gems Documentation

Release 0.3.5

Blake Printy

Dec 03, 2018

Contents

CHAPTER 1

Overview

The gems module provides specialized data structures to augment development. It's similar to the collections module, but contains different types of objects. For documentation on using these objects, see the Usage section of the documentation.

CHAPTER 2

Content:

2.1 Installation

2.1.1 Through pip

```
$ pip install gems
```

2.1.2 Via GitHub

```
$ git clone http://github.com/bprinty/gems.git
$ cd gems
$ python setup.py install
```

2.1.3 Questions/Feedback

File an issue in the GitHub issue tracker.

2.2 Usage

The gems module provides specialized data structures to augment development. It's similar to the collections module, but contains different types of objects.

Currently, the following objects are available (this list will grow with time and feedback):

Name	Description
composite	JSON-like data structure for easy data traversal.
filetree	JSON-like data structure for easy filesystem traversal.

2.2.1 composite

The *gems.composite* object abstracts away the complexity associated with managing heavily nested JSON-based structures, allowing easier access to internal properties, and providing operators that work with the data in an intuitive way. Here is a simple example of how to use the `composite` type in a project:

```
>>> from gems import composite
>>>
>>> data = composite({
>>>     'one': 1,
>>>     'two': [1, 2, 3],
>>>     'three': ['one', 2, {'three': 'four'}],
>>>     'four': {'five': [6, 7, 8], 'nine': 10, 'eleven': 'twelve'}
>>> })
>>> data.four.five[1] == 6
True
>>> data.two[0] == 1
True
```

In the example above, an arbitrary data structure is provided as an argument to the `composite` object, and is transformed into an object where properties can be traversed more gracefully (syntactically). You can also load a composite object from a json or yaml file like so:

```
>>> from gems import composite
>>>
>>> with open('data.json', 'r') as fi:
>>>     data = composite.load(fi)
>>>
>>> print data.four.five[1]
6
>>>
>>> with open('data.yml', 'r') as fi:
>>>     data = composite.load(fi)
>>>
>>> print data.four.five[1]
6
```

Some of the main features of `composite` objects that make them particularly useful are operators for interacting with the structure. For instance, if two composite objects or a composite object and another similar type are added, you get a `composite` object as a result that combines the objects in an intuitive way:

```
>>> # using the 'data' object from above
>>> obj = data + {'five': 6}
>>> obj.five == 6
True
>>> obj.two === [1, 2, 3]
True

>>> obj = data + [1, 2, 3]
>>> obj[0].one.two[0] == 1
True
>>> obj[1][1] == 2
True

>>> data2 = composite([
    1, 2, 3, {'four': 5}
])
```

(continues on next page)

(continued from previous page)

```
>>> obj = data2 + {'five': 6}
>>> obj[0][0] == 1
True
>>> obj[0][2].four == 5
True
>>> obj = data2 + ['seven', 8, 9]
>>> obj[4:6] == ['seven', 8]
True
```

Other operations like this also can be used with the `composite` object. For example:

```
>>> # using the 'data' object from above
>>> 'three' in data
True
>>> 7 in data.four.five
True
>>> data.four.five == [6, 7, 8]
True
>>> data == data2
False
```

Along with these operators, `composite` objects also extend set-based functionality for reducing data. For example:

```
>>> # initialize some data
>>> c1 = composite({
>>>     'one': 1,
>>>     'two': [1, 2],
>>>     'three': {'four': 5, 'five': 7},
>>>     'eight': 8
>>> })
>>> c2 = composite({
>>>     'one': 1,
>>>     'two': [1, 2, 3],
>>>     'three': {'four': 5, 'six': 7},
>>>     'eight': 9,
>>>     'nine': 10
>>> })
>>>
>>> # take the recursive intersection of the data structures
>>> print c1.intersection(c2)
{
    'one': 1,
    'two': [1, 2],
    'three': {'four': 5},
}
>>>
>>> # take the recursive difference of the data structures
>>> print c2.difference(c1)
{
    'two': [3],
    'three': {'six': 7},
    'eight': 9,
    'nine': 10
}
>>>
>>> # take the recursive union of the data structures
>>> print c1.union(c2)
```

(continues on next page)

(continued from previous page)

```
{
    'one': 1,
    'two': [1, 2, 3],
    'three': {'four': 5, 'five': 7, 'six': 7},
    'eight': [8, 9],
    'nine': 10
}
```

Finally, you can write composite objects back to JSON files easily:

```
>>> # change the data in the object
>>> data.four.five = 2
>>>
>>> with open('newdata.json', 'w') as nd:
>>>     data.write(nd)
```

By default, this will sort keys and pretty-print to the file, but if you just want to print the raw json to file, use `pretty=False`.

2.2.2 filetree

Traversal of a filetree is typically a pain in python. You could use `os.path.walk` recursively to accomplish it, but there should be an easier way. That's where the *gems.filetree* comes in handy. Here is an example of how to use the *gems.filetree* type in a project:

```
>>> from gems import filetree
>>>
>>> # mydir is a directory with the structure below
>>> ftree = filetree('mydir')
>>> print ftree
mydir/
     one/
        two.txt
        three.json
    two/
        three/
              four.txt
        five six/
                 seven.txt
        eight.config
```

The *gems.filetree* structure also allows for traversal of the file data like so:

```
>>> print data.one['two.txt']
/full/path/to/mydir/one/two.txt
>>>
>>> print data.two.three['four.txt']
/full/path/to/mydir/two/three/four.txt
>>>
>>> print data.two['five six']['eight.config']
/full/path/to/mydir/two/five six/eight.config
```

As you can see in the example above, using JSON-based access is much easier and cleaner than doing many `os.path.join` operations to create the full paths to objects on your filesystem. You can also create a json structure from the filetree:

```
>>> print data.json()
{
    "one": {
        "two.txt": "/path/to/mydir/one/two.txt",
        "three.json": "/path/to/mydir/one/three.json"
    },
    "two": {
        "three": {
            "four.txt": "/path/to/mydir/two/three/four.txt"
        },
        "five six": {
            "seven.txt": "/path/to/mydir/two/five six/seven.txt"
        },
        "eight.config": "/path/to/mydir/two/eight.config"
    }
}
```

Or, if you just want to see a list of all files in the filetree, you can do the following:

```
>>> print data.files()
'/path/to/mydir/one/two.txt'
'/path/to/mydir/one/three.json'
'/path/to/mydir/two/three/four.txt'
'/path/to/mydir/two/five six/seven.txt'
'/path/to/mydir/two/eight.config'
```

Finally, to prune the tree for specific files and create a new filetree object:

```
>>> newtree = data.prune(regex=".*.txt$")
>>> print newtree.files()
'/path/to/mydir/one/two.txt'
'/path/to/mydir/two/three/four.txt'
'/path/to/mydir/two/five six/seven.txt'
```

2.3 API

2.3.1 Data Management

class gems.**composite**(*data*)

Data structure for traversing object relationships via attributes instead of keys and indices.

Parameters **data** (*tuple, list, dict*) – Data to build composite datastructure from.

Example

```
>>> data = composite({
>>>     'one': 1,
>>>     'two': [1, 2, 3],
>>>     'three': ['one', 2, {'three': 'four'}],
>>>     'four': {'five': [6, 7, 8], 'nine': 10, 'eleven': 'twelve'}
>>> })
>>> data.four.five[1] == 6
True
```

(continues on next page)

(continued from previous page)

```
>>> data.two[0] == 1
True
```

append (*item*)
Append to object, if object is list.

difference (*other*, *recursive=True*)
Recursively compute difference of data. For dictionaries, items for specific keys will be reduced to differences. For lists, items will be reduced to differences. This method is meant to be analogous to set.difference for composite objects.

Parameters

- **other** (*composite*) – Other composite object to difference with.
- **recursive** (*bool*) – Whether or not to perform the operation recursively, for all nested composite objects.

extend (*item*)
Extend list from object, if object is list.

classmethod from_json (*fh*)
Load json from file handle.

Parameters **fh** (*file*) – File handle to load from.

Examlple:

```
>>> with open('data.json', 'r') as json:
>>>     data = composite.load(json)
```

classmethod from_string (*string*)
Load data from string.

Parameters **string** (*str*) – String to load from.

Examlple:

```
>>> with open('data.json', 'r') as json:
>>>     jdat = json.read()
>>> data = composite.from_string(jdat)
```

classmethod from_yaml (*fh*)
Load yaml from file handle.

Parameters **fh** (*file*) – File handle to load from.

Examlple:

```
>>> with open('data.yml', 'r') as json:
>>>     data = composite.load(json)
```

get (**args*, ***kwargs*)
Return item or None, depending on if item exists. This is meant to be similar to dict.get() for safe access of a property.

index (*item*)
Return index containing value.

intersection(*other*, *recursive=True*)
Recursively compute intersection of data. For dictionaries, items for specific keys will be reduced to unique items. For lists, items will be reduced to unique items. This method is meant to be analogous to set.intersection for composite objects.

Parameters

- **other** (*composite*) – Other composite object to intersect with.
- **recursive** (*bool*) – Whether or not to perform the operation recursively, for all nested composite objects.

items()
Return keys for object, if they are available.

json()
Return JSON representation of object.

keys()
Return keys for object, if they are available.

classmethod load(*fh*)
Load json or yaml data from file handle.

Parameters **fh** (*file*) – File handle to load from.

Examlple:

```
>>> with open('data.json', 'r') as json:
>>>     jsdata = composite.load(json)
>>>
>>> with open('data.yml', 'r') as yml:
>>>     ymldata = composite.load(yml)
```

pop(**args*, ***kwargs*)
Return item or None, depending on if item exists. This is meant to be similar to dict.pop() for safe access of a property.

union(*other*, *recursive=True*, *overwrite=False*)
Recursively compute union of data. For dictionaries, items for specific keys will be combined into a list, depending on the status of the overwrite= parameter. For lists, items will be appended and reduced to unique items. This method is meant to be analogous to set.union for composite objects.

Parameters

- **other** (*composite*) – Other composite object to union with.
- **recursive** (*bool*) – Whether or not to perform the operation recursively, for all nested composite objects.
- **overwrite** (*bool*) – Whether or not to overwrite entries with the same key in a nested dictionary.

update(*other*)
Update internal dictionary object. This is meant to be an analog for dict.update().

values()
Return keys for object, if they are available.

write(*fh*, *pretty=True*)
API niceness defaulting to composite.write_json().

write_json (*fh*, *pretty=True*)
Write composite object to file handle in JSON format.

Parameters

- **fh** (*file*) – File handle to write to.
- **pretty** (*bool*) – Sort keys and indent in output.

write_yaml (*fh*)
Write composite object to file handle in YAML format.

Parameters **fh** (*file*) – File handle to write to.

2.3.2 Filesystem Management

class gems.**filetree** (*directory*, *ignore='^[._]'*, *regex='.*'*)
Data structure for traversing directory structure and creating object for accessing relative file paths.

Parameters

- **directory** (*str*) – Directory to build filetree from.
- **ignore** (*str*) – Regular expression with items to ignore. If you wish to recurse through all directories (including hidden directories), set ignore=None. By default, this is set to "^[._]" (i.e. any files beginning with "." or "_").

Example

```
>>> data = filetree('mydir')
>>> print data
mydir/
     one/
        two.txt
        three.json
    two/
        three/
              four.txt
        five six/
                 seven.txt
        eight.config
>>> print data.one['two.txt']
/full/path/to/mydir/one/two.txt
>>> print data.two.three['four.txt']
/full/path/to/mydir/two/three/four.txt
>>> print data.two['five six']['eight.config']
/full/path/to/mydir/two/five six/eight.config
```

__str__()

Note: This needs to be completed – print filetree

filelist()
Return list of files in filetree.

get(*item*)
Safe way to get items, similar to __dict__.get().

Parameters **item** (*str*) – Item to get in file tree.

json()
Return JSON representation of object.

prune(*regex='.*'*)
Prune leaves of filetree according to specified regular expression.

Parameters **regex** (*str*) – Regular expression to use in pruning tree.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index