PostMonkey Documentation

Release 1.0b

Eric Rasmussen

November 10, 2016

Contents

PostMonkey is a simple Python (2.6+) wrapper for MailChimp's API version 1.3.

CHAPTER 1

Features

1. 100% test coverage
2. Connection handling via the excellent Requests library
3. Configurable timeout
4. Simple *Exceptions*

CHAPTER 2

Narrative Documentation

2.1 Basics

2.1.1 Overview

Once you create a *PostMonkey* instance with your MailChimp API key, you can use it to call MailChimp's API methods directly:

```
from postmonkey import PostMonkey
pm = PostMonkey('your_api_key')
pm.ping() # returns u"Everything's Chimpy!"
```

If the MailChimp method call accepts parameters, you can supply them in the form of keyword arguments. See *Examples* for common use cases, and refer to the MailChimp API v1.3 official documentation for a complete list of method calls, parameters, and response objects.

MailChimp has established guidelines/limits for API usage, so please refer to their FAQ for information.

Note: it is the caller's responsibility to supply valid method names and any required parameters. If MailChimp receives an invalid request, *PostMonkey* will raise a *postmonkey.exceptions.MailChimpException* containing the error code and message. See MailChimp API v1.3 - Exceptions for additional details.

2.1.2 Examples

Create a new *PostMonkey* instance with a 10 second timeout for requests:

```
from postmonkey import PostMonkey
pm = PostMonkey('your_api_key', timeout=10)
```

Get the IDs for your campaign lists:

```
lists = pm.lists()

# print the ID and name of each list
for list in lists['data']:
    print list['id'], list['name']
```

Subscribe "emailaddress" to list ID 5:

```
pm.listSubscribe(id=5, email_address="emailaddress")
```

Catch an exception returned by MailChimp (invalid list ID):

```
from postmonkey import MailChimpException
try:
    pm.listSubscribe(id=42, email_address="emailaddress")
except MailChimpException, e:
    print e.code  # 200
    print e.error # u'Invalid MailChimp List ID: 42'
```

Get campaign data for all "sent" campaigns:

```
campaigns = pm.campaigns(filters=[{'status': 'sent'}])

# print the name and count of emails sent for each campaign
for c in campaigns['data']:
    print c['title'], c['emails_sent']
```

2.2 Reading MailChimp's API

2.2.1 Parameters and Types

So let's be honest: these *PostMonkey* docs are lazy. You will find examples, but you won't find the complete API documented anywhere here. The main reason is MailChimp's API docs should be the one source of truth. Ultimately, no matter how much effort we put into keeping these docs up to date, they won't be a substitute for reading the actual API docs.

The problem is the actual API docs can be hard to read. Let's define some terms and work through a few examples so you can understand how they correspond to using *PostMonkey*.

A typical *PostMonkey* call looks like this:

```
postmonkey_instance.mailchimpMethodName(param_1=value_1,
                                        param_2=value_2,
                                        param_n=value_n)
```

The keyword arguments you supply to the method are automatically translated into a javascript object in the form:

```
{ 'param_1': value_1, 'param_2': value_2, 'param_3': value_3 }
```

Where each of the values is rendered into an appropriate JSON representation by python's json module.

This is all pretty standard so far, but the catch is that MailChimp doesn't document their API solely for JSON. The parameters they document use their own internal types that are (relatively) neutral for the different formats they support, but it's not always immediately obvious how they translate to json and back to python.

Here's a pretty table that should help:

MailChimp	Python
string	string
int	int
array	dict or list
boolean	bool

The only surprising one there is that when MailChimp documents a parameter as an array, they might mean you need a dict and they might mean you need a list. If the docs were only for JSON, we could hope they would distinguish between objects and arrays. Instead, their arrays appear to be the same as PHP arrays, which can mean one of two things for our purposes:

- an associative array mapping keys to values

- an indexed array of values (where the keys are implicit indices beginning at 0)

Let's look at a couple examples. Here are the documented parameters for the *listSubscribe* method:

```
listSubscribe(string apikey,
              string id,
              string email_address,
              array merge_vars,
              string email_type,
              bool double_optin,
              bool update_existing,
              bool replace_interests,
              bool send_welcome)
```

At first glance, the *merge_vars* parameter appears ambiguous now that we know it could be represented in python as a dict or a list. However, if you view the table with detailed descriptions on the documentation for listSubscribe, you'll see another listing of parameters and a description of fields that have names. This indicates that you need to supply an *associative array*, meaning a json object, meaning a python dict.

You can see an example in *merge_vars*.

Now let's look at *listBatchUnsubscribe*:

```
listBatchUnsubscribe(string apikey,
                     string id,
                     array emails,
                     boolean delete_member,
                     boolean send_goodbye,
                     boolean send_notify)
```

When you read the documentation for listBatchUnsubscribe and check the parameter descriptions, the "emails" parameter doesn't mention anything about names or keys, only an "array of email addresses". This is our clue that we can supply a python list like this:

```
postmonkey_instance.listBatchUnsubscribe(id='<your_list_id>',
                                         emails=['email_1', 'email_2'])
```

2.2.2 merge_vars in Depth

MailChimp allows you to store extra fields on subscriber accounts so you can personalize campaigns. The simplest example is having your campaign begin with:

*Hello, *|FNAME|*!*

Where *|FNAME|* will be replaced with the value of each user's FNAME merge variable.

The MailChimp API docs declare these *merge_vars* to be of type "array", but as we just discovered, that can mean different things depending on the context. In this case, they are defineitely referring to an associative array, which can be easily represented as a python dict.

Here's an example of calling *listSubscribe* with merge_vars:

```
pm.listSubscribe(id='<your_list_id>',
                 email_address='<subscriber_email>',
                 merge_vars={'FNAME': 'Mail', 'LNAME': 'Chimp'})
```

2.3 API Reference

2.3.1 PostMonkey

class postmonkey.**PostMonkey**(*apikey=''*, *endpoint=None*, *datacenter=None*, *timeout=None*, ***params*)

apikey

The API key for your MailChimp account, obtained from MailChimp.

endpoint

The URL used for API calls. Will be inferred from your API key unless you specifically override it (not recommended except for testing).

datacenter

The MailChimp supplied data center for your account. Will be inferred from your API key unless you specifically override it. Cannot be accessed or modified after initialization.

params

Any extra keyword arguments supplied on initialization will be made available in a dict via this attribute. Only include parameters that should be used on each and every API call. For instance, if you want to add a subscription form to your website, you can parameterize the list's ID.

postrequest

Defaults to `requests.post`, and should not be changed except for testing or if you have a really good reason. If you do override it, you must supply a function that takes a URL, a JSON encoded payload, a dict of HTTP headers, and optionally a timeout (in seconds). Must return a response object with a `text` attribute containing valid JSON.

postmonkey.**postmonkey_from_settings**(*settings*)

Factory method that takes a dict, finds keys prefixed with 'postmonkey.', and uses them to create a new `PostMonkey` instance. Intended for use with console scripts or web frameworks that load config file data into a dict.

2.3.2 Exceptions

exception postmonkey.exceptions.**DeserializationError**(*obj*)

Raised if MailChimp responds with anything other than valid JSON. *PostMonkey* uses MailChimp's JSON API exclusively so it is unlikely that this will be raised. The response that caused the error is made available via the `obj` attribute.

exception postmonkey.exceptions.**MailChimpException**(*code*, *error*)

If MailChimp returns an exception code as part of their response, this exception will be raised. Contains the unmodified `code` (int) and `error` (unicode) attributes returned by MailChimp.

exception postmonkey.exceptions.**PostRequestError**(*exc*)

If any exception is made during the POST request to MailChimp's server, *PostRequestError* will be raised. It wraps the underlying exception object and makes it available via the `exc` attribute.

exception postmonkey.exceptions.**SerializationError**(*obj*)

Raised if a method call contains parameters that cannot be serialized to JSON. The object that caused the error is made available via the `obj` attribute.

2.4 Contributing

2.4.1 Feature Additions/Requests

I'm very interested in discussing use cases that *PostMonkey* doesn't cover.

If you have an idea you want to discuss further, ping me (erasmas) on freenode, or feel free to open an issue/submit a pull request (tests included, please!).

If you would like to issue a pull request, I also ask that you make the request from a new feature.<your feature> branch so that I can spend some time testing the code before merging to master.

2.4.2 Notes on Testing

The test suite is written in a way that may be unusual to some, so if you submit a patch I only ask that you follow the testing methodology employed here. On a technical level it boils down to:

1. Parameterizing classes or functions that connect to outside systems
2. In tests, supplying dummy instances of those classes

In general for unit tests I prefer explicit dummies to mocking.

CHAPTER 3

Support and Documentation

The official documentation is available at http://postmonkey.readthedocs.org.

You can report bugs or open support requests in the github issue tracker, or you can discuss issues with me (erasmas) on irc.freenode.org. You can typically find me in #pyramid.

CHAPTER 4

Authors

Eric Rasmussen is the primary author. When he's not busy reinventing the wheel he occasionally blogs on Chromatic Leaves.

CHAPTER 5

License

PostMonkey is available under a FreeBSD-derived license. See LICENSE.txt for details.

p

D

M

P

S