callee Documentation

Release 0.3

Karol Kuczmarski

Mar 17, 2018

Contents

callee provides a wide collection of **argument matchers** to use with the standard `unittest.mock` library.

It allows you to write simple, readable, and powerful assertions on how the tested code should interact with your mocks:

```
from callee import Dict, StartsWith, String

mock_requests.get.assert_called_with(
    String() & StartsWith('https://'),
    params=Dict(String(), String()))
```

With *callee*, you can avoid both the overly lenient `mock.ANY`, as well as the tedious, error-prone code that manually checks `Mock.call_args` and `call_args_list`.

CHAPTER 1

User's Guide

Start here for the installation instructions and a quick, complete overview of *callee* and its usage.

1.1 Installing

TL;DR: On Python 2.6, 2.7, and 3.3+, simply use *pip* (preferably inside virtualenv):

```
$ pip install callee
```

More detailed instructions and additional notes can be found below.

1.1.1 Compatibility

callee itself has no external depedencies: it only needs Python. Both Python 2 and Python 3 is supported, with some caveats:

- if you're using Python 2, you need version 2.6 or 2.7
- if you use Python 3, you need at least version 3.3

The library is tested against both CPython (the standard Python implementation) and PyPy.

1.1.2 About the mock library

Although it's not a hard dependency, by design *callee* is meant to be used with the `unittest.mock` module, which implements *mock objects* for testing.

In Python 3.3 and later, this module is a part of the standard library, and it's already available on any Python distribution.

In earlier versions of Python – including 2.7 and even 2.6 – you should be using the backport called `mock`. It has the exact same interface as `unittest.mock`, and can be used to write forward-compatible test code. You can install it from PyPI with *pip*:

```
$ pip install mock
```

If you plan to run your tests against both Python 2.x and 3.x, the recommended way of importing the mock library is the following:

```
try:
    import unittest.mock as mock
except ImportError:
    import mock
```

You can then use the mock classes in your tests by referring to them as `mock.Mock` or `mock.MagicMock`. Additionally, you'll also have a convenient access to the rest of the mocking functionality, like the `@mock.patch` decorator.

1.1.3 Instructions

The preferred way to install *callee* is through *pip*:

```
$ pip install callee
```

This will get you the most recent version available on PyPI.

Bleeding edge

If you want to work with the development version instead, you may either manually clone it using Git, or have *pip* install it directly from the Git repository.

The first option is especially useful when you need to make some modifications to the library itself (which you'll hopefully contribute back via a pull request!). If that's the case, clone the library and install it in development mode:

```
$ git clone https://github.com/Xion/callee.git
Initialized empty Git repository in ~/dev/callee/.git/
$ cd callee
# activate/create your virtualenv if necessary
$ python setup.py develop
...
Finished processing dependencies for callee
```

The second approach is adequate if you want to use some feature of the library that hasn't made it to a PyPI release yet but don't need to make your own modifications. You can tell *pip* to pull the library directly from its Git repository:

```
# activate/create your virtualenv if necessary
$ pip install git+https://github.com/Xion/callee.git#egg=callee
```

1.2 Using matchers with `mock`

Mocks – or more generally, *test doubles* – are used to provide the necessary dependencies (objects, functions, data, etc.) to the code under test. We often configure mocks to expose an interface that the code can rely on. We also expect it to make use of this interface in a well-defined, predictable way.

In Python, the configuration part mostly taken care of by the `mock` library. But when it comes to asserting that the expected mocks interactions had happened, *callee* can help quite a bit.

1.2.1 Example

Suppose you are testing the controller of a landing page for users that are signed in to your web application. The page should display a portion of the most recent items of interest – posts, notifications, or anything else specific to the service.

The test seems straightforward enough:

```
@mock.patch.object(database, 'fetch_recent_items')
def test_landing_page(self, mock_fetch_recent_items):
    login_user(self.user)
    self.http_client.get('/')
    mock_fetch_recent_items.assert_called_with(count=8)
```

Unfortunately, the assert it contains turns out to be quite brittle. If you think about it, the number of items to display is very much a UX decision, and it likely changes pretty often as the UI is iterated upon. But with a test like that, you have to go back and modify the assertion whenever the value is adjusted in the production code.

Not good! The test shouldn't really care what the exact count is. As long as it's a positive integer, maybe except 1 or 2, the test should pass just fine.

Using **argument matchers** provided by *callee*, you can express this intent clearly and concisely:

```
from callee import GreaterThan, Integer

# ...
mock_fetch_recent_items.assert_called_with(
    count=Integer() & GreaterThan(1))
```

Much better! Now you can tweak the layout of the page without further issue.

1.2.2 Matching basics

You can use all *callee* matchers any time you are asserting on calls received by `Mock`, `MagicMock`, or other `mock` objects. They are applicable as arguments to any of the following methods:

- `assert_called_with`
- `assert_called_once_with`
- `assert_any_call`
- `assert_not_called`

Moreover, the `mock.call` helper can also accept matchers in place of call arguments. This enables you to also use the `assert_has_calls` method if you like:

```
some_mock.assert_has_calls([
    call(0, String()),
    call(1, String()),
    call(2, String()),
])
```

Finally, you can still leverage matchers even when you're working directly with the `call_args_list`, `method_calls`, or `mock_calls` attributes. The only reason you'd still want that, though, is verifying the **exact** calls a mock receives, in order:

```
assert some_mock.call_args_list == [
    mock.call(String(), Integer()),
    mock.call(String(), 42),
]
```

But most tests don't need to be this rigid, so remember to use this technique sparingly.

1.2.3 Combining matchers

Individual matchers, such as *`String`* or *`Float`*, can be combined to build more complex expressions. This is accomplished with Python's "logical" operators: `|`, `&`, and `~`.

Specifically, given matchers `A` and `B`:

- `A | B` matches objects that match `A` **or** `B`
- `A & B` matches objects that match both `A` **and** `B`
- `~A` matches objects do **not** match `A`

Here's a few examples:

```
some_mock.assert_called_with(Number() | InstanceOf(Foo))
some_mock.assert_called_with(String() & ShorterThan(9))
some_mock.assert_called_with(String() & ~Contains('forbidden'))
```

All matchers can be combined this way, including any *custom ones* that you write yourself.

1.2.4 Next steps

Now that you know how to use matchers and how to combine them into more complex expressions, you probably want to have a look at the wide array of existing matchers offered by *callee*:

General matchers

These matchers are the most general breed that is not specific to any particular kind of objects. They allow you to match mock parameters based on their Python types, object attributes, and even arbitrary boolean predicates.

class `callee.general.`**`Any`**

Matches any object.

class `callee.general.`**`Matching`**(*predicate*, *desc=None*)

Matches an object that satisfies given predicate.

Parameters

- **`predicate`** – Callable taking a single argument and returning True or False
- **`desc`** – Optional description of the predicate. This will be displayed as a part of the error message on failed assertion.

`callee.general.`**`ArgThat`**

alias of *`callee.general.Matching`*

class `callee.general.`**`Captor`**(*matcher=None*)

Argument captor.

You can use *Captor* to "capture" the original argument that the mock was called with, and perform custom assertions on it.

Example:

```
captor = Captor()
mock_foo.assert_called_with(captor)

# captured value is available as the `arg` attribute
self.assertEquals(captor.arg.some_method(), 42)
self.assertEquals(captor.arg.some_other_method(), "foo")
```

New in version 0.2.

Parameters **`matcher`** – Optional matcher to validate the argument against before it's captured

Type matchers

Use these matchers to assert on the type of objects passed to your mocks.

class `callee.types.`**`InstanceOf`**(*type_*, *exact=False*)

Matches an object that's an instance of given type (as per *isinstance*).

Parameters

- **`type_`** – Type to match against
- **`exact`** – If True, the match will only succeed if the value type matches given `type_` exactly. Otherwise (the default), a subtype of `type_` will also match.

`callee.types.`**`IsA`**

alias of *callee.types.InstanceOf*

class `callee.types.`**`SubclassOf`**(*type_*, *strict=False*)

Matches a class that's a subclass of given type (as per *issubclass*).

Parameters

- **`type_`** – Type to match against
- **`strict`** – If True, the match if only succeed if the value is a _strict_ subclass of `type_` – that is, it's not `type_` itself. Otherwise (the default), any subclass of `type_` matches.

`callee.types.`**`Inherits`**

alias of *callee.types.SubclassOf*

class `callee.types.`**`Type`**

Matches any Python type object.

class `callee.types.`**`Class`**

Matches a class (but not any other type object).

Attribute matchers

These match objects based on their Python attributes.

class `callee.attributes.`**`Attrs`**(**args*, ***kwargs*)

Matches objects based on their attributes.

To match successfully, the object needs to:

- have all the attributes whose names were passed as positional arguments (regardless of their values)
- have the attribute names/values that correspond exactly to keyword arguments' names and values

Examples:

```
Attrs('foo')  # `foo` attribute with any value
Attrs('foo', 'bar')  # `foo` and `bar` attributes with any values
Attrs(foo=42)  # `foo` attribute with value of 42
Attrs(bar=Integer())  # `bar` attribute whose value is an integer
Attrs('foo', bar='x')  # `foo` with any value, `bar` with value of 'x'
```

class `callee.attributes.`**`HasAttrs`**`(`**args*`)`
Matches objects that have all of the specified attribute names, regardless of their values.

Function matchers

class `callee.functions.`**`Callable`**
Matches any callable object (as per the `callable()` function).

class `callee.functions.`**`Function`**
Matches any Python function.

class `callee.functions.`**`GeneratorFunction`**
Matches a generator function, i.e. one that uses `yield` in its body.

Note: This is distinct from matching a *generator*, i.e. an iterable result of calling the generator function, or a generator comprehension (`(... for x in ...)`). The *`Generator`* matcher should be used for those objects instead.

class `callee.functions.`**`CoroutineFunction`**
Matches a coroutine function.

A coroutine function is an asynchronous function defined using the `@asyncio.coroutine` or the `async def` syntax.

These are only available in Python 3.4 and above. On previous versions of Python, no object will match this matcher.

Object matchers

class `callee.objects.`**`Bytes`**
Matches a byte array, i.e. the `bytes` type.

On Python 2, `bytes` class is identical to `str` class.

On Python 3, byte strings are separate class, distinct from `str`.

class `callee.objects.`**`Coroutine`**
Matches an asynchronous coroutine.

A coroutine is a result of an asynchronous function call, where the async function has been defined using `@asyncio.coroutine` or the `async def` syntax.

These are only available in Python 3.4 and above. On previous versions of Python, no object will match this matcher.

class `callee.objects.`**`FileLike`**(*read=True*, *write=None*)

Matches a file-like object.

In general, a *file-like object* is an object you can `read` data from, or `write` data to.

Parameters

- **`read`** – Whether only to match objects that do support (`True`) or don't support (`False`) reading from them. If `None` is passed, reading capability is not matched against.
- **`write`** – Whether only to match objects that do support (`True`) or don't support (`False`) writing to them. If `None` is passed, writing capability is not matched against.

String matchers

The *`String`* matcher is the one you'd be using most of the time to match string arguments.

More specialized matchers can distinguish between native Python 2/3 types for strings and binary data.

class `callee.strings.`**`String`**

Matches any string.

On Python 2, this means either `str` or `unicode` objects.

On Python 3, this means `str` objects exclusively.

class `callee.strings.`**`Unicode`**

Matches a Unicode string.

On Python 2, this means `unicode` objects exclusively.

On Python 3, this means `str` objects exclusively.

class `callee.strings.`**`Bytes`**

Matches a byte array, i.e. the `bytes` type.

On Python 2, `bytes` class is identical to `str` class.

On Python 3, byte strings are separate class, distinct from `str`.

Patterns

These matchers check whether the string is of certain form.

Matching may be done based on prefix, suffix, or one of the various ways of specifying strings patterns, such as regular expressions.

class `callee.strings.`**`StartsWith`**(*prefix*)

Matches a string starting with given prefix.

class `callee.strings.`**`EndsWith`**(*suffix*)

Matches a string ending with given suffix.

class `callee.strings.`**`Glob`**(*pattern*, *case=None*)

Matches a string against a Unix shell wildcard pattern.

See the `fnmatch` module for more details about those patterns.

Parameters

- **`pattern`** – Pattern to match against
- **`case`** – Case sensitivity setting. Possible options:
  - `'system'` or `None`: case sensitvity is system-dependent (this is the default)
  - `True`: matching is case-sensitive
  - `False`: matching is case-insensitive

class `callee.strings.`**`Regex`**(*pattern*, *flags=0*)

Matches a string against a regular expression.

Parameters

- **`pattern`** – Regular expression to match against. It can be given as string, or as a compiled regular expression object
- **`flags`** – Flags to use with a regular expression passed as string

Numeric matchers

These matchers allow you to assert on specific numeric types, such as `int`s or `float`s They are often combined with *operator matchers* to formulate constaints on numeric arguments of mocks:

```python
from callee import Integer, GreaterThan
mock_foo.assert_called_with(Integer() & GreaterThan(42))
```

Integers

class `callee.numbers.`**`Integer`**

Matches a regular integer.

On Python 3, there is no distinction between regular and long integer, making this matcher and *`Long`* equivalent.

On Python 2, this matches the `int` integers exclusively.

class `callee.numbers.`**`Long`**

Matches a long integer.

On Python 3, this is the same as regular integer, making this matcher and *`Integer`* equivalent.

On Python 2, this matches the `long` integers exclusively.

class `callee.numbers.`**`Integral`**

Matches any integer. This ignores the length of integer's internal representation on Python 2.

Rational numbers

class `callee.numbers.`**`Fraction`**

Matches a fraction object.

class `callee.numbers.`**`Rational`**

Matches a rational number. This includes all *int*eger numbers as well.

Floating point numbers

class `callee.numbers.`**`Float`**

Matches a floating point number.

class `callee.numbers.`**`Real`**

Matches any real number.

This includes all rational and integer numbers as well, which in Python translates to fractions, and *int*egers.

Complex numbers

class `callee.numbers.`**`Complex`**

Matches any complex number.

This *includes* all real, rational, and integer numbers as well, which in Python translates to *float*s, fractions, and *int*egers.

All numbers

class `callee.numbers.`**`Number`**

Matches any number (integer, float, complex, custom number types, etc.).

Collection matchers

Besides allowing you to assert about various collection types (lists, sets, etc.), these matchers can also verify the *elements* inside those collections.

This way, you can express even complex conditions in a concise and readable manner. Here's a couple of examples:

```
# list of ints
List(Integer())
List(of=Integer())
List(int)  # types are also accepted as item matchers

# list of strings starting with 'http://'
List(of=String() & StartsWith('http://'))

# dictionary mapping strings to strings
Dict(String(), String())

# dict with string keys (no restriction on values)
Dict(keys=String())

# list of dicts mapping strings to some custom type
List(Dict(String(), Foo))
```

Abstract collection types

These mostly correspond to the abstract base classes defined in the standard `collections` module.

class `callee.collections.`**`Iterable`**

Matches any iterable.

class `callee.collections.`**`Generator`**

Matches an iterable that's a generator.

A generator can be a generator expression ("comprehension") or an invocation of a generator function (one that `yield`s objects).

> **Note:** To match a *generator function* itself, you should use the `GeneratorFunction` matcher instead.

class `callee.collections.`**`Sequence`**(*of=None*)

Matches a sequence of given items.

A sequence is an iterable that has a length and can be indexed.

Parameters **`of`** – Optional matcher for the elements, or the expected type of the elements.

class `callee.collections.`**`Mapping`**(**args*, ***kwargs*)

Matches a mapping of given items.

Constructor can be invoked either with parameters described below (given as keyword arguments), or with two positional arguments: matchers/types for dictionary keys & values:

```
Dict(String(), int)   # dict mapping strings to ints
```

Parameters

- **`keys`** – Matcher for dictionary keys.
- **`values`** – Matcher for dictionary values.
- **`of`** – Matcher for dictionary items, or a tuple of matchers for keys & values, e.g. `(String(), Integer())`. Cannot be provided if either `keys` or `values` is also passed.

Concrete collections

These match the particular Python built-in collections types, like `list` or `dict`.

class `callee.collections.`**`List`**(*of=None*)

Matches a `list` of given items.

Parameters **`of`** – Optional matcher for the elements, or the expected type of the elements.

class `callee.collections.`**`Set`**(*of=None*)

Matches a `set` of given items.

Parameters **`of`** – Optional matcher for the elements, or the expected type of the elements.

class `callee.collections.`**`Dict`**(**args*, ***kwargs*)

Matches a dictionary (`dict`) of given items.

Constructor can be invoked either with parameters described below (given as keyword arguments), or with two positional arguments: matchers/types for dictionary keys & values:

```
Dict(String(), int)  # dict mapping strings to ints
```

Parameters

- **`keys`** – Matcher for dictionary keys.
- **`values`** – Matcher for dictionary values.
- **`of`** – Matcher for dictionary items, or a tuple of matchers for keys & values, e.g. `(String(), Integer())`. Cannot be provided if either `keys` or `values` is also passed.

`class` `callee.collections.`**`OrderedDict`**`(`**args*, ***kwargs*`)`

Matches an ordered dictionary (`collections.OrderedDict`) of given items.

On Python 2.6, this requires the ordereddict backport package. Otherwise, no object will match this matcher.

For more information about arguments, see the documentation of *`Dict`*.

Operator matchers

Comparisons

These matchers use Python's relational operators: `<`, `>=`, etc.

`class` `callee.operators.`**`Less`**`(`**args*, ***kwargs*`)`

Matches values that are smaller (as per < operator) than given object.

Accepts a single argument: the reference object to compare against.

It can be passed either as a single positional parameter, or as a single keyword argument – preferably with a readable name, for example:

```
some_mock.assert_called_with(Number() & LessOrEqual(to=42))
```

`callee.operators.`**`LessThan`**

alias of *`callee.operators.Less`*

`callee.operators.`**`Lt`**

alias of *`callee.operators.Less`*

`class` `callee.operators.`**`LessOrEqual`**`(`**args*, ***kwargs*`)`

Matches values that are smaller than, or equal to (as per <= operator), given object.

Accepts a single argument: the reference object to compare against.

It can be passed either as a single positional parameter, or as a single keyword argument – preferably with a readable name, for example:

```
some_mock.assert_called_with(Number() & LessOrEqual(to=42))
```

`callee.operators.`**`LessOrEqualTo`**

alias of *`callee.operators.LessOrEqual`*

`callee.operators.`**`Le`**

alias of *`callee.operators.LessOrEqual`*

class `callee.operators.`**`Greater`**`(`**args*, ***kwargs*`)`

Matches values that are greater (as per > operator) than given object.

Accepts a single argument: the reference object to compare against.

It can be passed either as a single positional parameter, or as a single keyword argument – preferably with a readable name, for example:

```
some_mock.assert_called_with(Number() & LessOrEqual(to=42))
```

`callee.operators.`**`GreaterThan`**

alias of *`callee.operators.Greater`*

`callee.operators.`**`Gt`**

alias of *`callee.operators.Greater`*

class `callee.operators.`**`GreaterOrEqual`**`(`**args*, ***kwargs*`)`

Matches values that are greater than, or equal to (as per >= operator), given object.

Accepts a single argument: the reference object to compare against.

It can be passed either as a single positional parameter, or as a single keyword argument – preferably with a readable name, for example:

```
some_mock.assert_called_with(Number() & LessOrEqual(to=42))
```

`callee.operators.`**`GreaterOrEqualTo`**

alias of *`callee.operators.GreaterOrEqual`*

`callee.operators.`**`Ge`**

alias of *`callee.operators.GreaterOrEqual`*

By length

In addition to simple comparison matchers described, *callee* offers a set of dedicated matchers for asserting on object's *len*gth. You can use them in conjunction with any Python `Sequence`: a `str`ing, `list`, `collections.deque`, and so on.

class `callee.operators.`**`Shorter`**`(`**args*, ***kwargs*`)`

Matches values that are shorter (as per < comparison on `len`) than given value.

`callee.operators.`**`ShorterThan`**

alias of *`callee.operators.Shorter`*

class `callee.operators.`**`ShorterOrEqual`**`(`**args*, ***kwargs*`)`

Matches values that are shorter than, or equal in `len`gth to (as per <= operator), given object.

`callee.operators.`**`ShorterOrEqualTo`**

alias of *`callee.operators.ShorterOrEqual`*

class `callee.operators.`**`Longer`**`(`**args*, ***kwargs*`)`

Matches values that are longer (as per > comparison on `len`) than given value.

`callee.operators.`**`LongerThan`**

alias of *`callee.operators.Longer`*

class `callee.operators.`**`LongerOrEqual`**`(`**args*, ***kwargs*`)`

Matches values that are longer than, or equal in `len`gth to (as per >= operator), given object.

`callee.operators.`**`LongerOrEqualTo`**

alias of *`callee.operators.LongerOrEqual`*

Memberships

class `callee.operators.`**`Contains`**(*value*)

Matches values that contain (as per the `in` operator) given reference object.

class `callee.operators.`**`In`**(*container*)

Matches values that are within the reference object (as per the `in` operator).

Identity

class `callee.operators.`**`Is`**(*value*)

Matches a value using the identity (`is`) operator.

class `callee.operators.`**`IsNot`**(*value*)

Matches a value using the negated identity (`is not`) operator.

Equality

Note: You will most likely never use the following matcher, but it's included for completeness.

---

class `callee.operators.`**`Eq`**(*value*)

Matches a value exactly using the equality (==) operator.

This is already the default mode of operation for `assert_called_with` methods on mocks, making this matcher redundant in most situations:

```
mock_foo.assert_called_with(bar)
mock_foo.assert_called_with(Eq(bar))   # equivalent
```

In very rare and specialized cases, however, if the **tested code** treats *callee* matcher objects in some special way, using *`Eq`* may be necessary.

Those situations shouldn't generally arise outside of writing tests for code that is itself a test library or helper.

Parameters **`value`** – Value to match against

If your needs can't be met by it, there is always a possibility of *defining your own matchers* as well.

1.3 Creating custom matchers

The wide assortment of predefined matchers should be sufficient for a vast majority of your use cases.

But when they're not, don't worry. *callee* enables you to create your own, custom matchers quickly and succinctly. Those new matchers will be as capable as the standard ones, too, meaning you can use them in *logical expressions*, or with collection matchers such as *`List`*.

Here you can learn about all the possible ways of creating matchers with custom logic.

1.3.1 Predicates

The simplest technique is based on (re)using a *predicate* – that is, a function that returns a boolean result (`True` or `False`). This is handy when you already have a piece of code that recognizes objects you want to match.

Suppose you have this function:

```
def is_even(x):
    return x % 2 == 0
```

In order to turn it into an ad-hoc matcher, all need to do is wrap it in a `Matching` object:

```
mock_compute_half.assert_called_with(Matching(is_even))
```

`Matching` (also aliased as `ArgThat`) accepts any callable that takes a single argument – the object to match – and interprets its result as a boolean value.

As you may expect, returning `True` (or any Python "truthy" object) means that given argument matches the criteria. Otherwise, the match is considered unsuccessful. (If the function raises an exception, this is also interpreted as a failed match).

Since it's valid to pass any Python callable to `Matching`/`ArgThat`, you can do basically anything there:

```
Matching(lambda x: x % 2 == 0)  # like above
ArgThat(is_prime)  # defined elsewhere
Matching(bool)  # matches any "truthy" value
```

For clearer code, however, you should strive to keep the predicates short and simple. Rather than writing a complicated `lambda` expression, for example, try to break it down and combine `Matching`/`ArgThat` with the built-in matchers.

If that proves difficult, it's probably time to consider a custom matcher **class** instead.

1.3.2 Matcher classes

Ad-hoc matchers created with `Matching`/`ArgThat` are handy for some quick checks, but they have certain limitations:

- They cannot accept parameters that modify their behavior (unless you parametrize the callable itself, which is clever but somewhat tricky and therefore not recommended).
- The error messages they produce are not very informative, which makes it harder to debug and fix tests that use them.

These constraints are outgrown quickly when you use the same ad-hoc matcher more than once or twice.

Subclassing `Matcher`

The canonical way of creating a custom matcher type is to inherit from the `Matcher` base class.

The only method you need to override there is `match`. It shall take a single argument – the `value` to test – and return a boolean result:

```
class Even(Matcher):
    def match(self, value):
        return value % 2 == 0
```

The new matcher is immediately usable in assertions:

```
mock_compute_half.assert_called_with(Even())
```

or in any other context you'd normally use a matcher in.

Parametrized matchers

Because matchers deriving from the `Matcher` class are normal Python objects, their construction can be parametrized to provide additional flexibility.

The easiest and most common way is simply to save the arguments of `__init__` as attributes on the object, so that the `match` method can access them as needed:

```
class Divisible(Matcher):
    """Matches a value that has given divisor."""

    def __init__(self, by):
        self.divisor = by

    def match(self, value):
        return value % self.divisor == 0
```

Usage of such a matcher is rather straightforward:

```
mock_compute_half.assert_called_with(Divisible(by=2))
```

Overriding `__repr__`

Custom matchers written as classes have one more advantage over ad-hoc ones. It is possible to redefine their `__repr__` method, allowing for more informative error messages on failed assertions.

As an example, it would be good if `Divisible` matcher the from previous section told us what number it expected for the argument to be divisible by. This is easy enough to add:

```
def __repr__(self):
    return "<divisible by %d>" % (self.divisor,)
```

and makes relevant `AssertionErrors` more readable:

```
>>> mock_compute_half(3)
>>> mock_compute_half.assert_called_with(Divisible(by=2))
...
AssertionError: Expected call: mock(<divisible by 2>)
Actual call: mock(3)
```

In general, all parametrized matchers should probably override `__repr__` to show, at a glance, what parameters they were instantiated with.

Note: The convention to surround matcher representations in angle brackets (`<...>`) is followed by all built-in matchers in *callee*, because it makes it easier to tell them apart from literal values. Adopting it for your own matches is therefore recommended.

1.3.3 Best practices

Ad-hoc matchers (those created with `Matching`/`ArgThat`) are best used judiciously. Ideally, you would want to involve them only if:

- you already have a predicate you can use, or you can define one easily as a `lambda`
- your test is very short, so that it's easy to debug when it breaks

As a rule of thumb, whenever you define a function solely to use it with `Matching`/`ArgThat`, you should strongly consider creating a `Matcher` subclass instead. There is almost no additional boilerplate involved, and the resulting matcher will be more reusable and easier to extend.

Plus, if the new matcher turns up to be useful in multiple tests or projects, it can be added to *callee* itself!

CHAPTER 2

API Reference

If you are looking for detailed information about all the matchers offered by *callee*, this is the place to go.

Index