
cidr-trie

Release 3.1.2

Jul 05, 2019

Contents

1	cidr_trie package	1
1.1	Submodules	1
1.2	cidr_trie.bits_util module	1
1.3	cidr_trie.cidr_util module	1
1.4	Module contents	2
2	cidr-trie	7
2.1	Installation	7
2.2	Usage	7
	Python Module Index	9
	Index	11

1.1 Submodules

1.2 cidr_trie.bits_util module

Contains various utility functions for interacting/manipulating integers on a bit-level.

`cidr_trie.bits_util.bit_not (n: int, numbits: int = 32) → int`

Return the bitwise NOT of 'n' with 'numbits' bits.

`cidr_trie.bits_util.ffs (x: int) → int`

Find first set - returns the index, counting from 0 (from the right), of the least significant set bit in *x*.

`cidr_trie.bits_util.flb (val: int, v6: bool) → int`

Find last set - returns the index, counting from 0 (from the right) of the most significant set bit in *val*.

`cidr_trie.bits_util.is_set (b: int, val: int, v6: bool) → bool`

Return whether *b*-th bit is set in integer 'val'.

Special case: when *b* < 0, it acts as if it were 0.

1.3 cidr_trie.cidr_util module

Contains various utility functions for interacting/manipulating IP and CIDR addresses.

`cidr_trie.cidr_util.cidr_atoi (cidr_string: str) → Tuple[int, int]`

Convert a CIDR string to a network and prefix length tuple. Supports IPv4 and IPv6.

Parameters `cidr_string` – The CIDR as a string, i.e. “192.168.0.0/16”

Returns A tuple containing the integer IP address of the network (the lowest IP inside) and the prefix length. i.e. (int(192.168.0.0), 16) for “192.168.0.0/16”.

`cidr_trie.cidr_util.get_subnet_mask(subnet: int, v6: bool) → int`

Get the subnet mask given a CIDR prefix 'subnet'.

`cidr_trie.cidr_util.ip_atoi(ip_string: str) → int`

Converts an IP string 'ip_string' to an integer.

`cidr_trie.cidr_util.ip_itoa(ip: int, v6: bool) → str`

Converts an IP integer 'ip' to a string.

`cidr_trie.cidr_util.is_v6(ip_string: str) → bool`

Returns True if a given IP string is v6, False otherwise.

`cidr_trie.cidr_util.longest_common_prefix_length(a: int, b: int, v6: bool) → int`

Find the longest common prefix length of 'a' and 'b'.

1.4 Module contents

Store CIDR IP addresses (both v4 and v6) in a PATRICIA trie for easy lookup.

A Patricia trie can be created, inserted to, and searched like this

```
trie = PatriciaTrie()
trie.insert("0.0.0.0/0", "Internet")
trie.insert("32.0.0.0/9", "RIR-A")
trie.insert("32.128.0.0/9", "RIR-B")
trie.insert("32.32.0.0/16", "another")
trie.insert("32.32.32.0/24", "third")
trie.insert("32.32.32.32/32", "you")
trie.insert("192.168.0.1/32", "totally different")
trie.insert("33.0.0.0/8", "RIR3")
trie.insert("64.0.0.0/8", "RIR2")

# find all node values on the way down
print(trie.find_all("32.32.32.32"))
```

class `cidr_trie.PatriciaNode` (*ip: int = 0, bit: int = 0, masks: Dict[int, Any] = {}*)

Bases: `object`

A node in the Patricia trie.

ip

The IP address associated with this node.

Type `int`

bit

How many bits along the IP the decision is made to branch.

Type `int`

masks

The data stored on this node. Maps netmasks to data.

Type `Dict[int, Any]`

left

The left subtrie of this node. Self pointer if no left node.

Type `PatriciaNode`

right

The right subtrie of this node. Self pointer if no right node.

Type *PatriciaNode*

parent

The parent node of this node. None if root node.

Type *PatriciaNode*

get_child_values (*prefix: str*) → List[Tuple[str, Any]]

Get all child values from this node by iterating through netmasks and checking to see if the given prefix is larger than the given netmask.

Parameters **prefix** – The prefix to use to check, i.e. “192.168.0.0/16”

Returns list of tuples of prefixes and values, i.e. [(“192.168.0.0/16”, 2856), ...]

Return type List[Tuple[str, Any]]

get_values (*prefix: str*) → List[Tuple[str, Any]]

Get values from this node by iterating through netmasks and checking to see if the given prefix is contained within.

Parameters **prefix** – The prefix to use to check, i.e. “192.168.0.0/16”

Returns list of tuples of prefixes and values, i.e. [(“192.168.0.0/16”, 2856), ...]

Return type List[Tuple[str, Any]]

class `cidr_trie.PatriciaTrie`

Bases: object

A Patricia trie that stores IP addresses and data.

root

The root element of the trie. Always exists as 0.0.0.0.

Type *PatriciaNode*

v6

Whether this trie stores IPv6 addresses or not.

Type bool

size

The number of nodes in this trie, not counting the root node.

Type int

A Patricia trie can be created, inserted to, and searched like this

```
trie = PatriciaTrie()
trie.insert("0.0.0.0/0", "Internet")
trie.insert("32.0.0.0/9", "RIR-A")
trie.insert("32.128.0.0/9", "RIR-B")
trie.insert("32.32.0.0/16", "another")
trie.insert("32.32.32.0/24", "third")
trie.insert("32.32.32.32/32", "you")
trie.insert("192.168.0.1/32", "totally different")
trie.insert("33.0.0.0/8", "RIR3")
trie.insert("64.0.0.0/8", "RIR2")

# find all node values on the way down
print(trie.find_all("32.32.32.32"))
```

check_value_exists (*prefix: str*) -> (<class 'bool'>, <class 'bool'>)

Check to see if a value exists in the trie already. Returns 2 bools, the first to indicate whether the IP existed in the trie, and the second to indicate whether the mask existed on that IP.

There can only be one of any IP stored in the trie, if you stored 2 prefixes, “192.168.0.0/16”, and “192.168.0.0/24”, the data would be stored on the same node, as the IP address “192.168.0.0” is the same. Even though these refer to 2 separate networks.

This method exists so you can check to see if this will happen. The first boolean returned indicates if an IP address is already present, for example in the case above, if called after inserting “192.168.0.0/16”, this method would return True in the first boolean for a prefix of “192.168.0.0/24” and other netmasks, and False for the second boolean. The second boolean is for checking if the mask is present – if called after inserting “192.168.0.0/16” with a prefix of “192.168.0.0/16” it would return True for both booleans.

Parameters **prefix** – The prefix to find in the trie, i.e. “192.168.0.0/16”

Returns A 2-tuple of bools indicating whether the IP existed and the mask existed, respectively.

Return type (bool, bool)

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

find (*prefix: str*) → `cidr_trie.PatriciaNode`

Find a value in the trie.

Parameters **prefix** – The prefix to find in the trie, i.e. “192.168.0.0/16”

Returns The node if found, None otherwise.

Return type *PatriciaNode*

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

find_all (*prefix: str, children: bool = False*) → `List[Tuple[str, Any]]`

Find all values for this prefix, traversing the trie at all levels.

With get all values from common prefixes of ‘prefix’, then traverse all children of ‘prefix’ to get their values too.

Parameters

- **prefix** – The prefix to find in the trie.
- **children** – Whether to find all child values of the exact node found. (Defaults to False, as this isn’t performant in large tries)

Returns list of tuples of prefixes and values, i.e. [(“192.168.0.0/16”, 2856), ...]

Return type `List[Tuple[str, Any]]`

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

insert (*prefix: str, data: Any*) → `cidr_trie.PatriciaNode`

Insert an IP and data into the trie.

If the IP was already in the trie it will overwrite the value.

```
trie = PatriciaTrie()
trie.insert("192.168.0.0/16", 1234)
```

Parameters

- **prefix** – The prefix to insert, i.e. “192.168.0.0/16”
- **data** – The value to associate with the IP and netmask.

Returns The node that was inserted in the trie.

Return type *PatriciaNode*

Raises `ValueError` – When trying to store an IPv4 address in a trie currently storing IPv6 addresses, and vice-versa.

traverse (*prefix: str*) → *cidr_trie.PatriciaNode*

Traverse the entire trie (from root) using a prefix.

Parameters **prefix** – The prefix to find in the trie, i.e. “192.168.0.0/16”

Yields *PatriciaNode* – The next node traversed when searching for ‘prefix’.

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

traverse_from_node (*node: cidr_trie.PatriciaNode, prefix: str*) → *cidr_trie.PatriciaNode*

Traverse the trie from a specific node using a prefix.

Parameters

- **node** – The node to start traversing from.
- **prefix** – The prefix to find in the trie, i.e. “192.168.0.0/16”

Yields *PatriciaNode* – The next node traversed when searching for ‘prefix’.

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

traverse_inorder () → *cidr_trie.PatriciaNode*

Perform an inorder traversal of the trie from the root node.

Yields *PatriciaNode* – The next node in the traversal.

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

traverse_inorder_from_node (*node: cidr_trie.PatriciaNode*) → *cidr_trie.PatriciaNode*

Perform an inorder traversal of the trie from a given node.

Parameters **node** – The node to traverse from.

Yields *PatriciaNode* – The next node in the traversal.

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

traverse_preorder () → *cidr_trie.PatriciaNode*

Perform a preorder traversal of the trie from the root node.

Yields *PatriciaNode* – The next node in the traversal.

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

traverse_preorder_from_node (*node: cidr_trie.PatriciaNode*) → *cidr_trie.PatriciaNode*

Perform a preorder traversal of the trie from a given node.

Parameters **node** – The node to traverse from.

Yields *PatriciaNode* – The next node in the traversal.

Raises `ValueError` – When trying to find an IPv4 address in a v6 trie and vice-versa.

validate_ip_type_for_trie (*ip: str*) → `None`

Make sure this IP is valid for this trie.

Raises `ValueError` – if trying to insert a v4 address into a v6 trie and vice-versa.

Store CIDR IP addresses (both v4 and v6) in a trie for easy lookup.

Read the documentation [here](#).

2.1 Installation

- Using pip:

```
$ pip install cidr-trie
```

- From source (Git):

```
$ git clone https://github.com/Figglewatts/cidr-trie.git
$ cd cidr-trie
$ python setup.py install
```

- From source (PyPI):

```
$ wget https://files.pythonhosted.org/packages/6b/53/
→118c09dc2c294f41b12007634d53ed33219d15366ea8a1903fb98eb47c25/cidr_trie-1.0.tar.gz
$ tar xvf cidr_trie-1.0.tar.gz
$ cd cidr_trie-1.0
$ python setup.py install
```

2.2 Usage

cidr-trie can be used to build a trie of IP networks, storing data on each node. The stored data can be of any type. Shown here is an example of building both IPv4 and IPv6 tries with data, and then retrieving data from both tries.

```
from cidr_trie import PatriciaTrie

# --- supports IPv4 ---
trie = PatriciaTrie()
trie.insert("0.0.0.0/0", "Internet")
trie.insert("32.0.0.0/9", "RIR-A")
trie.insert("32.128.0.0/9", "RIR-B")
trie.insert("32.32.0.0/16", "another")
trie.insert("32.32.32.0/24", "third")
trie.insert("32.32.32.32/32", "you")
trie.insert("192.168.0.1/32", "totally different")
trie.insert("33.0.0.0/8", "RIR3")
trie.insert("64.0.0.0/8", "RIR2")

# nodes: ['Internet', 'RIR-A', 'another', 'third', 'you']
nodes_for_prefix = trie.find_all("32.32.32.32")

# prints "Internet, RIR-A, another, third, you"
print(', '.join(n.value for n in nodes_for_prefix))

# nodes: ['Internet', 'totally different']
trie.find_all("192.168.0.1/32")

# nodes: ['Internet', 'RIR-B']
trie.find_all("32.192.0.0/10")

# --- supports IPv6 ---
trie = PatriciaTrie()
trie.insert("::/0", "Internet")
trie.insert("1234::/16", "Test")
trie.insert("1234:1001::/32", "Another one")
trie.insert("1234:1001:1920::/48", "A third")
trie.insert("1234:1001:1920:2000:2020::/96", "A fourth")
trie.insert("1234:1001:1920::ffff", "A different one")

# nodes: ['Internet', 'Test', 'Another one', 'A third', 'A fourth']
trie.find_all("1234:1001:1920:2000:2020::/128")

# nodes: ['Internet', 'Test', 'Another one', 'A third', 'A different one']
trie.find_all("1234:1001:1920::ffff")
```

C

- `cidr_trie`, 2
- `cidr_trie.bits_util`, 1
- `cidr_trie.cidr_util`, 1

B

`bit` (*cidr_trie.PatriciaNode* attribute), 2
`bit_not()` (in module *cidr_trie.bits_util*), 1

C

`check_value_exists()` (*cidr_trie.PatriciaTrie* method), 3
`cidr_atoi()` (in module *cidr_trie.cidr_util*), 1
`cidr_trie` (module), 2
`cidr_trie.bits_util` (module), 1
`cidr_trie.cidr_util` (module), 1

F

`ffs()` (in module *cidr_trie.bits_util*), 1
`find()` (*cidr_trie.PatriciaTrie* method), 4
`find_all()` (*cidr_trie.PatriciaTrie* method), 4
`fls()` (in module *cidr_trie.bits_util*), 1

G

`get_child_values()` (*cidr_trie.PatriciaNode* method), 3
`get_subnet_mask()` (in module *cidr_trie.cidr_util*), 1
`get_values()` (*cidr_trie.PatriciaNode* method), 3

I

`insert()` (*cidr_trie.PatriciaTrie* method), 4
`ip` (*cidr_trie.PatriciaNode* attribute), 2
`ip_atoi()` (in module *cidr_trie.cidr_util*), 2
`ip_itoa()` (in module *cidr_trie.cidr_util*), 2
`is_set()` (in module *cidr_trie.bits_util*), 1
`is_v6()` (in module *cidr_trie.cidr_util*), 2

L

`left` (*cidr_trie.PatriciaNode* attribute), 2
`longest_common_prefix_length()` (in module *cidr_trie.cidr_util*), 2

M

`masks` (*cidr_trie.PatriciaNode* attribute), 2

P

`parent` (*cidr_trie.PatriciaNode* attribute), 3
PatriciaNode (class in *cidr_trie*), 2
PatriciaTrie (class in *cidr_trie*), 3

R

`right` (*cidr_trie.PatriciaNode* attribute), 2
`root` (*cidr_trie.PatriciaTrie* attribute), 3

S

`size` (*cidr_trie.PatriciaTrie* attribute), 3

T

`traverse()` (*cidr_trie.PatriciaTrie* method), 5
`traverse_from_node()` (*cidr_trie.PatriciaTrie* method), 5
`traverse_inorder()` (*cidr_trie.PatriciaTrie* method), 5
`traverse_inorder_from_node()` (*cidr_trie.PatriciaTrie* method), 5
`traverse_preorder()` (*cidr_trie.PatriciaTrie* method), 5
`traverse_preorder_from_node()` (*cidr_trie.PatriciaTrie* method), 5

V

`v6` (*cidr_trie.PatriciaTrie* attribute), 3
`validate_ip_type_for_trie()` (*cidr_trie.PatriciaTrie* method), 5