Sections

Release 0.0.3

Trevor Edwin Pogue

Feb 27, 2022

CONTENTS

CHAPTER

ONE

[S E | C T | I O | N S]

Python package providing flexible tree data structures for organizing lists and dicts into sections.

Sections is designed to be:

- **Intuitive**: Start quickly, spend less time reading the docs.
- **Scalable**: Grow arbitrarily complex trees as your problem scales.
- **Flexible**: Rapidly build nodes with custom attributes, properties, and methods on the fly.
- **Fast**: Made with performance in mind - access lists and sub-lists/dicts in (1) time in many cases. See the Performance section for the full details.
- **Reliable**: Contains an exhaustive test suite and 100% code coverage.

1.1 Links

- GitHub
- Documentation

CHAPTER

TWO

USAGE

```
$ pip install sections
```

```
import sections

menu = sections(
    'Breakfast', 'Dinner',
    main=['Bacon&Eggs', 'Burger'],
    side=['HashBrown', 'Fries'],
)
```

```
$ print(menu)
 ___________________________
|  ______________________   |
| | 'Breakfast'          |  |
| | main = 'Bacon&Eggs'  |  |
| | side = 'HashBrown'   |  |
|  ----------------------   |
|                           |
|  __________________       |
| | 'Dinner'         |      |
| | main = 'Burger'  |      |
| | side = 'Fries'   |      |
|  ------------------       |
 ---------------------------
```

```
# menu's API with the expected results:
assert menu.mains == ['Bacon&Eggs', 'Burger']
assert menu.sides == ['HashBrown', 'Fries']
assert menu['Breakfast'].main == 'Bacon&Eggs'
assert menu['Breakfast'].side == 'HashBrown'
assert menu['Dinner'].main == 'Burger'
assert menu['Dinner'].side == 'Fries'
assert menu('sides', list) == ['HashBrown', 'Fries']
assert menu('sides', dict) == {'Breakfast': 'HashBrown', 'Dinner': 'Fries'}
# root section/node:
assert isinstance(menu, sections.Section)
# child sections/nodes:
assert isinstance(menu['Breakfast'], sections.Section)
assert isinstance(menu['Dinner'], sections.Section)
```

Scale in size:

```
library = sections(
    "My Bookshelf",
    sections({'Fiction'},
             'LOTR', 'Harry Potter',
             author=['JRR Tolkien', 'JK Rowling'],
             topic=[{'Fantasy'}, 'Hobbits', 'Wizards'],),
    sections({'Non-Fiction'},
             'General Relativity', 'A Brief History of Time',
             author=['Albert Einstein', 'Steven Hawking'],
             topic=[{'Physics'}, 'Time, Gravity', 'Black Holes'],
             ),
)
```

```
$ print(library)
 ________________________________________
| 'My Bookshelf'                         |
|   ______________________________       |
|  | 'Fiction'                    |      |
|  | topic = 'Fantasy'            |      |
|  |    ________________________   |      |
|  |   | 'LOTR'                 |  |      |
|  |   | author = 'JRR Tolkien' |  |      |
|  |   | topic  = 'Hobbits'     |  |      |
|  |    ------------------------   |      |
|  |    ________________________   |      |
|  |   | 'Harry Potter'         |  |      |
|  |   | author = 'JK Rowling'  |  |      |
|  |   | topic  = 'Wizards'     |  |      |
|  |    ------------------------   |      |
|   ------------------------------       |
|   ___________________________________    |
|  | 'Non-Fiction'                    |   |
|  | topic = 'Physics'                |   |
|  |    ____________________________   |   |
|  |   | 'General Relativity'       |  |   |
|  |   | author = 'Albert Einstein' |  |   |
|  |   | topic  = 'Time, Gravity'   |  |   |
|  |    ----------------------------   |   |
|  |    ____________________________   |   |
|  |   | 'A Brief History of Time'  |  |   |
|  |   | author = 'Steven Hawking'  |  |   |
|  |   | topic  = 'Black Holes'     |  |   |
|  |    ----------------------------   |   |
|   ----------------------------------    |
 ----------------------------------------
```

2.1 Attrs: Plural/singular hybrid attributes and more

Spend less time deciding between using the singular or plural form for an attribute name:

```
tasks = sections('pay bill', 'clean', status=['completed', 'started'])
assert tasks.statuses == ['completed', 'started']
assert tasks['pay bill'].status == 'completed'
assert tasks['clean'].status == 'started'
```

If you don't like this feature, simply turn it off as shown in the **Details - Plural/singular attributes** section.

2.2 Properties: Easily add on the fly

Properties and methods are automatically added to all nodes in a structure returned from a `sections()` call when passed as keyword arguments:

```
schedule = sections(
    'Weekdays', 'Weekend',
    hours_per_day=[[8, 8, 6, 10, 8], [4, 6]],
    hours=property(lambda self: sum(self.hours_per_day)),
)
assert schedule['Weekdays'].hours == 40
assert schedule['Weekend'].hours == 10
assert schedule.hours == 50
```

Adding properties and methods this way doesn't affect the class definitions of Sections/nodes from other structures. See the **Details - Properties/methods** section for how this works.

2.3 Construction: Build gradually or all at once

Construct section-by-section, section-wise, attribute-wise, or other ways:

```
def demo_different_construction_techniques():
    """Example construction techniques for producing the same structure."""
    # Building section-by-section
    books = sections()
    books['LOTR'] = sections(topic='Hobbits', author='JRR Tolkien')
    books['Harry Potter'] = sections(topic='Wizards', author='JK Rowling')
    demo_resulting_object_api(books)

    # Section-wise construction
    books = sections(
        sections('LOTR', topic='Hobbits', author='JRR Tolkien'),
        sections('Harry Potter', topic='Wizards', author='JK Rowling')
    )
    demo_resulting_object_api(books)

    # Attribute-wise construction
    books = sections(
        'LOTR', 'Harry Potter',
```

(continues on next page)

(continued from previous page)

```
        topics=['Hobbits', 'Wizards'],
        authors=['JRR Tolkien', 'JK Rowling']
    )
    demo_resulting_object_api(books)

    # setattr post-construction
    books = sections(
        'LOTR', 'Harry Potter',
    )
    books.topics = ['Hobbits', 'Wizards']
    books['LOTR'].author = 'JRR Tolkien'
    books['Harry Potter'].author = 'JK Rowling'
    demo_resulting_object_api(books)

def demo_resulting_object_api(books):
    """Example Section structure API and expected results."""
    assert books.names == ['LOTR', 'Harry Potter']
    assert books.topics == ['Hobbits', 'Wizards']
    assert books.authors == ['JRR Tolkien', 'JK Rowling']
    assert books['LOTR'].topic == 'Hobbits'
    assert books['LOTR'].author == 'JRR Tolkien'
    assert books['Harry Potter'].topic == 'Wizards'
    assert books['Harry Potter'].author == 'JK Rowling'

demo_different_construction_techniques()
```

CHAPTER

THREE

DETAILS

3.1 Section names

The non-keyword arguments passed into a `sections()` call define the section names and are accessed through the attribute `name`. The names are used like `keys` in a `dict` to access each child section of the root section node:

```
books = sections(
    'LOTR', 'Harry Potter',
    topic=['Hobbits', 'Wizards'],
    author=['JRR Tolkien', 'JK Rowling']
)
assert books.names == ['LOTR', 'Harry Potter']
assert books['LOTR'].name == 'LOTR'
assert books['Harry Potter'].name == 'Harry Potter'
```

Names are optional, and by default, children names are assigned as integer values corresponding to indices in an array, while a root has a default keyvalue of `sections.SectionNone`:

```
sect = sections(x=['a', 'b'])
assert sect.sections.names == [0, 1]
assert sect.name is sections.SectionNone

# the string representation of sections.SectionNone is 'section':
assert str(sect.name) == 'sections'
```

3.2 Parent names and attributes

A parent section name can optionally be provided as the first argument in a `sections()` call by defining it in a set (surrounding it with curly brackets). This strategy avoids an extra level of braces when instantiating Section objects. This idea applies also for defining parent attributes:

```
library = sections(
    {"My Bookshelf"},
    [{'Fantasy'}, 'LOTR', 'Harry Potter'],
    [{'Academic'}, 'Advanced Mathematics', 'Physics for Engineers'],
    topic=[{'All my books'},
           [{'Imaginary things'}, 'Hobbits', 'Wizards'],
           [{'School'}, 'Numbers', 'Forces']],
)
```

(continues on next page)

(continued from previous page)

```
assert library.name == "My Bookshelf"
assert library.sections.names == ['Fantasy', 'Academic']
assert library['Fantasy'].sections.names == ['LOTR', 'Harry Potter']
assert library['Academic'].sections.names == [
    'Advanced Mathematics', 'Physics for Engineers'
]
assert library['Fantasy']['Harry Potter'].name == 'Harry Potter'
assert library.topic == 'All my books'
assert library['Fantasy'].topic == 'Imaginary things'
assert library['Academic'].topic == 'School'
```

3.3 Return attributes as a list, dict, or iterable

Access the data in different forms with the `gettype` argument in Section.__call__() as follows:

```
menu = sections('Breakfast', 'Dinner', sides=['HashBrown', 'Fries'])

# return as list always, even if a single element is returned
assert menu('sides', list) == ['HashBrown', 'Fries']
assert menu['Breakfast']('side', list) == ['HashBrown']

# return as dict
assert menu('sides', dict) == {'Breakfast': 'HashBrown', 'Dinner': 'Fries'}
assert menu['Breakfast']('side', dict) == {'Breakfast': 'HashBrown'}

# return as iterator over elements in list (fastest method, theoretically)
for i, value in enumerate(menu('sides', iter)):
    assert value == ['HashBrown', 'Fries'][i]
for i, value in enumerate(menu['Breakfast']('side', iter)):
    assert value == ['HashBrown'][i]
```

See the Section.__call__() method in the References section of the docs for more options.

Set the default return type when accessing structure attributes by changing `Section.default_gettype` as follows:

```
menu = sections('Breakfast', 'Dinner', sides=['HashBrown', 'Fries'])

menu['Breakfast'].default_gettype = dict  # set for only 'Breakfast' node
assert menu.sides == ['HashBrown', 'Fries']
assert menu['Breakfast']('side') == {'Breakfast': 'HashBrown'}

menu.cls.default_gettype = dict           # set for all nodes in `menu`
assert menu('sides') == {'Breakfast': 'HashBrown', 'Dinner': 'Fries'}
assert menu['Breakfast']('side') == {'Breakfast': 'HashBrown'}

sections.Section.default_gettype = dict   # set for all structures
tasks1 = sections('pay bill', 'clean', status=['completed', 'started'])
tasks2 = sections('pay bill', 'clean', status=['completed', 'started'])
assert tasks1('statuses') == {'pay bill': 'completed', 'clean': 'started'}
assert tasks2('statuses') == {'pay bill': 'completed', 'clean': 'started'}
```

The above will also work for accessing attributes in the form `object.attr` but only if the node does not contain the attribute `attr`, otherwise it will return the non-iterable raw value for `attr`. Therefore, for consistency, access attributes using Section.__call__() like above if you wish to **always receive an iterable** form of the attributes.

3.4 Plural/singular attributes

When an attribute is not found in a Section node, both the plural and singular forms of the word are then checked to see if the node contains the attribute under those forms of the word. If they are still not found, the node will recursively repeat the same search on each of its children, concatenating the results into a list or dict. The true attribute name in each node supplied a corresponding value is whatever name was given in the keyword argument's key (i.e. `status` in the example below).

If you don't like this feature, simply turn it off using the following:

```
import pytest
tasks = sections('pay bill', 'clean', status=['completed', 'started'])
assert tasks.statuses == ['completed', 'started']
# turn off for all future structures:
sections.Section.use_pluralsingular = False
tasks = sections('pay bill', 'clean', status=['completed', 'started'])
with pytest.raises(AttributeError):
    tasks.statuses  # this now raises an AttributeError
```

Note, however, that this will still traverse descendant nodes to see if they contain the requested attribute. To stop using this feature also, access attributes using the Section.get_node_attr() method instead.

3.5 Properties/methods

Each `sections()` call returns a structure containing nodes of a unique class created in a class factory function, where the unique class definition contains no logic except that it inherits from the Section class. This allows properties/methods added to one structure's class definition to not affect the class definitions of nodes from other structures.

3.6 Subclassing

Inheriting Section is easy, the only requirement is to call `super().__init__(**kwds)` at some point in `__init__()` like below if you override that method:

```
class Library(sections.Section):
    """My library class."""

    def __init__(self, price="Custom default value", **kwds):
        """Pass **kwds to super."""
        super().__init__(**kwds)
        self.price = price

    @property
    def genres(self):
        """A synonym for sections."""
        if self.isroot:
```

(continues on next page)

(continued from previous page)

```
            return self.sections
        else:
            raise AttributeError('This library has only 1 level of genres')

    @property
    def books(self):
        """A synonym for leaves."""
        return self.leaves

    @property
    def titles(self):
        """A synonym for names."""
        return self.leaves.names

    def critique(self, review="Haven't read it yet", rating=0):
        """Set the book price based on the rating."""
        self.review = review
        self.price = rating * 2

library = Library(
    [{'Fantasy'}, 'LOTR', 'Harry Potter'],
    [{'Academic'}, 'Advanced Math.', 'Physics for Engineers']
)
assert library.genres.names == ['Fantasy', 'Academic']
assert library.books.titles == [
    'LOTR', 'Harry Potter', 'Advanced Math.', 'Physics for Engineers'
]
library.books['LOTR'].critique(review='Good but too long', rating=7)
library.books['Harry Potter'].critique(
    review="I don't like owls", rating=4)
assert library.books['LOTR'].review == 'Good but too long'
assert library.books['LOTR'].price == 14
assert library.books['Harry Potter'].review == "I don't like owls"
assert library.books['Harry Potter'].price == 8
import pytest
with pytest.raises(AttributeError):
    library['Fantasy'].genres
```

`Section.__init__()` assigns the kwds values passed to it to the object attributes, and the passed kwds are generated during instantiation by a metaclass.

3.7 Performance

Each non-leaf Section node keeps a cache containing quickly readable references to attribute dicts previously parsed from manually traversing through descendant nodes in an earlier read. The caches are invalidated accordingly for modified nodes and their ancestors when the tree structure or node attribute values change.

The caches allow instant reading of sub-lists/dicts in (1) time and can often make structure attribute reading faster by 5x, or even much more when the structure is rarely being modified. If preferred, turn this feature off to avoid the extra memory consumption it causes by modifying the node or structure's class attribute `use_cache` to `False` as follows:

```
sect = sections(*[[[[[42] * 10] * 10] * 10] * 10] * 10])
sect.use_cache = False              # turn off for just the root node
sect.cls.use_cache = False          # turn off for all nodes in `sect`
sections.Section.use_cache = False  # turn off for all structures
```

CHAPTER

FOUR

REFERENCE

The following describes the available interface with the Section class, the class representing each node object in a sections tree structure.

class sections.**Section**(**args: SectionKeysOrObjects*, *parent: Optional[SectionParent] = None*, ***kwds: SectionAttr*)

Objects instantiated by `Section` are nodes in a sections tree structure. Each node has useful methods and properties for organizing lists/dicts into sections and for conveniently accessing/modifying the sub-list/dicts from each section/subsection.

__call__(*name: str = <sections.types.SectionNoneType object>*, *gettype: GetType = 'default'*, *default: typing.Any = <sections.types.SectionNoneType object>*) → Union[Any, List[Any]]

Run `get_nearest_attr`. This returns attribute *name* from self if self contains the attribute in either the singular or plural form for *name*. Else, try the same pattern for each of self's children, putting the returned results from each child into a list. Else, raise AttributeError.

Parameters

- **name** – The name of the attribute to find in self or self's descendants.
- **gettype** – Valid values are *'default'*, *'hybrid' list*, *iter*, *dict*, *'self'*. Setting to *'default'* uses the value of self.default_gettype for gettype (its default is 'hybrid'). Setting to *'hybrid'* returns a list if more than 1 element is found, else returns the non-iterable raw form of the element. Setting to *list* returns a list containing the attribute values. Setting to *iter* returns an iterable iterating through the attribute values. Setting to *dict* returns a dict containing pairs of the containing node's name with the attribute value. Setting to *'self'* will only search for attrs in self, and will never wrap the attr in an iterable form like the dict/list/iter options.

searches for attributes only in self. Setting to *'nearest'* also searches through

Parameters **default** – If not provided, AttributeError will be raised if attr *name* not found. If given, return default if attr *name* not found.

Returns An iterable or non-iterable form of the attribute *name* formed from self or descendant nodes. Depends on the value given to *gettype*.

__getattr__(*name: str*) → Any

Called if self node does not have attribute *name*, in which case try finding attribute *name* from `__call__`.

__getitem__(*names: Any*) → Section

x.__getitem__(y) <==> x[y]

__iter__() → Iterable[Section]

By default iterate over child nodes instead of their names/keys.

__setattr__(*name: str*, *value: Any*, *_invalidate_cache=True*) → None
If value is a list, recursively setattr for each child node with the corresponding value element from the value list.

__setitem__(*name: Any*, *value: Union[Section, AnyDict]*) → None
Add a child *name* to self. Ensure added children are converted to the same unique Section type as the rest of the nodes in the structure, and update its name to *name*, and its parent to self.

property children: Section
Get self nodes's children. Returns a Section node that has no public attrs and has shallow copies of self node's children as its children. This can be useful if self has an attr *attr* but you want to access a list of the childrens' attr *attr*, then write section.children.attr to access the attr list.

clear() → None. Remove all items from od.

property cls: Type[Section]
The unique structure-wide class of each node.

copy() → a shallow copy of od

property descendants: Section
Similar to *leaves* except all nodes in structure are returned.

property descendants_iter: iter
Return iterator that iterates through self and all self's descendants.

descendants_str() → str
Print the output of `node_str <Section.node_str()` for self and all of its descendants.

Parameters **breadthfirst** – Set True to print descendants in a breadth-first pattern or False for depth-first.

property entries: Section
A synonym for property *leaves*.

property flat: Section
Synonym for *descendants*.

fromkeys(**args: Any*, ***kwds: Any*) → None
Not supported.

get(*name: Any*, *default: Optional[Any] = None*) → None
Return the value for key if key is in the dictionary, else default.

insert(*i: int*, *child: Section*) → None
Insert *child' at index `i* of dict. The key for *child* will be taken from child's *name* attribute. If *i* is negative, insert at end of dict.

insertitem(*i: int*, *name: Any*, *child: Section*) → None
Insert *child' at index `i* of dict. The key for *child* will be taken from child's *name* attribute. If *i* is negative, insert at end of dict.

property ischild: bool
True iff self node has a parent.

property isleaf: bool
True iff self node has no children.

property isparent: bool
True iff self node has any children.

property isroot: bool
True iff self node has no parent.

items() → Tuple[Iterable[Any], Iterable[Any]]
Return iterator over child names and children.

keys() → Iterable[Any]
Return iterator over child names.

property leaves: Section
Get all leaf node descendants of self. Returns a Section node that has no public attrs and has shallow copies of self node's leaves as its children. This can be useful if self has an attr *attr* but you want to access a list of the leaves' attr *attr*, then write section.leaves.attr to access the leaf attr list.

property leaves_iter: iter
Return iterator that iterates through all self's leaf node descendants.

move_to_end(*name: Any*, *last: bool = True*) → None
Move an existing child to either end of ordered children dict.

property node: Section
Return a shallow copy of self with no children. Useful for searching for attributes only in self.

node_str() → str
Neatly print the public attributes of the Section node and its class, as well as its types property output.

node_withchildren_fromiter(*itr: iter*) → Section
Perform a general form of the task performed in *leaves*. Return a Section node with any children referenced in the iterable from the *itr* argument.

property nofchildren: int
Nunber of children Sections/nodes.

pop(*name_or_i: Union[Any, int]*) → Any
Remove child *name_or_i* from self. If there is no child with that name and *name_or_i* is int, remove child in position *name_or_i*.

popitem(*last=True*) → Tuple[Any, Any]
Remove last added child from self.

property sections: Section
A synonym for property *children*.

setdefault(*name: Any*, *default: Section*) → Any
If self has a child *name*, return it. If not, set child *default* with name *name* default and return *default*.

structure_change()
Will be called every time there is a change in structure, i.e. whenever a node is added or removed or rearranged in child order. Meant for use when overriding.

update(*other: Section*) → None
Add all children from *other* to self.

values() → Iterable[Any]
Return iterator over children.

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

Sections could always use more documentation, whether as part of the official Sections docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/trevorpogue/sections/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *sections* for local development:

1. Fork sections (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/sections.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

   Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

CHAPTER

SIX

AUTHORS

- Trevor Edwin Pogue - trevorpogue@gmail.com

CHAPTER

SEVEN

CHANGELOG

7.1 0.0.0 (2021-06-23)

- First release on PyPI

7.2 0.0.1 (2021-06-25)

- Refactor code into smaller classes and files
- Update Section.deep_string()
- Update readme/docs

7.3 0.0.2 (2021-06-26)

- Fix bug when using Section.leaves or Section.children
- Add tests/test_indepth_usage.py
- Update readme/docs

7.4 0.0.3

- improve __str__ to be visually intuitive
- add descendants, flat properties
- add insert methods
- add feature for default attr to search for in __call__
- can add lists as node attrs if attr name starts with '_'
- make plural_singular work for properties/methods also
- add structure_change() function for use when subclassing
- add testcases
- safer internal attrs prefix/rename classes with Section prefix

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX