

---

# **iptools**

***Release 0.6.1***

**Dec 12, 2018**



---

## Contents

---

<b>1</b>	<b>Using with Django INTERNAL_IPS</b>	<b>3</b>
<b>2</b>	<b>Python Version Compatibility</b>	<b>5</b>
<b>3</b>	<b>Installation</b>	<b>7</b>
<b>4</b>	<b>API</b>	<b>9</b>
4.1	iptools . . . . .	9
4.2	iptools.ipv4 . . . . .	14
4.3	iptools.ipv6 . . . . .	20
<b>5</b>	<b>Indices and tables</b>	<b>25</b>
	<b>Python Module Index</b>	<b>27</b>



The **iptools** package is a collection of utilities for dealing with IP addresses.

The project was inspired by a desire to be able to use **CIDR** address notation to designate `INTERNAL_IPS` in a **Django** project's settings file.



---

## Using with Django INTERNAL\_IPS

---

An `iptools.IpRangeList` object can be used in a Django settings file to allow CIDR and/or (start, end) ranges to be used in the `INTERNAL_IPS` list.

There are many internal and add-on components for Django that use the `INTERNAL_IPS` configuration setting to alter application behavior or make debugging easier. When you are developing and testing an application by yourself it's easy to add the ip address that your web browser will be coming from to this list. When you are developing in a group or testing from many ips it can become cumbersome to add more and more ip addresses to the setting individually.

The `iptools.IpRangeList` object can help by replacing the standard tuple of addresses recommended by the Django docs with an intelligent object that responds to the membership test operator `in`. This object can be configured with dotted quad IP addresses like the default `INTERNAL_IPS` tuple (eg. `'127.0.0.1'`), CIDR block notation (eg. `'127/8'`, `'192.168/16'`) for entire network blocks, and/or (start, end) tuples describing an arbitrary range of IP addresses.

Django's internal checks against the `INTERNAL_IPS` tuple take the form `if addr in INTERNAL_IPS` or `if addr not in INTERNAL_IPS`. This works transparently with the `iptools.IpRangeList` object because it implements the magic method `__contains__` which python calls when the `in` or `not in` operators are used.

```
import iptools

INTERNAL_IPS = iptools.IpRangeList(
    '127.0.0.1',          # single ip
    '192.168/16',        # CIDR network block
    ('10.0.0.1', '10.0.0.19'), # arbitrary inclusive range
    '::1',               # single IPv6 address
    'fe80::/10',         # IPv6 CIDR block
    '::ffff:172.16.0.2'  # IPv4-mapped IPv6 address
)
```






## CHAPTER 2

---

### Python Version Compatibility

---

Travis CI automatically runs tests against python 2.5, 2.6, 2.7, 3.2, 3.3 and pypy.

Current test status:  build passing



## CHAPTER 3

---

### Installation

---

Install the latest stable version from PyPi using `pip`:

```
$ pip install iptools
```

or `setuptools`:

```
$ easy_install iptools
```

Install the latest development version:

```
$ git clone https://github.com/bd808/python-iptables.git
$ cd python-iptables
$ python setup.py install
```



## 4.1 iptools

### 4.1.1 iptools.IpRangeList

**class** iptools.IpRangeList(\*args)

List of IpRange objects.

Converts a list of ip address and/or CIDR addresses into a list of IpRange objects. This list can perform `in` and `not in` tests and iterate all of the addresses in the range.

**Parameters** *\*args* (list of str and/or tuple) – List of ip addresses or CIDR notation and/or (start, end) tuples of ip addresses.

**\_\_contains\_\_** (item)

Implements membership test operators `in` and `not in` for the address ranges contained in the list.

```
>>> r = IpRangeList('127.0.0.1', '10/8', '192.168/16')
>>> '127.0.0.1' in r
True
>>> '10.0.0.1' in r
True
>>> 2130706433 in r
True
>>> 'invalid' in r
Traceback (most recent call last):
...
TypeError: expected ip address, 32-bit integer or 128-bit integer
```

**Parameters** *item* (str) – Dotted-quad ip address.

**Returns** True if address is in list, False otherwise.

**\_\_init\_\_** (\*args)

*x.\_\_init\_\_*(...) initializes x; see `help(type(x))` for signature

`__iter__()`

Return an iterator over all ip addresses in the list.

```
>>> iter = IpRangeList('127.0.0.1').__iter__()
>>> next(iter)
'127.0.0.1'
>>> next(iter)
Traceback (most recent call last):
...
StopIteration
>>> iter = IpRangeList('127.0.0.1', '10/31').__iter__()
>>> next(iter)
'127.0.0.1'
>>> next(iter)
'10.0.0.0'
>>> next(iter)
'10.0.0.1'
>>> next(iter)
Traceback (most recent call last):
...
StopIteration
```

`__len__()`

Return the length of all ranges in the list.

```
>>> len(IpRangeList('127.0.0.1'))
1
>>> len(IpRangeList('127.0.0.1', '10/31'))
3
>>> len(IpRangeList('1/24'))
256
>>> len(IpRangeList('192.168.0.0/22'))
1024
```

`__repr__()`

```
>>> repr(IpRangeList('127.0.0.1', '10/8', '192.168/16'))
...
"IpRangeList(IpRange('127.0.0.1', '127.0.0.1'),
IpRange('10.0.0.0', '10.255.255.255'),
IpRange('192.168.0.0', '192.168.255.255'))"
>>> repr(
...     IpRangeList(IpRange('127.0.0.1', '127.0.0.1'),
...     IpRange('10.0.0.0', '10.255.255.255'),
...     IpRange('192.168.0.0', '192.168.255.255')))
...
"IpRangeList(IpRange('127.0.0.1', '127.0.0.1'),
IpRange('10.0.0.0', '10.255.255.255'),
IpRange('192.168.0.0', '192.168.255.255'))"
```

`__str__()`

```
>>> str(IpRangeList('127.0.0.1', '10/8', '192.168/16'))
...
"(('127.0.0.1', '127.0.0.1'),
```

(continues on next page)

(continued from previous page)

```
( '10.0.0.0', '10.255.255.255'),
( '192.168.0.0', '192.168.255.255')) "
```

**\_\_weakref\_\_**

list of weak references to the object (if defined)

## 4.1.2 iptools.IpRange

**class** iptools.IpRange (start, end=None)

Range of ip addresses.

Converts a CIDR notation address, ip address and subnet, tuple of ip addresses or start and end addresses into a smart object which can perform `in` and `not in` tests and iterate all of the addresses in the range.

```
>>> r = IpRange('127.0.0.1', '127.255.255.255')
>>> '127.127.127.127' in r
True
>>> '10.0.0.1' in r
False
>>> 2130706433 in r
True
>>> # IPv4 mapped IPv6 addresses are valid in an IPv4 block
>>> '::ffff:127.127.127.127' in r
True
>>> # but only if they are actually in the block :)
>>> '::ffff:192.0.2.128' in r
False
>>> '::ffff:c000:0280' in r
False
>>> r = IpRange('127/24')
>>> print(r)
('127.0.0.0', '127.0.0.255')
>>> r = IpRange('127/30')
>>> for ip in r:
...     print(ip)
127.0.0.0
127.0.0.1
127.0.0.2
127.0.0.3
>>> print(IpRange('127.0.0.255', '127.0.0.0'))
('127.0.0.0', '127.0.0.255')
>>> r = IpRange('127/255.255.255.0')
>>> print(r)
('127.0.0.0', '127.0.0.255')
>>> r = IpRange('::ffff:0000:0000', '::ffff:ffff:ffff')
>>> '::ffff:192.0.2.128' in r
True
>>> '::ffff:c000:0280' in r
True
>>> 281473902969472 in r
True
>>> '192.168.2.128' in r
False
>>> 2130706433 in r
False
```

(continues on next page)

(continued from previous page)

```
>>> r = IpRange('::ffff:ffff:0000/120')
>>> for ip in r:
...     print(ip)
::ffff:ffff:0 ... ::ffff:ffff:6d ... ::ffff:ffff:ff
```

### Parameters

- **start** (*str* or *tuple*) – Ip address in dotted quad format, CIDR notation, subnet format or (*start*, *end*) tuple of ip addresses in dotted quad format.
- **end** (*str*) – Ip address in dotted quad format or None.

### `__contains__` (*item*)

Implements membership test operators `in` and `not in` for the address range.

```
>>> r = IpRange('127.0.0.1', '127.255.255.255')
>>> '127.127.127.127' in r
True
>>> '10.0.0.1' in r
False
>>> 2130706433 in r
True
>>> 'invalid' in r
Traceback (most recent call last):
...
TypeError: expected ip address, 32-bit integer or 128-bit integer
```

**Parameters** *item* (*str*) – Dotted-quad ip address.

**Returns** True if address is in range, False otherwise.

### `__eq__` (*other*)

```
>>> IpRange('127.0.0.1') == IpRange('127.0.0.1')
True
>>> IpRange('127.0.0.1') == IpRange('127.0.0.2')
False
>>> IpRange('10/8') == IpRange('10', '10.255.255.255')
True
```

### `__getitem__` (*index*)

```
>>> r = IpRange('127.0.0.1', '127.255.255.255')
>>> r[0]
'127.0.0.1'
>>> r[16777214]
'127.255.255.255'
>>> r[-1]
'127.255.255.255'
>>> r[len(r)]
Traceback (most recent call last):
...
IndexError: index out of range
```



```

>>> r[:]
IpRange('127.0.0.1', '127.255.255.255')
>>> r[1:]
IpRange('127.0.0.2', '127.255.255.255')
>>> r[-2:]
IpRange('127.255.255.254', '127.255.255.255')
>>> r[0:2]
IpRange('127.0.0.1', '127.0.0.2')
>>> r[0:-1]
IpRange('127.0.0.1', '127.255.255.254')
>>> r[:-2]
IpRange('127.0.0.1', '127.255.255.253')
>>> r[::2]
Traceback (most recent call last):
...
ValueError: slice step not supported

```

**\_\_hash\_\_()**

```

>>> a = IpRange('127.0.0.0/8')
>>> b = IpRange('127.0.0.0', '127.255.255.255')
>>> a.__hash__() == b.__hash__()
True
>>> c = IpRange('10/8')
>>> a.__hash__() == c.__hash__()
False
>>> b.__hash__() == c.__hash__()
False

```

**\_\_init\_\_(start, end=None)**

x.\_\_init\_\_(...) initializes x; see help(type(x)) for signature

**\_\_iter\_\_()**

Return an iterator over ip addresses in the range.

```

>>> iter = IpRange('127/31').__iter__()
>>> next(iter)
'127.0.0.0'
>>> next(iter)
'127.0.0.1'
>>> next(iter)
Traceback (most recent call last):
...
StopIteration

```

**\_\_len\_\_()**

Return the length of the range.

```

>>> len(IpRange('127.0.0.1'))
1
>>> len(IpRange('127/31'))
2
>>> len(IpRange('127/22'))
1024

```

**\_\_repr\_\_()**

```
>>> repr(IpRange('127.0.0.1'))
"IpRange('127.0.0.1', '127.0.0.1') "
>>> repr(IpRange('10/8'))
"IpRange('10.0.0.0', '10.255.255.255') "
>>> repr(IpRange('127.0.0.255', '127.0.0.0'))
"IpRange('127.0.0.0', '127.0.0.255') "
```

`__str__()`

```
>>> str(IpRange('127.0.0.1'))
"('127.0.0.1', '127.0.0.1') "
>>> str(IpRange('10/8'))
"('10.0.0.0', '10.255.255.255') "
>>> str(IpRange('127.0.0.255', '127.0.0.0'))
"('127.0.0.0', '127.0.0.255') "
```

**index** (*item*)

Return the 0-based position of *item* in this IpRange.

```
>>> r = IpRange('127.0.0.1', '127.255.255.255')
>>> r.index('127.0.0.1')
0
>>> r.index('127.255.255.255')
16777214
>>> r.index('10.0.0.1')
Traceback (most recent call last):
...
ValueError: 10.0.0.1 is not in range
```

**Parameters** *item* (*str*) – Dotted-quad ip address.

**Returns** Index of ip address in range

`__weakref__`

list of weak references to the object (if defined)

## 4.2 iptools.ipv4

`iptools.ipv4.cidr2block` (*cidr*)

Convert a CIDR notation ip address into a tuple containing the network block start and end addresses.

```
>>> cidr2block('127.0.0.1/32')
('127.0.0.1', '127.0.0.1')
>>> cidr2block('127/8')
('127.0.0.0', '127.255.255.255')
>>> cidr2block('127.0.1/16')
('127.0.0.0', '127.0.255.255')
>>> cidr2block('127.1/24')
('127.1.0.0', '127.1.0.255')
>>> cidr2block('127.0.0.3/29')
('127.0.0.0', '127.0.0.7')
>>> cidr2block('127/0')
('0.0.0.0', '255.255.255.255')
```

**Parameters** `cidr` (*str*) – CIDR notation ip address (eg. '127.0.0.1/8').

**Returns** Tuple of block (start, end) or None if invalid.

**Raises** TypeError

`iptools.ipv4.hex2ip(hex_str)`

Convert a hex encoded integer to a dotted-quad ip address.

```
>>> hex2ip('00000001')
'0.0.0.1'
>>> hex2ip('7f000001')
'127.0.0.1'
>>> hex2ip('7fffffff')
'127.255.255.255'
>>> hex2ip('80000001')
'128.0.0.1'
>>> hex2ip('ffffffff')
'255.255.255.255'
```

**Parameters** `hex_str` (*str*) – Numeric ip address as a hex-encoded string.

**Returns** Dotted-quad ip address or None if invalid.

`iptools.ipv4.ip2hex(addr)`

Convert a dotted-quad ip address to a hex encoded number.

```
>>> ip2hex('0.0.0.1')
'00000001'
>>> ip2hex('127.0.0.1')
'7f000001'
>>> ip2hex('127.255.255.255')
'7fffffff'
>>> ip2hex('128.0.0.1')
'80000001'
>>> ip2hex('128.1')
'80000001'
>>> ip2hex('255.255.255.255')
'ffffffff'
```

**Parameters** `addr` (*str*) – Dotted-quad ip address.

**Returns** Numeric ip address as a hex-encoded string or None if invalid.

`iptools.ipv4.ip2long(ip)`

Convert a dotted-quad ip address to a network byte order 32-bit integer.

```
>>> ip2long('127.0.0.1')
2130706433
>>> ip2long('127.1')
2130706433
>>> ip2long('127')
2130706432
>>> ip2long('127.0.0.256') is None
True
```

**Parameters** `ip` (*str*) – Dotted-quad ip address (eg. '127.0.0.1').

**Returns** Network byte order 32-bit integer or *None* if ip is invalid.

`iptools.ipv4.ip2network(ip)`

Convert a dotted-quad ip to base network number.

This differs from `ip2long()` in that partial addresses are treated as all network instead of network plus host (eg. '127.1' expands to '127.1.0.0')

**Parameters** `ip (str)` – dotted-quad ip address (eg. '127.0.0.1').

**Returns** Network byte order 32-bit integer or *None* if ip is invalid.

`iptools.ipv4.long2ip(l)`

Convert a network byte order 32-bit integer to a dotted quad ip address.

```
>>> long2ip(2130706433)
'127.0.0.1'
>>> long2ip(MIN_IP)
'0.0.0.0'
>>> long2ip(MAX_IP)
'255.255.255.255'
>>> long2ip(None)
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for >>: 'NoneType' and 'int'
>>> long2ip(-1)
Traceback (most recent call last):
...
TypeError: expected int between 0 and 4294967295 inclusive
>>> long2ip(374297346592387463875)
Traceback (most recent call last):
...
TypeError: expected int between 0 and 4294967295 inclusive
>>> long2ip(MAX_IP + 1)
Traceback (most recent call last):
...
TypeError: expected int between 0 and 4294967295 inclusive
```

**Parameters** `l (int)` – Network byte order 32-bit integer.

**Returns** Dotted-quad ip address (eg. '127.0.0.1').

**Raises** `TypeError`

`iptools.ipv4.netmask2prefix(mask)`

Convert a dotted-quad netmask into a CIDR prefix.

```
>>> netmask2prefix('255.0.0.0')
8
>>> netmask2prefix('255.128.0.0')
9
>>> netmask2prefix('255.255.255.254')
31
>>> netmask2prefix('255.255.255.255')
32
>>> netmask2prefix('0.0.0.0')
0
>>> netmask2prefix('127.0.0.1')
0
```

**Parameters** `mask` (*str*) – Netmask in dotted-quad notation.

**Returns** CIDR prefix corresponding to netmask or 0 if invalid.

`iptools.ipv4.subnet2block` (*subnet*)

Convert a dotted-quad ip address including a netmask into a tuple containing the network block start and end addresses.

```
>>> subnet2block('127.0.0.1/255.255.255.255')
('127.0.0.1', '127.0.0.1')
>>> subnet2block('127/255')
('127.0.0.0', '127.255.255.255')
>>> subnet2block('127.0.1/255.255')
('127.0.0.0', '127.0.255.255')
>>> subnet2block('127.1/255.255.255.0')
('127.1.0.0', '127.1.0.255')
>>> subnet2block('127.0.0.3/255.255.255.248')
('127.0.0.0', '127.0.0.7')
>>> subnet2block('127/0')
('0.0.0.0', '255.255.255.255')
```

**Parameters** `subnet` (*str*) – dotted-quad ip address with netmask (eg. '127.0.0.1/255.0.0.0').

**Returns** Tuple of block (start, end) or None if invalid.

**Raises** `TypeError`

`iptools.ipv4.validate_cidr` (*s*)

Validate a CIDR notation ip address.

The string is considered a valid CIDR address if it consists of a valid IPv4 address in dotted-quad format followed by a forward slash (/) and a bit mask length (1-32).

```
>>> validate_cidr('127.0.0.1/32')
True
>>> validate_cidr('127.0/8')
True
>>> validate_cidr('127.0.0.256/32')
False
>>> validate_cidr('127.0.0.0')
False
>>> validate_cidr('LOOPBACK')
True
>>> validate_cidr('127.0.0.1/33')
False
>>> validate_cidr(None)
Traceback (most recent call last):
...
TypeError: expected string or buffer
```

**Parameters** `s` (*str*) – String to validate as a CIDR notation ip address.

**Returns** `True` if a valid CIDR address, `False` otherwise.

**Raises** `TypeError`

`iptools.ipv4.validate_ip` (*s*)

Validate a dotted-quad ip address.

The string is considered a valid dotted-quad address if it consists of one to four octets (0-255) separated by periods (.).

```
>>> validate_ip('127.0.0.1')
True
>>> validate_ip('127.0')
True
>>> validate_ip('127.0.0.256')
False
>>> validate_ip(LOCALHOST)
True
>>> validate_ip(None)
Traceback (most recent call last):
...
TypeError: expected string or buffer
```

**Parameters** *s* (*str*) – String to validate as a dotted-quad ip address.

**Returns** True if a valid dotted-quad ip address, False otherwise.

**Raises** TypeError

`iputils.ipv4.validate_netmask(s)`

Validate that a dotted-quad ip address is a valid netmask.

```
>>> validate_netmask('0.0.0.0')
True
>>> validate_netmask('128.0.0.0')
True
>>> validate_netmask('255.0.0.0')
True
>>> validate_netmask('255.255.255.255')
True
>>> validate_netmask(BROADCAST)
True
>>> validate_netmask('128.0.0.1')
False
```

**Parameters** *s* (*str*) – String to validate as a dotted-quad notation netmask.

**Returns** True if a valid netmask, False otherwise.

**Raises** TypeError

`iputils.ipv4.validate_subnet(s)`

Validate a dotted-quad ip address including a netmask.

The string is considered a valid dotted-quad address with netmask if it consists of one to four octets (0-255) separated by periods (.) followed by a forward slash (/) and a subnet bitmask which is expressed in dotted-quad format.

```
>>> validate_subnet('127.0.0.1/255.255.255.255')
True
>>> validate_subnet('127.0/255.0.0.0')
True
>>> validate_subnet('127.0/255')
True
>>> validate_subnet('127.0.0.256/255.255.255.255')
```

(continues on next page)

(continued from previous page)

```
False
>>> validate_subnet('127.0.0.1/255.255.255.256')
False
>>> validate_subnet('127.0.0.0')
False
>>> validate_subnet(None)
Traceback (most recent call last):
...
TypeError: expected string or unicode
```

**Parameters** *s* (*str*) – String to validate as a dotted-quad ip address with netmask.

**Returns** True if a valid dotted-quad ip address with netmask, False otherwise.

**Raises** TypeError

```
iptools.ipv4.BENCHMARK_TESTS = '198.18.0.0/15'
    Inter-network communications testing (RFC 2544)

iptools.ipv4.BROADCAST = '255.255.255.255'
    Broadcast messages to the current network (only valid as destination address) (RFC 919)

iptools.ipv4.CURRENT_NETWORK = '0.0.0.0/8'
    Broadcast messages to the current network (only valid as source address) (RFC 5735)

iptools.ipv4.DUAL_STACK_LITE = '192.0.0.0/29'
    Dual-Stack Lite link address (RFC 6333)

iptools.ipv4.IETF_PROTOCOL_RESERVED = '192.0.0.0/24'
    IETF protocol assignments reserved block (RFC 5735)

iptools.ipv4.IPV6_TO_IPV4_RELAY = '192.88.99.0/24'
    6to4 anycast relay (RFC 3068)

iptools.ipv4.LINK_LOCAL = '169.254.0.0/16'
    Autoconfiguration when no IP address available (RFC 3972)

iptools.ipv4.LOCALHOST = '127.0.0.1'
    Common localhost address (RFC 5735)

iptools.ipv4.LOOPBACK = '127.0.0.0/8'
    Loopback addresses on the local host (RFC 5735)

iptools.ipv4.MAX_IP = 4294967295
    Maximum IPv4 integer

iptools.ipv4.MIN_IP = 0
    Minimum IPv4 integer

iptools.ipv4.MULTICAST = '224.0.0.0/4'
    Multicast reserved block (RFC 5771)

iptools.ipv4.MULTICAST_INTERNETWORK = '224.0.1.0/24'
    Forwardable multicast (RFC 5771)

iptools.ipv4.MULTICAST_LOCAL = '224.0.0.0/24'
    Link local multicast (RFC 5771)

iptools.ipv4.PRIVATE_NETWORK_10 = '10.0.0.0/8'
    Private network (RFC 1918)
```

```

iputils.ipv4.PRIVATE_NETWORK_172_16 = '172.16.0.0/12'
    Private network (RFC 1918)

iputils.ipv4.PRIVATE_NETWORK_192_168 = '192.168.0.0/16'
    Private network (RFC 1918)

iputils.ipv4.RESERVED = '240.0.0.0/4'
    Former Class E address space. Reserved for future use (RFC 1700)

iputils.ipv4.SHARED_ADDRESS_SPACE = '100.64.0.0/10'
    Carrier-grade NAT private network (RFC 6598)

iputils.ipv4.TEST_NET_1 = '192.0.2.0/24'
    Documentation and example network (RFC 5737)

iputils.ipv4.TEST_NET_2 = '198.51.100.0/24'
    Documentation and example network (RFC 5737)

iputils.ipv4.TEST_NET_3 = '203.0.113.0/24'
    Documentation and example network (RFC 5737)

```

## 4.3 iputils.ipv6

`iputils.ipv6.cidr2block(cidr)`  
 Convert a CIDR notation ip address into a tuple containing the network block start and end addresses.

```

>>> cidr2block('2001:db8::/48')
('2001:db8::', '2001:db8:0:ffff:ffff:ffff:ffff:ffff')
>>> cidr2block('::/0')
('::', 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff')

```

**Parameters** `cidr` (*str*) – CIDR notation ip address (eg. '127.0.0.1/8').

**Returns** Tuple of block (start, end) or None if invalid.

**Raises** `TypeError`

`iputils.ipv6.ip2long(ip)`  
 Convert a hexadecimal IPv6 address to a network byte order 128-bit integer.

```

>>> ip2long('::') == 0
True
>>> ip2long('::1') == 1
True
>>> expect = 0x20010db885a300000000008a2e03707334
>>> ip2long('2001:db8:85a3::8a2e:370:7334') == expect
True
>>> ip2long('2001:db8:85a3:0:0:8a2e:370:7334') == expect
True
>>> ip2long('2001:0db8:85a3:0000:0000:8a2e:0370:7334') == expect
True
>>> expect = 0x20010db80000000000000010000000000001
>>> ip2long('2001:db8::1:0:0:1') == expect
True
>>> expect = 281473902969472
>>> ip2long('::ffff:192.0.2.128') == expect
True

```

(continues on next page)



(continued from previous page)

```

>>> expect = 0xffffffffffffffffffffffffffffffff
>>> ip2long('ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff') == expect
True
>>> ip2long('ff::ff:ff') == None
True

```

**Parameters** `ip` (*str*) – Hexidecimal IPv6 address

**Returns** Network byte order 128-bit integer or None if ip is invalid.

`iptools.ipv6.long2ip(l)`

Convert a network byte order 128-bit integer to a canonical IPv6 address.

```

>>> long2ip(2130706433)
'::7f00:1'
>>> long2ip(42540766411282592856904266426630537217)
'2001:db8::1:0:0:1'
>>> long2ip(MIN_IP)
'::1'
>>> long2ip(MAX_IP)
'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff'
>>> long2ip(None)
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for >>: 'NoneType' and 'int'
>>> long2ip(-1)
Traceback (most recent call last):
...
TypeError: expected int between 0 and <really big int> inclusive
>>> long2ip(MAX_IP + 1)
Traceback (most recent call last):
...
TypeError: expected int between 0 and <really big int> inclusive

```

**Parameters** `l` (*int*) – Network byte order 128-bit integer.

**Returns** Canonical IPv6 address (eg. `'::1'`).

**Raises** `TypeError`

`iptools.ipv6.validate_cidr(s)`

Validate a CIDR notation ip address.

The string is considered a valid CIDR address if it consists of a valid IPv6 address in hextet format followed by a forward slash (/) and a bit mask length (0-128).

```

>>> validate_cidr('::/128')
True
>>> validate_cidr('::/0')
True
>>> validate_cidr('fc00::/7')
True
>>> validate_cidr('::ffff:0:0/96')
True
>>> validate_cidr(':::')
False

```

(continues on next page)

(continued from previous page)

```
>>> validate_cidr('::/129')
False
>>> validate_cidr(None)
Traceback (most recent call last):
...
TypeError: expected string or buffer
```

**Parameters** *s* (*str*) – String to validate as a CIDR notation ip address.

**Returns** True if a valid CIDR address, False otherwise.

**Raises** TypeError

`iputils.ipv6.validate_ip(s)`

Validate a hexadecimal IPv6 ip address.

```
>>> validate_ip('::')
True
>>> validate_ip(':::1')
True
>>> validate_ip('2001:db8:85a3::8a2e:370:7334')
True
>>> validate_ip('2001:db8:85a3:0:0:8a2e:370:7334')
True
>>> validate_ip('2001:0db8:85a3:0000:0000:8a2e:0370:7334')
True
>>> validate_ip('2001:db8::1:0:0:1')
True
>>> validate_ip('ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff')
True
>>> validate_ip('::ffff:192.0.2.128')
True
>>> validate_ip('::ff::ff')
False
>>> validate_ip('::fffff')
False
>>> validate_ip('::ffff:192.0.2.300')
False
>>> validate_ip(None)
Traceback (most recent call last):
...
TypeError: expected string or buffer
```

**Parameters** *s* (*str*) – String to validate as a hexadecimal IPv6 ip address.

**Returns** True if a valid hexadecimal IPv6 ip address, False otherwise.

**Raises** TypeError

`iputils.ipv6.DOCUMENTATION_NETWORK = '2001::db8::/32'`

Documentation and example network (RFC 3849)

`iputils.ipv6.IPV4_MAPPED = '::ffff:0:0/96'`

IPv4 mapped to IPv6 (not globally routable) (RFC 4291)

`iputils.ipv6.IPV6_TO_IPV4_NETWORK = '2002::/16'`

6to4 Address block (RFC 3056)

```

iptools.ipv6.LINK_LOCAL = 'fe80::/10'
    Link-Local unicast networks (not globally routable) (RFC 4291)

iptools.ipv6.LOCALHOST = '::1/128'
    Common localhost address (RFC 4291)

iptools.ipv6.LOOPBACK = '::1/128'
    Loopback addresses on the local host (RFC 4291)

iptools.ipv6.MAX_IP = 340282366920938463463374607431768211455L
    Maximum IPv6 integer

iptools.ipv6.MIN_IP = 0
    Minimum IPv6 integer

iptools.ipv6.MULTICAST = 'ff00::/8'
    Multicast reserved block (RFC 5771)

iptools.ipv6.MULTICAST_GLOBAL = 'ff0e::/16'
    Organization-Local multicast

iptools.ipv6.MULTICAST_LOCAL = 'ff02::/16'
    Link-Local multicast

iptools.ipv6.MULTICAST_LOCAL_DHCP = 'ff02::1:2'
    All DHCP servers and relay agents on the local segment

iptools.ipv6.MULTICAST_LOCAL_NODES = 'ff02::1'
    All nodes on the local segment

iptools.ipv6.MULTICAST_LOCAL_ROUTERS = 'ff02::2'
    All routers on the local segment

iptools.ipv6.MULTICAST_LOOPBACK = 'ff01::/16'
    Interface-Local multicast

iptools.ipv6.MULTICAST_SITE = 'ff08::/16'
    Organization-Local multicast

iptools.ipv6.MULTICAST_SITE = 'ff08::/16'
    Organization-Local multicast

iptools.ipv6.MULTICAST_SITE_DHCP = 'ff05::1:3'
    All DHCP servers and relay agents on the local site

iptools.ipv6.PRIVATE_NETWORK = 'fd00::/8'
    Private network (RFC 4193)

iptools.ipv6.TEREDO_NETWORK = '2001::/32'
    Teredo addresses (RFC 4380)

iptools.ipv6.UNSPECIFIED_ADDRESS = '::/128'
    Absence of an address (only valid as source address) (RFC 4291)

```



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `search`



### i

- `iptables`, [9](#)
- `iptables.ipv4`, [14](#)
- `iptables.ipv6`, [20](#)





## Symbols

\_\_contains\_\_() (iptools.IpRange method), 12  
 \_\_contains\_\_() (iptools.IpRangeList method), 9  
 \_\_eq\_\_() (iptools.IpRange method), 12  
 \_\_getitem\_\_() (iptools.IpRange method), 12  
 \_\_hash\_\_() (iptools.IpRange method), 13  
 \_\_init\_\_() (iptools.IpRange method), 13  
 \_\_init\_\_() (iptools.IpRangeList method), 9  
 \_\_iter\_\_() (iptools.IpRange method), 13  
 \_\_iter\_\_() (iptools.IpRangeList method), 10  
 \_\_len\_\_() (iptools.IpRange method), 13  
 \_\_len\_\_() (iptools.IpRangeList method), 10  
 \_\_repr\_\_() (iptools.IpRange method), 13  
 \_\_repr\_\_() (iptools.IpRangeList method), 10  
 \_\_str\_\_() (iptools.IpRange method), 14  
 \_\_str\_\_() (iptools.IpRangeList method), 10  
 \_\_weakref\_\_ (iptools.IpRange attribute), 14  
 \_\_weakref\_\_ (iptools.IpRangeList attribute), 11

## B

BENCHMARK\_TESTS (in module iptools.ipv4), 19  
 BROADCAST (in module iptools.ipv4), 19

## C

cidr2block() (in module iptools.ipv4), 14  
 cidr2block() (in module iptools.ipv6), 20  
 CURRENT\_NETWORK (in module iptools.ipv4), 19

## D

DOCUMENTATION\_NETWORK (in module ip-  
tools.ipv6), 22  
 DUAL\_STACK\_LITE (in module iptools.ipv4), 19

## H

hex2ip() (in module iptools.ipv4), 15

## I

IETF\_PROTOCOL\_RESERVED (in module ip-  
tools.ipv4), 19

index() (iptools.IpRange method), 14  
 ip2hex() (in module iptools.ipv4), 15  
 ip2long() (in module iptools.ipv4), 15  
 ip2long() (in module iptools.ipv6), 20  
 ip2network() (in module iptools.ipv4), 16  
 IpRange (class in iptools), 11  
 IpRangeList (class in iptools), 9  
 iptools (module), 9  
 iptools.ipv4 (module), 14  
 iptools.ipv6 (module), 20  
 IPV4\_MAPPED (in module iptools.ipv6), 22  
 IPV6\_TO\_IPV4\_NETWORK (in module iptools.ipv6),  
22  
 IPV6\_TO\_IPV4\_RELAY (in module iptools.ipv4), 19

## L

LINK\_LOCAL (in module iptools.ipv4), 19  
 LINK\_LOCAL (in module iptools.ipv6), 22  
 LOCALHOST (in module iptools.ipv4), 19  
 LOCALHOST (in module iptools.ipv6), 23  
 long2ip() (in module iptools.ipv4), 16  
 long2ip() (in module iptools.ipv6), 21  
 LOOPBACK (in module iptools.ipv4), 19  
 LOOPBACK (in module iptools.ipv6), 23

## M

MAX\_IP (in module iptools.ipv4), 19  
 MAX\_IP (in module iptools.ipv6), 23  
 MIN\_IP (in module iptools.ipv4), 19  
 MIN\_IP (in module iptools.ipv6), 23  
 MULTICAST (in module iptools.ipv4), 19  
 MULTICAST (in module iptools.ipv6), 23  
 MULTICAST\_GLOBAL (in module iptools.ipv6), 23  
 MULTICAST\_INTERNETWORK (in module ip-  
tools.ipv4), 19  
 MULTICAST\_LOCAL (in module iptools.ipv4), 19  
 MULTICAST\_LOCAL (in module iptools.ipv6), 23  
 MULTICAST\_LOCAL\_DHCP (in module iptools.ipv6),  
23

MULTICAST\_LOCAL\_NODES (in module ip-  
tools.ipv6), [23](#)  
MULTICAST\_LOCAL\_ROUTERS (in module ip-  
tools.ipv6), [23](#)  
MULTICAST\_LOOPBACK (in module iptools.ipv6), [23](#)  
MULTICAST\_SITE (in module iptools.ipv6), [23](#)  
MULTICAST\_SITE\_DHCP (in module iptools.ipv6), [23](#)

## N

netmask2prefix() (in module iptools.ipv4), [16](#)

## P

PRIVATE\_NETWORK (in module iptools.ipv6), [23](#)  
PRIVATE\_NETWORK\_10 (in module iptools.ipv4), [19](#)  
PRIVATE\_NETWORK\_172\_16 (in module iptools.ipv4),  
[19](#)  
PRIVATE\_NETWORK\_192\_168 (in module ip-  
tools.ipv4), [20](#)

## R

RESERVED (in module iptools.ipv4), [20](#)

## S

SHARED\_ADDRESS\_SPACE (in module iptools.ipv4),  
[20](#)  
subnet2block() (in module iptools.ipv4), [17](#)

## T

TEREDO\_NETWORK (in module iptools.ipv6), [23](#)  
TEST\_NET\_1 (in module iptools.ipv4), [20](#)  
TEST\_NET\_2 (in module iptools.ipv4), [20](#)  
TEST\_NET\_3 (in module iptools.ipv4), [20](#)

## U

UNSPECIFIED\_ADDRESS (in module iptools.ipv6), [23](#)

## V

validate\_cidr() (in module iptools.ipv4), [17](#)  
validate\_cidr() (in module iptools.ipv6), [21](#)  
validate\_ip() (in module iptools.ipv4), [17](#)  
validate\_ip() (in module iptools.ipv6), [22](#)  
validate\_netmask() (in module iptools.ipv4), [18](#)  
validate\_subnet() (in module iptools.ipv4), [18](#)