

Zope 3 Cookbook

Tarek Ziadé - ziade.tarek@gmail.com

Contents

| | |
|---|------------|
| Introduction | iii |
| 1 Understanding interfaces | 1 |
| 2 Writing unit tests for Python | 7 |
| 3 Writing functional tests | 13 |
| 4 Understanding adapters | 19 |
| 5 Understanding events | 23 |
| 6 Writing reSTructuredText for documentation and doctests | 27 |
| 7 Finding the good pace between coding, documenting, unit testing and functional testing | 31 |
| 8 Recording a user web session to write tests | 35 |
| 9 Creating an RSS feed for any container | 39 |
| 10 Retrieving an object URL | 45 |
| 11 Setting up and using a mail delivery queue | 47 |

Introduction

This books contains recipes for Zope 3.2.1, from the zope-cookbook.org project. Zope 3 continues to evolve but thos recipes are still usable for most of them.

This book was written by Tarek Ziadé and is distributed under the CC licence Attribution-NonCommercial-NoDerivs 2.5

<http://creativecommons.org/licenses/by-nc-nd/2.5/>

Chapter 1

Understanding interfaces

Interfaces are the basis of component oriented programming and therefore of Zope 3. This recipe presents how it works.

Understanding the mechanism

Interfaces comes from the standardization of C header files. Separating the declarative part from its implementation let the developer hide the code to the user, that only gets the definitions of provided functionalities. Therefore, changing the code can be done without interfering with the rest of the system.

This principle has been added with different flavors in many modern object-oriented languages, like Java and its *protocols*, or Delphi, in a similar way C++ does.

Python doesn't *yet* integrate interfaces, even though it has been discussed many times over the past five years by the community. Zope has therefore implemented its own system, which has been reused by other Python frameworks like *Twisted*.

In Zope, an interface defines a contract, made of methods and attributes. A class can *implement* this interface, by providing the code for all parts of the signature. A given class can implement many interfaces.

This is useful in Zope, where tools need to interact with each other through abstractions, instead of playing directly with classes that implement those abstractions. ZCML directives for example can combine an action to an interface. The published objects will trigger this behavior if they implements the given interface.

Defining an interface

`zope.interface` provides a set of tools to declare and manipulate interfaces.

Elements to declare interfaces:

- *Interface* : base class to define an interface;
- *Attribute* : base attribute;
- *invariant* : constraint applied to candidate objects for a given interface.

Interface

Interface is a special class, similar to Python's *object* one, that provides a base class for all interfaces. An interface is a class with empty methods, that is a bit similar to C headers: only signatures are defined.

The IRule interface:

```
>>> from zope.interface import Interface
>>> class IRule(Interface):
...     def bigtime(language):
...         """tells if the language is cool"""
...     
```

Notes:

In interface classes, the *self* implicit attribute disappears

The body of the methods is only composed of a docstring

Any class can then implement the corresponding code, and get linked to the interface class through the *implements* directive.

The Rule class:

```
>>> from zope.interface import implements
>>> class Rule(object):
...     implements(IRule)
...     def bigtime(self, language):
...         if language == 'python':
...             return True
...         return False
...
>>> rules = Rule()
>>> rules.bigtime('python')
True
>>> rules.bigtime('java')
False
>>> rules.bigtime('ruby')
False
```

Attribute

Attributes can also be added to the interface to extend the signature. *Attribute* is a class that get instanciated with a simple string.

The IDocument interface:

```
>>> from zope.interface import Attribute
>>> class IDocument(Interface):
...     title = Attribute('Document title')
...     description = Attribute('Document description')
...     def retrieve():
...         """returns title and description"""
...     
```


max.size checks that the *Book* instance doesn't weight more than 250 pages, otherwise it raises a *BigBookError* .

The *Book* instances can then be checked with the *validateInvariants* *Interface* method.

invariant tests:

```
>>> programming_python = Book(570)
>>> programming_python.pages
570
>>> programming_python.display(45)
'Ha ha ha. Buy it.'
>>> IBook.validateInvariants(programming_python)
Traceback (most recent call last):
...
BigBookError: <...Book object at ...>
>>> news_from_charleston = Book(46)
>>> IBook.validateInvariants(news_from_charleston)
```

Using interfaces

Besides the *implements* function that links a class to an interface, *zope.interface* provides a mapping style access.

Using IDocument:

```
>>> elements = list(IDocument)
>>> elements.sort()
>>> elements
['description', 'retrieve', 'title']
>>> IDocument['retrieve']
<zope.interface.interface.Method object at ...>
>>> title = IDocument['title']
>>> title
<zope.interface.interface.Attribute object at ...>
>>> title.__name__
'title'
>>> title.__doc__
'Document title'
```

A collection of functions and *Interface* methods can also be used to manipulate interfaces.

The most used are:

- *Interface.providedBy(object)* : return *True* if *object* implements the interface.
- *implementedBy(class)* : returns the list of interfaces implemented by *class* .
- *directlyProvides(class, interface)* : let the developer manually specify that *class* directly provides *interface* , without having to change *class* code to add an *implements* directive.

- *directlyProvidedBy(class)* : returns an object that lists all interfaces linked to *class* with *directlyProvides* .

Interface manipulation examples:

```
>>> class IDoIt(Interface):
...     def well():
...         """complex, smart code"""
...
>>> class CanDoIt(object):
...     implements(IDoIt)
...     def well(self):
...         return 1 + 1
...
>>> ok = CanDoIt()
>>> ok.well()
2
>>> IDoIt.providedBy(ok)
True
>>> from zope.interface import directlyProvides
>>> from zope.interface import directlyProvidedBy
>>> class CouldHaveDoneIt(object):
...     def well(self):
...         """too"""
...         return 2 + 2
...
>>> directlyProvides(CouldHaveDoneIt, IDoIt)
>>> list(directlyProvidedBy(CouldHaveDoneIt))
[<InterfaceClass ...IDoIt>]
```

Understanding the use of interfaces in Zope

A majority of ZCML directives are based on the combination of an interface and a behavior. These behavior are often described themselves by other interfaces, that points the qualified code to be run.

Zope framework, that manipulates concrete elements, like persistent objects or view classes, translates abstractions defined by interfaces and apply them in the real execution context. It invokes, using utilities described in this recipe, the good objects at the right time. This loose, component-oriented approach, helps the classes to evolve independently one from each other and helps the developer apply the *KISS* principle.

Chapter 2

Writing unit tests for Python

Bruce Eckel explained one day that strong static typing was to be opposed to strong Test Driven Development (TDD): where a C compiler complains about a function call made with parameters that are not of the expected type, Python developer have to provide tests that fills the lack of pre-execution checks. But Bruce explains that this is not sufficient to make C or C++ programs higher in quality.

TDD programming is more than that: it's a philosophy that increases a lot the quality of programs and the productivity of developers. Even C developers, when they understand what TDD brings, comes to it naturally.

This recipes resumes the TDD principles, and how it can be done in Python.

Understanding TDD

TDD claims that each piece of code has to come with tests, that validate that all uses cases are fullfilled.

An *average* function for example, can be tested with a serie of asserts.

The average function and its tests:

```
>>> def average(*elements):
...     return sum(elements) / len(elements)
...
>>> assert average(5, 9, 0, 7) == 5
>>> assert average(5) == 5
>>> assert average(9, 18, 0) == 9
```

Theses tests can be grouped in another function, specialized in *average* testing.

test_average:

```
>>> def test_average():
...     assert average(5, 9, 0, 7) == 5
```

```

...     assert average(5) == 5
...     assert average(9, 18, 0) == 9
...
>>> test_average()

```

Building the test function and the code function at the same time has several advantages:

- The developer sees her code with a fresh, user-oriented point of view. By this new look over her code, she corrects 90% of bugs and design errors.
- Tests that are not immediately done are never done.
- Non-regression is kept : adding code with tests, then running again all tests prevents previous features to get broken.

When a bug is found, the test function is completed with the new case that raises the problem *before* it is corrected. When the test reproduces exactly the problem, the code is corrected.

In our example, a buggy behavior could happen when an empty sequence is given to the function. After the case is added to the tests and reproduced, the code is corrected.

Correcting a ZeroDivision error:

```

>>> elements = []
>>> average(*elements)
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
>>> def test_average():
...     assert average(5, 9, 0, 7) == 5
...     assert average(5) == 5
...     average(9, 18, 0) == 9
...     elements = []
...     assert average(*elements) == 0
...
>>> test_average()
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
>>> def average(*elements):
...     if len(elements) == 0:
...         return 0
...     return sum(elements) / len(elements)
...
>>> test_average()

```

Notes:

Previous tests are kept and still launched: the test function is getting richer.

This technique offers a high level of code quality on projects where numerous developers interact: tests become a security for the code not to be broken and a

developer documentation. People can therefore work on code pieces they didn't created with ease.

Presenting TDD principle in a recipe is a bit short, and reading books about it can provide people a better knowledge.

Writing unit tests with Python tools

Python offers in its standard library tools to speed up TDD work: *unittest*, a *JUnit* port, and *doctest*, an original test system based on docstrings.

unittest

The *unittest* module provides helpers to write tests. The main element is a class based on *JUnit* test model.

This class can be used as a base class to gather tests and make a *test suite*. It also provides hooks at the start and end of tests launch. These hooks are called *test fixture*, and can be used to setup, when needed, a test environment.

The tests from the previous example can be gather in such a class and saved in a Python module. Each code module *code.py* can have a corresponding *test_code.py* module with its tests.

the test_average.py file

```
import unittest
# the average function has been put
# into an 'average.py' file
from average import average

class TestAverage(unittest.TestCase):

    def test_average_simple(self):
        res = average(5, 9, 0, 7)
        self.assertEqual(res, 5)

    def test_average_one_element(self):
        self.assertEqual(average(5), 5)

    def test_average_zero_div(self):
        self.assertEqual(average(*[]), 0)

if __name__ == '__main__':
    unittest.main()
```

All methods that starts with *test* are executed as tests when the script is launched.

Launching test_average.py:

```
tziade@dabox:~/Desktop$ python test_average.py
```

```
...
```

```
-----
Ran 3 tests in 0.000s
```

OK

Test modules are gathered for each Python package in a subfolder called *tests*, that can be scanned by a test script. This script generally scans all modules in this folder which name starts with *test*.

Doctests

Another system to create tests is called *doctests* and based on using code comments (docstrings). Code sessions, like what could be typed in a Python prompt, can be inserted.

Tests for average, using docstrings:

```
>>> def average(*elements):
...     """ Does an average, and does it well
...     >>> average(5, 9, 0, 7)
...     5
...     >>> average(5)
...     5
...     >>> average([])
...     0
...     """
...     if len(elements) == 0:
...         return 0
...     return sum(elements) / len(elements)
... 
```

This writing offers a better readability of tests, that appears like small prompt sessions. But the module can get a bit obfuscated when a lot of tests are added all over the docstrings. Therefore, these tests can be separated in another file, like classical unit tests.

docstrings in a separated file:

```
Average
=====
```

```
Average calculates the average of a sequence::
```

```
>>> average(5, 9, 0, 7)
5
>>> average(5)
5
```

```
Average is pretty cool, and doesn't die when it's asked
the average of an empty sequence::
```

```
>>> average([])
0
```

(Even though this kind of call is pretty weird)

This way of writing tests let also combines explanations over small prompt sequences: it becomes a documentation, also called *executable documentation* or *agile documentation*.

This is the preferred shape for tests in Zope 3 world, as developers can document and test their packages within the same move.

Classical unit tests are reserved to lower level tests, when a lot of test fixture is needed and would make the doctest unreadable.

Chapter 3

Writing functional tests

Functional or "acceptance tests" are complementary to unit tests: They are black boxes that make sure that the functionality of the whole system, is working as it was defined.

These tests are only covering the use of the system from the point of view of a normal user, in order to make sure that all the expected functionalities are present and work as expected.

They are an infallible sign of quality for the end users, who may even occasionally write or extend these tests themselves.

In terms of proportion, a system should have approximately 2/3 of unit tests and 1/3 functional tests, given that the latter are on a higher level.

For web applications, these tests need a special execution environment, which imitates a browser as best as possible, and offers an API for launching requests and study the response, just like it would be done by FireFox or Internet Explorer.

Understanding how it works

By definition, a functional tests only concentrates on the functionality of the application, and doesn't care for implementation details, as opposed to unit tests. It is a practical test as performed by an end user, who can for example browse the menus of the application in order to test each of them. He then uses a check list to make sure that all expected functionality is there and works fine. The system is said to be "validated" once all items in the check are verified to be correct and in conformity with the specification.

Functional tests for Python and of course for Zope, are done by the means of test classes derived from unit test classes, or in doctests.

The only difference between unit tests and functional tests is the interface between the system and the test writer. The tester writer has to write the tests under the same conditions that an end user would have at their disposal.

Writing functional tests with classes

Zope provides a special module in the `zope.app.testing` package called `functional`, whose classes inherit from `unittest.TestCase`. The most commonly used class is `BrowserTestCase`.

It has the following methods:

- `getRootFolder()`: returns the Zope root folder. It's used to prepare the site before analysing a publication.
- `publish(path, basic=None)`: Returns the rendered object behind the path in the form of a `Response` object. the optional `basic` parameter has the form `'login:password'`. The string `'mgr:mgrpw'` can be used to authenticate with manager rights.
- `checkForBrokenLinks(body,path, basic=None)`: Checks for broken links in the `body`, which is a string representing the body of the response (fetched with `Response.getBody()`). `path` gives the relative path that corresponds to the object at the origin of the response. This parameter is needed by the method in order to make sense of relative links in the site.

The test utility augments the `Response` object with methods for retrieving information easily, such as:

- `getBody()`: returns the response body
- `getOutput()`: returns the complete response (body and headers)
- `getStatus()`: returns the status
- `getHeaders()`: for retrieving the headers

In the following example, a folder is created under the root folder, and its rendering is verified:

Verification class of the rendering of a folder:

```
>>> import transaction
>>> from zope.app.testing import functional
>>> from zope.app.folder import Folder
>>> class FolderTest(functional.BrowserTestCase):
...     def test_folder(self):
...         root = self.getRootFolder()
...
...         # create a Folder
...         root['cool_folder'] = Folder()
...         transaction.commit()
...
...         # get the root content through the publisher
...         response = self.publish('/cool_folder', basic='mgr:mgrpw')
...         self.assert_(' <title>Z3: cool_folder</title>' in
...                        response.getBody())
```

Notes:

After having validated the modification in the site with the code `transaction.commit()`, the object is available for the `publish` call.

The test is executed like a classic unit test.

Starting the test:

```
>>> import unittest
>>> suite = unittest.makeSuite(FolderTest)
>>> test_runner = unittest.TextTestRunner()
>>> test_runner.run(suite)
<unittest._TextTestResult run=1 errors=0 failures=0>
```

Functionaltests with doctests

Functional tests are always very high level tests, and come very close to example code that could be part of a documentation. In Python, the both the need for documentation and for testing can be satisfied by the means of doctests.

Understanding Python doctests

The idea of putting tests into Python doctests is quite intriguing: The code is written in the docstrings as a series of interactions on the Python prompt.

Doctest example:

```
>>> def theTruth():
...     """
...     >>> theTruth()
...     'doctests rule'
...     """
...     return 'doctests rule'
```

The Python `doctest` module contains utilities for extracting and executing these tests.

Execution of a doctest:

```
>>> import doctest
>>> doctest.run_docstring_examples(theTruth, globals(), verbose=True)
Finding tests in NoName
Trying:
    theTruth()
Expecting:
    'doctests rule'
ok
```

Once doctests are added to all modules, classes and methods, the readability of the code may suffer. In order to avoid this code obfuscation, it is recommended to group the functional tests in a separate file, which is then "executed". This opens the way for explaining the examples in more depth, and to follow logical steps. In this case, we can speak of "agile documentation", or "executable documentation".

The documents can then be called by the test module.

Agile document:

```
Le module of truth
=====
```

This document doesn't contain any functional tests, but a unit test.

It doesn't matter at all.

The truth module contains the function 'theTruth()', which can be called for showing a message. This method doesn't serve any real purpose.

Usage example::

```
>>> from truth import theTruth
>>> theTruth()
'doctests rule'
```

Understanding functional doctests in Zope

In order to easily manipulate doctests in Zope, the *zope.testbrowser* package offers the *Browser* class, which allows the simulation of a browser and its functionalities. By convention, this class is invoked with the URL starting with `http://localhost`, followed by the path to the Zope server.

This is how you can write a test to reach the page situated at the root:

```
>>> from zope.testbrowser.testing import Browser
>>> browser = Browser('http://localhost')
>>> browser.url
'http://localhost'
>>> browser.contents
'...<title>Z3: </title>...'
```

Notes:

The *contents* property is used to fetch the content of the page in the form of a string.

The '...' (or ellipsis) have a precise meaning in the doctests: They stand for an arbitrary amount of text. In the above example, they allow us to make sure that *contents* does contain the given HTMLcode.

Besides *contents* and *url*, the main elements provided by *Browser* are:

- *open(url)* : allows to open the given URL.
- *isHtml* : is the content written in HTML ?
- *title* : returns the title of the page.
- *headers* : returns the Python object *httplib.HTTPMessage* containing the headers.

- *getLink(text)* : fetches the *Link* object for the link given in *text* . This link can then be followed using the *click()* method, whereby the new page is loaded in the *Browser* instance.

You will be able to learn about further elements offered by this utility in some other recipes.

Limitations of functional tests

Functional tests for the web are somewhat limited in what they can do if compared to browsers: It is impossible to test JavaScript contained in the pages, because each browser comes with its own engine. For this, you need to look at other technologies, like *Selenium*, which runs the tests inside a real browser.

Chapter 4

Understanding adapters

When applications gets bigger, there's a side effect on the code, called the *spaghetti effect*: interactions between classes can lead to unwanted dependencies and the code turns into a monolithic bloc.

Adapters provides a way to prevent from this, by implementing the *Liskov substitution principle* .

Understanding how it works

Adapters provide a cooperation mechanism between any given object and a particular context, using interfaces. They allow an arbitrary type of class to be compatible with a given interface, by giving a compatibility layer.

This mechanism is used in systems like Microsoft COM's QueryAdapter, and let the developer gathers objects in a specific functional context. This also known as *glue code*.

Adapters provides several advantages:

- They can gather class instances in contexts they where not implemented for, without having to change their code or make them depend on each other.
- They offer a smooth way to gather generic features, that can be applied on several kind of classes.

Adapters can be seen as a formalized *duck typing* and where proposed some years ago in *PEP 246*. There are also Python implementations of it, like *PyProtocols*.

Defining an adapter: IClassDoc

Adapters are built with a class instance, and implements an interface.

For example, the *IClassDoc* interface, which defines methods to get a documentation about a given class instance, no matter the class type.

IClassDoc interface:

```
>>> from zope.interface import Interface
>>> class IClassDoc(Interface):
...     def getMethods():
...         """ renvoie un texte qui liste les méthodes de la classe """
...     def getAttributes():
...         """ renvoie un texte qui liste les attributs de la classe """
... 
```

An adapter for this interface could be named ClassDoc..

ClassDoc class:

```
>>> from zope.interface import implements
>>> class ClassDoc(object):
...     implements(IClassDoc)
...     def __init__(self, context):
...         self.context = context
...     def getMethods(self):
...         methods = [(element, getattr(self.context, element).__doc__)
...                     for element in dir(self.context)
...                     if callable(getattr(self.context, element))]
...         for method, doc in methods:
...             print '%s: %s' % (method, doc)
...     def getAttributes(self):
...         attributes = [element
...                        for element in dir(self.context)
...                        if not callable(getattr(self.context, element))]
...         for attribute in attributes:
...             print attribute
... 
```

```
>>> objet = ['une', 'liste']
>>> doc = ClassDoc(objet)
>>> doc.getAttributes()
__doc__
>>> doc.getMethods()
__add__: x.__add__(y) <==> x+y
... 
```

```
append: L.append(object) -- append object to end
count: L.count(value) -> integer -- return number of occurrences of value
extend: L.extend(iterable) -- extend list by appending elements from the iterable
index: L.index(value, [start, [stop]]) -> integer -- return first index of value
insert: L.insert(index, object) -- insert object before index
pop: L.pop([index]) -> item -- remove and return item at index (default last)
remove: L.remove(value) -- remove first occurrence of value
reverse: L.reverse() -- reverse *IN PLACE*
sort: L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1
```

Registering ClassDoc into Zope

Zope 3 provides a registry for Adapters, that centralizes all of them. An adapter is registered with a ZCML directive, which associates the class with the interface to provide. This class is called a factory.

The adapter can also, depending on use cases, adapt an Interface to another. The *adapter* directive parameters are::

- *factory* : the adapter class or classes;
- *provides* : the interface the adapter implements;
- *for* : the classes and interfaces which are adapted;
- *permission* : a permission that filters the adapter access;
- *name* : the adapter name;
- *trusted* : a boolean, that tells if the adapter class is trusted. If so, the security won't be check on objects called by the class.
- *locate* : a boolean, that make the adapter located. (**todo**explain here)

Registering the IClassDoc adapter:

```
>>> from zope.configuration import xmlconfig
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'>
...         <include package="zope.app.component" file="meta.zcml" />
...         <adapter
...             factory="recipe.ClassDoc"
...             for="*"
...             provides="recipe.IClassDoc"
...             />
...     </configure>
... """)
```

Notes:

The *include* directive is here for the test to work, and does not need to be added.

Using IClassDoc adapter

IClassDoc adapter can be called with *zapi.getAdapter(object, interface=Interface, name="", context=None)* , which parameters are:

- *object* : the object to adapt;
- *interface* : the wanted interface;

- *name* : the name of adapter, if given;
- *context* : the context, for local Adapters;

Using adapter:

```
>>> from zope.app import zapi
>>> la_coupe = ['est', 'pleine']
>>> la_coupe_doc = zapi.getAdapter(la_coupe, IClassDoc)
>>> la_coupe_doc.getMethods()
__add__: x.__add__(y) <==> x+y
...
append: L.append(object) -- append object to end
count: L.count(value) -> integer -- return number of occurrences of value ...
```

Chapter 5

Understanding events

When an action has to be triggered on a given event that can come from several components, there's a risk of interdependencies if links are directly made between all elements.

Event programming presented in this recipe resolves this issue through the *Observer* design pattern.

Understanding the Observer design pattern

The easiest way to avoid interdependency between actors is to trigger an event that is seen by observers in charge of running a piece of code. This is done by the *Observer* design pattern (DP).

This DP settles a notification system, where each observer registers itself for a given event id into a dedicated registering object. Actors that provokes an event can notify the register which then calls all registered objects.

Basic event register in Python

```
>>> class EventManager(object):
...     observers = {}
...     def subscribe(self, event_id, caller):
...         if event_id in self.observers:
...             if caller not in self.observers[event_id]:
...                 self.observers[event_id].append(caller)
...         else:
...             self.observers[event_id] = [caller]
...     def notify(self, event_id):
...         if event_id not in self.observers:
...             return
...         for caller in self.observers[event_id]:
...             caller()
...
>>> manager = EventManager()
>>> class Observer(object):
...     def __init__(self, id):
...         self.id = id
```

```

...     manager.subscribe('moved', self)
...     def __call__(self):
...         print '%s: she moved !' % self.id
...
>>> class Mover(object):
...     def __call__(self):
...         print "Ok, let's move"
...         manager.notify('moved')
...         print "Yes I did"
...
>>> sarah = Mover()
>>> camera1 = Observer('camera 1')
>>> camera2 = Observer('camera 2')
>>> sarah()
Ok, let's move
camera 1: she moved !
camera 2: she moved !
Yes I did

```

This separation allows to implement behaviors without having to make involved classes dependent. In Zope this mechanism is often used and a lot of events are already available, like:

- Object creation;
- Object move or deletion;
- etc.

Using observer in Zope

The *zope.component* implements a *subscriber* directive that let the developer link a callable with an interface that represents an Event. When an object that implements the given interface is notified through the global function *zope.event.notify*, the subscribed function is called.

In the example below, *groupie_1* and *groupie_2* are called when a *ILeMessie* event occurs.

Zope events:

```

>>> from zope.interface import Interface, implements, Attribute
>>> class IKing(Interface):
...     name = Attribute('Name of the king')
...     groupies = Attribute('Number of groupies')
...
>>> class TheKing(object):
...     implements(IKing)
...     def __init__(self, name):
...         self.name = name
...         self.groupies = 0
...
>>> def groupie_1(event):

```

```

...     print 'Long life to %s !' % event.name
...     event.groupies += 1
...
>>> def groupie_2(event):
...     print 'Glory to our king %s !' % event.name
...     event.groupies += 1
...
>>> from zope.configuration import xmlconfig
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'>
...         <include package="zope.app.component" file="meta.zcml" />
...         <subscriber
...             handler="recipe.groupie_1"
...             for="recipe.IKing"
...         />
...         <subscriber
...             handler="recipe.groupie_2"
...             for="recipe.IKing"
...         />
...     </configure>
... """)
>>> arthur = TheKing('Arthur')
>>> from zope.event import notify
>>> notify(arthur)
Long life to Arthur !
Glory to our king Arthur !
>>> arthur.groupies
2

```

The *subscriber* directive can also be used with richer configuration based on adapters, but the example shown is sufficient for all kind of event needs in an application.

Current events in Zope

Zope comes in:

- *zope.app.event.interfaces* ;
- *zope.app.component* ;
- *zope.app.publication* ;
- *zope.app.container* .

with a serie of interfaces for events used into the framework for base manipulations, like:

- *IObjectEvent*: base interface, triggered for all events on an object;
- *IObjectCreatedEvent*: triggered when an object is created;

- `IObjectCopiedEvent`: triggered when an object is copied;
- `IObjectModifiedEvent`: triggered when an object is modified;
- `IObjectAnnotationsModifiedEvent`: triggered when an object annotation is changed;
- `IObjectContentModifiedEvent`: triggered when an object content is modified;
- `IRegistrationEvent`: base interface, triggered fro register events;
- `IRegistrationActivatedEvent`: triggered when a register is activated;
- `IRegistrationDeactivatedEvent`: triggered when a register is deactivated;
- `IObjectMovedEvent`: triggered when an object has move in a container;
- `IObjectAddedEvent`: triggered when an object has been added into a container;
- `IObjectRemovedEvent`: triggered when an object has been removed from a container;
- `IContainerModifiedEvent`: triggered when a reordering, deletion or adding occurs in a container;
- `IBeforeTraverseEvent`: triggered when the *publisher* starts the traversal;
- `IEndRequestEvent`: triggered when the *publisher* has finished the request calculation.

Chapter 6

Writing reSTructuredText for documentation and doctests

The whole Zope documentation available inside its packages is written in reSTructuredText or reST, which is an evolution of StructuredText. This structured text is more readable as is than LaTeX, since the tags don't change the text itself. Tools allow then to convert reST files to publishable formats like HTML, XML or PDF.

This recipe is a minimal *cheatsheet* to write doctests for Zope packages using reST. A complete documentation is available here: <http://docutils.sourceforge.net/rst.html>.

Structuring the file

To organise a file into sections, subsections and so forth, the section title just need to be underlined with a character from:

```
= - ' : ' " ~ ^ _ * + # < >
```

Whenever the reST engine meet such underlines, it links the given level to the character in use, for the rest of the document.

a reST example:

```
Title  
=====
```

```
Ok, so I have two sections, enjoy !
```

```
Section 1  
~~~~~
```

```
The title shows off. I talk about it in a subsection.
```

Subsection 1

Section 1 asked me to make fun of Title, but I won't, since I am trying to get promoted.

Section 2

~~~~~

Let's calm down folks, the document is over...

## Inserting code examples

Inserting a code bloc is done by indenting it with at least one character. (most of the time people do 2 or 4). The bloc must be preceded by a ":" sign and an empty line, then forwarded by an empty line.

*Code insertion:*

The Math module

=====

Python has a math module that has a function for power::

```
>>> import math
>>> math.pow(56, 3)
175616.0
```

It has pi as well::

```
>>> math.pi
3.1415926535897931
```

Notes:

Zope 3 use a 2 spaces indenting, but some developers use 4, like Python code.

## Inserting links and formatting the text

reST provide tags to display bold or italic text.

*Bold and italic:*

Facts

=====

They aren't more **people** in 'Italia' than in other part of the world.

Italic is used to emphasis a module name or a inline piece of code. Bold is dedicated to regular text emphasis.

For links, the most common types are links to Web resources (URL), and kickers to other parts of the current document.

Like in HTML, links can be associated to a piece of text, which is quoted, followed by an underline. This text is repeated at the bottom of the document, preceded by two dots, a space and an underline, then forwarded by the URL, with a ":" prefix.

*Link example:*

```
Daily job
=====
```

```
Morning and evening : a visit to 'zope-cookbook'_
Noon: Don't forget your pill.
```

```
.. _'zope-cookbook': http://zope-cookbook.org
```

Notes:

The footer reference is removed when the text is rendered in HTML, PDF or another publishing format.

To define links within the document itself, the same kind of markup is used, but the target is marked with the reference followed by ":".

*Links in the documents:*

```
Daily job
=====
```

```
Morning and evening : a visit to 'zope-cookbook'_
_'pill':
Noon: Don't forget your pill.
```

```
_'zope-cookbook':
Zope-cookbook.org is a neat place, but before you go there,
don't forget to take your 'pill'.
```

## Adopting good practices

A few good practices, unordered:

- Make sure the reST file always compiles. With *rest2html* for example;
- group link references in the bottom of the document;
- reST files are package documentation, they should be well written;
- Module, class and method docstrings can use reST as well.



## Chapter 7

# Finding the good pace between coding, documenting, unit testing and functional testing

Coding, testing, documenting, coding, testing, coding, testing...

The good pace of the different activities of a developer daily job is hard to find. In ends of projects, the rush and the stress often reduce the testing and the documenting parts, so the work can be finished on time. But this shortcut often leads to backdrafts later: end of projects are often periods where the code is highly refactored.

This problem can be partly reduced with good practices explained in this recipe. They apply to Zope of course, but the principles are the same for other Python project.

### Understanding the developer duties

A developer has three duties:

- Coding;
- documenting;
- testing;

The first phase is of course TDD coding. The a cycle of functional tests is made. The the code might be changed, to add new features, changes, etc. Theses cycles are well documented and explained in methodologies like RUP (IBM Rational Unified Process). This is out of topic though, but interested readers can get complementary informations on Rational website:

<http://www.rational.com>

## Creating the code

To create a new feature, the developer gets a feature request, made of functional specifications and sometimes technical specifications.

Using this document, she creates the code following these steps:

- Translating in a reST document (doctest) the specifications;
- inserting code examples, that fullfills the features asked;
- coding the underlying code, so the examples in the document works for real.

## Translating the need into a doctest file

Translating in doctests the needs is done by summarizing the specifications, and by describing the future package structure.

Let's code for example a package that handles the retrieval of RSS feeds to store them in persistent Python objects, then provide web views. The developer decides to organize the package in three modules:

- *rssfetcher.py* : module for RSS retrieval;
- *rssobject.py* : module for creating and storing data;
- *rssview.py* : module for viewing data.

She starts to write four doctests files, that shows the future code structure:

- *README.txt*
- *rssfetcher.txt*
- *rssobject.txt*
- *rssview.txt*

The *README.txt* file describe in a few sentences the package goal, and present its structure.

*README.txt* file:

```
rssreader
=====
```

This package knows how to display RSS feeds. It is organized in three modules.

- *rssfetcher*: knows how to read RSS 1.0 feeds
- *rssobject*: store the feeds and provides accessors
- *rssview*: know how to display a feed in the browser

Each module then has its own doctest file.

*rssfetcher.txt* file:

```
rssfetcher
=====
```

rssfetcher module transforms a RSS 1.0 XML file into a Python usable structure. It is called with the feed URL and returns a list with the feed entries.

At this point the developer didn't write any code, but has a first draft of the package structure and technical specifications. This step reveals sometimes some incoherences in the needs.

## Inserting code examples

Using these text files, the developer can start to define each module high level public elements, that will be used by other modules.

Changing *rssfetcher.txt*:

```
rssfetcher
=====
```

rssfetcher module transforms a RSS 1.0 XML file into a Python usable structure. It is called with the feed URL and returns a list with the feed entries.

It provides a 'fetchFeed(source)' function that returns a Python structure, given an URL or a file ::

```
>>> from rssfetcher import fetchFeed
>>> fetchFeed('tests/rss.xml')
[{'Subject': 'entry one', 'content': "le contenu est ici"}]
```

Notes:

At this point the package will need a *rss.xml* file to be used as a sample for tests.

This principle is used for all modules that might provide public functionalities.

## Coding functionalities

The previous doctest cannot work until the functionalitie is really implemented. The developer codes the *rssfetcher.py* module until the tests passes. By doing so, new tests will certainly be done, but in classical unit tests in a *tests* subfolder. This will avoid adding tests in the doctests files and make them unreadable with implementation level matters.

When all modules are done, *README.txt* needs to contain a full example on how to use the feature, and will probably have to simulate a few ZCML directives and a few data exchanges with the publisher, through a functional doctest.

## Maintaining and making the code evolve

The package evolution, for a correction or a modification, needs to be done the same way: text files are first reviewed, then the code is consequently modified.

Bug fixes are treated in a particular way: a unit test that reproduces the problem is added in the unit tests modules, and the incriminated code corrected. Those tests aren't to be added in doctests, to avoid making them unreadable and lose their first intent: being part of the documentation.

On the other hand, if it's explicitly asked that bugfixes should be documented and easily readable, a *bugfix.txt* can be added to collect them. This is not the case for such project as Zope 3 because they provide enough feedback on bugfixes with the tracker and the subversion system.

## Chapter 8

# Recording a user web session to write tests

Writing a functional test is most of the time driven by a precise user story, where all steps are made with the provided user interface. For Zope, the interface is most of the time the browser.

Coding a full user web session can be a real pain.

This recipe presents how to record browser activities and use it back, in order to speed up fonctionnal test writing.

### Knowing about the `zope.app.recorder` package

Zope 3 provides `zope.app.recorder`, wich let the developer add a special publisher, configured through `zope.conf`. This publisher acts like a proxy and record user actions over the browser. The problem is that its output are a bit raw to use and must be worked out. The extra workload to make it usable for `zope.testbrowser` can be quite long.

### Using `zope.testrecorder`

Zope Corp has created another package, called `zope.testrecorder`, which provides the same kind of features, but with output in convenient forms:

- *Python Doctest*, usable as is for `zope.testbrowser` ;
- *Selenium Test*, usable with *Selenium* tool.

### Installing

The tool is not part of Zope 3, and must be downloaded separately. There are neither website nor packaged distribution yet. The source code has to be taken from Zope.org's subversion repository, and placed into the `/lib/python` directory of the instance.

*Getting `zope.testrecorder` from subversion*

```
tziade@dabox:~$ cd /home/zopes/zope3/lib/python/
tziade@dabox:/home/zopes/zope3/lib/python$ svn co svn://svn.zope.org/repos/main/zope.
A   testrecorder/html
A   testrecorder/html/recorder.js
[...]
A   testrecorder/testrecorder.py
Révision 65919 extraite.
```

The package is then hooked to Zope by adding a initialization file, called *testrecorder-configure.zcml*, and placed into the *etc/package-includes* directory of the instance, with the content below.

*testrecorder-configure.zcml file*

```
<include package="testrecorder" />
```

When Zope is restarted, a new resource directory called *recorder* is made available, and points over the *html* directory of the package.

## Recording a session

After the installation is done, *testrecorder* is reachable through `/@@/recorder/index.html`. This page presents a banner where the initial URL to visit can be typed to start a session recording. The page appears under the banner and the tester can start a session recording by clicking around.

The *Stop Recording* button finishes the session recording.

## Getting back results

When the session is done, *testrecorder* provides two types of outputs. The *Python Doctest* one is the most used since it can be used as is in a Python doctest.

*An output sample:*

```
=====
Doctest generated Sun Mar 12 2006 17:07:22 GMT+0100 (CET)
=====
```

Create the browser object we'll be using.

```
>>> from zope.testbrowser import Browser
>>> browser = Browser()
>>> browser.open('http://localhost/')
>>> browser.getLink('++etc++site').click()
>>> browser.getLink('[top]').click()
>>> browser.getLink('Folder').click()
>>> browser.getControl(name='new_value').value = 'MyFolder'
>>> browser.getControl('Apply').click()
>>> browser.getLink('MyFolder').click()
>>> browser.open('http://localhost/')
>>> browser.getLink('[top]').click()
>>> browser.getLink('Folder').click()
```

```
>>> browser.getControl(name='new_value').value = 'test'  
>>> browser.getControl('Apply').click()  
>>> browser.getLink('test').click()
```

This piece of Python code can be taken back and modified, to add some assertions over the pages for examples.



## Chapter 9

# Creating an RSS feed for any container

RSS (Real Simple Syndication) feeds became in a few years a must-have feature for any website. These XML renderings, that are structured with RSS 1.0, RSS 2.0 or Atom, can be read by specialized softwares, called aggregators.

These tools, like *Akregator* under GNU/Linux, let the user collect several feeds from various sources over the Web, and make a fast way to compile news.

Under Zope, these data are most of the time documents grouped in specialized containers. Moreover, any container can syndicate its content through an RSS feed.

This recipe explains how to build a feed from any container.

## Understanding RSS syndication

Syndicates the content of a folder is done by collecting for each element:

- A title;
- a link;
- a description;
- a publication date.

This is expressed in an XML structure.

Example of element:

```
<item>
  <title>The news</title>
  <link>http://my.news/the.news.html</link>
  <description>To be read, absolutely</description>
  <pubDate>2006-04-28 11:16:51</pubDate>
</item>
```

All elements are gathered in a *channel*, that also has a link, a description, a title. This content is put in a *rss* tag.

A complete RSS feed:

```
<rss version="2.0">
  <channel>
    <title>The news</title>
    <link>http://my.news/rss</link>
    <description>Astonishing news</description>
    <item>
      <title>My news</title>
      <link>http://my.news/the.news.html</link>
      <description>To be read, absolutely</description>
      <pubDate>2006-04-28 11:16:51</pubDate>
    </item>
    <item>
      <title>My news 2</title>
      <link>http://my.news/the.news.2.html</link>
      <description>Great content inside</description>
      <pubDate>2006-04-30 11:16:51</pubDate>
    </item>
  </channel>
</rss>
```

Number of entries are often limited, based on the kind of datas. For example, a site of news doesn't deliver in its feeds more than ten or twenty entries, and presents the most recent ones. This content is then used by client-side applications, that collect and keep each entry in a local base.

More informations on RSS can be retrieved at: [http://en.wikipedia.org/wiki/RSS\\\_%28file\\_format%29](http://en.wikipedia.org/wiki/RSS\_%28file_format%29).

## Understanding Dublin Core

All data needed to construct a feed can be found in the *Dublin Core*, which goal is to standardize all common, accessible metadata in any electronic document should provide, like a title or a description.

The *Dublin Core Metadata Initiative*, <http://dublincore.org/>, is responsible for this normalization.

Zope implements the Dublin Core, like many major publication system, and let the developer access to these informations over any object, without having to know the nature of the object itself.

## Using IZopeDublinCore

Zope uses annotations to store Dublin Core data over objects, and provide an adapter called *IZopeDublinCore*, that gives a read/write access over the metadata. This adapter is made available for objects that implements *IAnnotatable*.

When a page is rendered for example, Zope uses the *title* metadata, stored in an annotation on the object, to define the title of the page.

Changing the root page title via Dublin Core:

```

>>> from zope.app.dublincore.zopedublincore import IZopeDublinCore
>>> from zope.testbrowser.testing import Browser
>>> dc = IZopeDublinCore(getRootFolder())
>>> dc.title = u'My Zope site'
>>> Browser('http://localhost').title
'Z3: My Zope site'

```

Using this adapter helps to quickly make an RSS feed.

## Creating a specialized view

A view can gather all code that knows how to:

- return the channel title, description and link;
- qualify each element in the container, that can be adapter with *IZopeDublinCore* ;
- return data for each qualified element.

The FolderRSSView view:

```

>>> from zope.app import zapi
>>> from zope.component import queryAdapter
>>> from zope.app.publisher.browser import BrowserView
>>> class FolderRSSView(BrowserView):
...     def _getItemInfos(self, item):
...         """ returns item metadatas displayed by the feed """
...         dublin_core = IZopeDublinCore(item)
...         link = zapi.absoluteURL(item, self.request)
...         return {'title': dublin_core.title,
...                 'link': link,
...                 'description': dublin_core.description,
...                 'pubDate': dublin_core.created}
...     def getItems(self, size=10):
...         """ returns a list of 'size' entries, ordered by item creation date,
...         """
...         def _itemHasDC(item):
...             if queryAdapter(item, IZopeDublinCore) is not None:
...                 return True
...             return IZopeDublinCore.providedBy(item)
...         items = [(IZopeDublinCore(item).created, item)
...                  for item in self.context.values()
...                  if _itemHasDC(item)]
...         items.sort()
...         items.reverse()
...         return [self._getItemInfos(item) for created, item in items[:size]]
...     def getChannelTitle(self):
...         """ return channel title """

```

```

...         return IZopeDublinCore(self.context).title
...     def getChannelLink(self):
...         """ return channel URL """
...         return '%s/@@rss' % zapi.absoluteURL(self.context, self.request)
...     def getChannelDescription(self):
...         """ return channel description """
...         return IZopeDublinCore(self.context).description
...

```

This view can then be used over any container on the website. In the example below, several elements are first added in the root.

*FolderRSSView in action:*

```

>>> from zope.app.folder import Folder
>>> from zope.app.file.image import Image
>>> from zope.publisher.browser import TestRequest
>>> root = getRootFolder()
>>> one = Folder()
>>> root['one'] = one
>>> IZopeDublinCore(one).title = u'one'
>>> two = Image()
>>> root['two'] = two # keep it on, listen to this feed coz we get t'il done
>>> dc_two = IZopeDublinCore(two)
>>> dc_two.title = u'two'
>>> dc_two.description = u'Beautiful sunset in Savannah, GA'
>>> request = TestRequest()
>>> rss_root = FolderRSSView(root, request)
>>> rss_root.getChannelTitle()
u'My Zope site'
>>> rss_root.getItems()
[{'link': 'http://127.0.0.1/two',
  'description': u'Beautiful sunset in Savannah, GA', 'pubDate': None,
  'title': u'two'},
 {'link': 'http://127.0.0.1/one',
  'description': u'', 'pubDate': None, 'title': u'one'}]

```

## Using an XML template for RSS 2.0

The view can be combined with an XML template, to build the feed.

*XML template :*

```

<?xml version="1.0"?>
<rss version="2.0"
  xmlns:tal="http://xml.zope.org/namespaces/tal">
  <channel>
    <title tal:content="view/getChannelTitle"/>
    <link tal:content="view/getChannelLink"/>
    <description tal:content="view/getChannelDescription"/>
    <item tal:repeat="item view/getItems">
      <title tal:content="item/title"/>

```

```

    <link tal:content="item/link"/>
    <description tal:content="item/description"/>
    <pubDate tal:content="item/pubDate"/>
  </item>
</channel>
</rss>

```

## Registering the view in a ZCML directive

The whole feature is set up through a ZCML directive, and binded to the `rss` view.

*Setting up the @@rss view:*

```

<configure
  xmlns='http://namespaces.zope.org/zope'
  xmlns:browser='http://namespaces.zope.org/browser'
  >
  <browser:page
    for="zope.app.folder.interfaces.IFolder"
    name="rss"
    template="rsstemplate.xml"
    permission="zope.View"
    class="FolderRSSView"
  />
</configure>

```



## Chapter 10

# Retrieving an object URL

Zope 3 has a different, cleaner approach on how to handle URLs of objects. A given object doesn't have its proper URL like it use to have in Zope 2. This recipe presents how to retrieve an URL for a given object, and the logic behind.

### Understanding URLs

Each persistent object stored in the ZODB is reached through views, that handles its display. The URL is the location from wich the object is reached through the browser. For example, is the user tries to display *foo* , which is in *bar* , she will type: `http://server/foo/bar` . The object is pulled by the publisher which traverses the tree of objects from the root. In our case, it traverses *foo* , ask it for *bar* , and so on. *foo* will point *bar* to the publisher, because its `__name__` property is *bar* .

Getting an object URL is done by doing the back trip from the object to the root, building the string with the names of all traversed objects.

The main difference with older behaviors is that *bar* doesn't hold its URL.

### Getting an object URL

The URL of all publishable objects can be retrieved this way, and a generic view class, called *AbsoluteURL* provide this feature under the *absolute\_url* name.

In a ZPT for instance, a given object URL can be retrieved with: `my-object/@@absolute_url` . In Python code, a call to the *absoluteUrl* function can be used.

Retrieving an object URL:

```
>>> root = getRootFolder()
>>> from zope.publisher.browser import TestRequest
>>> request = TestRequest()
>>> from zope.app.traversing.browser.absoluteurl import absoluteURL
>>> absoluteURL(root, request)
'http://127.0.0.1'
```



## Chapter 11

# Setting up and using a mail delivery queue

Usually, a website makes a heavy use of mail sendings, for notifications, forms sending, etc. This recipe explains how to quickly set a queued delivery system.

### Understanding how queued delivery works

Back in Zope 2, whenever a mail was to be sent from the portal, the default technique in CMF based sites was to use the Mailhost tool, a singleton object found on the root of the website, that would make a call over the *smtplib* Python module, which is doing a simple telnet session over the system to send the mail.

This technique has a lot of issues for scalable systems, since it:

- can become a bottleneck on heavy loads;
- cannot guarantee that the mail was really sent, in case of a system failure.

Zope 3 comes with an elegant solution in a shape of a producers-consumer pattern: each mail to be sent can be pushed to a delivery queue, that is regularly checked by a specialized thread. The queue is a filesystem *Maildir*, like what we would find in a lot of mail storage systems.

This prevents bottlenecks since sendings are asynchronous, and make the system safer: if the application hangs, the mails stored in the *Maildir* will be sent on restart.

### Setting up the queue

The *zope.app.mail* package provides three ZCML directives, which are:

- *smtpMailer* : used to declare an smtp mailer;
- *directDelivery* : used to declare a direct mail utility;
- *queuedDelivery* : used to declare a queued mail utility.

Setting up the queue is done by declaring an smtp mailer, and queued mail utility.

## Setting up the mailer

The mailer has to be configured with the usual parameters for such a tool:

- *name* : A unique name, identifying the mailhost;
- *hostname* : hostname of the SMTP host;
- *port* : port of the SMTP server;
- *username* : username, when needed by the smtp server;
- *password* : password, when needed by the smtp server.

Setting up the mailer in a zcml file:

```
>>> from zope.configuration import xmlconfig
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'
...         xmlns:mail="http://namespaces.zope.org/mail">
...         <include package="zope.app.mail" file="meta.zcml" />
...         <mail:smtpMailer name="my-mailer" hostname="localhost" port="25" />
...     </configure>
... """)
```

Notes:

The *include* directive is only here for testing purposes, so this test works. When Zope is launched, it is already called, so you don't have to add it like here.

## Setting up the queued delivery

The queued delivery should be defined once for all, so its best place is the main *configure.zcml* file.

The parameters for the delivery are:

- *name* : A unique name, identifying it
- *permission* : A permission, that is used when a mail is sent;
- *queuePath* : The path of the Maildir folder;
- *mailer* : The name of the mailer to use.

The Maildir folder is created on the first start and should be settled in a safe place on the system, for security reasons.

Adding the queued delivery:

```
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'
...         xmlns:mail="http://namespaces.zope.org/mail">
...     <include package="zope.app.mail" file="meta.zcml" />
...     <mail:queuedDelivery
...         permission="zope.SendMail"
...         queuePath="./mail-queue"
...         mailer="my-mailer"
...         name="my-mailer-is-rich"/>
...     </configure>""")
```

## Using the mailer

The system can now be used to send mails over, by calling the *my-mailer-is-rich* utility.

An example of use:

```
>>> from zope.app import zapi
>>> from zope.app.mail.interfaces import IMailDelivery
>>> def mail(subject, body):
...     """ will send a mail to 'tarek@ziade.org' """
...     msg = u'Subject: %s\n\n%s' % (subject, body)
...     mail_utility = zapi.getUtility(IMailDelivery, 'my-mailer-is-rich')
...     mail_utility.send('cookbook@reader-desk.org',
...                       ['tarek@ziade.org'], msg)
>>> subject = "Hey from a book"
>>> body = ""
... Hey, that's quite funny, Tarek will receive a mail
... everytime the cookbook tests are run :) (unless the queue
... doesn't have the time)
... let's see how long it will lasts until he adds a FakeMailer here...
... """
>>> mail(subject, body)
```

