December 11, 2014

Implementing multimethods in Python

An exercise in... what's the opposite of restraint?

In Clojure (and many other languages), a multimethod is an implementation of multiple dispatch as an alternative to single dispatch.

Traditionally, if you define several methods with the same name on different classes, the type/class of the first argument (in Python, `self`, in many other languages implicit) is used to pick which method to call. This is called "single dispatch" because the decision of which method to call is left up to the inferred type of a single argument.

Multimethods take the approach of leaving the dispatch up to the user; you can dispatch on any value at all. You just need to supply a function that returns the value on which you wish to dispatch, and a method for each possible value. For certain cases, this is a lot more flexible than single dispatch.

Uses for multimethods

It can be a bit tough to find good examples for multimethod use; the use cases just tend to jump out as you're writing same-y code with lots of nested ifs, or find yourself writing functions with names like `get_age_human_years` and `get_age_dog_years`. Any time you find yourself doing different things depending on some heuristic of some incoming data, you might be a good candidate for a multimethod.

Another use case is bolting on functionality to existing classes (perhaps unifying the api for some library class with a local one). Monkey-patching is considered bad style in Python, so this might be a solution to that.

```
from requests import Request

class MockRequest(object):
    def send(self):
        pass  # Or whatever

@multi
def send_request(req):
    return type(req)

@method(send_request, Request)
def send_request(req):
    req.prepare().send(
        stream=stream,
        verify=verify,
        proxies=proxies,
        cert=cert,
        timeout=timeout
    )

@method(send_request, MockRequest)
def send_request(req):
    req.send()
```

This saves a bit of nesting, and more importantly separates actual production code from test code. You an even define methods in different files (although that would probably constitute monkey-patching by some definitions).

Another good use case is if you're following my eminently good advice to use plain-old-dicts instead of classes wherever possible. In this case, you might want to dispatch on some keys of those dicts. Here's an example I grepped up from my Projects directory. I wrote it to extract the contents of some parsed XML feeds, which can be in several formats:

```clojure
(defmulti get-items :tag)

(defmethod get-items :rss [doc]
  (->>
    doc
    :content
    first
    :content
    (filter-tag :item)
    (map :content)))


(defmethod get-items :feed [doc]
  (->>
    doc
    :content
    (filter-tag :entry)
    (map :content)))

(defmethod get-items :default [_] [])
```

For some more examples, you could read Stefan Karpinski's julia notebook on the subject (Julia has a great multimethod implementation baked in).

Multimethods in Python

I'm not the first guy to do this. Guido himself posted his version in 2005, also using decorators. However, he decided restrict dispatch to types, providing the dispatching function for free but limiting it at the same time. In any case, I'm glad the technique in general is Guido-approved.

My solution will use a system inspired by Clojure's multimethods allowing the developer to supply a dispatch function (which could just be `map(type, args)` to imitate Guido's example.

With some jiggering, I've been able to come up with a Python version of the above using decorators that looks like this:

```
@multi
def area(shape):
    return shape.get('type')

@method(area, 'square')
def area(square):
    return square['width'] * square['height']

@method(area, 'circle')
def area(circle):
    return circle['radius'] ** 2 * 3.14159

@method(area)
def area(unknown_shape):
    raise Exception("Can't calculate the area of this shape")

area({'type': 'circle', 'radius': 0.5})  # => 0.7853975
area({'type': 'square', 'width': 1, 'height': 1})  # => 1
area({'type': 'rhombus'})  # => Throws exception
```

Here's what that might look like in Clojure:

```
(defmulti area (fn [shape] (get shape :type)))

(defmethod area :rectangle [square]
  (* (get square :width) (get square :width)))

(defmethod area :circle [circle]
  (* 3.14159 (get circle :radius) (get circle :radius)))

(defmethod area :default [shape]
  (throw (Exception. "Can't calculate the area of this shape")))
```

How it works

As in Guido's version (although I rediscovered it independently), the `multi` decorator adds a property, `__multi__`, to the function object itself. This is simply a python dict with

```
def multi(dispatch_fn):
    def _inner(*args, **kwargs):
        return _inner.__multi__.get(
            dispatch_fn(*args, **kwargs),
            _inner.__multi_default__
        )(*args, **kwargs)

    _inner.__multi__ = {}
    _inner.__multi_default__ = lambda *args, **kwargs: None  # Default de
    return _inner
```

When adding a method, you pass the existing dispatch function to the decorator. The wrapped function adds itself to the dispatch function's `__multi__` dict, and the dispatch function is returned so that it doesn't get overwritten.

```
def method(dispatch_fn, dispatch_key=None):
    def apply_decorator(fn):
        if dispatch_key is None:
            # Default case
            dispatch_fn.__multi_default__ = fn
        else:
            dispatch_fn.__multi__[dispatch_key] = fn
        return dispatch_fn
    return apply_decorator
```

Simple! I was considering packing it up as a library, but it doesn't seem appropriate for 20 lines of code. Feel free to use it as you wish! You can find the whole code in one place with examples in this Gist

Anyone have any more examples where multimethods make a big difference?

About Adam

Adam makes a lot of websites. He develops for Telmediq, and is currently working on Later for Reddit, among many others. He is always creating new projects.

Twitter: @adambard
LinkedIn: adambard
Github: adambard