

---

# **Sections**

***Release 0.0.0***

**Trevor Edwin Pogue**

**Jun 25, 2021**



# CONTENTS

<b>1</b>	<b>[ s e   c t   i o   n s ]</b>	<b>1</b>
1.1	Usage . . . . .	1
1.2	Details . . . . .	3
<b>2</b>	<b>Reference</b>	<b>9</b>
<b>3</b>	<b>Contributing</b>	<b>13</b>
3.1	Bug reports . . . . .	13
3.2	Documentation improvements . . . . .	13
3.3	Feature requests and feedback . . . . .	13
3.4	Development . . . . .	14
<b>4</b>	<b>Authors</b>	<b>17</b>
<b>5</b>	<b>Changelog</b>	<b>19</b>
5.1	0.0.0 (2021-06-14) . . . . .	19
<b>6</b>	<b>Indices and tables</b>	<b>21</b>
	<b>Index</b>	<b>23</b>



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

Flexible tree data structures for organizing lists and dicts into sections.

sections is designed to be:

- **Intuitive:** Start quickly and 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.

See the GitHub page at <https://github.com/trevorpogue/sections>. See also the [References](#) section of these docs for more thorough documentation of all the interfacing methods for Section objects.

## 1.1 Usage

```
pip install sections
```

```
import sections

menu = sections(
    'Breakfast', 'Dinner',
    mains=['Bacon&Eggs', 'Burger'],
    sides=['HashBrown', 'Fries'],
)

# Resulting structure's API and 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']
```

(continues on next page)

(continued from previous page)

```

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)

```

### 1.1.1 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 attribute settings** section.

### 1.1.2 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 **Detail - Properties/methods** section for how this works.

### 1.1.3 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(

```

(continues on next page)

(continued from previous page)

```

    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',
    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()

```

## 1.2 Details

### 1.2.1 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',
    topics=['Hobbits', 'Wizards'],
    authors=['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 will be 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) == 'section'
```

## 1.2.2 Parent names and attributes

A parent section name can optionally be provided as the first argument in a list or Section instantiation 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'],
    topics=[{'All my books'},
             [{'Imaginary things'}, 'Hobbits', 'Wizards'],
             [{'School'}, 'Numbers', 'Forces']],
)
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'
```

## 1.2.3 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)):
```

(continues on next page)



(continued from previous page)

```

    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.

## 1.2.4 Plural/singular attribute settings

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']
sections.Section.use_pluralsingular = False # turn off for all future objs
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.

## 1.2.5 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.

## 1.2.6 Printing

Section structures can be visualized through the `Section.deep_str()` method as follows:

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

Output:

```
#####  
<class 'Section'> structure  
  
'My Bookshelf' = <root, parent>  
  parent = None  
  children = ['Fantasy', 'Academic']  
  topics = 'All my books'  
  
'Fantasy' = <child, parent>  
  parent = 'My Bookshelf'  
  children = ['LOTR', 'Harry Potter']  
  topics = 'Imaginary things'  
  
'Academic' = <child, parent>  
  parent = 'My Bookshelf'  
  children = ['Advanced Mathematics', 'Physics for Engineers']  
  topics = 'School'  
  
'LOTR' = <child, leaf>  
  parent = 'Fantasy'  
  topics = 'Hobbits'  
  
'Harry Potter' = <child, leaf>  
  parent = 'Fantasy'  
  topics = 'Wizards'  
  
'Advanced Mathematics' = <child, leaf>  
  parent = 'Academic'  
  topics = 'Numbers'  
  
'Physics for Engineers' = <child, leaf>
```

(continues on next page)

(continued from previous page)

```

parent = 'Academic'
topics = 'Forces'
#####

```

See the [References](#) section of the docs for more printing options.

## 1.2.7 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__(price="Custom default value", **kwds):
        """Pass **kwds to super."""
        super().__init__(**kwds)

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

    def critique(self, impression="Haven't read it yet", rating=0):
        """Set the book price based on the impression."""
        self.review = impression
        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(impression='Good but too long', rating=7)
library.books['Harry Potter'].critique(
    impression="I don't like owls", rating=4)

```

(continues on next page)

(continued from previous page)

```
assert library.books['LOTR'].price == 14
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.

## 1.2.8 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. The downside is that it also increases memory usage by roughly 5x as well. This is not a concern on a general-purpose computer for structures representing lists/dicts with less than 1000 - 10,000 elements. However, for structures in this range or larger, it is recommended to consider changing the node or structure's class attribute use\_cache to False. This can be done as follows:

```
sect = sections(*[[[[[42] * 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
```

The dict option for gettype in the Section.\_\_call\_\_() method is currently slower than the other options. For performance-critical uses, use the other options for gettype. Alternatively, if a dict is required just for visual printing purposes, use the faster 'full\_dict' option for gettype instead. This option returns dicts with valid values with keys that also have string representations of the node names, but the raw form of the keys are references to node objects and cannot be referenced by the user through strings. See the Section.\_\_call\_\_() method in the References section of the docs for more details on the gettype options.

## 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, \*\*kws: *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, gettype: *GetType* = 'default') → 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`.

For argument *gettype*, 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 'full\_dict' is faster than *dict* and returns a dict containing pairs of a reference to each node and its attribute value. 'full\_dict' output is visually identical to *dict* for printing purposes except that it will contain all attributes even if some source nodes have duplicate names. The only downside to 'full\_dict' is that the keys cannot be referenced by name like with *dict*, but all values() are still valid.

#### Parameters

- **name** – The name of the attribute to find in self or self's descendants.
- **gettype** – Valid values are 'default', 'hybrid' *list*, *iter*, *dict*, 'full\_dict'. See method's description body above for explanation of what each value does.

**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\_\_**(name: Any) → Section

Return child node *name* of self.

**\_\_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

Not supported.

**property cls: Type[Section]**

The unique structure-wide class of each node.

**copy()** → None

Not supported.

**deep\_str**(*breadthfirst: bool = True*) → 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](#).

**fromkeys**(*\*args: Any, \*\*kwds: Any*) → None

Not supported.

**get**(*\*args: Any, \*\*kwds: Any*) → None

Not supported.

**get\_nearest\_attr**(*name: str, gettype: GetType = 'default'*) → Union[Any, List[Any], Iterable[Any], AnyDict]

Default method called by `__call__`. See the docstring of `__call__` for the full details of what this method does.

**get\_node\_attr**(*name: str, gettype: GetType = 'default'*) → Any

Return attribute *name* only from self as opposed to searching for attribute *attr* in descendant nodes as well.

**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 not 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.

**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`

Number of children Sections/nodes.

**pop**(*name: Any*) → `Any`

Remove child *name* from self.

**popitem()** → `Tuple[Any, Any]`

Remove last added child from self.

**property sections:** `Section`

A synonym for property [children](#).

**setdefault**(*\*args: Any, \*\*kwds: Any*) → `Any`

Not supported.

**update**(*\*args: Any, \*\*kwds: Any*) → `None`

Not supported.

**values()** → `Iterable[Any]`

Return iterator over chilren.





## CONTRIBUTING

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

### 3.1 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.

### 3.2 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.

### 3.3 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 :)

## 3.4 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:

Win- dows	<pre>set PYTEST_ADDOPTS=--cov-append tox</pre>
Other	<pre>PYTEST_ADDOPTS=--cov-append tox</pre>

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.

### 3.4.1 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*)<sup>1</sup>.
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*.

---

<sup>1</sup> 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 ...

### 3.4.2 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  
**FOUR**

---

**AUTHORS**

- Trevor Edwin Pogue - [trevorpogue@gmail.com](mailto:trevorpogue@gmail.com)



## CHANGELOG

### 5.1 0.0.0 (2021-06-14)

- First release on PyPI.





## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## Symbols

[\\_\\_call\\_\\_\(\) \(sections.Section method\), 9](#)  
[\\_\\_getattr\\_\\_\(\) \(sections.Section method\), 9](#)  
[\\_\\_getitem\\_\\_\(\) \(sections.Section method\), 9](#)  
[\\_\\_iter\\_\\_\(\) \(sections.Section method\), 9](#)  
[\\_\\_setattr\\_\\_\(\) \(sections.Section method\), 9](#)  
[\\_\\_setitem\\_\\_\(\) \(sections.Section method\), 9](#)

## C

[children \(sections.Section property\), 10](#)  
[clear\(\) \(sections.Section method\), 10](#)  
[cls \(sections.Section property\), 10](#)  
[copy\(\) \(sections.Section method\), 10](#)

## D

[deep\\_str\(\) \(sections.Section method\), 10](#)

## E

[entries \(sections.Section property\), 10](#)

## F

[fromkeys\(\) \(sections.Section method\), 10](#)

## G

[get\(\) \(sections.Section method\), 10](#)  
[get\\_nearest\\_attr\(\) \(sections.Section method\), 10](#)  
[get\\_node\\_attr\(\) \(sections.Section method\), 10](#)

## I

[ischild \(sections.Section property\), 10](#)  
[isleaf \(sections.Section property\), 10](#)  
[isparent \(sections.Section property\), 10](#)  
[isroot \(sections.Section property\), 10](#)  
[items\(\) \(sections.Section method\), 10](#)

## K

[keys\(\) \(sections.Section method\), 10](#)

## L

[leaves \(sections.Section property\), 10](#)  
[leaves\\_iter \(sections.Section property\), 10](#)

## N

[node\\_str\(\) \(sections.Section method\), 11](#)  
[node\\_withchildren\\_fromiter\(\) \(sections.Section method\), 11](#)  
[nofchildren \(sections.Section property\), 11](#)

## P

[pop\(\) \(sections.Section method\), 11](#)  
[popitem\(\) \(sections.Section method\), 11](#)

## S

[Section \(class in sections\), 9](#)  
[sections \(sections.Section property\), 11](#)  
[setdefault\(\) \(sections.Section method\), 11](#)

## U

[update\(\) \(sections.Section method\), 11](#)

## V

[values\(\) \(sections.Section method\), 11](#)