ctrie Documentation

Release 0.1.1

Baptiste Fontaine

Oct 26, 2017

Contents

`ctrie` is a compact trie implementation for Python.

CHAPTER 1

Install

With `pip`:

```
[sudo] pip install ctrie
```

CHAPTER 2

Guide

API Reference

ctrie

This module provides a compact trie (`CTrie`) implementation for Python.

class `ctrie.`**`CTrie`**(*terminal=False*)

A compact trie. A trie, or prefix tree, is a data structure to efficiently store a set of strings with a lot of shared prefixes, like URLs.

Compact tries are recursively defined, a trie has zero or more other child tries. A trie is said to be terminal if it contains the empty string. Each child is labelled with a prefix for its own children. For example, one compact trie that contains words "foo", "bar" and "foo42" has two children, one labelled "foo" and the other labelled "bar". The last one is terminal because once we read "bar" we can end our walk in the trie. The first one is terminal for the same reason, but have one terminal child labelled "42". In the following symbolic representation, a terminated child is represented with a dot at the end of its label:

```
<empty>
   |
   +-- bar.
   |
   +-- foo.
        |
        +-- 42.
```

Each possible walk in the trie, starting from the root and ending on a terminal node is a contained word. In the previous example, the empty string is not present in the trie because the root node is not terminal.

A non-compact trie contains only one char per node.

```
>>> from ctrie import CTrie
>>> ct = CTrie()
>>> c.add("foo", "bar", "qux")
>>> "foo" in c
True
>>> "Bar" in c
False
>>> c.remove("foo")
>>> "foo" in c
False
```

```
>>> len(c)
2
```

Keyword arguments:

- `terminal` (`bool`, default: `False`): specifies if the trie contains the empty string. This is the same as:

```
>>> ct = CTrie()
>>> ct.add('')  # empty string
```

add(**words*)

Add one or more words in the trie. The function returns `True` if all words were successfully added. It'll return `False` if one or more words were already present in the trie.

```
>>> ct = CTrie()
>>> ct.add('foo', 'bar')
True
>>> 'foo' in ct
True
>>> ct.add('bar')
False
>>> ct.add('qux')
True
```

classmethod from_dict(*d*)

Take a dictionary in the format returned by `to_dict` and build a new instance.

height()

Compute the height of the trie. An empty trie has a zero height, and the maximum height of a compacted trie is the length of its longuest word.

```
>>> ct = CTrie()
>>> ct.height()
0
>>> ct.add(''); ct.height()
0
>>> ct.add('foo'); ct.height()
1
>>> ct.add('bar'); ct.height()
1
>>> ct.add('boo'); ct.height()
2
```

New in version 0.1.0.

is_empty()

Return `True` if the trie is empty.

```
>>> ct = CTrie()
>>> ct.is_empty()
True
>>> ct.add('')
True
>>> ct.is_empty()
False
```

pretty_string()
Return a pretty-printed version of the internal state of the tree. It helps understand how words are split to fit in a tree.

remove(**words*)
Remove one or more words from the trie. The function returns `True` if all the words were successfully removed (i.e. they were in the trie before) or `False` if one or more words couldn't be found in the trie.

```
>>> ct = CTrie()
>>> ct.add('foo', 'qux')
True
>>> 'foo' in ct
True
>>> ct.remove('foo', 'bar')
False
>>> 'foo' in ct
False
>>> ct.remove('qux')
>>> True
```

subtree(*prefix*)
Return a subtree that's equivalent to the current one with all values stripped from the given prefix. Values that don't start with this prefix are not included. An empty trie is returned if the prefix doesn't match any value.

Note that the returned subtree may share its nodes with the current one.

```
>>> ct = CTrie()
>>> ct.add('foo', 'bar', 'foobar', 'fooo', 'qux')
True
>>> foo = ct.subtree('foo')
>>> list(foo)  # actual order may vary
['', 'bar', 'o']
>>> ct.subtree('nope').is_empty()
True
```

to_dict()
Recursively convert the tree to a dictionary.

values()
Yield all values from this trie in arbitrary order. The order in which they're yielded doesn't depend on the order in which they're inserted.

```
>>> ct = CTrie()
>>> ct.add('foo', 'bar', 'qux')
True
>>> for x in ct.values(): print x
...
qux
foo
bar
```

New in version 0.1.0.

write_pretty_string(*writer*)
Equivalent of `pretty_string` that uses the given writer's `.write` method.

Changes

v0.2.0 (upcoming version)

- `__eq__` added to compare two compact tries
- `subtree` added to extract a subtree from a given trie
- `values` method fixed for empty strings

v0.1.1

- `__iter__` and `__contains__` added
- `__iadd__` implemented, allowing you to extend a trie with += and an iterable

v0.1.0

- memory footprint reduced
- `len` support
- `height` and `values` methods added

v0.0.1

- initial release
- support for `add` and `remove` operations
- full online documentation

C