probability to observe a value at least as far from the mean as x if x indeed follows the assumed distribution
  - Compare the p-value to a predefined acceptance limit α: if $p > \alpha$ the test is passed
- Interpretation: Let, e.g., $\alpha = 0.01 = 1\%$. If we observe a value x with a p-value less than $\alpha = 1\%$, this means that the value x belongs to the outer tail of the assumed distribution, among those values that make up the 1% least likely values in the distribution. We thus assume that x did not come from the expected distribution and declare the test failed.
- Note: By construction, this test will fail in 1% of all cases even if x follows the assumed distribution. Thus, failures need to be inspected carefully.
- See, e.g., Knuth, The Art of Computer Programming, vol 2.

10.2.1 Examples of statistical tests

- Z-test
  - Strictly speaking, tests whether the mean of n random values drawn independently from the same distribution is from a Gaussian distribution of given mean and variance
  - Due to the central limit theorem, it can also be applied in many other cases as an approximation provided we are considering averages of many trials
  - If the variance of the Gaussian distribution is not know a priori, one should use Student's t-test instead
- Binomial test
  - An explicit test for binomially distributed quantities, e.g., the number of successes in n Bernoulli experiments (coin flips)
  - See also GraphPad for an explanation of the test. The binomial test in SciPy uses the same approach as GraphPad
- `scipy.stats` provides a number of statistical test functions

[ ]:
```

```