pelecanus Documentation

Release 0.4.1

Erik Aker

Sep 28, 2017

Contents

Contents:

CHAPTER 1

pelecanus

A Python3 application for navigating and editing nested JSON, named 'pelecanus' after Pelecanus occidentalis, the brown Pelican of California and the Eastern Pacific, which is a wonderful bird, but also named such because I got tired of writing "NestedJson".

This application has been built-for and tested on Python3.3 and Python3.4.

Project Goals

Often, it's necessary to explore a JSON object without knowing precisely where things are (in the case of Hypermedia, for example). By creating a recursive data structure, we can facilitate such tasks as retrieving key-value pairs, iterating through the data structure, and searching for elements in the data structure.

How to Use

To install for Python3.3+, simply do:

```
$ pip install pelecanus
```

pelecanus offers *PelicanJson* objects, which are nested dictionaries created from valid JSON objects. *PelicanJson* objects provide a few methods to make it easier to navigate and edit nested JSON objects.

To create a PelicanJson object, you can pass the constructor a Python dictionary created from a JSON dump (or a simple Python dictionary that could be a valid JSON object):

```
>>> content = {'links': {'alternate': [{'href': 'somelink'}]}}
>>> from pelecanus import PelicanJson
>>> pelican = PelicanJson(content)
```

Enumerate

Once you have a *PelicanJson* object, probably one of the most useful things to do is to find all the nested paths and the values located at those paths. The *enumerate* method has been provided for this purpose:

```
>>> for item in pelican.enumerate():
...   print(item)
(['links', 'alternate', 0, 'href'], 'somelink')
...
```

In JSON, only strings may be used as keys [(see JSON spec)](http://json.org/), so the integers that appear in the nested path represent list indices. In this case, *['links', alternate', 0, 'href']* actually represents:

```
1. A dictionary with a key `links`, which points to...
2. Another dictionary which contains a key 'alternate', which contains...
3. A list, the first item of which...
4. Is a dictionary containing the key `href`.
```

enumerate, like most methods in a *PelicanJson* object, returns a generator. If you want just the paths and not their associated values, use the *paths* method:

```
>>> for item in pelican.paths():
...   print(item)
['links', 'alternate', 0, 'href']
```

Getting and Setting Values

You can retrieve the value from a nested path using *get_nested_value*:

```
>>> pelican.get_nested_value(['links', 'alternate', 0, 'href'])
'somelink'
```

If you want to change a nested value, you can use the *set_nested_value* method:

```
>>> pelican.set_nested_value(['links', 'alternate', 0, 'href'], 'newvalue')
>>> pelican.get_nested_value(['links', 'alternate', 0, 'href'])
'newvalue'
```

If you attempt to set a nested value for a path that does not exist, an exception will be raised:

```
>>> pelican.set_nested_value(['links', 'BADKEY'], 'newvalue')
Traceback (most recent call last):
...
KeyError: 'BADKEY'
```

However, you can create a new path and set it equal to a new value if you pass in *force=True* when you call *set_nested_value*:

```
>>> pelican.set_nested_value(['links', 'BADKEY'], 'newvalue', force=True)
>>> pelican.get_nested_value(['links', 'BADKEY'])
'newvalue'
```

Because integers will *always* be interpreted as list-indices, this works for creating ad-hoc lists or adding elements to lists, but be advised: when setting a new path with *force=True*, a *PelicanJson* object will back-fill any missing list indices with *None* (simliar to assigning to a non-existent array index in Ruby):

```
>>> new_path = ['links', 'NewKey', 4, 'NewNestedKey']
>>> pelican.set_nested_value(new_path, 'LIST Example', force=True)
>>> pelican.get_nested_value(new_path)
'LIST EXAMPLE'
>>> pelican.get_nested_value(['links', 'NewKey'])
[None, None, None, None, {'NestedKey': 'LIST EXAMPLE'}]
```

In this example, the *PelicanJson* object found the integer and realized this must be a list index. However, the list was missing, so it created the list and then created all of the items at indices *before* the missing index, at which point it inserted the missing item, a new object with the key-value pair of *NewNestedKey* and *LIST EXAMPLE*. If unexpected, this behavior could be kind of annoying, but the goal is to *force* the path into existence and expected path is now present.

Keys, Values, Items, etc.

A *PelicanJson* object is a modified version of a Python dictionary, so you can use all of the normal dictionary methods, but it will mostly return nested results (which means you will often get duplicate *keys*). The length of the object too will be based on all the nested keys present:

```
>>> list(pelican.keys())
['links', 'attributes', 'href']
>>> len(pelican)
3
```

values is only going to return values that exist at endpoints, which are the inside-most points of all nested objects, leaves in the tree, in other words:

```
>>> list(pelican.values())
['somelink']
```

While *items* attempts to do double-duty, returning each key in the tree and its corresponding value:

```
>>> list(pelican.items())
[('links', <PelicanJson: {'attributes': [<PelicanJson: {'href': 'somelink'}>]}>), (
↪'attributes', [<PelicanJson: {'href': 'somelink'}>]), ('href', 'somelink')]
```

You can also use *in* to see if a key is somewhere inside the dictionary (even if it's a nested key):

```
>>> 'attributes' in pelican
True
```

Turning it back into a plain dictionary or JSON

Other useful methods include *convert* and *serialize* for turning the object back into a plain Python dictionary and for returning a JSON dump, respectively:

```
>>> pelican.convert() == content
True
>>> pelican.serialize()
'{"links": {"attributes": [{"href": "somelink"}]}}'
>>> import json
>>> json.loads(pelican.serialize()) == content
True
```

Searching Keys and Values

You can also use the methods *search_key* and *search_value* in order to find all the paths that lead to keys or values you are searching for (data comes from the Open Library API):

```
>>> book = {'ISBN:9780804720687': {'preview': 'noview', 'bib_key': 'ISBN:9780804720687
↪', 'preview_url': 'https://openlibrary.org/books/OL7928788M/Between_Pacific_Tides',
↪'info_url': 'https://openlibrary.org/books/OL7928788M/Between_Pacific_Tides',
↪'thumbnail_url': 'https://covers.openlibrary.org/b/id/577352-S.jpg'}}
>>> pelican = PelicanJson(book)
>>> for path in pelican.search_key('preview'):
...   print(path)
['ISBN:9780804720687', 'preview']
>>> for path in pelican.search_value('https://covers.openlibrary.org/b/id/577352-S.jpg
↪'):
...  print(path)
['ISBN:9780804720687', 'thumbnail_url']
```

In addition, *pluck* is for retrieving the whole object that contains a particular key-value pair:

```
>>> list(pelican.pluck('preview', 'noview'))
[<PelicanJson: {'preview': 'noview', 'thumbnail_url': 'https://covers.openlibrary.org/
↪b/id/577352-S.jpg', 'bib_key': 'ISBN:9780804720687', 'preview_url': 'https://
↪openlibrary.org/books/OL7928788M/Between_Pacific_Tides', 'info_url': 'https://
↪openlibrary.org/books/OL7928788M/Between_Pacific_Tides'}>]
```

Find and Replace

Finally, there is also a *find_and_replace* method which searches for a particular value and replaces it with a passed-in replacement value:

```
>>> for path in pelican.search_value('https://covers.openlibrary.org/b/id/577352-S.jpg
↪'):
...  print(path)
['ISBN:9780804720687', 'thumbnail_url']
>>> pelican.find_and_replace('https://covers.openlibrary.org/b/id/577352-S.jpg',
↪'SOME NEW URL')
>>> pelican.get_nested_value(['ISBN:9780804720687', 'thumbnail_url'])
'SOME NEW URL'
```

This can, of course, be dangerous, so use with caution.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search