How I learned to stop worrying and love unit tests

https://github.com/krasch

Is my code correct? What if I publish my results and everything was wrong ??

Unit testing will give you

- better sleep
- faster workflow
- better (simpler) code
- better documentation

Outline

1. Very brief context
2. Let’s make some tests
3. When, how, how often to run tests?
4. Unit testing for data scientists

Types of testing

Unit testing	test individual functions in isolation
Integration testing	test groups of functions / modules
Validation testing	is software following the spec?
System testing	test whole system, on the hardware where it is supposed to be run
Stress testing	test system beyond normal operational capacity
Penetration testing	test security aspects of the system

White box vs black box testing

White box: you know the implementation of the system under test

Black box: you do not know anything about the implementation

Unit tests are typically white box

Running example: Function to impute an array

- take one numpy array
- calculate mean
- replace all NaN with mean
- return numpy array

Function to impute an array

```python
def impute(data):
    # which indexes hold actual values, which hold NaN?
    is_nan = np.isnan(data)
    is_finite = ~is_nan

    # replace all NaN with mean value
    data = data.copy()
    data[is_nan] = np.mean(data[is_finite])

    return data
```

Example result of applying the impute function

```
input: [1.2, 2.8, np.nan, 8]
output: [1.2, 2.8, 4, 8]
```

-> lets make this into a unit test!

First unit test

```python
from numpy.testing import assert_array_equal

def test_impute_one_value():
    # 1. Define some input data
    data = np.array([1.2, 2.8, np.nan, 8])
    # 2. Define what is expected to happen
    expected = np.array([1.2, 2.8, 4, 8])
    # 3. Run function and record what actually happens
    actual = impute(data)
    # 4. Make sure expected and actual are equal
    assert_array_equal(expected, actual)
```

Typically unit tests follow this pattern

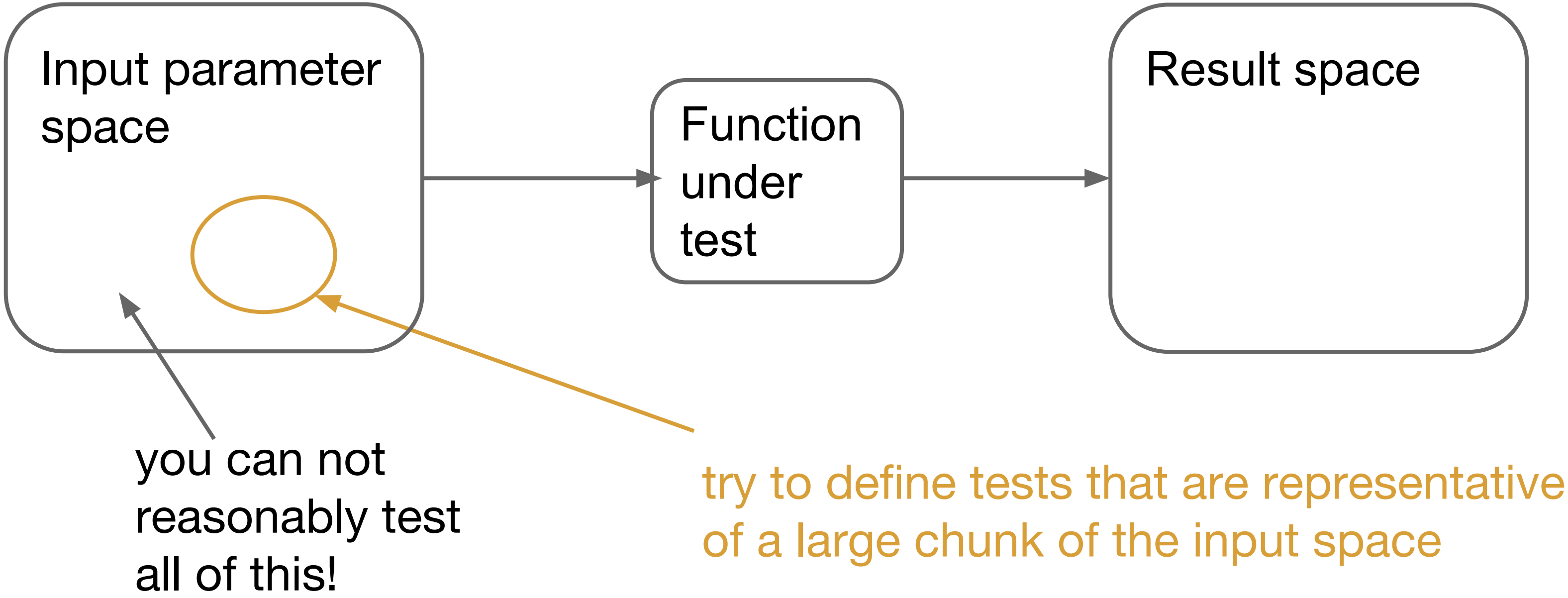
1. Define some input data
2. Define what is expected to happen
3. Run function and record what actually happens
4. Make sure expected and actual results are equal

Assertions compare actual and expected results

```python
# you could use the built-in assert function
assert(5 == 3)

# but there are many specialised functions that make your life easier
from nose.tools import assert_equal, assert_false,
                 assert_list_equal, assert_dict_contains_subset
from numpy.testing import assert_array_equal,
                         assert_array_almost_equal
from pandas.util.testing import assert_frame_equal
```

(each of these libraries contain many more assert methods)

Try to write representative tests



Input parameter space

Function under test

Result space

you can not reasonably test all of this!

try to define tests that are representative of a large chunk of the input space

Representative tests for the impute function

We tested:

```
[1.2, 2.8, np.nan, 8]
```

It does not make sense to test

```
[1.3, 0.2, np.nan, 2], [3.2, 18.2, np.nan, 7]
```

It could make sense to test

```
[np.nan, 0.2], [3.2, np.nan, np.nan, 7],
```

It definitely makes sense to test

```
[1, 2, 3, 8], [1, -2, -11, np.nan],

[1], [], [np.nan, np.nan, np.nan]
```

Second unit test

```python
def test_nothing_to_impute():
    # 1. Define some input data
    data = np.array([1.2, 4, 8, 7])
    # 2. Define what is expected to happen
    expected = np.array([1.2, 4, 8, 7])
    # 3. Run function and record what actually happens
    actual = impute(data)
    # 4. Make sure expected and actual are equal
    assert_array_equal(expected, actual)
```

Third unit test

```
def test_all_values_nan():
    # 1. Define some input data
    data = np.array([np.nan, np.nan, np.nan])
    # 2. Define what is expected to happen
    ??????
```

What should happen?

- Function returns `[np.nan, np.nan, np.nan]` or
- Function throws an error

Update impute function

```
class ImputationError(Exception):  # our own exception!
    pass

def impute(data):
    # which indexes hold actual values, which hold NaN?
    is_nan = np.isnan(data)
    is_finite = ~is_nan
    # at least one value should be an actual number
    if is_finite.sum() == 0:
        raise ImputationError("All values are NaN")
    # replace all NaN with mean value
    data = data.copy()
    data[is_nan] = np.mean(data[ is_finite])
    return data
```

Back to third unit test

```
from nose.tools import raises

@raises(ImputationError) # 2. What is expected to happen
def test_all_values_nan():
   # 1. Define some input data
   data = np.array([np.nan, np.nan, np.nan])
   # 3. Call function
   impute(data)
```

(@raises takes care of step 4, if exception is not thrown, the test will fail)

Impute function should not change input array

```python
def test_input_array_unchanged():
    # 1. Define some input data
    data = np.array([1.2, 4, np.nan, 7])
    # 2. Define what is expected to happen
    expected = data.copy()
    # 3. Run function
    impute(data)
    actual = data
    # 4. Make sure expected and actual are equal
    assert_array_equal(expected, actual)
```

Representative test cases ≈ edge cases

Things that are typically worth testing:

- empty arrays
- arrays with only one value
- zero, negative numbers
- very large / very small numbers
- different data types, e.g. int vs float
- non-ascii characters (*Liberté, égalité, fraternité*)

One test tests only one specific thing

```python
def test_various_things():
    data1 = np.array([1.2, 2.8, np.nan, 8])
    expected1 = np.array([1.2, 2.8, 4, 8])
    actual1 = impute(data1)
    assert_array_equal(expected1, actual1)
    # never do this!
    data2 = np.array([2, 2, np.nan, 8])
    expected2 = np.array([2, 2, 4, 8])
    actual2 = impute(data2)
    assert_array_equal(expected2, actual2)
```

This is really bad practice.
If test fails, you will not be able to immediately see which inputs failed

Bundle tests and setup code in test classes

```
from unittest import TestCase

class TestVariousRelatedSomethings(TestCase):
    # this function will be called before every test method
    def setUp(self):
        # e.g. set up some temporary workdir
    # this function will be called after every test method
    def tearDown(self):
        # delete that temporary workdir

    def test_something(self):
        # ...
    def test_something_else(self):
        # ...
```

calling stuff Test...
and test_... makes sure
that all tests are found
and run

Unit tests can live in the same folder as your code

Code layout:

- config.py
- model
    - features.py
    - util.py <- *contains impute function*
    - test_util.py

Unit tests can live in an extra “test” folder

Code layout:

- config.py
- model
    - features.py
    - util.py   <- *includes impute function*
- test
    - test_util.py

Unit tests can live in your documentation (doctests)

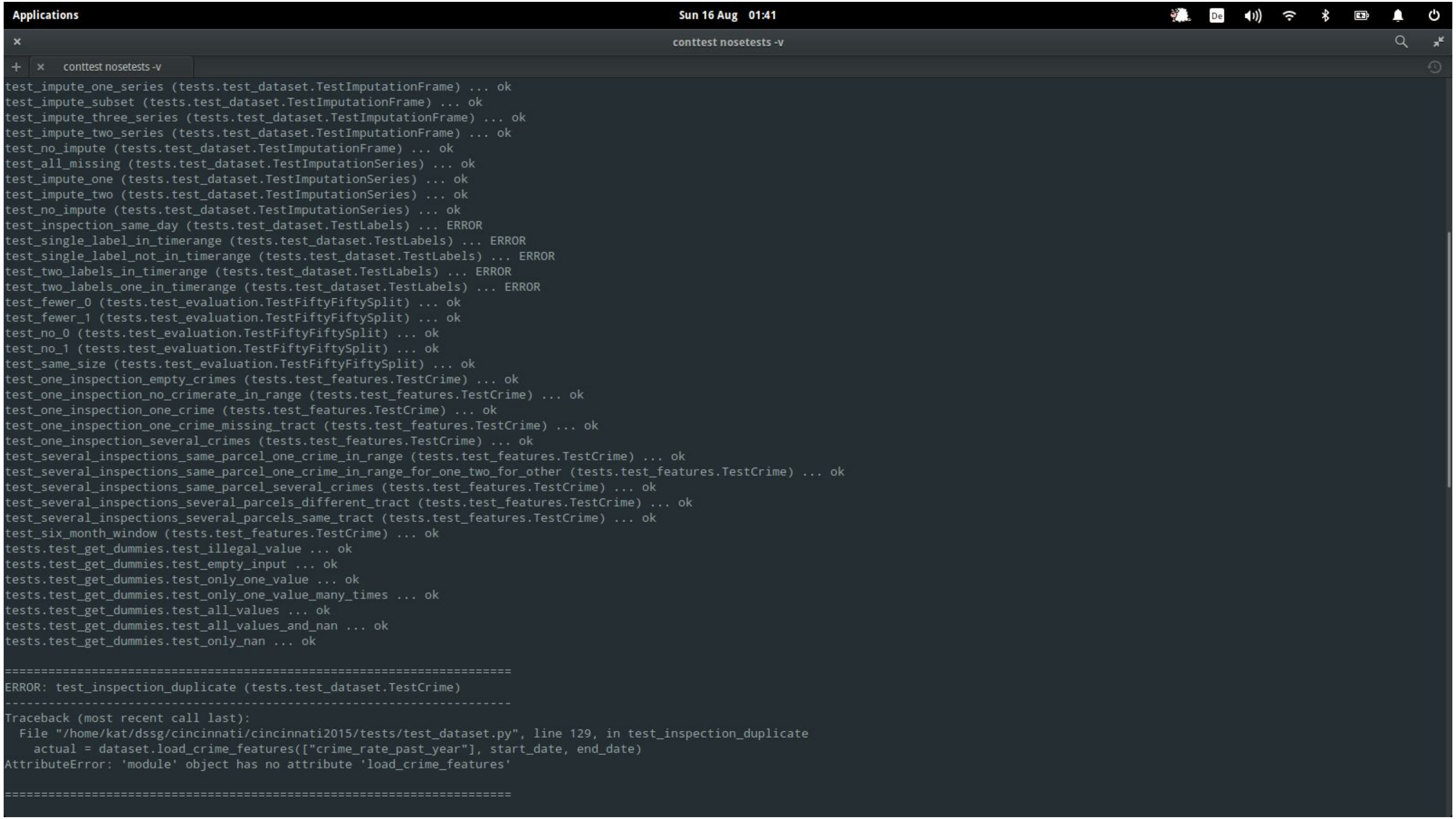
```
def impute(data):
    """
    >>> impute(np.array([1.2, 2.8, np.nan, 8]))
    array([ 1.2,  2.8,  4. ,  8. ])

    >>> impute(np.array([1, 2, 3, 4]))
    array([1, 2, 3, 4])
    """
    [Continue with function implementation here]
```

Running unit tests

- I like using the nose library
- it looks for everything that has a name that includes “test” and tries to run it as a unit test

```
pip install nose or conda install nose
nosetests
```

Nose tells you which tests passed/failed

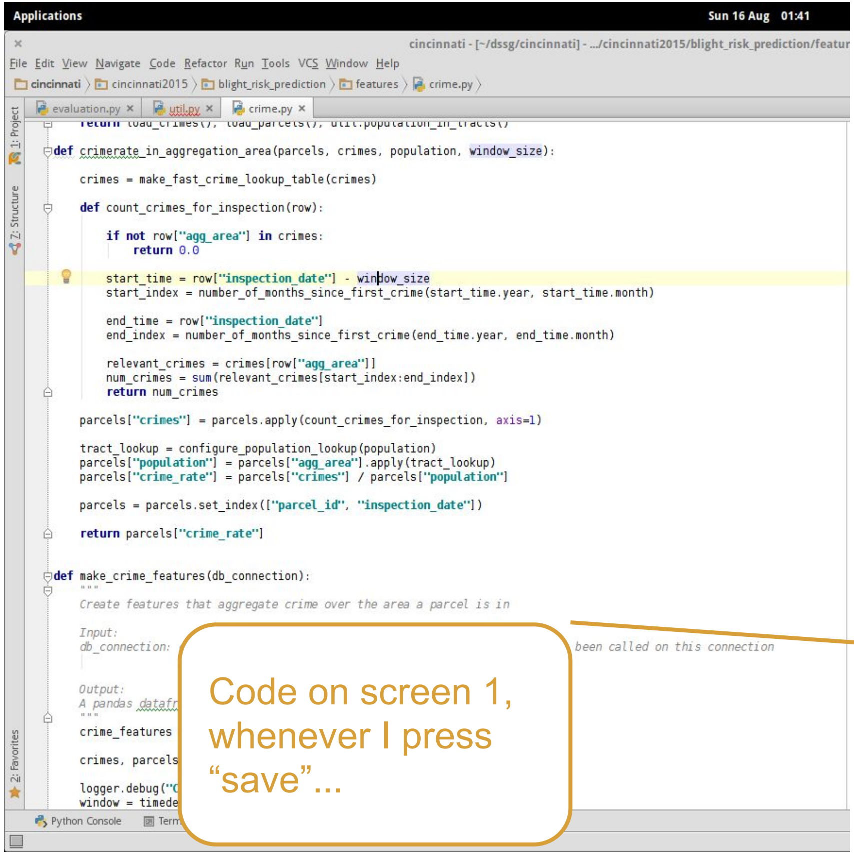


```
test_impute_one_series (tests.test_dataset.TestImputationFrame) ... ok
test_impute_subset (tests.test_dataset.TestImputationFrame) ... ok
test_impute_three_series (tests.test_dataset.TestImputationFrame) ... ok
test_impute_two_series (tests.test_dataset.TestImputationFrame) ... ok
test_no_impute (tests.test_dataset.TestImputationFrame) ... ok
test_all_missing (tests.test_dataset.TestImputationSeries) ... ok
test_impute_one (tests.test_dataset.TestImputationSeries) ... ok
test_impute_two (tests.test_dataset.TestImputationSeries) ... ok
test_no_impute (tests.test_dataset.TestImputationSeries) ... ok
test_inspection_same_day (tests.test_dataset.TestLabels) ... ERROR
test_single_label_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_single_label_not_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_two_labels_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_two_labels_one_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_fewer_0 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_fewer_1 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_no_0 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_no_1 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_same_size (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_one_inspection_empty_crimes (tests.test_features.TestCrime) ... ok
test_one_inspection_no_crimerate_in_range (tests.test_features.TestCrime) ... ok
test_one_inspection_one_crime (tests.test_features.TestCrime) ... ok
test_one_inspection_one_crime_missing_tract (tests.test_features.TestCrime) ... ok
test_one_inspection_several_crimes (tests.test_features.TestCrime) ... ok
test_several_inspections_same_parcel_one_crime_in_range (tests.test_features.TestCrime) ... ok
test_several_inspections_same_parcel_one_crime_in_range_for_one_two_for_other (tests.test_features.TestCrime) ... ok
test_several_inspections_same_parcel_several_crimes (tests.test_features.TestCrime) ... ok
test_several_inspections_several_parcels_different_tract (tests.test_features.TestCrime) ... ok
test_several_inspections_several_parcels_same_tract (tests.test_features.TestCrime) ... ok
test_six_month_window (tests.test_features.TestCrime) ... ok
tests.test_get_dummies.test_illegal_value ... ok
tests.test_get_dummies.test_empty_input ... ok
tests.test_get_dummies.test_only_one_value ... ok
tests.test_get_dummies.test_only_one_value_many_times ... ok
tests.test_get_dummies.test_all_values ... ok
tests.test_get_dummies.test_all_values_and_nan ... ok
tests.test_get_dummies.test_only_nan ... ok

======================================================================
ERROR: test_inspection_duplicate (tests.test_dataset.TestCrime)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/kat/dssg/cincinnati2015/tests/test_dataset.py", line 129, in test_inspection_duplicate
    actual = dataset.load_crime_features(["crime_rate_past_year"], start_date, end_date)
AttributeError: 'module' object has no attribute 'load_crime_features'

======================================================================
```

Re-run tests everytime you change your code

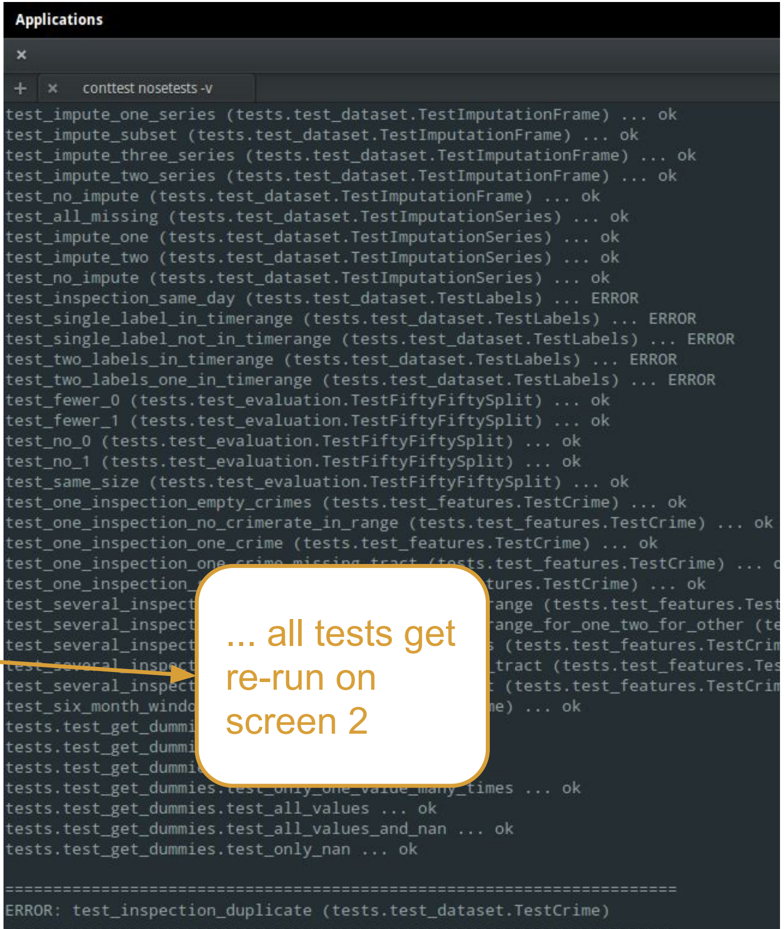
guards against regression (something that used to work that was broken by some idiot (probably you))

Continuous testing using conttest library


Applications
Sun 16 Aug 01:41
cincinnati - [~/dssg/cincinnati] - .../cincinnati2015/blight_risk_prediction/featur
File Edit View Navigate Code Refactor Run Tools VCS Window Help
cincinnati › cincinnati2015 › blight_risk_prediction › features › crime.py
evaluation.py util.py crime.py
1: Project
Z: Structure
2: Favorites
Python Console
Term

```
    return load_crimes(), load_parcels(), util.population_in_tracts()

def crimerate_in_aggregation_area(parcels, crimes, population, window_size):

    crimes = make_fast_crime_lookup_table(crimes)

    def count_crimes_for_inspection(row):

        if not row["agg_area"] in crimes:
            return 0.0

        start_time = row["inspection_date"] - window_size
        start_index = number_of_months_since_first_crime(start_time.year, start_time.month)

        end_time = row["inspection_date"]
        end_index = number_of_months_since_first_crime(end_time.year, end_time.month)

        relevant_crimes = crimes[row["agg_area"]]
        num_crimes = sum(relevant_crimes[start_index:end_index])
        return num_crimes

    parcels["crimes"] = parcels.apply(count_crimes_for_inspection, axis=1)

    tract_lookup = configure_population_lookup(population)
    parcels["population"] = parcels["agg_area"].apply(tract_lookup)
    parcels["crime_rate"] = parcels["crimes"] / parcels["population"]

    parcels = parcels.set_index(["parcel_id", "inspection_date"])

    return parcels["crime_rate"]


def make_crime_features(db_connection):
    """
    Create features that aggregate crime over the area a parcel is in

    Input:
    db_connection:                                  been called on this connection


    Output:
    A pandas datafr
    """
    crime_features

    crimes, parcels

    logger.debug("C
    window = timede
```

Code on screen 1, whenever I press "save"...



```
test_impute_one_series (tests.test_dataset.TestImputationFrame) ... ok
test_impute_subset (tests.test_dataset.TestImputationFrame) ... ok
test_impute_three_series (tests.test_dataset.TestImputationFrame) ... ok
test_impute_two_series (tests.test_dataset.TestImputationFrame) ... ok
test_no_impute (tests.test_dataset.TestImputationFrame) ... ok
test_all_missing (tests.test_dataset.TestImputationSeries) ... ok
test_impute_one (tests.test_dataset.TestImputationSeries) ... ok
test_impute_two (tests.test_dataset.TestImputationSeries) ... ok
test_no_impute (tests.test_dataset.TestImputationSeries) ... ok
test_inspection_same_day (tests.test_dataset.TestLabels) ... ERROR
test_single_label_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_single_label_not_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_two_labels_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_two_labels_one_in_timerange (tests.test_dataset.TestLabels) ... ERROR
test_fewer_0 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_fewer_1 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_no_0 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_no_1 (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_same_size (tests.test_evaluation.TestFiftyFiftySplit) ... ok
test_one_inspection_empty_crimes (tests.test_features.TestCrime) ... ok
test_one_inspection_no_crimerate_in_range (tests.test_features.TestCrime) ... ok
test_one_inspection_one_crime (tests.test_features.TestCrime) ... ok
test_one_inspection_one_crime_missing_tract (tests.test_features.TestCrime) ...
test_one_inspection_                          tures.TestCrime) ... ok
test_several_inspect                          ange (tests.test_features.Test
test_several_inspect                          ange_for_one_two_for_other (te
test_several_inspect                          s (tests.test_features.TestCrim
test_several_inspect                          tract (tests.test_features.Tes
test_several_inspect                          t (tests.test_features.TestCrim
test_six_month_windo                          me) ... ok
tests.test_get_dummi
tests.test_get_dummi
tests.test_get_dummi
tests.test_get_dummies.test_only_one_value_many_times ... ok
tests.test_get_dummies.test_all_values ... ok
tests.test_get_dummies.test_all_values_and_nan ... ok
tests.test_get_dummies.test_only_nan ... ok

======================================================================
ERROR: test_inspection_duplicate (tests.test_dataset.TestCrime)
```

... all tests get re-run on screen 2

When to write tests

Alternative 1: first write code, then write tests

Alternative 2: first write tests, then write code (test-driven development)

or be pragmatic: start implementing your function; when you feel you should try it out, just write a test instead of a main()

write a new test whenever you find a bug!

Testable code = every function does one thing only

Bad function:

- read data from database
- make features
- impute features

So many things can go wrong here and we would have to test every possible combination of things

Instead: split function into three functions, test each of them in isolation

Functions should have no outside dependencies

```
some_global_variable = True
def my_stupid_function(some_param):
    if some_global_variable:
      # do something with some_param
```

This will be really hard to unit test

How can we unit test typical steps in the data science process?

1. Clean and transform data and put into db
2. Read data from db
3. Make features and pre-process (e.g. imputation)
4. Machine learning stuff (scikit-learn probably)
5. Evaluation

Clean and transform data

- Usually one-off scripts, not proper modules
- My pragmatic approach: do not unit test this, but litter code with asserts that check that the data conforms to what you expect

```
assert(len(line.split(‘|’)) == 31)
assert(len(parcel_id) == 12)
```

(asserts can be turned off -> prefer custom exceptions in production code)

Read data from database

Testing with databases is HARD

Bad: test against your real database

- probably very slow (because database huge)
- you can not rely on the contents of the database

  -> your tests might fail
- not everybody will have access to your db
- inserting and deleting test data is error prone

Read data from database (continued)

Better: test against test database

- install database locally or use sqlite
- in setUp method fill database with only the necessary data
- pro: database contains what you expect, tests run fast
- con: much annoying admin stuff

Best:

- no idea...

Make features

- working with databases is hard

--> separate database querying from feature generation code as much as possible!

- test feature generation code with small test datasets

Testing crime-rate feature generation

```python
def test_several_inspections_several_parcels_different_tract():
    crimes = [("16Sep2014", "tract567", 3),
              ("18Oct2014", "tract568", 1),
              ("14Jul2014", "tract568", 6)]
    parcels = [("parcelA", "01Dec2014", "tract567"),
               ("parcelB", "18Nov2014", "tract568"),]
    population = [("tract567", 1234),
                  ("tract568", 203)]
    window = datetime.timedelta(days=365)

    expected = [("parcelA", "01Dec2014", 3.0 / 1234),
                ("parcelB", "18Nov2014", 7.0 / 203)]
    actual = crime.crimerate_in_aggregation_area(parcels, crimes, population, window)
    assert_array_equal(expected, actual)
```

Machine learning stuff

let’s just hope scikit-learn tested their code...

Evaluation

if you write your own evaluation code, you should unit test it (just imagine that your model is doing great but you don't know because of a bug in your evaluation function)

Unit tests are good documentation

- they are basically a spec by example
- inputs, outputs, expected failures

Testable code is better code

- every function does only one thing
- means that functions will not be super long
- functions have no external dependencies (e.g. evil global variables)

Unit tests make your workflow faster

- you work on functions in isolation
- you work with only a subset of the data
- that subset is well-defined

Caveats

Just because you unit tested something does not mean you know it is correct

There are bugs in your software that you never ever considered testing for

Also, maybe you understood the spec wrong
-> outside testers necessary

Summary

- Better sleep
- Better documentation
- Better code
- Faster workflow