Effective Programming Practices for Economists

15. Testing

Hans-Martin von Gaudecker

Department of Economics, Universität Bonn

universitätbonn

Why bother about testing?

- Extra code = extra work = inefficient?
- Nobody enjoys it
- Just walk out of the class if:
  - . . . your programs always work correctly, or . . .
  - . . . you don't care if they're correct or not, so long . . .
  - . . . as their output looks plausible, and . . .
  - . . . you like being inefficient
- **The more you invest in quality, the less total time it takes to get your code to work**

Testing tells you . . .

- . . . if the program is doing what it's supposed to
- . . . what the program actually is supposed to do
- **Tests are runnable specifications**
  - Less likely to fall out of sync with the program than documentation

Quality is not testing

Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often.

Steve McConnell

- Good tests localize problems
- Speed up debugging enormously
- Which is the main factor contributing to development time

Programming error isn't "classical"

Buggy code is biased towards expected results (↗):

> *If your software simulates some complex phenomena, you don't know what it's supposed to do; that's why you're simulating. Errors are easier to spot in consumer software. A climate model needs a higher level of quality assurance than a word processor because bugs in the latter are more obvious. Genomic analysis may contain egregious errors and no one will ever know, but a bug in an MP3 player is sure to annoy users.*
>
> John D. Cook

How much can you test?

- Test a function comparing 7-digit phone numbers
- 10^7 possible numbers
- $(10^7)^2$ possible pairs of numbers
- At one million tests per second, that's 155 days . . .
  - . . . for one simple function
  - And how would you actually *write* 10^{14} tests?

How much can you test?

- A failing test shows you there might be a problem
- Passing tests can never guarantee the absence of errors
- But you know that . . .
  - Hypothesis testing
  - Theory of science
- Nevertheless, we make progress in science . . .
  - Same here – it simply works very well

Different types of testing

- **Unit testing**
  - Test components in a program one-by-one in isolation
- **Integration testing**
  - Test how different but related units interact
- **System testing**
  - Integration tests taken to the extreme
- **Regression testing**
  - Test whether result is the same as before
  - Replication, essentially
  - Important when replacing an easy-to-understand algorithm by a fast, complicated one (e.g. written in C or Fortran)

Why return codes are not recommended

Assume you want to read from a file that may or may not exist

```
def read_parameters(parameters_file):
    if os.path.isfile(parameters_file):
        data = open(parameters_file, 'r').read()
        status = 0
    else:
        data = None
        status = 1
    return data, status


data, status = read_parameters(some_file)
if status != 0:
    print("Can't read {}, using default data".format(some_file))
    data = default_data
```

Why return codes are not recommended

- Expected flow of control and error handling intertwined
- Hard to see the forest (normal behaviour) for the trees (exceptional case)
- Net result: People stop bothering to check the status codes when programs become complicated
- Errors even harder to track down when they occur

Introducing exceptions

Separate “normal” operation from “exceptional” cases

```
def read_parameters_clean(parameters_file):
    return open(parameters_file, 'r').read()
```

```
try:
    data = read_parameters_clean(some_file)
except:
    print("Can't read {}, using default data".format(some_file))
    data = default_data
```

Exceptions are not new . . .

You've seen exceptions before:

```
>>> values = [0, 1, 2]
>>> values[99]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Catching exceptions

- But now we did not halt the program at the exception . . .
  - . . . which is Python's default behaviour, along with printing out the type of exception and the location where it occured
- The code in the `try` block is executed
- If an exception occurs, control moves to the `except` block
- Both blocks can hold an arbitrary number of lines
- We can handle different exception types in different ways

Catching exceptions

```
try:
    data = read_parameters_clean(some_file)
    data_array = np.array(data)
    this_will_raise_a_name_error
except IOError:
    print("Can't read {}, using default for data_array".format(some_file))
    data_array = np.array(default_data)
except ValueError as err:
    print("Data in {} is badly formatted".format(some_file))
    raise err
```

- Upon an error in the `try` block, Python searches for a compatible `except` statement
- If it finds one, the appropriate block is executed
- If no appropriate `except` statement is found, it's back to default exception handling (print the error, halt program)

Catching exceptions

- File not found (or no read permissions) leads to a message being printed and the default data being used
- Corrupted data in `some_file` leads to a message being printed, the standard exception being raised and the program halts
- The undefined variable `this_will_raise_a_name_error` leads to a `NameError` and the program halts as usual

Catching exceptions

- Store an exception object in `variable_name`:

```
except SomeException as variable_name:
```

- Retains all information about the exception
- `raise SomeException` let's you insert errors manually
- The following statements catch any type of exception:

```
except:
except Exception:
```

- Use with great care – hide errors you do **not** expect!!!

User-defined exceptions

- You could (should!) define exceptions yourself
- Easier to recognise when they occur
- More discrimination possibilities when catching them
- Do this by subclassing `Exception`

User-defined exceptions

```
class LambdaDomainError(Exception):

    def __init__(self, value):
        self.value = value

    def __str__(self):
        return 'Loss aversion must be positive, got lambda_={}' \
               .format(self.value)


class AgentKinked:

    def __init__(self, lambda_):
        self.set_preferences(lambda_)

    def set_preferences(self, lambda_):
        """Set the preference parameter and make a domain check."""
        if not lambda_ > 0:
            raise LambdaDomainError(lambda_)
        self.lambda_ = lambda_
```

User-defined exceptions

```
# Custom errors are easier to recognise.
AgentKinked(-5.0)
```

```
Traceback (most recent call last):
  File "error_handling_own_exception.py", line 22, in <module>
    AgentKinked(-5.0)
  File "error_handling_own_exception.py", line 18, in __init__
    self.set_preferences(lambda_)
  File "error_handling_own_exception.py", line 14, in set_preferences
    if not lambda_ > 0: raise LambdaDomainError(lambda_)
__main__.LambdaDomainError: Loss aversion must be positive, got lambda_=-
```

User-defined exceptions

```
# Custom errors allow better discrimination in exception handling.
# (imagine a more complex operation in the try block)
try:
    goofy = AgentKinked(-5.0)
except LambdaDomainError:
    goofy = None
```

Custom error messages

- Customise error messages of built-in exceptions using:

```
raise SomeBuiltInException("Custom message")
```

- Much quicker to program than custom exceptions

- Which one to use?
  - Think **DRY**
  - Necessary to catch specific exceptions?
  - Easier to do for custom exceptions

Custom error messages

```
class AgentKinked:

    def __init__(self, lambda_):
        self.set_preferences(lambda_)

    def set_preferences(self, lambda_):
        """Set the preference parameter and make a domain check."""
        if not lambda_ > 0:
            raise ValueError(
                'Loss aversion must be positive, got lambda_={}'.format(l
            )
        self.lambda_ = lambda_
```

Custom error messages

```
# Custom messages are quicker to program.
AgentKinked(-5.0)
```

```
Traceback (most recent call last):
  File "error_handling_custom_message.py", line 14, in <module>
    AgentKinked(-5.0)
  File "error_handling_custom_message.py", line 10, in __init__
    self.set_preferences(lambda_)
  File "error_handling_custom_message.py", line 6, in set_preferences +
    'got lambda_={}'.format(lambda_))
ValueError: Loss aversion must be positive, got lambda_=-5.0
```

Throw low, catch high

- The possibility to store an error means you can retain all information from an exception that occured deep down in your program and handle it at the appropriate level

- E.g. an OverflowError occurs in `exponential_utility()`
  - If you're using an optimiser that can handle infinity, print a warning and set the function value to infinity
  - Else, let the error halt the program or re-start the optimiser with different initial values

Throw low, catch high

- Remember the Zen . . .
  - Errors should never pass silently
  - Unless explicitly silenced
- The built-in exceptions are listed at

  `http://docs.python.org/3/library/exceptions.html`

Ingredients to a unit test

Fixture	Start with these two
Action	This is the “real” code we care about
Expected result	Start with these two
Actual result	
Report	

Why tests first?

- Disciplining effect: Think first, then code
  - I've been down **many** dead-end roads for sure
  - Tests first would have avoided a lot of them
- Pinpoint bugs right away, before they grow large
- Define the interface of a function without thinking too much about the internals
  - Higher level of abstraction

Why tests first?

- Analogy in business:
  - A CEO should think (mostly) about what the company's product does for the customer
  - Implementation is left to the engineer
- You're both in one person
- But you can't keep everything in mind at the same time

Testing the power utility function: Valid input

- Start with a very simple case, namely testing our code for

$$\frac{z^{1-\gamma}}{1-\gamma}$$

- **Fixture**: $z = 100$ and $\gamma = \frac{1}{2}$

- **Expected result**: 20

Testing the power utility function: Valid input

```
# Test the behaviour of utility_crra with typical input values.
if utility_crra(100.0, 0.5) == 20.0:
    print('Test of utility_crra(100.0, 0.5) ... okay.')
else:
    print('''Test of utility_crra(100.0, 0.5) ... ERROR.
Expected result: 20.0
Got result: {}'''.format(utility_crra(100.0, 0.5)))
```

Testing the power utility function: How to handle exceptions?

- Neeed to check the domains of z and γ
- Could use either of the following:

```
assert condition

if not condition:
    raise SomeException
```

Testing the power utility function: How to handle exceptions?

- Use `assert` to test conditions that should never happen while developing your programs
  - Crash early in the case of a corrupt program state
  - Aside: `assert` statements are removed when you run programs with optimisation turned on
- Use manually triggered exceptions for errors that can conceivably happen
  - Simpler to distinguish between various types of errors
  - Best create your own exception classes
  - Examples use string exceptions to show why . . .

Testing the power utility function: Negative argument

- Function to test remains the same:

$$\frac{z^{1-\gamma}}{1-\gamma}$$

- **Fixture**: $z = -100$ and $\gamma = \frac{1}{2}$

- **Expected result**:

```
ValueError("All values of 'argument' must be positive")
```

Testing the power utility function: Negative argument

```
# Test the behaviour of utility_crra with negative argument.
try:
    utility_crra(-100.0, 0.5)
    print('''Test of utility_crra(-100.0, 0.5) ... ERROR.
Expected result: ValueError("All values of 'argument' must be positive.")
Got result: {}'''.format(utility_crra(-100.0, 0.5)))
except ValueError as err:
    if str(err) == "All values of 'argument' must be positive.":
        print('Test of utility_crra(-100.0, 0.5) ... okay.')
    else:
        print('''Test of utility_crra(-100.0, 0.5) ... ERROR.
Expected result: ValueError("All values of 'argument' must be positive.")
Got result: ValueError({})'''.format(str(err)))
```

Testing the power utility function: Log utility

- **Fixture**: $z = 100$ and $\gamma = 1$

- **Expected result**: `ExponentDomainError`

```
class ExponentDomainError(Exception):

    def __str__(self):
        return "All values of 'exponent' must differ from one."
```

Testing the power utility function: Log utility

```
# Test the behaviour of utility_crra under log utility.
try:
    utility_crra(100.0, 1.0)
    print('''Test of utility_crra(100.0, 1.0) ... ERROR.
Expected result: ExponentDomainError
Got result: {}'''.format(utility_crra(100.0, 1.0)))
except ExponentDomainError:
    print('Test of utility_crra(100.0, 1.0) ... okay.')
except Exception as err:
    print('''Test of utility_crra(100.0, 1.0) ... ERROR.
Expected result: ExponentDomainError
Got another error: ''')
    raise err
```

Testing the power utility function: The function being tested

```
def utility_crra(argument, exponent):
    """Return the CRRA utility evaluation of *argument* where the Arrow-Pratt
    coefficient of relative risk aversion takes on the value *exponent*.

    Neither handle cases where (any of the elements of) *argument* is less than
    zero (raise ValueError), nor log utility, i.e. exponent == 1  (raise
    ExponentDomainError)

    """

    if (np.asarray(argument) <= 0).any():
        raise ValueError("All values of 'argument' must be positive.")
    else:
        pass
    if (np.asarray(exponent) == 1.0).any():
        raise ExponentDomainError
    else:
        pass

    return argument ** (1.0 - exponent) / (1.0 - exponent)
```

Testing the power utility function: Test results

```
Test of utility_crra(100.0, 0.5) ... okay.
Test of utility_crra(-100.0, 0.5) ... okay.
Test of utility_crra(100.0, 1.0) ... okay
```

Testing the power utility function: What to test?

- No clear-cut rules what ought to be tested
- Rule of thumb:
  - A couple of standard values
  - Corner cases
  - All exceptions
  - Bugs that did not make previous tests fail

Testing the power utility function: Checking risk neutrality

```
# Test the behaviour of utility_crra under risk neutrality.
if utility_crra(100.0, 0.0) == 108.0:
    print('Test of utility_crra(100.0, 0.0) ... okay.')
else:
    print('''Test of utility_crra(100.0, 0.0) ... ERROR.
Expected result: 100.0
Got result: {}'''.format(utility_crra(100.0, 0.0)))
```

Test results:

```
Test of utility_crra(100.0, 0.0) ... ERROR.
Expected result: 100.0
Got result: 100.0
```

Automate, automate, automate . . .

- We could have gotten by with a lot less code and fewer errors (in our tests) had we had a testing library
  - Even some hand-written functions would have helped
- Still not good: Had `utility_crra` raised an incompatible error (e.g. had we passed in arrays for z and γ of different sizes), the rest of the tests would not have run
  - Less information, more difficult to find bugs
- So don't try this manual approach at home . . .

. . . but remember where you came from

All testing libraries follow the pattern we have used:

1. Set up a fixture and define the expected result
2. Perform the action that is to be tested
3. Compare the expected result with the actual result
4. Report a summary of the comparison

Basic workflow for testing using `nose`

- Put each test into a function, whose name begins with†

  ```
  test_
  ```

- Group related tests in files, whose names begin or end with†

  ```
  test
  ```

- In the shell, run the command

  ```
  nosetests
  ```

  which automatically searches the current directory and certain sub-directories† / all package contents for tests

†Sneak preview: `nose` looks for directories, modules, and functions matching the following regular expression:

```
(?:^|[b_.-])[Tt]est)
```

First steps with `nose`

1. Set up a fixture and define the expected result

```
from nose.tools import *
from src.examples.testing.utility_crra import utility_crra, Exponer


def test_utility_crra_typical_input():
    assert_equal(first=utility_crra(100.0, 0.5), second=20.0)
```

2. Perform the action that is to be tested

3. Compare the expected result with the actual result

4. Report a **v**erbose summary of the comparison

```
$ nosetests -v
```

First steps with `nose`

Results:

```
$ nosetests -v
utility_crra_test.test_utility_crra_negative_argument ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Can it really be that simple?

Yes, yes, and yes

Consider testing for the right exception

Can it really be that simple?

```
def test_utility_crra_negative_argument():
    assert_raises_regexp(
        expected_exception=ValueError,
        expected_regex=r"All values of 'argument' must be positive",
        callable_obj=utility_crra,
        argument=-100.0,
        exponent=0.5
    )
```

```
$ nosetests
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Arguments can be passed by order . . .

```
def test_utility_crra_log_utility():
    assert_raises(ExponentDomainError, utility_crra, 100.0, 1.0)


def test_utility_crra_risk_lovingness():
    assert_equal(utility_crra(50000.0, -1.0), 125000000.0)


def test_utility_crra_risk_neutrality():
    assert_equal(utility_crra(50000.0, 0.0), 50000.0)
```

All tests are run, regardless of failures

```
$ nosetests
...F.
======================================================================
FAIL: utility_crra_test.test_utility_crra_risk_lovingness
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/me/miniconda3/envs/prog-econ/lib/python3.4/site-packages/nos
    self.test(*self.arg)
  File "/home/me/econ/prog-econ/src/examples/testing/utility_crra_test.py"
    assert_equal(utility_crra(50000.0, -1.0), 125000000.0)
AssertionError: 1250000000.0 != 125000000.0

----------------------------------------------------------------------
Ran 5 tests in 0.002s

FAILED (failures=1)
```

Running a single test from a module

Single out one test by appending `module.py:test` to `nosetests`:

```
$ nosetests utility_crra_test.py:test_utility_crra_risk_neutrality
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Recipes for names to use with `nose`

- Name modules with tests by prepending `test_` or appending `_test` to the module being tested
  - For testing functions in `my_module.py`, use either `my_module_test.py` or `test_my_module.py`
  - Choice depends on how you like your directory listing
- Same for functions, but add text so they are self-explanatory
  - Rule of thumb: Shouldn't need docstrings for simple tests
  - Don't be afraid of long function names!

The `nose` workflow

- Define fixture and expected result
- To compare the two, pick the appropriate `assert` statement from `nose.tools`
  - You will find more than you could imagine
  - Your friends are called:

```
dir(nose.tools)
nose.tools.assert_something.__doc__
```

  - NumPy provides extensions for arrays based on `nose`
- Let `nose` worry about the rest!

`nose` and Waf

- Automate test runs as part of the build
- Add this to the bottom of `utility_crra_test`:

```
if __name__ == '__main__':
    import nose
    nose.runmodule()
```

- Use `run_py_script` in `wscript` file – no need for a target:

```
ctx(
    features='run_py_script',
    source='utility_crra_test.py',
    deps='utility_crra.py'
)
```

Forgot to test a case with strong risk aversion . . .

- You'll learn to love cases with parameters you did not expect when you a minimiser picks them for you
  - Providing meaningful error messages is crucial
  - Write parameter values etc. to disk when things go wrong

```
def test_utility_crra_strong_risk_aversion():
    assert_equal(result_arg_ten_exp_nine, -1.25e-9)
```

Test result

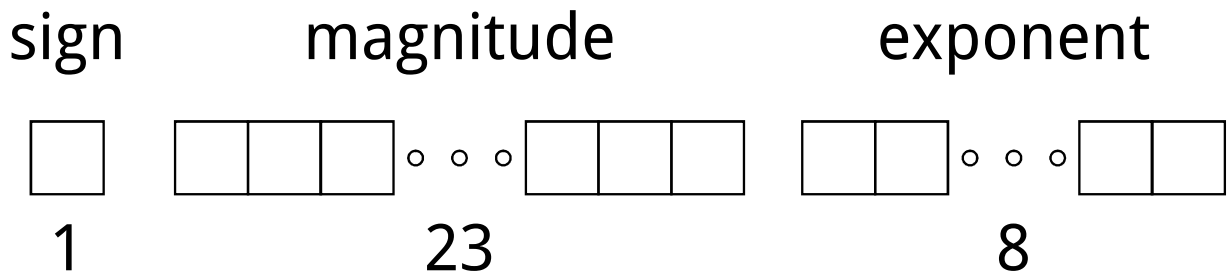
```
$ nosetests utility_crra_test.py:test_utility_crra_strong_risk_aversion
F
======================================================================
FAIL: src.examples.testing.utility_crra_test.test_utility_crra_strong_risk_aver
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/me/miniconda3/envs/prog-econ/lib/python3.4/site-packages/nose/cas
    self.test(*self.arg)
  File "/home/me/econ/prog-econ/src/examples/testing/utility_crra_test.py", lin
    assert_equal(result_arg_ten_exp_nine, -1.25e-9)
AssertionError: -1.2499999924031613e-09 != -1.25e-09

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

What is happening?

- Impossible to represent an infinite number of real values with a finite set of bit patterns
- You can't even get closer – although you may get ever more precise, there is always an infinite number of values between any two values that you **can** represent . . .
- **What follows is oversimplified!!!**

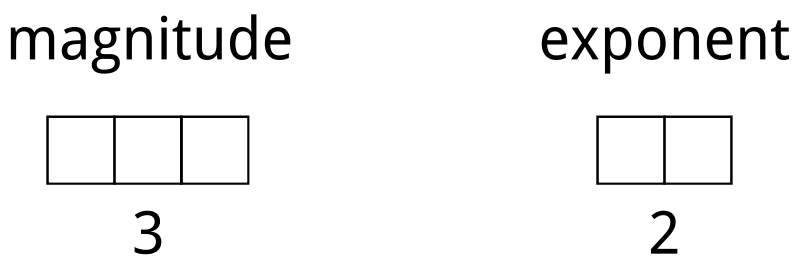
A (too) quick tour of floating points

- Based on Wilson (2010)
- Read Goldberg (1991) before doing heavy computing
- Sign, magnitude, exponent
- 32-bit IEEE standard:

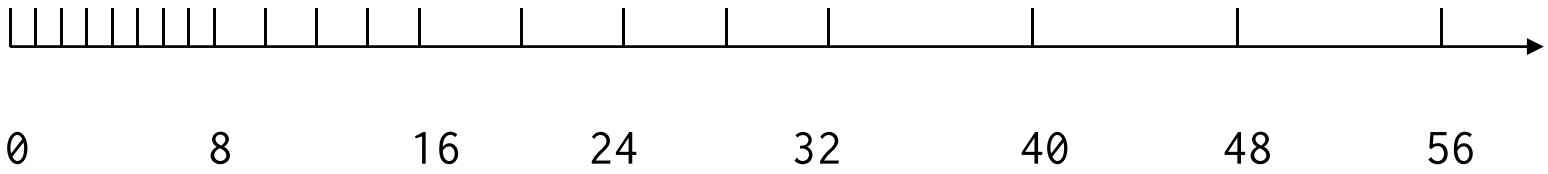


A (too) quick tour of floating points

- Use a simpler format for illustration of problems
- 5 bits, only positive values without fractions



The set of representable numbers



- No representation for 9
- So 8+1 must be either 8 or 10
- If 8+1 = 8, what is 8+1+1?
- (8+1)+1 = 8+1 (if we round down) = 8 again
- But 8+(1+1) = 8+2 = 10, which we can represent
- “Sort then sum” would give the same answer . . .

Some consequences

- Mathematically correct, but computationally different, ways to come up with the same floating point number might yield different results
- In comparisons, >, >=, and friends are safe to use with floating point numbers
- However, == and != are **not!**
- Need to account for approximation error **(hard!)**

The set of representable numbers and some consequences

- Spacing is uneven
- But relative spacing stays the same
  - Multiplying the few mantissas by ever-larger exponents
- Two concepts of error:

$$\text{Absolute error} = |\text{value} - \text{approximation}|$$

$$\text{Relative error} = \frac{|\text{value} - \text{approximation}|}{\text{value}}$$

Which error type is more useful?

- Absolute error is generally simpler to grasp
- And more relevant in some domains, e.g. monetary units
- But it makes little sense to say “off by 0.01" if the value you are approximating is 0.000000001 . . .

Which error type is more useful?

- Python's / nose's `assert_almost_equal` functions use absolute error tolerances
  - At least it's consistent and easy to understand
- Use `numpy.allclose()` for relative error tolerance
  - Need to use it with some `assert` statement for tests
- More on floating-point comparison algorithms:

  `http://www.boost.org/doc/libs/1_34_0/libs/test/doc/components/test_tools/floating_point_comparison.html`

Syntax of `assert_almost_equal`

```
assert_almost_equal(first, second[, places[, msg[, delta]]])
```

- Test that *first* and *second* are approximately equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero

- You could use a different error specification:
  - If *delta* is supplied instead of *places* then the difference between *first* and *second* must be less than *delta*

- Useful in our context?

Approximations in action

```
def test_strong_risk_aversion_abs():
    assert_almost_equal(first=result_arg_ten_exp_nine, second=-1.25e-9)
```

```
$ nosetests utility_crra_test.py:test_strong_risk_aversion_abs
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Approximations in action

Approximations in action

Approximations in action

Confession and what to take away from this

- Nevertheless, this **is** a real problem
- Main message of the floating point interlude:

  Be careful!!!

  There are no golden rules or “sensible” defaults. No way around understanding your problem

More everyday consequences

- Don’t live in blissful ignorance, but be cautious
- Use established routines wherever possible
- Try to avoid calculations with known imprecisions
  - Like inverting (large) matrices . . . (Cook, 2010)
  - Like code from “Numerical Recipes” books

Further reading

Good series of blog posts:
`http://randomascii.wordpress.com/category/floating-point/`

Start from here:
`http://randomascii.wordpress.com/2012/01/11/tricks-with-the-floating-point-format/`

Interface and implementation

- Difference is one of the most important ideas in computing

Interface	How something interacts with the world
Implementation	How it does what it does

- Simple example:

```
def integrate(func, x1, x2):
    ... math goes here ...
    return result
```

Interface	$\{\text{func}, x_1, x_2\} \rightarrow \text{integral}$
Implementation	We don't (have to) care

Irrelevance of implementation details simplifies unit testing

- Want to test components in program one by one
- But components depend on each other
- How to isolate the component under test from the others?
- Replace the other components with things that have the same interfaces, but simpler implementations
  - Sometimes requires refactoring
  - Or up-front design

Design for test

- Design components so they rely exclusively on interfaces
  - Then you can create simple replacements for components with a complicated internal structure
  - Both just need to share the same interface
- Empirics show that interfaces are much more stable
  - So tests don't have to be rewritten over and over
- Isolate interactions with outside world
  - Like opening files
- Make things you are going to examine deterministic
  - Set the seed of the random number generator, etc.

Classes and testing

- The object-oriented approach and unit testing form a natural pair
- Both built on encapsulation and exclusive reliance on interfaces
- For each class whose behaviour you want to test, create one test class
- Inside the test classes, define test methods using the same naming conventions as for test functions before

Defining the parameter values of the environment

- Did you notice the duplication of the function parameters?
  - Duplicated code usually implies scope for simplification
  - The only question is how
- Want to ensure the same environment for all tests
- Could define global parameters at the module level
  - This is what we did in the last floating point example
  - Defined `rel_diff` globally, used it in two tests

This can be a shortcut, but . . .

- What if you need to modify the fixtures?
- What if the functions you test do it?
- Back to defining them over and over, in every test?
- `nose` helps out again . . .

Solution I: Define a class for testing

- Define a test class with a `setUp` method
- Will automatically be called before **each** test
- Guess when a `tearDown` method will be called . . .
  - Sometimes, you need to destroy the environment again (e.g. close the connection to a database)
- Yes, the naming convention is violated

```
from nose.tools import *
from src.examples.testing.utility_crra import utility_crra, ExponentDomainError


class TestUtilityCRRA:
    """Unit tests for the *utility_crra* module."""

    def setUp(self):
        self.argument = 100.0
        self.exponent = 0.5

    def test_utility_crra_typical_input(self):
        assert_equal(utility_crra(self.argument, self.exponent), 20.0)

    def test_utility_crra_negative_argument(self):
        self.argument = -100.0
        assert_raises_regexp(
            ValueError,
            r" 'argument' must be positive\.",
            utility_crra,
            self.argument,
            self.exponent
        )

    def test_utility_crra_log_utility(self):
        self.exponent = 1.0
        assert_raises(
            ExponentDomainError,
            utility_crra,
            self.argument,
            self.exponent
        )
```

Solution II: *decorators* for per-test setup

(Simple story) decorators are functions which are executed before another function call

```
import sys
from nose import with_setup


def setup_each():
    "set up test fixtures ..."


@with_setup(setup_each)
def test_one():
    "first test ..."


@with_setup(setup_each)
def test_two():
    "second test ..."
```

```
from nose.tools import *
from src.examples.testing.utility_crra import utility_crra, ExponentDomainError


def setup_typical():
    global argument, exponent
    argument = 100.0
    exponent = 0.5


@with_setup(setup_typical)
def test_utility_crra_typical_input():
    assert_equal(utility_crra(argument, exponent), 20.0)


@with_setup(setup_typical)
def test_utility_crra_negative_argument():
    argument = -100.0
    assert_raises_regexp(
        ValueError,
        r" 'argument' must be positive\.",
        utility_crra,
        argument,
        exponent
    )


@with_setup(setup_typical)
def test_utility_crra_log_utility():
    exponent = 1.0
    assert_raises(ExponentDomainError, utility_crra, argument, exponent)
```

When to use this?

- Decorators are not magic, but somewhat tricky
- No need to understand how they work (for our purposes)
- Need to understand . . .
  - what `@with_setup` does
  - when and why to use it
- In example, globals are enough (only immutable objects)
- Tearing things down works analogously:

```
@with_setup(setup_func, teardown_func)
def test_1():
    '''Before running test_1, setup_func is called.
    After running test_1, teardown_func is called'''
```

When fixtures are more than parameters

- How to unit-test a function that calls another function?
- Simply use it?
  - Integration testing – complement, but no substitute
  - If a bug turns up, it's not clear which function is responsible
- Mock objects to the rescue
  - Simple objects that imitate behaviour of the real function
  - Like the mockingbird sings the songs of other birds without comprehending them
- Not enough time to cover it here
  - Check documentation of Python libraries `fudge`, or `Mocker`

What coverage tools do

- Your mind still must keep track of what you are testing
  - Scope for automation!!!
- Test coverage tells you which lines of code have been visited
  - Use Python library `coverage`
- Integrates seamlessly with `nose`

```
nosetests --with-coverage
```

- Option to restrict analysis to one module / package, great when you import stuff like NumPy:

```
--cover-package=module_name
```

Nose & coverage in action

```
$ nosetests --with-coverage --cover-package=.
..........................
Name                    Stmts   Miss  Cover   Missing
-----------------------------------------------------
ez_utility                 70     32    54%   20-34, 133-140, 150-188
ez_utility_test            88      1    99%   160
-----------------------------------------------------
TOTAL                     229     34    85%
-----------------------------------------------------
Ran 26 tests in 0.273s

OK
```

Nose & coverage in action

```
$ nosetests --with-coverage --cover-package=ez_utility
.........................
Name                   Stmts   Miss  Cover   Missing
----------------------------------------------------
ez_utility                70     32    54%   20-34, 133-140, 150-188
----------------------------------------------------
Ran 26 tests in 0.273s

OK
```

Some caveats

- Complete coverage is only a necessary condition . . .
- What happens at non-exhaustive if-statements?
- For meaningful coverage in such cases, use:

```
if sometimes_true:
    do(something)
else:
    pass
```

- Coverage data from previous runs is saved – can mess up your results when testing. If in doubt, use the option:

```
--cover-erase
```

Once more: Why testing?

- If you do trial & error in a complex system, it's much more difficult to find bugs – much harder to pin down error
- Allows you to change one thing at a time, easily
  - Without doing harm to other parts of the program
  - Medical researchers start by killing rats as well . . .
- Allows you to check correctness of your functions immediately after programming
  - Don't (want to) remember the exact choices much longer
  - Testing upfront often obviates you from this need
- Done right, it ensures that you don't reintroduce bugs

Why interfaces matter so much more

- Typical referee worry:

  I don't believe the distributional assumptions in you ML routine. Show that everything goes through with GMM.

- In Stata, you'll likely write most of the program twice

- In a well-programmed Python / Matlab / R / Gauss / Ox project, you need to replace a couple of functions
  - Core parts don't care whether they're called by ML or GMM
  - Outer shell doesn't care whether it supplies its parameters to ML or GMM

References I


Arbuckle, Daniel (2010). *Python Testing: Beginner's Guide*. Birmingham, UK: Packt Publishing.


Cook, John D. (2010). *Don'T Invert that Matrix*. `http://www.johndcook.com/blog/2010/01/19/dont-invert-that-matrix/`.


Feathers, Michael C. (2005). *Working Effectively with Legacy Code*. Prentice Hall.


Goldberg, David (1991). "What Every Computer Scientist Should Know about Floating-Point Arithmetic". In: *ACM Computing Surveys* 23.1, pp. 5–48.


Wilson, Gregory V. (2010). *Testing with Floating Point Numbers*. `http://software-carpentry.org/4_0/test/float/`.

Acknowledgements

- This course is designed after and borrows a lot from the `Software Carpentry` course designed by `Greg Wilson` for scientists and engineers.

- The Software Carpentry course material is made available under a `Creative Commons Attribution License`, as is this course's material.

License for the course material

[Links to the full `legal` text and the source text for `this` page.] You are free:

- **to Share** to copy, distribute and transmit the work
- **to Remix** to adapt the work

Under the following conditions:

- **Attribution** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:

- **Waiver** Any of the above conditions can be waived if you get permission from the copyright holder.
- **Public Domain** Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- **Other Rights** In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to `this` web page.