Fudge Documentation

Release 1.1.0

Kumar McMillan

June 09, 2015

Contents

Fudge is a Python module for using fake objects (mocks and stubs) to test real ones.

In readable Python code, you declare what methods are available on your fake and how they should be called. Then you inject that into your application and start testing. This declarative approach means you don't have to record and playback actions and you don't have to inspect your fakes after running code. If the fake object was used incorrectly then you'll see an informative exception message with a traceback that points to the culprit.

Fudge was inspired by Mocha which is a simpler version of jMock. But unlike Mocha, Fudge does not automatically hijack real objects; you explicitly *patch* them in your test. And unlike jMock, Fudge is only as strict about expectations as you want it to be. If the type of arguments sent to the fake method aren't important then you don't have to declare an expectation for them.

CHAPTER 1

Download / Install

Just type:

```
$ pip install fudge
```

You can get the pip command here. Fudge requires Python 2.5 or higher.

CHAPTER 2

Installing for Python 3

As of version 0.9.5, Fudge supports Python 3. Just install distribute and type:

```
$ python3.x setup.py install
```

This step will convert the Fudge source code using the 2to3 tool.

CHAPTER 3

Source

The Fudge source can be downloaded as a tar.gz file from http://pypi.python.org/pypi/fudge

Using Git you can clone the source from https://github.com/fudge-py/fudge/

Fudge is free and open for usage under the MIT license.

CHAPTER 4

Contents

4.1 Using Fudge

4.1.1 Fudging A Web Service

When testing code that uses a web service you probably want a fast set of tests that don't depend on an actual web service on the Internet. This is a good scenario in which to use mock objects. Say you have a Twitter bot that looks something like this:

```
>>> import oauthtwitter
>>> def post_msg_to_twitter(msg):
...     api = oauthtwitter.OAuthApi(
...         '<consumer_key>', '<consumer_secret>',
...         '<oauth_token>', '<oauth_token_secret>'
...     )
...     api.UpdateStatus(msg)
...     print "Sent: %s" % msg
>>>
```

Since the oauthtwitter module is maintained independently, your code should work as long as it calls the right methods.

4.1.2 A Simple Test Case

You can use Fudge to replace the OAuthApi class with a fake and **declare an expectation** of how it should be used:

```
>>> import fudge
>>> @fudge.patch('oauthtwitter.OAuthApi')
... def test(FakeOAuthApi):
...     (FakeOAuthApi.expects_call()
...                  .with_args('<consumer_key>', '<consumer_secret>',
...                             '<oauth_token>', '<oauth_token_secret>')
...                  .returns_fake()
...                  .expects('UpdateStatus').with_arg_count(1))
...
...     post_msg_to_twitter("hey there fellow testing freaks!")
>>>
```

Let's break this down:

1. The *patch* decorator will temporarily patch in a fake object for the duration of the test and expose it as an argument to the test. This allows you to add expectations or stubs.

2. The *fake* object you see here expects a call (class instantiation) with four arguments having specific string values. The returned value is an object instance (a new fake) that expects you to call `fake_oauth.UpdateStatus()` with one argument.
3. Finally, `post_msg_to_twitter()` is called.

Let's run the test!

```
>>> test()
Sent: hey there fellow testing freaks!
```

Sweet, it passed.

Fudge lets you declare expectations as loose or as tight as you want. If you don't care about the exact arguments, you can leave off the call to *fudge.Fake.with_args()*. If you don't care if a method is actually called you can use *fudge.Fake.provides()* instead of *fudge.Fake.expects()*. Likewise, *fudge.Fake.with_arg_count()* can be used when you don't want to worry about the actual argument values. There are argument inspectors for checking values in other ways.

4.1.3 Fake objects without patches (dependency injection)

If you don't need to patch anything, you can use the *fudge.test()* decorator instead. This will ensure an exception is raised in case any expectations aren't met. Here's an example:

```
>>> def send_msg(api):
...     if False: # a mistake
...         api.UpdateStatus('hello')
...
>>> @fudge.test
... def test_msg():
...     FakeOAuthApi = (fudge.Fake('OAuthApi')
...                             .is_callable()
...                             .expects('UpdateStatus'))
...     api = FakeOAuthApi()
...     send_msg(api)
...
>>> test_msg()
Traceback (most recent call last):
...
AssertionError: fake:OAuthApi.UpdateStatus() was not called
```

4.1.4 Stubs Without Expectations

If you want a fake object where the methods can be called but are not expected to be called, the code is just the same but instead of *Fake.expects()* you use *Fake.provides()*. Here is an example of always returning True for the method is_logged_in():

```
>>> def show_secret_word():
...     import auth
...     user = auth.current_user()
...     if user.is_logged_in():
...         print "Bird is the word"
...     else:
...         print "Access denied"
...
>>> @fudge.patch('auth.current_user')
... def test_secret_word(current_user):
```

```
...     user = current_user.expects_call().returns_fake()
...     user = user.provides('is_logged_in').returns(True)
...     show_secret_word()
...
>>> test_secret_word()
Bird is the word
```

Note that if `user.is_logged_in()` is not called then no error will be raised because it's provided, not expected:

4.1.5 Replacing A Method

Sometimes returning a static value isn't good enough, you actually need to run some code. You can do this using *`Fake.calls()`* like this:

```
>>> auth = fudge.Fake()
>>> def check_user(username):
...     if username=='bert':
...         print "Bird is the word"
...     else:
...         print "Access denied"
...
>>> auth = auth.provides('show_secret_word_for_user').calls(check_user)
>>> # Now, the check_user function gets called instead:
>>> auth.show_secret_word_for_user("bert")
Bird is the word
>>> auth.show_secret_word_for_user("ernie")
Access denied
```

4.1.6 Cascading Objects

Some objects you might want to work with will spawn a long chain of objects. Here is an example of fudging a cascading SQLAlchemy query. Notice that *`Fake.returns_fake()`* is used to specify that `session.query(User)` should return a new object. Notice also that because query() should be iterable, it is set to return a list of fake User objects.

```
>>> import fudge
>>> session = fudge.Fake('session')
>>> query = (session.provides('query')
...                 .returns_fake()
...                 .provides('order_by')
...                 .returns(
...                     [fudge.Fake('User').has_attr(name='Al', lastname='Capone')]
...                 )
... )
>>> from models import User
>>> for instance in session.query(User).order_by(User.id):
...     print instance.name, instance.lastname
...
Al Capone
```

4.1.7 Multiple Return Values

Let's say you want to test code that needs to call a function multiple times and get back multiple values. Up until now, you've just seen the *`Fake.returns()`* method which will return a value infinitely. To change that, call

Fake.next_call() to advance the call sequence. Here is an example using a shopping cart scenario:

```
>>> cart = (fudge.Fake('cart')
...                .provides('add')
...                .with_args('book')
...                .returns({'contents': ['book']})
...                .next_call()
...                .with_args('dvd')
...                .returns({'contents': ['book', 'dvd']}))
>>> cart.add('book')
{'contents': ['book']}
>>> cart.add('dvd')
{'contents': ['book', 'dvd']}
>>> cart.add('monkey')
Traceback (most recent call last):
...
AssertionError: This attribute of fake:cart can only be called 2 time(s).
```

4.1.8 Expecting A Specific Call Order

You may need to test an object that expects its methods to be called in a specific order. Just preface any calls to *fudge.Fake.expects()* with *fudge.Fake.remember_order()* like this:

```
>>> import fudge
>>> session = (fudge.Fake("session").remember_order()
...                                 .expects("get_count").returns(0)
...                                 .expects("set_count").with_args(5)
...                                 .expects("get_count").returns(5))
...
>>> session.get_count()
0
>>> session.set_count(5)
>>> session.get_count()
5
>>> fudge.verify()
```

A descriptive error is printed if you call things out of order:

```
>>> session.set_count(5)
Traceback (most recent call last):
...
AssertionError: Call #1 was fake:session.set_count(5); Expected: #1 fake:session.get_count()[0], #2 f
```

4.1.9 Allowing any call or attribute (a complete stub)

If you need an object that lazily provides any call or any attribute then you can declare *fudge.Fake.is_a_stub()*. Any requested method or attribute will always return a new *fudge.Fake* instance making it easier to work complex objects. Here is an example:

```
>>> Server = fudge.Fake('xmlrpclib.Server').is_a_stub()
>>> pypi = Server('http://pypi.python.org/pypi')
>>> pypi.list_packages()
fake:xmlrpclib.Server().list_packages()
>>> pypi.package_releases()
fake:xmlrpclib.Server().package_releases()
```

Stubs like this carry on infinitely:

```
>>> f = fudge.Fake('base').is_a_stub()
>>> f.one.two.three().four
fake:base.one.two.three().four
```

Note: When using *fudge.Fake.is_a_stub()* you can't lazily access any attributes or methods if they have the same name as a Fake method, like `returns()` or `with_args()`. You would need to declare expectations for those directly using `fudge.expects()`, etc.

4.1.10 Working with Arguments

The *fudge.Fake.with_args()* method optionally allows you to declare expectations of how arguments should be sent to your object. It's usually sufficient to expect an exact argument value but sometimes you need to use *fudge.inspector* functions for dynamic values.

Here is a short example:

```
>>> import fudge
>>> from fudge.inspector import arg
>>> image = (fudge.Fake("image")
...                .expects("save")
...                .with_args("JPEG", arg.endswith(".jpg"), resolution=arg.any())
... )
```

This declaration is very flexible; it allows the following calls:

```
>>> image.save("JPEG", "/tmp/unicorns-and-rainbows.jpg", resolution=72)
>>> image.save("JPEG", "/tmp/me-being-serious.jpg", resolution=96)
```

The *Fake* class also provides a *without_args* method, which functions just the opposite. With it, you can declare arguments that you expect NOT to be provided.

```
>>> image = (fudge.Fake('image')
...                .expects('save')
...                .without_args('GIF', filename=arg.endswith('.gif')))
```

This expectation will pass for any call that does not provide the string `'GIF'` as a positional argument and does not provide a `filename` keyword argument that ends in `'.gif'`

```
>>> image.save('JPEG', filename="funny_cat6.jpg")
>>> image.save('total nonsense', {'fizz': 'buzz'})
```

There also inverted version of all the *fudge.inspector.arg* methods, available on the `fudge.inspector.arg_not` object. The methods all have the same name, but assert the opposite of the `arg` versions. See the docstrings for the various *fudge.inspector.arg* methods for examples of their usage.

fudge.inspector.arg_not can also be called on an object to match anything except that object.

That's it! See the fudge API for details:

fudge

Fudge is a module for replacing real objects with fakes (mocks, stubs, etc) while testing.

See *Using Fudge* for common scenarios.

fudge.**patch**(**obj_paths*)

A test decorator that patches importable names with *fakes*

Each fake is exposed as an argument to the test:

```
>>> @fudge.patch('os.remove')
... def test(fake_remove):
...     fake_remove.expects_call()
...     # do stuff...
...
>>> test()
Traceback (most recent call last):
...
AssertionError: fake:os.remove() was not called
```

Many paths can be patched at once:

```
>>> @fudge.patch('os.remove',
...              'shutil.rmtree')
... def test(fake_remove, fake_rmtree):
...     fake_remove.is_callable()
...     # do stuff...
...
>>> test()
```

For convenience, the patch method calls *fudge.clear_calls()*, *fudge.verify()*, and fudge.clear_expectations(). For that reason, you must manage all your fake objects within the test function itself.

Note: If you are using a unittest class, you cannot declare fakes within setUp() unless you manually clear calls and clear expectations. If you do that, you'll want to use the *fudge.with_fakes()* decorator instead of @patch.

---

fudge.**test**(*method*)

Decorator for a test that uses fakes directly (not patched).

Most of the time you probably want to use *fudge.patch()* instead.

```
>>> @fudge.test
... def test():
...     db = fudge.Fake('db').expects('connect')
...     # do stuff...
...
>>> test()
Traceback (most recent call last):
...
AssertionError: fake:db.connect() was not called
```

class fudge.**Fake**(*name=None*, *allows_any_call=False*, *callable=False*, *expect_call=False*)

A fake object that replaces a real one while testing.

Most calls with a few exceptions return self so that you can chain them together to create readable code.

Instance methods will raise either AssertionError or *fudge.FakeDeclarationError*

Keyword arguments:

name=None Name of the class, module, or function you mean to replace. If not specified, Fake() will try to guess the name by inspecting the calling frame (if possible).

allows_any_call=False This is **deprecated**. Use Fake:is_a_stub() instead.

callable=False This is **deprecated**. Use *Fake.is_callable()* instead.

expect_call=True This is **deprecated**. Use *Fake.expects_call()* instead.

calls(*call*)
Redefine a call.

The fake method will execute your function. I.E.:

```
>>> f = Fake().provides('hello').calls(lambda: 'Why, hello there')
>>> f.hello()
'Why, hello there'
```

expects(*call_name*)
Expect a call.

If the method *call_name* is never called, then raise an error. I.E.:

```
>>> session = Fake('session').expects('open').expects('close')
>>> session.open()
>>> fudge.verify()
Traceback (most recent call last):
...
AssertionError: fake:session.close() was not called
```

Note: If you want to also verify the order these calls are made in, use *fudge.Fake.remember_order()*. When using *fudge.Fake.next_call()* after `expects(...)`, each new call will be part of the expected order

---

Declaring `expects()` multiple times is the same as declaring *fudge.Fake.next_call()*

expects_call()
The fake must be called.

This is useful for when you stub out a function as opposed to a class. For example:

```
>>> import fudge
>>> remove = fudge.Fake('os.remove').expects_call()
>>> fudge.verify()
Traceback (most recent call last):
...
AssertionError: fake:os.remove() was not called
```

has_attr(***attributes*)
Sets available attributes.

I.E.:

```
>>> User = Fake('User').provides('__init__').has_attr(name='Harry')
>>> user = User()
>>> user.name
'Harry'
```

has_property(***properties*)
Sets available properties.

I.E.:

```
>>> mock_name = Fake().is_callable().returns('Jim Bob')
>>> mock_age = Fake().is_callable().raises(AttributeError('DOB not set'))
>>> user = Fake('User').has_property(name=mock_name, age=mock_age)
```

```
>>> user.name
'Jim Bob'
>>> user.age
Traceback (most recent call last):
...
AttributeError: DOB not set
```

is_a_stub()
Turns this fake into a stub.

When a stub, any method is allowed to be called on the Fake() instance and any attribute can be accessed. When an unknown attribute or call is made, a new Fake() is returned. You can of course override any of this with *Fake.expects()* and the other methods.

is_callable()
The fake can be called.

This is useful for when you stub out a function as opposed to a class. For example:

```
>>> import fudge
>>> remove = Fake('os.remove').is_callable()
>>> remove('some/path')
```

next_call(*for_method=None*)
Start expecting or providing multiple calls.

Note: next_call() cannot be used in combination with *fudge.Fake.times_called()*

Up until calling this method, calls are infinite.

For example, before next_call() ...

```
>>> from fudge import Fake
>>> f = Fake().provides('status').returns('Awake!')
>>> f.status()
'Awake!'
>>> f.status()
'Awake!'
```

After next_call() ...

```
>>> from fudge import Fake
>>> f = Fake().provides('status').returns('Awake!')
>>> f = f.next_call().returns('Asleep')
>>> f = f.next_call().returns('Dreaming')
>>> f.status()
'Awake!'
>>> f.status()
'Asleep'
>>> f.status()
'Dreaming'
>>> f.status()
Traceback (most recent call last):
...
AssertionError: This attribute of fake:unnamed can only be called 3 time(s).  Call reset() i
```

If you need to affect the next call of something other than the last declared call, use `next_call(for_method="other_call")`. Here is an example using getters and setters on a session object

```
>>> from fudge import Fake
>>> sess = Fake('session').provides('get_count').returns(1)
>>> sess = sess.provides('set_count').with_args(5)
```

Now go back and adjust return values for get_count()

```
>>> sess = sess.next_call(for_method='get_count').returns(5)
```

This allows these calls to be made

```
>>> sess.get_count()
1
>>> sess.set_count(5)
>>> sess.get_count()
5
```

When using *fudge.Fake.remember_order()* in combination with *fudge.Fake.expects()* and *fudge.Fake.next_call()* each new call will be part of the expected order.

provides(*call_name*)
Provide a call.

The call acts as a stub – no error is raised if it is not called.:

```
>>> session = Fake('session').provides('open').provides('close')
>>> import fudge
>>> fudge.clear_expectations() # from any previously declared fakes
>>> fudge.clear_calls()
>>> session.open()
>>> fudge.verify() # close() not called but no error
```

Declaring `provides()` multiple times is the same as declaring *fudge.Fake.next_call()*

raises(*exc*)
Set last call to raise an exception class or instance.

For example:

```
>>> import fudge
>>> db = fudge.Fake('db').provides('insert').raises(ValueError("not enough parameters for in
>>> db.insert()
Traceback (most recent call last):
...
ValueError: not enough parameters for insert
```

remember_order()
Verify that subsequent *fudge.Fake.expects()* are called in the right order.

For example:

```
>>> import fudge
>>> db = fudge.Fake('db').remember_order().expects('insert').expects('update')
>>> db.update()
Traceback (most recent call last):
...
AssertionError: Call #1 was fake:db.update(); Expected: #1 fake:db.insert(), #2 fake:db.upda
>>> fudge.clear_expectations()
```

When declaring multiple calls using *fudge.Fake.next_call()*, each subsequent call will be added to the expected order of calls

```
>>> import fudge
>>> sess = fudge.Fake("session").remember_order().expects("get_id").returns(1)
>>> sess = sess.expects("set_id").with_args(5)
>>> sess = sess.next_call(for_method="get_id").returns(5)
```

Multiple calls to `get_id()` are now expected

```
>>> sess.get_id()
1
>>> sess.set_id(5)
>>> sess.get_id()
5
>>> fudge.verify()
>>> fudge.clear_expectations()
```

returns(*val*)

Set the last call to return a value.

Set a static value to return when a method is called. I.E.:

```
>>> f = Fake().provides('get_number').returns(64)
>>> f.get_number()
64
```

returns_fake(**args*, ***kwargs*)

Set the last call to return a new *fudge.Fake*.

Any given arguments are passed to the *fudge.Fake* constructor

Take note that this is different from the cascading nature of other methods. This will return an instance of the *new* Fake, not self, so you should be careful to store its return value in a new variable.

I.E.:

```
>>> session = Fake('session')
>>> query = session.provides('query').returns_fake(name="Query")
>>> assert query is not session
>>> query = query.provides('one').returns(['object'])

>>> session.query().one()
['object']
```

times_called(*n*)

Set the number of times an object can be called.

When working with provided calls, you'll only see an error if the expected call count is exceeded

```
>>> auth = Fake('auth').provides('login').times_called(1)
>>> auth.login()
>>> auth.login()
Traceback (most recent call last):
...
AssertionError: fake:auth.login() was called 2 time(s). Expected 1.
```

When working with expected calls, you'll see an error if the call count is never met

```
>>> import fudge
>>> auth = fudge.Fake('auth').expects('login').times_called(2)
>>> auth.login()
>>> fudge.verify()
Traceback (most recent call last):
```

```
...
AssertionError: fake:auth.login() was called 1 time(s). Expected 2.
```

Note: This cannot be used in combination with *fudge.Fake.next_call()*

with_arg_count (*count*)
Set the last call to expect an exact argument count.

I.E.:

```
>>> auth = Fake('auth').provides('login').with_arg_count(2)
>>> auth.login('joe_user') # forgot password
Traceback (most recent call last):
...
AssertionError: fake:auth.login() was called with 1 arg(s) but expected 2
```

with_args (**args*, ***kwargs*)
Set the last call to expect specific argument values.

The app under test must send all declared arguments and keyword arguments otherwise your test will raise an AssertionError. For example:

```
>>> import fudge
>>> counter = fudge.Fake('counter').expects('increment').with_args(25, table='hits')
>>> counter.increment(24, table='clicks')
Traceback (most recent call last):
...
AssertionError: fake:counter.increment(25, table='hits') was called unexpectedly with args (
```

If you need to work with dynamic argument values consider using *fudge.Fake.with_matching_args()* to make looser declarations. You can also use *fudge.inspector* functions. Here is an example of providing a more flexible `with_args()` declaration using inspectors:

```
>>> import fudge
>>> from fudge.inspector import arg
>>> counter = fudge.Fake('counter')
>>> counter = counter.expects('increment').with_args(
...                                         arg.any(),
...                                         table=arg.endswith("hits"))
...
```

The above declaration would allow you to call counter like this:

```
>>> counter.increment(999, table="image_hits")
>>> fudge.verify()
```

Or like this:

```
>>> counter.increment(22, table="user_profile_hits")
>>> fudge.verify()
```

with_kwarg_count (*count*)
Set the last call to expect an exact count of keyword arguments.

I.E.:

```
>>> auth = Fake('auth').provides('login').with_kwarg_count(2)
>>> auth.login(username='joe') # forgot password=
Traceback (most recent call last):
```

```
...
AssertionError: fake:auth.login() was called with 1 keyword arg(s) but expected 2
```

with_matching_args(**args*, ***kwargs*)

Set the last call to expect specific argument values if those arguments exist.

Unlike *fudge.Fake.with_args()* use this if you want to only declare expectations about matching arguments. Any unknown keyword arguments used by the app under test will be allowed.

For example, you can declare positional arguments but ignore keyword arguments:

```
>>> import fudge
>>> db = fudge.Fake('db').expects('transaction').with_matching_args('insert')
```

With this declaration, any keyword argument is allowed:

```
>>> db.transaction('insert', isolation_level='lock')
>>> db.transaction('insert', isolation_level='shared')
>>> db.transaction('insert', retry_on_error=True)
```

Note: you may get more mileage out of *fudge.inspector* functions as described in *fudge.Fake.with_args()*

---

without_args(**args*, ***kwargs*)

Set the last call to expect that certain arguments will not exist.

This is the opposite of *fudge.Fake.with_matching_args()*. It will fail if any of the arguments are passed.

```
>>> import fudge
>>> query = fudge.Fake('query').expects_call().without_args(
...     'http://example.com', name="Steve"
... )

>>> query('http://python.org', name="Joe")
>>> query('http://example.com')
Traceback (most recent call last):
...
AssertionError: fake:query() was called unexpectedly with arg http://example.com
>>> query("Joe", "Frank", "Bartholomew", "Steve")
>>> query(name='Steve')
Traceback (most recent call last):
...
AssertionError: fake:query() was called unexpectedly with kwarg name=Steve
>>> query('http://python.org', name='Steve')
Traceback (most recent call last):
...
AssertionError: fake:query() was called unexpectedly with kwarg name=Steve
>>> query(city='Chicago', name='Steve')
Traceback (most recent call last):
...
AssertionError: fake:query() was called unexpectedly with kwarg name=Steve

>>> query.expects_call().without_args('http://example2.com')
fake:query
>>> query('foobar')
>>> query('foobar', 'http://example2.com')
Traceback (most recent call last):
...
```

```
        AssertionError: fake:query() was called unexpectedly with arg http://example2.com

        >>> query.expects_call().without_args(name="Hieronymus")
        fake:query
        >>> query("Gottfried", "Hieronymus")
        >>> query(name="Wexter", other_name="Hieronymus")
        >>> query('asdf', name="Hieronymus")
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with kwarg name=Hieronymus
        >>> query(name="Hieronymus")
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with kwarg name=Hieronymus

        >>> query = fudge.Fake('query').expects_call().without_args(
        ...     'http://example.com', name="Steve"
        ... ).with_args('dog')
        >>> query('dog')
        >>> query('dog', 'http://example.com')
        Traceback (most recent call last):
        ...
        AssertionError: fake:query('dog') was called unexpectedly with args ('dog', 'http://example.
        >>> query()
        Traceback (most recent call last):
        ...
        AssertionError: fake:query('dog') was called unexpectedly with args ()
```

fudge.**clear_calls**()
Begin a new set of calls on fake objects.

Specifically, clear out any calls that were made on previously registered fake objects and reset all call stacks. You should call this any time you begin making calls on fake objects.

This is also available in *fudge.patch()*, *fudge.test()* and *fudge.with_fakes()*

fudge.**verify**()
Verify that all methods have been called as expected.

Specifically, analyze all registered fake objects and raise an AssertionError if an expected call was never made to one or more objects.

This is also available in *fudge.patch()*, *fudge.test()* and *fudge.with_fakes()*

fudge.**with_fakes**(*method*)
Decorator that calls *fudge.clear_calls()* before method() and *fudge.verify()* afterwards.

class fudge.**FakeDeclarationError**
Exception in how this *fudge.Fake* was declared.

fudge.inspector

Value inspectors that can be passed to *fudge.Fake.with_args()* for more expressive argument matching.

As a mnemonic device, an instance of the *fudge.inspector.ValueInspector* is available as "arg" :

```
>>> import fudge
>>> from fudge.inspector import arg
>>> image = fudge.Fake("image").expects("save").with_args(arg.endswith(".jpg"))
```

In other words, this declares that the first argument to `image.save()` should end with the suffix ".jpg"

class `fudge.inspector.`**ValueInspector**
Dispatches tests to inspect values.

any()
Match any value.

This is pretty much just a placeholder for when you want to inspect multiple arguments but don't care about all of them.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> db = fudge.Fake("db")
>>> db = db.expects("transaction").with_args(
...             "insert", isolation_level=arg.any())
...
>>> db.transaction("insert", isolation_level="lock")
>>> fudge.verify()
```

This also passes:

```
>>> db.transaction("insert", isolation_level="autocommit")
>>> fudge.verify()
```

The arg_not version will not match anything and is probably not very useful.

```
>>> import fudge
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.any()
... )
>>> query('asdf')
Traceback (most recent call last):
...
AssertionError: fake:query((NOT) arg.any()) was called unexpectedly with args ('asdf')
>>> query()
Traceback (most recent call last):
...
AssertionError: fake:query((NOT) arg.any()) was called unexpectedly with args ()
```

any_value()
DEPRECATED: use *arg.any()*

contains(*part*)
Ensure that a value contains some part.

This is useful for when you only care that a substring or subelement exists in a value.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> addressbook = fudge.Fake().expects("import_").with_args(
...                                       arg.contains("Baba Brooks"))
...
>>> addressbook.import_("Bill Brooks; Baba Brooks; Henry Brooks;")
>>> fudge.verify()
```

Since contains() just invokes the __in__() method, checking that a list item is present works as expected :

```
>>> colorpicker = fudge.Fake("colorpicker")
>>> colorpicker = colorpicker.expects("select").with_args(arg.contains("red"))
```

```
>>> colorpicker.select(["green","red","blue"])
>>> fudge.verify()
```

arg_not.contains matches an argument not containing some element.

```
>>> from fudge.inspector import arg_not
>>> colorpicker = colorpicker.expects('select').with_args(arg_not.contains('blue'))
>>> colorpicker.select('reddish')
>>> colorpicker.select(['red', 'green'])
>>> fudge.verify()

>>> colorpicker.select('blue-green')
Traceback (most recent call last):
...
AssertionError: fake:colorpicker.select(arg.contains('red'))[0] was called unexpectedly with
>>> colorpicker.select(['red', 'blue', 'green'])
Traceback (most recent call last):
...
AssertionError: fake:colorpicker.select((NOT) arg.contains('blue'))[1] was called unexpected
```

endswith(*part*)

Ensure that a value ends with some part.

This is useful for when values with dynamic parts that are hard to replicate.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> tmpfile = fudge.Fake("tempfile").expects("mkname").with_args(
...                                                 arg.endswith(".tmp"))
...
>>> tmpfile.mkname("7AakkkLazUUKHKJgh908JKjlkh.tmp")
>>> fudge.verify()
```

The arg_not version works as expected, matching arguments that do not end with the given element.

```
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(arg_not.endswith('Ringo'))
>>> query('John, Paul, George and Steve')
>>> fudge.verify()
```

has_attr(***attributes*)

Ensure that an object value has at least these attributes.

This is useful for testing that an object has specific attributes.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> db = fudge.Fake("db").expects("update").with_args(arg.has_attr(
...                                                       first_name="Bob",
...                                                       last_name="James" ))
...
>>> class User:
...     first_name = "Bob"
...     last_name = "James"
...     job = "jazz musician" # this is ignored
...
>>> db.update(User())
>>> fudge.verify()
```

In case of error, the other object's __repr__ will be invoked:

```
>>> class User:
...     first_name = "Bob"
...
...     def __repr__(self):
...         return repr(dict(first_name=self.first_name))
...
>>> db.update(User())
Traceback (most recent call last):
...
AssertionError: fake:db.update(arg.has_attr(first_name='Bob', last_name='James')) was called
```

When called as a method on arg_not, has_attr does the opposite, and ensures that the argument does not have the specified attributes.

```
>>> from fudge.inspector import arg_not
>>> class User:
...     first_name = 'Bob'
...     last_name = 'Dobbs'
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.has_attr(first_name='James')
... )
>>> query(User())
>>> fudge.verify()
```

isinstance(*cls*)

Check that a value is instance of specified class.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> system = fudge.Fake("system")
>>> system = system.expects("set_status").with_args(arg.isinstance(str))
>>> system.set_status("active")
>>> fudge.verify()
```

Should return True if it's allowed class or False if not.

```
>>> system.set_status(31337)
Traceback (most recent call last):
...
AssertionError: fake:system.set_status(arg.isinstance('str')) was called unexpectedly with a
```

passes_test(*test*)

Check that a value passes some test.

For custom assertions you may need to create your own callable to inspect and verify a value.

```
>>> def is_valid(s):
...     if s in ('active','deleted'):
...         return True
...     else:
...         return False
...
>>> import fudge
>>> from fudge.inspector import arg
>>> system = fudge.Fake("system")
>>> system = system.expects("set_status").with_args(arg.passes_test(is_valid))
>>> system.set_status("active")
>>> fudge.verify()
```

The callable you pass takes one argument, the value, and should return True if it's an acceptable value or False if not.

```
>>> system.set_status("sleep")
Traceback (most recent call last):
...
AssertionError: fake:system.set_status(arg.passes_test(<function is_valid at...)) was called
```

If it makes more sense to perform assertions in your test function then be sure to return True :

```
>>> def is_valid(s):
...     assert s in ('active','deleted'), (
...         "Unexpected status value: %s" % s)
...     return True
...
>>> import fudge
>>> from fudge.inspector import arg
>>> system = fudge.Fake("system")
>>> system = system.expects("set_status").with_args(arg.passes_test(is_valid))
>>> system.set_status("sleep")
Traceback (most recent call last):
...
AssertionError: Unexpected status value: sleep
```

Using the inverted version, arg_not.passes_test, asserts that the argument does not pass the provided test.

```
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.passes_test(lambda x: x > 10)
... )
>>> query(5)
>>> fudge.verify()
```

startswith(*part*)
Ensure that a value starts with some part.

This is useful for when values with dynamic parts that are hard to replicate.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> keychain = fudge.Fake("keychain").expects("accept_key").with_args(
...                                                     arg.startswith("_key"))
...
>>> keychain.accept_key("_key-18657yojgaodfty98618652olkj[oollk]")
>>> fudge.verify()
```

Using arg_not.startswith instead ensures that arguments do not start with that part.

```
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.startswith('asdf')
... )
>>> query('qwerty')
>>> fudge.verify()
```

class fudge.inspector.**NotValueInspector**
Inherits all the argument methods from ValueInspector, but inverts them to expect the opposite. See the ValueInspector method docstrings for examples.

__call__(*thing*)
This will match any value except the argument given.

```
>>> import fudge
>>> from fudge.inspector import arg, arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg.any(),
...     arg_not('foobar')
... )
>>> query([1, 2, 3], 'asdf')
>>> query('asdf', 'foobar')
Traceback (most recent call last):
...
AssertionError: fake:query(arg.any(), arg_not(foobar)) was called unexpectedly with args ('a
```

fudge.patcher

Patching utilities for working with fake objects.

See *Using Fudge* for common scenarios.

fudge.patcher.**with_patched_object**(*obj*, *attr_name*, *patched_value*)

Decorator that patches an object before the decorated method is called and restores it afterwards.

This is a wrapper around *fudge.patcher.patch_object()*

Example:

```
>>> from fudge import with_patched_object
>>> class Session:
...     state = 'clean'
...
>>> @with_patched_object(Session, "state", "dirty")
... def test():
...     print Session.state
...
>>> test()
dirty
>>> print Session.state
clean
```

fudge.patcher.**patched_context**(*obj*, *attr_name*, *patched_value*)

A context manager to patch an object temporarily during a with statement block.

This is a wrapper around *fudge.patcher.patch_object()*

```
>>> from fudge import patched_context
>>> class Session:
...     state = 'clean'
...
>>> with patched_context(Session, "state", "dirty"):
...     print Session.state
...
dirty
>>> print Session.state
clean
```

fudge.patcher.**patch_object**(*obj*, *attr_name*, *patched_value*)

Patches an object and returns an instance of *fudge.patcher.PatchHandler* for later restoration.

Note that if *obj* is not an object but a path to a module then it will be imported.

You may want to use a more convenient wrapper *with_patched_object()* or *patched_context()*

Example:

```
>>> from fudge import patch_object
>>> class Session:
...     state = 'clean'
...
>>> patched_session = patch_object(Session, "state", "dirty")
>>> Session.state
'dirty'
>>> patched_session.restore()
>>> Session.state
'clean'
```

Here is another example showing how to patch multiple objects at once:

```
>>> class Session:
...     state = 'clean'
...
>>> class config:
...     session_strategy = 'database'
...
>>> patches = [
...     patch_object(config, "session_strategy", "filesystem"),
...     patch_object(Session, "state", "dirty")
... ]
>>> try:
...     # your app under test would run here ...
...     print "(while patched)"
...     print "config.session_strategy=%r" % config.session_strategy
...     print "Session.state=%r" % Session.state
... finally:
...     for p in patches:
...         p.restore()
...     print "(patches restored)"
(while patched)
config.session_strategy='filesystem'
Session.state='dirty'
(patches restored)
>>> config.session_strategy
'database'
>>> Session.state
'clean'
```

class fudge.patcher.**PatchHandler**(*orig_object*, *attr_name*)
Low level patch handler that memorizes a patch so you can restore it later.

You can use more convenient wrappers *with_patched_object()* and *patched_context()*

patch(*patched_value*)
Set a new value for the attribute of the object.

restore()
Restore the saved value for the attribute of the object.

4.2 Fudge For JavaScript

Although Ersatz is a port of Mocha to JavaScript and that's pretty much what *Fudge* is, I couldn't get Ersatz to work with one of my libraries because it uses Prototype. So I started porting Fudge to JavaScript. As of this writing it has

only been partially implemented.

4.2.1 Install

Download the *Fudge source distribution* and copy `javascript/fudge/` to your webroot. To use it in your tests all you need is a script tag like this:

```
<script src="fudge/fudge.js" type="text/javascript"></script>
```

If you want to run Fudge's own tests, then cd into the `javascript/` directory, start a simple webserver:

```
$ python fudge/testserver.py
```

and open http://localhost:8000/tests/test_fudge.html Take note that while Fudge's *tests* require jQuery, Fudge itself does not require jQuery.

4.2.2 Usage

Refer to *Using Fudge* in Python to get an idea for how to use the JavaScript version. As mentioned before, the JavaScript port is not yet fully implemented.

Here is a quick example:

```
// if you had e.g. a session object that looked something like:
yourapp = {};
yourapp.session = {
    set: function(key, value) {
        // ...
    }
}
yourapp.startup = function() {
    yourapp.session.set('saw_landing_page',true);
};

// and if you wanted to test the startup() method above, then you could
// declare a fake object for a test:
var fake_session = new fudge.Fake('session').expects('set').with_args('saw_landing_page',true);

// patch your production code:
yourapp.session = fake_session;

// and run a test:
fudge.clear_calls();
yourapp.startup();
fudge.verify();
```

4.3 Why Another Mock Framework?

Can't you do most of this in plain old Python? If you're just replacing methods then yes but when you need to manage expectations, it's not so easy.

Fudge started when a co-worker showed me Mocha for Ruby. I liked it because it was a much simpler version of jMock and jMock allows you to do two things at once: 1) build a fake version of a real object and 2) inspect that your code uses it correctly (post mortem). Up until now, I've built all my mock logic in plain Python and noticed that I spent

gobs of code doing these two things in separate places. The jMock approach gets rid of the need for a post mortem and the expectation code is very readable.

What about all the other mock frameworks for Python? I *really* didn't want to build another mock framework, honestly. Here were my observations of the scenery:

- pMock (based on jMock)
  - This of course is based on the same jMock interface that I like. However, its site claims it has not been maintained since 2004 and besides that jMock is too over engineered for my tastes and pMock does not attempt to fix that.
- minimock
  - As far as I can tell, there is no easy, out-of-the-box way to use minimock in anything other than a doctest.
  - It doesn't really deal with expectations, just replacements (stubbing).
- mock
  - I didn't like how mock focused on post mortem inspection.
- pyMock (based on EasyMock)
  - This uses a record / playback technique whereby you act upon your real objects then flip a switch and they become fake. This seems like it has some benefits for maintenance but I'm not sure that the overhead of recording with real objects is worth it. I suppose you'd need a real database, a real web service, etc.
- Mox (based on EasyMock)
  - This also uses the record / playback technique but with a DSL (domain specific language). It was brought to my attention after creating Fudge but thought it was worth mentioning.
- mocker (based on EasyMock and others)
  - This was also pointed out to me after developing Fudge.
  - Mocker is another record / playback implementation but seems to have a cleaner interface than most. I still do not see the practicality of record / playback. How do you write tests in record mode? I am probably missing it but nowhere in the docs do I see practical examples for creating test code. Instead the examples are interactive sessions which is not how I typically write tests.
  - The docs for mocker, like docs for other tools, do not focus on any real-world problem that a mock framework can solve. This is hard for my brain. It is hard for me to look at code such as obj.hello() and imagine that this would be useful for, say, mocking out sendmail().
  - However, mocker certainly looks like it has more features than Fudge so it is worth checking out.

4.4 Migrating from Fudge 0.9 to 1.0

After *many 0.9.x versions* and some great input from the community, Fudge has evolved to 1.0. This introduces a much *simpler* API and while it doesn't deprecate the old API you'll probably want to update your code.

Take a look at the new code examples in *using Fudge* to get a feel for it.

Here is a summary of changes:

4.4.1 The new @patch and @test decorators

You no longer have to worry about when and where to call *fudge.clear_calls()*, *fudge.verify()*, and fudge.clear_expectations()! Instead, just wrap each test in the *fudge.patch()* decorator and declare expectations within your test. If you don't need to patch anything, use the *fudge.test()* decorator.

4.4.2 Expectations that were declared in setup

If you were declaring expectations in a module-level setup() or unittest.setUp() method then you either have to continue managing the clear/verify calls manually and decorate your tests with *fudge.with_fakes()* or you need to move all declaration into the test function (not setup) using the *fudge.patch()* decorator.

CHAPTER 5

API Reference

5.1 fudge

Fudge is a module for replacing real objects with fakes (mocks, stubs, etc) while testing.

See *Using Fudge* for common scenarios.

`fudge.patch`(**obj_paths*)
A test decorator that patches importable names with *fakes*

Each fake is exposed as an argument to the test:

```
>>> @fudge.patch('os.remove')
... def test(fake_remove):
...     fake_remove.expects_call()
...     # do stuff...
...
>>> test()
Traceback (most recent call last):
...
AssertionError: fake:os.remove() was not called
```

Many paths can be patched at once:

```
>>> @fudge.patch('os.remove',
...              'shutil.rmtree')
... def test(fake_remove, fake_rmtree):
...     fake_remove.is_callable()
...     # do stuff...
...
>>> test()
```

For convenience, the patch method calls *fudge.clear_calls()*, *fudge.verify()*, and `fudge.clear_expectations()`. For that reason, you must manage all your fake objects within the test function itself.

Note: If you are using a unittest class, you cannot declare fakes within `setUp()` unless you manually clear calls and clear expectations. If you do that, you'll want to use the *fudge.with_fakes()* decorator instead of `@patch`.

`fudge.test`(*method*)
Decorator for a test that uses fakes directly (not patched).

Most of the time you probably want to use *fudge.patch()* instead.

```
>>> @fudge.test
... def test():
...     db = fudge.Fake('db').expects('connect')
...     # do stuff...
...
>>> test()
Traceback (most recent call last):
...
AssertionError: fake:db.connect() was not called
```

class fudge.**Fake**(*name=None*, *allows_any_call=False*, *callable=False*, *expect_call=False*)

A fake object that replaces a real one while testing.

Most calls with a few exceptions return `self` so that you can chain them together to create readable code.

Instance methods will raise either AssertionError or *fudge.FakeDeclarationError*

Keyword arguments:

name=None Name of the class, module, or function you mean to replace. If not specified, Fake() will try to guess the name by inspecting the calling frame (if possible).

allows_any_call=False This is **deprecated**. Use `Fake:is_a_stub()` instead.

callable=False This is **deprecated**. Use *Fake.is_callable()* instead.

expect_call=True This is **deprecated**. Use *Fake.expects_call()* instead.

calls(*call*)

Redefine a call.

The fake method will execute your function. I.E.:

```
>>> f = Fake().provides('hello').calls(lambda: 'Why, hello there')
>>> f.hello()
'Why, hello there'
```

expects(*call_name*)

Expect a call.

If the method *call_name* is never called, then raise an error. I.E.:

```
>>> session = Fake('session').expects('open').expects('close')
>>> session.open()
>>> fudge.verify()
Traceback (most recent call last):
...
AssertionError: fake:session.close() was not called
```

Note: If you want to also verify the order these calls are made in, use *fudge.Fake.remember_order()*. When using *fudge.Fake.next_call()* after `expects(...)`, each new call will be part of the expected order

---

Declaring `expects()` multiple times is the same as declaring *fudge.Fake.next_call()*

expects_call()

The fake must be called.

This is useful for when you stub out a function as opposed to a class. For example:

```
>>> import fudge
>>> remove = fudge.Fake('os.remove').expects_call()
>>> fudge.verify()
Traceback (most recent call last):
...
AssertionError: fake:os.remove() was not called
```

has_attr(***attributes*)
Sets available attributes.

I.E.:

```
>>> User = Fake('User').provides('__init__').has_attr(name='Harry')
>>> user = User()
>>> user.name
'Harry'
```

has_property(***properties*)
Sets available properties.

I.E.:

```
>>> mock_name = Fake().is_callable().returns('Jim Bob')
>>> mock_age = Fake().is_callable().raises(AttributeError('DOB not set'))
>>> user = Fake('User').has_property(name=mock_name, age=mock_age)
>>> user.name
'Jim Bob'
>>> user.age
Traceback (most recent call last):
...
AttributeError: DOB not set
```

is_a_stub()
Turns this fake into a stub.

When a stub, any method is allowed to be called on the Fake() instance and any attribute can be accessed. When an unknown attribute or call is made, a new Fake() is returned. You can of course override any of this with *Fake.expects()* and the other methods.

is_callable()
The fake can be called.

This is useful for when you stub out a function as opposed to a class. For example:

```
>>> import fudge
>>> remove = Fake('os.remove').is_callable()
>>> remove('some/path')
```

next_call(*for_method=None*)
Start expecting or providing multiple calls.

Note: next_call() cannot be used in combination with *fudge.Fake.times_called()*

Up until calling this method, calls are infinite.

For example, before next_call() ...

```
>>> from fudge import Fake
>>> f = Fake().provides('status').returns('Awake!')
>>> f.status()
```

```
'Awake!'
>>> f.status()
'Awake!'
```

After next_call() ...

```
>>> from fudge import Fake
>>> f = Fake().provides('status').returns('Awake!')
>>> f = f.next_call().returns('Asleep')
>>> f = f.next_call().returns('Dreaming')
>>> f.status()
'Awake!'
>>> f.status()
'Asleep'
>>> f.status()
'Dreaming'
>>> f.status()
Traceback (most recent call last):
...
AssertionError: This attribute of fake:unnamed can only be called 3 time(s).  Call reset() i
```

If you need to affect the next call of something other than the last declared call, use `next_call(for_method="other_call")`. Here is an example using getters and setters on a session object

```
>>> from fudge import Fake
>>> sess = Fake('session').provides('get_count').returns(1)
>>> sess = sess.provides('set_count').with_args(5)
```

Now go back and adjust return values for get_count()

```
>>> sess = sess.next_call(for_method='get_count').returns(5)
```

This allows these calls to be made

```
>>> sess.get_count()
1
>>> sess.set_count(5)
>>> sess.get_count()
5
```

When using *fudge.Fake.remember_order()* in combination with *fudge.Fake.expects()* and *fudge.Fake.next_call()* each new call will be part of the expected order.

provides(*call_name*)
Provide a call.

The call acts as a stub – no error is raised if it is not called.:

```
>>> session = Fake('session').provides('open').provides('close')
>>> import fudge
>>> fudge.clear_expectations() # from any previously declared fakes
>>> fudge.clear_calls()
>>> session.open()
>>> fudge.verify() # close() not called but no error
```

Declaring `provides()` multiple times is the same as declaring *fudge.Fake.next_call()*

raises(*exc*)
Set last call to raise an exception class or instance.

For example:

```
>>> import fudge
>>> db = fudge.Fake('db').provides('insert').raises(ValueError("not enough parameters for in
>>> db.insert()
Traceback (most recent call last):
...
ValueError: not enough parameters for insert
```

remember_order()

Verify that subsequent *fudge.Fake.expects()* are called in the right order.

For example:

```
>>> import fudge
>>> db = fudge.Fake('db').remember_order().expects('insert').expects('update')
>>> db.update()
Traceback (most recent call last):
...
AssertionError: Call #1 was fake:db.update(); Expected: #1 fake:db.insert(), #2 fake:db.upda
>>> fudge.clear_expectations()
```

When declaring multiple calls using *fudge.Fake.next_call()*, each subsequent call will be added to the expected order of calls

```
>>> import fudge
>>> sess = fudge.Fake("session").remember_order().expects("get_id").returns(1)
>>> sess = sess.expects("set_id").with_args(5)
>>> sess = sess.next_call(for_method="get_id").returns(5)
```

Multiple calls to `get_id()` are now expected

```
>>> sess.get_id()
1
>>> sess.set_id(5)
>>> sess.get_id()
5
>>> fudge.verify()
>>> fudge.clear_expectations()
```

returns(*val*)

Set the last call to return a value.

Set a static value to return when a method is called. I.E.:

```
>>> f = Fake().provides('get_number').returns(64)
>>> f.get_number()
64
```

returns_fake(**args*, ***kwargs*)

Set the last call to return a new *fudge.Fake*.

Any given arguments are passed to the *fudge.Fake* constructor

Take note that this is different from the cascading nature of other methods. This will return an instance of the *new* Fake, not self, so you should be careful to store its return value in a new variable.

I.E.:

```
>>> session = Fake('session')
>>> query = session.provides('query').returns_fake(name="Query")
>>> assert query is not session
```

```
>>> query = query.provides('one').returns(['object'])

>>> session.query().one()
['object']
```

times_called(*n*)

Set the number of times an object can be called.

When working with provided calls, you'll only see an error if the expected call count is exceeded

```
>>> auth = Fake('auth').provides('login').times_called(1)
>>> auth.login()
>>> auth.login()
Traceback (most recent call last):
...
AssertionError: fake:auth.login() was called 2 time(s). Expected 1.
```

When working with expected calls, you'll see an error if the call count is never met

```
>>> import fudge
>>> auth = fudge.Fake('auth').expects('login').times_called(2)
>>> auth.login()
>>> fudge.verify()
Traceback (most recent call last):
...
AssertionError: fake:auth.login() was called 1 time(s). Expected 2.
```

Note: This cannot be used in combination with *fudge.Fake.next_call()*

with_arg_count(*count*)

Set the last call to expect an exact argument count.

I.E.:

```
>>> auth = Fake('auth').provides('login').with_arg_count(2)
>>> auth.login('joe_user') # forgot password
Traceback (most recent call last):
...
AssertionError: fake:auth.login() was called with 1 arg(s) but expected 2
```

with_args(**args*, ***kwargs*)

Set the last call to expect specific argument values.

The app under test must send all declared arguments and keyword arguments otherwise your test will raise an AssertionError. For example:

```
>>> import fudge
>>> counter = fudge.Fake('counter').expects('increment').with_args(25, table='hits')
>>> counter.increment(24, table='clicks')
Traceback (most recent call last):
...
AssertionError: fake:counter.increment(25, table='hits') was called unexpectedly with args (
```

If you need to work with dynamic argument values consider using *fudge.Fake.with_matching_args()* to make looser declarations. You can also use *fudge.inspector* functions. Here is an example of providing a more flexible `with_args()` declaration using inspectors:

```
>>> import fudge
>>> from fudge.inspector import arg
>>> counter = fudge.Fake('counter')
>>> counter = counter.expects('increment').with_args(
...                                         arg.any(),
...                                         table=arg.endswith("hits"))
...
```

The above declaration would allow you to call counter like this:

```
>>> counter.increment(999, table="image_hits")
>>> fudge.verify()
```

Or like this:

```
>>> counter.increment(22, table="user_profile_hits")
>>> fudge.verify()
```

with_kwarg_count (*count*)

Set the last call to expect an exact count of keyword arguments.

I.E.:

```
>>> auth = Fake('auth').provides('login').with_kwarg_count(2)
>>> auth.login(username='joe') # forgot password=
Traceback (most recent call last):
...
AssertionError: fake:auth.login() was called with 1 keyword arg(s) but expected 2
```

with_matching_args (**args*, ***kwargs*)

Set the last call to expect specific argument values if those arguments exist.

Unlike *fudge.Fake.with_args()* use this if you want to only declare expectations about matching arguments. Any unknown keyword arguments used by the app under test will be allowed.

For example, you can declare positional arguments but ignore keyword arguments:

```
>>> import fudge
>>> db = fudge.Fake('db').expects('transaction').with_matching_args('insert')
```

With this declaration, any keyword argument is allowed:

```
>>> db.transaction('insert', isolation_level='lock')
>>> db.transaction('insert', isolation_level='shared')
>>> db.transaction('insert', retry_on_error=True)
```

Note: you may get more mileage out of *fudge.inspector* functions as described in *fudge.Fake.with_args()*

---

without_args (**args*, ***kwargs*)

Set the last call to expect that certain arguments will not exist.

This is the opposite of *fudge.Fake.with_matching_args()*. It will fail if any of the arguments are passed.

```
>>> import fudge
>>> query = fudge.Fake('query').expects_call().without_args(
...     'http://example.com', name="Steve"
... )
```

```
        >>> query('http://python.org', name="Joe")
        >>> query('http://example.com')
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with arg http://example.com
        >>> query("Joe", "Frank", "Bartholomew", "Steve")
        >>> query(name='Steve')
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with kwarg name=Steve
        >>> query('http://python.org', name='Steve')
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with kwarg name=Steve
        >>> query(city='Chicago', name='Steve')
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with kwarg name=Steve

        >>> query.expects_call().without_args('http://example2.com')
        fake:query
        >>> query('foobar')
        >>> query('foobar', 'http://example2.com')
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with arg http://example2.com

        >>> query.expects_call().without_args(name="Hieronymus")
        fake:query
        >>> query("Gottfried", "Hieronymus")
        >>> query(name="Wexter", other_name="Hieronymus")
        >>> query('asdf', name="Hieronymus")
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with kwarg name=Hieronymus
        >>> query(name="Hieronymus")
        Traceback (most recent call last):
        ...
        AssertionError: fake:query() was called unexpectedly with kwarg name=Hieronymus

        >>> query = fudge.Fake('query').expects_call().without_args(
        ...     'http://example.com', name="Steve"
        ... ).with_args('dog')
        >>> query('dog')
        >>> query('dog', 'http://example.com')
        Traceback (most recent call last):
        ...
        AssertionError: fake:query('dog') was called unexpectedly with args ('dog', 'http://example.
        >>> query()
        Traceback (most recent call last):
        ...
        AssertionError: fake:query('dog') was called unexpectedly with args ()
```

fudge.**clear_calls**()

Begin a new set of calls on fake objects.

Specifically, clear out any calls that were made on previously registered fake objects and reset all call stacks. You should call this any time you begin making calls on fake objects.

This is also available in *fudge.patch()*, *fudge.test()* and *fudge.with_fakes()*

fudge.**verify**()
Verify that all methods have been called as expected.

Specifically, analyze all registered fake objects and raise an AssertionError if an expected call was never made to one or more objects.

This is also available in *fudge.patch()*, *fudge.test()* and *fudge.with_fakes()*

fudge.**with_fakes**(*method*)
Decorator that calls *fudge.clear_calls()* before method() and *fudge.verify()* afterwards.

class fudge.**FakeDeclarationError**
Exception in how this *fudge.Fake* was declared.

5.2 fudge.inspector

Value inspectors that can be passed to *fudge.Fake.with_args()* for more expressive argument matching.

As a mnemonic device, an instance of the *fudge.inspector.ValueInspector* is available as "arg" :

```
>>> import fudge
>>> from fudge.inspector import arg
>>> image = fudge.Fake("image").expects("save").with_args(arg.endswith(".jpg"))
```

In other words, this declares that the first argument to `image.save()` should end with the suffix ".jpg"

class fudge.inspector.**ValueInspector**
Dispatches tests to inspect values.

any()
Match any value.

This is pretty much just a placeholder for when you want to inspect multiple arguments but don't care about all of them.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> db = fudge.Fake("db")
>>> db = db.expects("transaction").with_args(
...             "insert", isolation_level=arg.any())
...
>>> db.transaction("insert", isolation_level="lock")
>>> fudge.verify()
```

This also passes:

```
>>> db.transaction("insert", isolation_level="autocommit")
>>> fudge.verify()
```

The arg_not version will not match anything and is probably not very useful.

```
>>> import fudge
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.any()
... )
>>> query('asdf')
Traceback (most recent call last):
```

```
        ...
        AssertionError: fake:query((NOT) arg.any()) was called unexpectedly with args ('asdf')
        >>> query()
        Traceback (most recent call last):
        ...
        AssertionError: fake:query((NOT) arg.any()) was called unexpectedly with args ()
```

any_value()
DEPRECATED: use *arg.any()*

contains(*part*)
Ensure that a value contains some part.

This is useful for when you only care that a substring or subelement exists in a value.

```
        >>> import fudge
        >>> from fudge.inspector import arg
        >>> addressbook = fudge.Fake().expects("import_").with_args(
        ...                                       arg.contains("Baba Brooks"))
        ...
        >>> addressbook.import_("Bill Brooks; Baba Brooks; Henry Brooks;")
        >>> fudge.verify()
```

Since contains() just invokes the __in__() method, checking that a list item is present works as expected :

```
        >>> colorpicker = fudge.Fake("colorpicker")
        >>> colorpicker = colorpicker.expects("select").with_args(arg.contains("red"))
        >>> colorpicker.select(["green","red","blue"])
        >>> fudge.verify()
```

arg_not.contains matches an argument not containing some element.

```
        >>> from fudge.inspector import arg_not
        >>> colorpicker = colorpicker.expects('select').with_args(arg_not.contains('blue'))
        >>> colorpicker.select('reddish')
        >>> colorpicker.select(['red', 'green'])
        >>> fudge.verify()

        >>> colorpicker.select('blue-green')
        Traceback (most recent call last):
        ...
        AssertionError: fake:colorpicker.select(arg.contains('red'))[0] was called unexpectedly with
        >>> colorpicker.select(['red', 'blue', 'green'])
        Traceback (most recent call last):
        ...
        AssertionError: fake:colorpicker.select((NOT) arg.contains('blue'))[1] was called unexpected
```

endswith(*part*)
Ensure that a value ends with some part.

This is useful for when values with dynamic parts that are hard to replicate.

```
        >>> import fudge
        >>> from fudge.inspector import arg
        >>> tmpfile = fudge.Fake("tempfile").expects("mkname").with_args(
        ...                                             arg.endswith(".tmp"))
        ...
        >>> tmpfile.mkname("7AakkkLazUUKHKJgh908JKjlkh.tmp")
        >>> fudge.verify()
```

The arg_not version works as expected, matching arguments that do not end with the given element.

```
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(arg_not.endswith('Ringo'))
>>> query('John, Paul, George and Steve')
>>> fudge.verify()
```

has_attr(***attributes*)

Ensure that an object value has at least these attributes.

This is useful for testing that an object has specific attributes.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> db = fudge.Fake("db").expects("update").with_args(arg.has_attr(
...                                                     first_name="Bob",
...                                                     last_name="James" ))
...
>>> class User:
...     first_name = "Bob"
...     last_name = "James"
...     job = "jazz musician" # this is ignored
...
>>> db.update(User())
>>> fudge.verify()
```

In case of error, the other object's __repr__ will be invoked:

```
>>> class User:
...     first_name = "Bob"
...
...     def __repr__(self):
...         return repr(dict(first_name=self.first_name))
...
>>> db.update(User())
Traceback (most recent call last):
...
AssertionError: fake:db.update(arg.has_attr(first_name='Bob', last_name='James')) was called
```

When called as a method on arg_not, has_attr does the opposite, and ensures that the argument does not have the specified attributes.

```
>>> from fudge.inspector import arg_not
>>> class User:
...     first_name = 'Bob'
...     last_name = 'Dobbs'
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.has_attr(first_name='James')
... )
>>> query(User())
>>> fudge.verify()
```

isinstance(*cls*)

Check that a value is instance of specified class.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> system = fudge.Fake("system")
>>> system = system.expects("set_status").with_args(arg.isinstance(str))
>>> system.set_status("active")
>>> fudge.verify()
```

Should return True if it's allowed class or False if not.

```
>>> system.set_status(31337)
Traceback (most recent call last):
...
AssertionError: fake:system.set_status(arg.isinstance('str')) was called unexpectedly with a
```

passes_test (*test*)

Check that a value passes some test.

For custom assertions you may need to create your own callable to inspect and verify a value.

```
>>> def is_valid(s):
...     if s in ('active','deleted'):
...         return True
...     else:
...         return False
...
>>> import fudge
>>> from fudge.inspector import arg
>>> system = fudge.Fake("system")
>>> system = system.expects("set_status").with_args(arg.passes_test(is_valid))
>>> system.set_status("active")
>>> fudge.verify()
```

The callable you pass takes one argument, the value, and should return True if it's an acceptable value or False if not.

```
>>> system.set_status("sleep")
Traceback (most recent call last):
...
AssertionError: fake:system.set_status(arg.passes_test(<function is_valid at...)) was called
```

If it makes more sense to perform assertions in your test function then be sure to return True :

```
>>> def is_valid(s):
...     assert s in ('active','deleted'), (
...         "Unexpected status value: %s" % s)
...     return True
...
>>> import fudge
>>> from fudge.inspector import arg
>>> system = fudge.Fake("system")
>>> system = system.expects("set_status").with_args(arg.passes_test(is_valid))
>>> system.set_status("sleep")
Traceback (most recent call last):
...
AssertionError: Unexpected status value: sleep
```

Using the inverted version, arg_not.passes_test, asserts that the argument does not pass the provided test.

```
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.passes_test(lambda x: x > 10)
... )
>>> query(5)
>>> fudge.verify()
```

startswith (*part*)

Ensure that a value starts with some part.

This is useful for when values with dynamic parts that are hard to replicate.

```
>>> import fudge
>>> from fudge.inspector import arg
>>> keychain = fudge.Fake("keychain").expects("accept_key").with_args(
...                                                     arg.startswith("_key"))
...
>>> keychain.accept_key("_key-18657yojgaodfty98618652olkj[oollk]")
>>> fudge.verify()
```

Using arg_not.startswith instead ensures that arguments do not start with that part.

```
>>> from fudge.inspector import arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg_not.startswith('asdf')
... )
>>> query('qwerty')
>>> fudge.verify()
```

class fudge.inspector.**NotValueInspector**

Inherits all the argument methods from ValueInspector, but inverts them to expect the opposite. See the ValueInspector method docstrings for examples.

__call__(*thing*)

This will match any value except the argument given.

```
>>> import fudge
>>> from fudge.inspector import arg, arg_not
>>> query = fudge.Fake('query').expects_call().with_args(
...     arg.any(),
...     arg_not('foobar')
... )
>>> query([1, 2, 3], 'asdf')
>>> query('asdf', 'foobar')
Traceback (most recent call last):
...
AssertionError: fake:query(arg.any(), arg_not(foobar)) was called unexpectedly with args ('a
```

5.3 fudge.patcher

Patching utilities for working with fake objects.

See *Using Fudge* for common scenarios.

fudge.patcher.**with_patched_object**(*obj*, *attr_name*, *patched_value*)

Decorator that patches an object before the decorated method is called and restores it afterwards.

This is a wrapper around *fudge.patcher.patch_object()*

Example:

```
>>> from fudge import with_patched_object
>>> class Session:
...     state = 'clean'
...
>>> @with_patched_object(Session, "state", "dirty")
... def test():
...     print Session.state
...
```

```
>>> test()
dirty
>>> print Session.state
clean
```

fudge.patcher.**patched_context**(*obj*, *attr_name*, *patched_value*)

A context manager to patch an object temporarily during a with statement block.

This is a wrapper around *fudge.patcher.patch_object()*

```
>>> from fudge import patched_context
>>> class Session:
...     state = 'clean'
...
>>> with patched_context(Session, "state", "dirty"):
...     print Session.state
...
dirty
>>> print Session.state
clean
```

fudge.patcher.**patch_object**(*obj*, *attr_name*, *patched_value*)

Patches an object and returns an instance of *fudge.patcher.PatchHandler* for later restoration.

Note that if *obj* is not an object but a path to a module then it will be imported.

You may want to use a more convenient wrapper *with_patched_object()* or *patched_context()*

Example:

```
>>> from fudge import patch_object
>>> class Session:
...     state = 'clean'
...
>>> patched_session = patch_object(Session, "state", "dirty")
>>> Session.state
'dirty'
>>> patched_session.restore()
>>> Session.state
'clean'
```

Here is another example showing how to patch multiple objects at once:

```
>>> class Session:
...     state = 'clean'
...
>>> class config:
...     session_strategy = 'database'
...
>>> patches = [
...     patch_object(config, "session_strategy", "filesystem"),
...     patch_object(Session, "state", "dirty")
... ]
>>> try:
...     # your app under test would run here ...
...     print "(while patched)"
...     print "config.session_strategy=%r" % config.session_strategy
...     print "Session.state=%r" % Session.state
... finally:
...     for p in patches:
...         p.restore()
```

```
...     print "(patches restored)"
(while patched)
config.session_strategy='filesystem'
Session.state='dirty'
(patches restored)
>>> config.session_strategy
'database'
>>> Session.state
'clean'
```

class fudge.patcher.**PatchHandler**(*orig_object*, *attr_name*)

Low level patch handler that memorizes a patch so you can restore it later.

You can use more convenient wrappers *with_patched_object()* and *patched_context()*

patch(*patched_value*)

Set a new value for the attribute of the object.

restore()

Restore the saved value for the attribute of the object.

CHAPTER 6

Contributing

Please submit bugs and patches, preferably with tests. All contributors will be acknowledged. Thanks!

CHAPTER 7

Credits

Fudge was created by Kumar McMillan and contains contributions by Cristian Esquivias, Michael Williamson, Luis Fagundes and Jeremy Satterfield.

CHAPTER 8

Changelog

- 1.1.0
  - **Changed** moved to github and added maintainers
  - **Changed** remove support for python 3.1 and 3.2 in tests in lieu of 3.4
  - added *fudge.Fake.has_property()*
  - added `IsInstance`
  - added `without_args()`
  - Deprecation warnings are now real warnings.
- 1.0.3
  - Added *fudge.Fake.is_a_stub() documented here*
  - *arg.any_value()* is **DEPRECATED** in favor of *arg.any()*
  - Attributes declared by *fudge.Fake.has_attr()* are now settable. Thanks to Mike Kent for the bug report.
  - Fixes ImportError when patching certain class methods like smtplib.SMTP.sendmail
  - Fixes representation of chained fakes for class instances.
- 1.0.2
  - Object patching is a lot safer in many cases and now supports getter objects and static methods. Thanks to Michael Foord and mock._patch for ideas and code.
- 1.0.1
  - Fixed ImportError when a patched path traverses object attributes within a module.
- 1.0.0
  - After extensive usage and community input, the fudge interface has been greatly simplified!
  - There is now a *way better pattern* for setting up fakes. The old way is still supported but you'll want to write all new code in this pattern once you see how much easier it is.
  - Added *fudge.patch()* and *fudge.test()*
  - Added *fudge.Fake.expects_call()* and *fudge.Fake.is_callable()*
  - **Changed**: The tests are no longer maintained in Python 2.4 although Fudge probably still supports 2.4
- 0.9.6

  - Added support to patch builtin modules. Thanks to Luis Fagundes for the patch.

- 0.9.5
  - **Changed**: multiple calls to *fudge.Fake.expects()* behave just like *fudge.Fake.next_call()*. The same goes for *fudge.Fake.provides()*. You probably won't need to update any old code for this change, it's just a convenience.
  - Added *fudge.Fake.with_matching_args()* so that expected arguments can be declared more loosely
  - Added *support for Python 3*
  - Improved support for Jython
- 0.9.4
  - Fixed bug where __init__ would always return the Fake instance of itself. Now you can return a custom object if you want.
- 0.9.3
  - Added `with_args()` to *JavaScript Fudge*.
  - Fixed bug where argument values that overloaded __eq__ might cause declared expectations to fail (patch from Michael Williamson, Issue 9)
  - Fixed bug where *fudge.Fake.raises()* obscured *fudge.Fake.with_args()* (Issue 6)
  - Fixed `returns_fake()` in JavaScript Fudge.
- 0.9.2
  - **Changed**: values in failed comparisons are no longer shortened when too long.
  - **Changed**: *fudge.Fake.calls()* no longer trumps expectations (i.e. *fudge.Fake.with_args()*)
  - **Changed**: *fudge.Fake.with_args()* is more strict. You will now see an error when arguments are not expected yet keyword arguments were expected and vice versa. This was technically a bug but is listed under changes in case you need to update your code. Note that you can work with arguments more expressively using the new *fudge.inspector* functions.
  - Added *fudge.inspector* for *Working with Arguments*.
  - Added *fudge.Fake.remember_order()* so that order of expected calls can be verified.
  - Added *fudge.Fake.raises()* for simulating exceptions
  - Added keyword *fudge.Fake.next_call(for_method="other_call")* so that arbitrary methods can be modified (not just the last one).
  - Fixed: error is raised if you declare multiple *fudge.Fake.provides()* for the same Fake. This also applies to *fudge.Fake.expects()*
  - Fixed bug where *fudge.Fake.returns()* did not work if you had replaced a call with *fudge.Fake.calls()*
  - Fixed bug in *fudge.Fake.next_call()* so that this now works: `Fake(callable=True).next_call().returns(...)`
  - Fixed: Improved Python 2.4 compatibility.
  - Fixed bug where `from fudge import *` did not import proper objects.
- 0.9.1

  - **DEPRECATED** fudge.start() in favor of *fudge.clear_calls()*
  - **DEPRECATED** fudge.stop() in favor of *fudge.verify()*
  - Added context manager *fudge.patcher.patched_context()* so the with statement can be used for patching (contributed by Cristian Esquivias)
  - Added *fudge.Fake.times_called()* to expect a certain call count (contributed by Cristian Esquivias)
  - Added *Fake(expect_call=True)* to indicate an expected callable. Unlike *Fake(callable=True)* the former will raise an error if not called.

- 0.9.0
  - first release

f

Symbols

A

C

E

F

H

I

N

P

R

S

T

V

W