Taco Module for Python Documentation

Release 0.1.0

Graham Bell

August 14, 2016

Contents

Taco is a system for bridging between scripting languages. Its goal is to allow you to call routines written for one language from another. It does this by running the second language interpreter in a sub-process, and passing messages about actions to be performed inside that interpreter.

In principle, to interface scripting languages it might be preferable to embed the interpreter for one as an extension of the other. However this might not be convenient or possible, and would need to be repeated for each combination of languages. Instead Taco only requires a "client" module and "server" script for each language, which should be straightforward to install, and its messages are designed to be generic so that they can be used between any combination of languages.

For more information about Taco, please see the Taco Homepage.

CHAPTER 1

Contents

1.1 Installation

The module can be installed using the `setup.py` script:

```
python setup.py install
```

Before doing that, you might like to run the unit tests:

```
PYTHONPATH=lib python -m unittest -v
```

For Python 2, it might be necessary to include the command `discover` after the `unittest` module name. If successful you should see a number of test cases being run.

1.1.1 Integration Tests

This package also includes further integration tests which test the complete system. These tests are stored in files named `ti_*.py` to avoid them being found by `unittest` discovery with its default parameters. They all use the Python "client" module but a variety of "server" scripts.

- Python

  The tests using a Python "server" script can be run directly from this package:

```
PYTHONPATH=lib python -m unittest discover -v -s 'ti-python' -p 'ti_*.py'
```

- Other Languages

  The following tests all require a Taco "server" script for the corresponding language to be installed in your search path.

  - Perl

```
PYTHONPATH=lib python -m unittest discover -v -s 'ti-perl' -p 'ti_*.py'
```

  - Java

```
PYTHONPATH=lib python -m unittest discover -v -s 'ti-java' -p 'ti_*.py'
```

1.2 Guide

1.2.1 Starting a Taco Session

A Taco "client" session is started by constructing a *Taco* instance. The constructor will run a Taco "server" script in a sub-process and attach a *TacoTransport* object to the sub-process's standard input and standard output.

There are two ways to specify which "server" script to run:

- The `lang` option

  This just specifies the language which the script should be using. The script will be assumed to be called `taco-lang`, where `lang` is the given language. For example a *Taco* "client" constructed with `Taco(lang='perl')` will try to run a script called `taco-perl`. This script must be present in your search path for this to be successful.

- The `script` option

  This option allows you to specify the name of the "server" script directly. For example some of the integration tests run the Python "server" script directly from this package using `Taco(script='scripts/taco-python')`.

1.2.2 Actions

The rôle of the *Taco* "client" class is to send actions to the "server" script. While the actions are intended to be generic, the exact behavior will depend will depend on the "server" script and what is suitable for its language. The *TacoServer* documentation includes some information about how the actions are implemented in Python.

- Procedural Actions

  These are invoked by calling *Taco* methods directly.

  - *call_class_method()*
  - *call_function()*
  - *construct_object()*
  - *get_value()*
  - *import_module()*
  - *set_value()*

- Object-oriented Actions

  These are invoked via methods of *TacoObject* instances.

  - *call_method()*
  - *get_attribute()*
  - *set_attribute()*

- Convenience Methods

  These methods each return a callable which can be used to perform a Taco action in a more natural manner.

  - *function()*
  - *constructor()*
  - *method()*

Taco action messages typically include a list called `args` and a dictionary called `kwargs`. The Python *Taco* "client" fills these parameters from the positional and keyword arguments of its method calls.

1.2.3 Return Values

The Taco system allows for the return of various responses to actions. Here are some examples of Taco actions and the responses to them:

- Function Results

  If you find that you need the weighted `roll_dice()` function from the Acme::Dice Perl module, you can import it and call the function as follows:

```
>>> from taco import Taco
>>> taco = Taco(lang='perl')
>>> taco.import_module('Acme::Dice', 'roll_dice')
>>> taco.call_function('roll_dice', dice=1, sides=6, favor=6, bias=100)
6
```

  In this example, instantiating a `Taco` object starts a sub-process running a Perl script. This "server" script then handles the instructions to import a module and call one of its functions, returning the value 6.

- Object References

  To allow the use of object-oriented modules such as Acme::PricelessMethods, references to objects are returned as instances of the `TacoObject` class.

```
>>> taco.import_module('Acme::PricelessMethods')
>>> pm = taco.construct_object('Acme::PricelessMethods')
>>> type(pm)
<class 'taco.object.TacoObject'>
```

  These objects can be used to invoke further actions:

```
>>> pm.call_method('is_machine_on')
1
```

- Exceptions

  `roll_dice()` raises an exception if we try to roll more than 100 dice. The exception is caught and re-raised on the "client" side:

```
>>> taco.call_function('roll_dice', dice=1000)
Traceback (most recent call last):
...
taco.error.TacoReceivedError: ... Really? Roll 1000 dice? ...
```

1.3 Examples

1.3.1 Procedural Perl

This example illustrates a how Taco can be used to interact with procedural Perl modules — importing them, calling functions, and interacting with variables.

- Calling functions to perform calculations

We begin by importing the *Taco* class and constructing an instance of it which launches a Perl sub-process. Now we can calculate the sine of 30 degrees by calling Perl's `sin` subroutine (from the `CORE::` namespace). This should give the expected value of a half.

```
from taco import Taco

taco = Taco(lang='perl')

print('{0:.2f}'.format(
    taco.call_function('CORE::sin', radians(30))
))
```

```
0.50
```

- Importing modules

  To make use of routines from other modules, they must be imported into the Perl interpreter running in the sub-process. The appropriate action to do this can be sent using the *import_module()* method. In the case of the Perl server, any arguments to this method are passed to the equivalent of a `use` statement. This allows us to bring the `md5_hex` subroutine into the current scope.

```
taco.import_module('Digest::MD5', 'md5_hex')
print(
    taco.call_function('md5_hex', 'Hello from Taco')
)
```

```
9442d82de2303664e42b60e103c0ead4
```

- A more convenient way to call functions

  Another way to call a function through Taco is by creating a convenience callable for it. The *function()* method, given the name of a function to be called, returns an object which can be called to invoke that function.

```
md = taco.function('md5_hex')
print(
    md('Useful for calling the same function multiple times')
)
```

```
47a533e6b83934f58c976de5f2b2dc5a
```

- Getting values

  We can retrieve the value of variables using *get_value()*. In this example, importing the "English" module gives a readable name for the variable containing the operating system name.

```
taco.import_module('English')
print(
    taco.get_value('$OSNAME')
)
```

```
linux
```

- Setting values

  set_value() can be used to assign a variable on the server side. In the case of Perl, setting the output field separator variable `$,` will configure the spacing between things which are printed out.

```
taco.set_value('$,', '**')
```

At this stage we can make use of some object-oriented code to check that the setting of `$,` has taken effect. For more information about using Taco with object-oriented Perl modules, see the object-oriented Perl example. Here we print the strings `'X'`, `'Y'` and `'Z'` to an IO::String object and check the result.

```
taco.import_module('IO::String')
s = taco.construct_object('IO::String')
s.call_method('print', 'X', 'Y', 'Z')
s.call_method('pos', 0)
print(s.call_method('getline'))
```

```
X**Y**Z
```

1.3.2 Object-Oriented Perl: Astro::Coords

This example demonstrates interaction with object-oriented Perl libraries via Taco. The libraries Astro::Coords, Astro::Telescope and DateTime are used to perform some calculations related to the position of Mars over Jodrell Bank during the Queen's speech on Christmas Day, 2010.

- Construct DateTime object

  First we set up a *Taco* object running the default Taco server script implementation for Perl. The server is then instructed to load the DateTime module and to construct an instance of that class. A set of Python keyword arguments is given to the constructor and will be turned into a flattened list of keywords and values as required by the DateTime constructor. Finally the DateTime object's `strftime` method is called to allow us to check that the date has been set correctly.

```
from taco import Taco
taco = Taco(lang='perl')

taco.import_module('DateTime')
qs = taco.construct_object('DateTime', year=2010, month=12, day=25,
                           hour=15, minute=0, second=0)
print(
    qs.call_method('strftime', '%Y-%m-%d %H:%M:%S')
)
```

```
2010-12-25 15:00:00
```

Note: The actual DateTime object will be stored in an object cache on the server side. The *TacoObject* simply refers to it by an object number. When the *TacoObject*'s `__del__` method is called, a `destroy_object` action will be sent, allowing the object to be cleared from the cache.

- Construct Astro::Coords object for Mars

  Next we import the Astro::Coords module and construct an object representing the coordinates of Mars. Since we may want to construct several similar objects, we use the *constructor()* convenience method to get a callable which we can use to call the class constructor.

```
taco.import_module('Astro::Coords')
coords = taco.constructor('Astro::Coords')
mars = coords(planet='mars')
print(
    mars.call_method('name')
)
```

```
mars
```

- Construct Astro::Telescope object and apply it to Mars

  The Astro::Telescope class offers information about a number of telescopes. It has a class method which can be used to fetch a list of supported telescope identifiers. This Perl method needs to be called in list context so we specify `context='list'` in the method call. If you come across a function or method which requires a keyword argument called `context`, this facility can be disabled by setting the `disable_context` attribute of the *Taco* object, for example by specifying `disable_context=True` in its constructor.

```
taco.import_module('Astro::Telescope')
telescopes = taco.call_class_method('Astro::Telescope',
                                    'telNames', context='list')
print('JODRELL1' in telescopes)
```

```
True
```

  Now that we have confirmed that the Perl module knows about Jodrell Bank, we can set this as the location in our object representing Mars. The Python positional argument `'JODRELL1'` to the *construct_object()* method is passed to the Perl constructor at the start of its list of arguments.

  In this example, *construct_object()* will return a *TacoObject*, but when this is passed to another Taco method — in this case *call_method()* — it will automatically be converted to a reference to the object in the cache on the server side.

  We also need to set the date and time, which we can do by calling the Astro::Coords object's `datetime` method. However as we will want to be able to repeat this easily, we can use the convenience routine *method()* to get a Python callable for it. This can then be called with the object representing the date and time of the Queen's speech, which is again automatically converted to a reference to the corresponding Perl object.

  Finally we can have our Astro::Coords object calculate the elevation of Mars for this time and place.

```
mars.call_method('telescope',
                 taco.construct_object('Astro::Telescope', 'JODRELL1'))
datetime = mars.method('datetime')
datetime(qs)
print('{0:.1f}'.format(
    mars.call_method('el', format='degrees')
))
```

```
8.2
```

- Investigate the transit time

  So Mars was above the horizon (positive elevation), but it was still pretty low in the sky. We can have Astro::Coords determine the transit time — the time at which it was highest. (In this method call, `event=0` requests the nearest transit, either before or after the currently configured time.)

```
mt = mars.call_method('meridian_time', event=0)
print(
    type(mt)
)
print(
    mt.call_method('strftime','%H:%M')
)
```

```
<class 'taco.object.TacoObject'>
12:52
```

Note: The Perl `meridian_time` method has returned an object, which is now being referred to by a *`TacoObject`* instance. Taco handles objects returned from functions and methods in the same way as objects explicitly constructed with *`construct_object()`*.

We can now set the Mars object's time to the meridian time using our convenience callable, and find the corresponding elevation.

```
datetime(mt)
print('{0:.1f}'.format(
    mars.call_method('el', format='degrees')
))
```

```
13.0
```

- Check the distance to the Sun

  As a final example, we will calculate the distance (across the sky) between Mars and the Sun. First we construct an object representing the Sun's position.

```
sun = coords(planet='sun')
print(
    sun.call_method('name')
)
```

```
sun
```

Then, after setting the Sun object to the same time, we can request the distance between the two objects. Astro::Coords returns the distance as another object, but we can call its `degrees` method to obtain a value in degrees.

```
sun.call_method('datetime', mt)
print('{0:.1f}'.format(
    mars.call_method('distance', sun).call_method('degrees')
))
```

```
9.9
```

1.4 Client API

1.4.1 taco

The `taco` module imports the *`Taco`* class from the *`taco.client`* module, allowing it to be imported as follows:

```
from taco import Taco
```

1.4.2 taco.client

class `taco.client.`**`Taco`**(*lang=None*, *script=None*, *disable_context=False*)

Taco client class.

Example:

```
from taco import Taco

taco = Taco(lang='python')

taco.import_module('time', 'sleep')
taco.call_function('sleep', 5)
```

call_class_method(*class_*, *name*, **args*, ***kwargs*)
Invoke a class method call in the connected server.

The context (void / scalar / list) can be specified as a keyword argument "context" unless the "disable_context" attribute has been set.

call_function(*name*, **args*, ***kwargs*)
Invoke a function call in the connected server.

The context (void / scalar / list) can be specified as a keyword argument "context" unless the "disable_context" attribute has been set.

construct_object(*class_*, **args*, ***kwargs*)
Invoke an object constructor.

If successful, this should return a *TacoObject* instance which references the new object. The given arguments and keyword arguments are passed to the object constructor.

get_class_attribute(*class_*, *name*)
Request the value of a class (static) attribute.

get_value(*name*)
Request the value of the given variable.

import_module(*name*, **args*, ***kwargs*)
Instruct the server to load the specified module.

The interpretation of the arguments depends on the language of the Taco server implementation.

set_class_attribute(*class_*, *name*, *value*)
Set the value of a class (static) attribute.

set_value(*name*, *value*)
Set the value of the given variable.

function(*name*)
Convience method giving a function which calls call_function.

This example is equivalent to that given for this class:

```
sleep = taco.function('sleep')
sleep(5)
```

constructor(*class_*)
Convience method giving a function which calls construct_object.

For example constructing multiple datetime objects:

```
taco.import_module('datetime', 'datetime')
afd = taco.construct_object('datetime', 2000, 4, 1)
```

Could be done more easily:

```
datetime = taco.constructor('datetime')
afd = datetime(2000, 4, 1)
```

1.4.3 taco.object

class taco.object.**TacoObject**(*client*, *number*)
Taco object class.

This class is used to represent objects by Taco actions. Instances of this class will returned by methods of Taco objects and should not normally be constructed explicitly.

The objects reside on the server side and are referred to by instances of this class by their object number. When these instances are destroyed the destroy_object action is sent automatically.

call_method(**args*, ***kwargs*)
Invoke the given method on the object.

The first argument is the method name.

The context (void / scalar / list) can be specified as a keyword argument "context" unless the "disable_context" attribute of the client has been set.

get_attribute(**args*, ***kwargs*)
Retrieve the value of the given attribute.

set_attribute(**args*, ***kwargs*)
Set the value of the given attribute.

method(*name*)
Convenience method giving a function which calls a method.

Returns a function which can be used to invoke a method on the server object. For example:

```
strftime = afd.method('strftime')
print(strftime('%Y-%m-%d'))
```

1.4.4 taco.error

exception taco.error.**TacoError**
Base class for specific Taco client exceptions.

Note that the client can also raise general exceptions, such as ValueError, if its methods are called with invalid arguments.

exception taco.error.**TacoReceivedError**
An exception raised by the Taco client.

Raised if the client receives an exception action. The exception message will contain any message text received in the exception action.

exception taco.error.**TacoUnknownActionError**
An exception raised by the Taco client.

Raised if the client receives an action of an unknown type.

1.5 Internal API

1.5.1 Server

taco.server

class taco.server.**TacoServer**
Taco server class.

This class implements a Taco server for Python.

run()
Main server function.

Enters a message handling loop. The loop exits on failure to read another message.

call_class_method(*message*)
Call the class method specified in the message.

The context, if present in the message, is ignored.

call_function(*message*)
Call the function specified in the message.

The context, if present in the message, is ignored.

call_method(*message*)
Call an object method.

Works similarly to call_function.

construct_object(*message*)
Call an object constructor.

Works similarly to call_function.

destroy_object(*message*)
Remove an object from the objects dictionary.

get_attribute(*message*)
Get an attribute value from an object.

get_class_attribute(*message*)
Get a static attribute from a class.

get_value(*message*)
Get the value of a variable.

If the variable name contains "."-separated components, then it is looked up using the _find_attr function.

import_module(*message*)
Import a module or names from a module.

Without arguments, the module is imported and the top level package name is inserted into the "ns" dictionary.

With "args" specified, it is used as a list of names to import from the module, and those names are inserted into the "ns" dictionary.

Currently any "kwargs" in the message are ignored.

set_attribute(*message*)
Set an attribute value of an object.

set_class_attribute(*message*)
Set a static attribute from a class.

set_value(*message*)
Set the value of a variable.

If the variable name contains "."-separated components, then it is looked up using the _find_attr function.

1.5.2 Transport

taco.transport

class taco.transport.**TacoTransport**(*in_*, *out*, *from_obj=None*, *to_obj=None*)
Taco transport class.

Implements the communication between Taco clients and servers.

read()
Read a message from the input stream.

The decoded message is returned as a data structure, or None is returned if nothing was read.

write(*message*)
Write a message to the output stream.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

t

C

D

F

G

I

M

R

S

T

W