Package ‘ipaddress’

May 12, 2020

Title Tidy IP Addresses
Version 0.3.0

Description Classes and functions for working with IP (Internet Protocol) addresses and networks, inspired by the Python 'ipaddress' module. Offers full support for both IPv4 and IPv6 (Internet Protocol versions 4 and 6) address spaces. It is specifically designed to work well with the 'tidyverse'.

License MIT + file LICENSE

URL https://davidchall.github.io/ipaddress,
https://github.com/davidchall/ipaddress

BugReports https://github.com/davidchall/ipaddress/issues

Depends R (>= 3.3.0)
Imports blob (>= 1.2.0), methods, Rcpp, rlang (>= 0.4.0), vctrs (>= 0.3.0)

Suggests crayon, knitr, pillar (>= 1.4.0), rmarkdown, testthat (>= 2.2.0)

LinkingTo AsioHeaders, BH, Rcpp
VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.1.0

SystemRequirements C++11

NeedsCompilation yes

Author David Hall [aut, cre] (<https://orcid.org/0000-0002-2193-0480>)

Maintainer David Hall <david.hall.physics@gmail.com>

Repository CRAN

Date/Publication 2020-05-12 18:00:02 UTC
**R topics documented:**

- address_in_network .................................................. 2
- ipv6-transition .......................................................... 3
- ip_address ................................................................. 4
- ip_interface ............................................................... 6
- ip_network ................................................................. 8
- ip_to_binary ............................................................... 10
- ip_to_bytes ............................................................... 11
- ip_to_hostname ........................................................... 12
- ip_to_integer ............................................................. 13
- is_ipv6 ...................................................................... 14
- is_reserved ............................................................... 15
- max_prefix_length ...................................................... 16
- netmask ................................................................. 17
- network_in_network .................................................... 18
- network_size ........................................................... 19
- sample ................................................................. 20
- sequence ............................................................... 21

**Index**  

| address_in_network | Network membership of addresses |

**Description**

These functions check whether an address falls within a network.

- `is_within()` performs a one-to-one matching between addresses and networks.
- `is_within_any()` checks if each address falls within any of the networks.

**Usage**

```r
is_within(address, network)
```

```r
is_within_any(address, network)
```

**Arguments**

- `address` An `ip_address` vector
- `network` An `ip_network` vector

**Value**

A logical vector
See Also

Use `is_subnet()` to check if an `ip_network` is within another `ip_network`.

Examples

```r
is_within(ip_address("192.168.2.6"), ip_network("192.168.2.0/28"))

is_within(ip_address("192.168.3.6"), ip_network("192.168.2.0/28"))

is_within_any(ip_address("192.168.3.6"), ip_network(c("192.168.2.0/28", "192.168.3.0/28")))
```

---

**ipv6-transition**

**IPv6 transition mechanisms**

### Description

There are multiple mechanisms designed to help with the transition from IPv4 to IPv6. These functions make it possible to extract the embedded IPv4 address from an IPv6 address.

### Usage

```r
is_ipv4_mapped(x)

is_6to4(x)

is_teredo(x)

extract_ipv4_mapped(x)

extract_6to4(x)

extract_teredo_server(x)

extract_teredo_client(x)
```

### Arguments

- `x` An `ip_address` vector

### Details

The IPv6 transition mechanisms are described in the IETF memos:

- IPv4-mapped: RFC 4291
- 6to4: RFC 3056
- Teredo: RFC 4380
Value

- `is_xxx()` functions return a logical vector
- `extract_xxx()` functions return an `ip_address` vector.

Examples

```r
# these examples show the reserved networks
is_ipv4_mapped(ip_network("::ffff:0.0.0.0/96"))

is_6to4(ip_network("2002::/16"))

is_teredo(ip_network("2001::/32"))

# these examples show embedded IPv4 addresses
extract_ipv4_mapped(ip_address("::ffff:192.168.0.1"))

extract_6to4(ip_address("2002:c000:0204::"))


```

---

**ip_address**

*Vector of IP addresses*

Description

- `ip_address()` constructs a vector of IP addresses.
- `is_ip_address()` checks if an object is of class `ip_address`.
- `as_ip_address()` casts an object to `ip_address`.

Usage

```r
ip_address(x = character())

is_ip_address(x)

as_ip_address(x)

## S3 method for class 'character'
as_ip_address(x)

## S3 method for class 'ip_interface'
as_ip_address(x)

## S3 method for class 'ip_address'
as.character(x, ...)
```
## S3 method for class 'ip_address'

format(x, ...)

### Arguments

- **x**
  - For `ip_address()`: A character vector of IP addresses, in dot-decimal notation (IPv4) or hexadecimal notation (IPv6)
  - For `is_ip_address()`: An object to test
  - For `as_ip_address()`: An object to cast
  - For `as.character()`: An `ip_address` vector

... Included for S3 generic consistency

### Details

An address in IPv4 space uses 32-bits. It is usually represented as 4 groups of 8 bits, each shown as decimal digits (e.g. 192.168.0.1). This is known as dot-decimal notation.

An address in IPv6 space uses 128-bits. It is usually represented as 8 groups of 16 bits, each shown as hexadecimal digits (e.g. 2001:0db8::8a2e:0370:7334). This representation can also be compressed by removing leading zeros and replacing consecutive groups of zeros with double-colon (e.g. 2001:db8:8a2e:0370:7334). Finally, there is also the dual representation. This expresses the final two groups as an IPv4 address (e.g. 2001:db8:8a2e:3.112.115.52).

The `ip_address()` constructor accepts a character vector of IP addresses in these two formats. It checks whether each string is a valid IPv4 or IPv6 address, and converts it to an `ip_address` object. If the input is invalid, a warning is emitted and NA is stored instead.

When casting an `ip_address` object back to a character vector using `as.character()`, IPv6 addresses are reduced to their compressed representation. A special case is IPv4-mapped IPv6 addresses (see `is_ipv4_mapped()`), which are returned in the dual representation (e.g. ::ffff:192.168.0.1).

Integers can be added to or subtracted from `ip_address` vectors. This class also supports bitwise operations: `!` (NOT), `&` (AND), `|` (OR) and `^` (XOR).

### Value

An S3 vector of class `ip_address`

### Examples

# supports IPv4 and IPv6 simultaneously
ip_address(c("192.168.0.1", "2001:db8::8a2e:370:7334"))

# validates inputs and replaces with NA
ip_address(c("255.255.255.256", "192.168.0.1/32"))

# addition of integers
ip_address("192.168.0.1") + -2:2

# bitwise NOT
!ip_address("192.168.0.1")
# bitwise AND
ip_address("192.168.0.1") & ip_address("255.0.0.255")

# bitwise OR
ip_address("192.168.0.0") | ip_address("255.0.0.255")

# bitwise XOR
ip_address("192.168.0.0") ^ ip_address("255.0.0.255")

---

**ip_interface**

*Vector of IP interfaces*

---

### Description

This hybrid class stores both the host address and the network it is on. 
-ip_interface() constructs a vector of IP interfaces. 
is_ip_interface() checks if an object is of class ip_interface. 
as_ip_interface() casts an object to ip_interface.

### Usage

ip_interface(...)

## Default S3 method:
ip_interface(x = character(), ...)

## S3 method for class 'ip_address'
ip_interface(address, prefix_length, ...)

is_ip_interface(x)

as_ip_interface(x)

## S3 method for class 'character'
as_ip_interface(x)

## S3 method for class 'ip_interface'
as.character(x, ...)

## S3 method for class 'ip_interface'
format(x, ...)

### Arguments

... Included for S3 generic consistency
**ip_interface**  

For `ip_interface()`: A character vector of IP interfaces, in CIDR notation (IPv4 or IPv6)

For `is_ip_interface()`: An object to test

For `as_ip_interface()`: An object to cast

For `as.character()`: An `ip_interface` vector

### address
An `ip_address` vector

### prefix_length
An integer vector

#### Details

Constructing an `ip_interface` vector is conceptually like constructing an `ip_network` vector, except the host bits are retained.

The `ip_interface` class inherits from the `ip_address` class. This means it can generally be used in places where an `ip_address` vector is expected. A few exceptions to this rule are:

- It does not support addition and subtraction of integers
- It does not support bitwise operations
- It cannot be compared to `ip_address` vectors

The `ip_interface` class additionally supports a few functions typically reserved for `ip_network` vectors: `prefix_length()`, `netmask()` and `hostmask()`.

For other purposes, you can extract the address and network components using `as_ip_address()` and `as_ip_network()`.

#### Value

An S3 vector of class `ip_interface`

#### Examples

```r
# construct from character vector
ip_interface(c("192.168.0.1/10", "2001:db8:c3::abcd/45"))

# construct from address + prefix length objects
ip_interface(ip_address(c("192.168.0.1", "2001:db8:c3::abcd")), c(10L, 45L))

# extract IP address
x <- ip_interface(c("192.168.0.1/10", "2001:db8:c3::abcd/45"))
as_ip_address(x)

# extract IP network (with host bits masked)
as_ip_network(x)
```
ip_network

Vector of IP networks

Description

ip_network() constructs a vector of IP networks.

is_ip_network() checks if an object is of class ip_network.

as_ip_network() casts an object to ip_network.

Usage

ip_network(...)

## Default S3 method:
ip_network(x = character(), strict = TRUE, ...)

## S3 method for class 'ip_address'
ip_network(address, prefix_length, strict = TRUE, ...)

is_ip_network(x)

as_ip_network(x)

## S3 method for class 'character'
as_ip_network(x)

## S3 method for class 'ip_interface'
as_ip_network(x)

## S3 method for class 'ip_network'
as.character(x, ...)

## S3 method for class 'ip_network'
format(x, ...)

Arguments

...  Included for S3 generic consistency

x  
- For ip_network(): A character vector of IP networks, in CIDR notation (IPv4 or IPv6)
- For is_ip_network(): An object to test
- For as_ip_network(): An object to cast
- For as.character(): An ip_network vector

strict  If TRUE (the default) and the input has host bits set, then a warning is emitted and NA is returned. If FALSE, the host bits are set to zero and a valid IP network
An IP network corresponds to a contiguous range of IP addresses (also known as an IP block). CIDR notation represents an IP network as the routing prefix address (which denotes the start of the range) and the prefix length (which indicates the size of the range) separated by a forward slash. For example, 192.168.0.0/24 represents addresses from 192.168.0.0 to 192.168.0.255.

The prefix length indicates the number of bits reserved by the routing prefix. This means that larger prefix lengths indicate smaller networks. The maximum prefix length is 32 for IPv4 and 128 for IPv6. These would correspond to an IP network of a single IP address.

The \texttt{ip_network()} constructor accepts a character vector of IP networks in CIDR notation. It checks whether each string is a valid IPv4 or IPv6 network, and converts it to an \texttt{ip_network} object. If the input is invalid, a warning is emitted and \texttt{NA} is stored instead.

An alternative constructor accepts an \texttt{ip_address} vector and an integer vector containing the network address and prefix length, respectively.

When casting an \texttt{ip_network} object back to a character vector using \texttt{as.character()}, IPv6 addresses are reduced to their compressed representation.

\textbf{Value}

An S3 vector of class \texttt{ip_network}

\textbf{See Also}

\texttt{prefix_length()}, \texttt{network_address()}, \texttt{netmask()}, \texttt{hostmask()}

\textbf{Examples}

\begin{verbatim}
# construct from character vector
ip_network(c("192.168.0.0/24", "2001:db8::/48"))

# construct from address + prefix length objects
ip_network(ip_address(c("192.168.0.0", "2001:db8::")), c(24L, 48L))

# validates inputs and replaces with NA
ip_network(c("192.168.0.0/33", "192.168.0.0"))

# IP networks should not have any host bits set
ip_network("192.168.0.1/22")

# but we can mask the host bits if desired
ip_network("192.168.0.1/22", strict = FALSE)
\end{verbatim}
ip_to_binary

Represent address as binary

Description

Encode or decode an ip_address as a binary bit string.

Usage

ip_to_binary(x)

binary_to_ip(x)

Arguments

x  
• For ip_to_binary(): An ip_address vector
• For binary_to_ip(): A character vector containing only 0 and 1 characters

Details

The bits are stored in network order (also known as big-endian order), which is part of the IP standard.

IPv4 addresses use 32 bits, IPv6 addresses use 128 bits, and missing values are encoded as NA.

Value

• For ip_to_binary(): A character vector
• For binary_to_ip(): An ip_address vector

See Also

• ip_to_integer() and integer_to_ip()
• ip_to_bytes() and bytes_to_ip()

Examples

x <- ip_address(c("192.168.0.1", "2001:db8:8a2e:370:7334", NA))

ip_to_binary(x)

binary_to_ip(ip_to_binary(x))
### ip_to_bytes

Represent address as raw bytes

**Description**

Encode or decode an `ip_address` as a vector of raw bytes.

**Usage**

- `ip_to_bytes(x)`
- `bytes_to_ip(x)`

**Arguments**

- `x`  
  - For `ip_to_bytes()`: An `ip_address` vector
  - For `bytes_to_ip()`: A `blob::blob` vector

**Details**

The bytes are stored in network order (also known as big-endian order), which is part of the IP standard.

IPv4 addresses use 4 bytes, IPv6 addresses use 16 bytes, and missing values are encoded as NULL.

**Value**

- For `ip_to_bytes()`: A `blob::blob` vector
- For `bytes_to_ip()`: An `ip_address` vector

**See Also**

- `ip_to_integer()` and `integer_to_ip()`
- `ip_to_binary()` and `binary_to_ip()`

**Examples**

```r
x <- ip_address(c("192.168.0.1", "2001:db8::8a2e:370:7334", NA))
ip_to_bytes(x)

bytes_to_ip(ip_to_bytes(x))
```
ip_to_hostname  

Translate address to/from hostname

Description

Perform reverse and forward DNS resolution

Usage

ip_to_hostname(x, multiple = FALSE)
hostname_to_ip(x, multiple = FALSE)

Arguments

x
  • For ip_to_hostname(): An ip_address vector
  • For hostname_to_ip(): A character vector of hostnames

multiple
  A logical scalar indicating if all resolved endpoints are returned, or just the first endpoint (the default). This determines whether a vector or list of vectors is returned.

Details

These functions require an internet connection. Before processing the input vector, we first check that a known hostname can be resolved. If this fails, an error is raised.

If DNS lookup cannot resolve an input, then NA is returned for that input. If an error occurs during DNS lookup, then a warning is emitted and NA is returned for that input.

DNS resolution performs a many-to-many mapping between IP addresses and hostnames. For this reason, these two functions can potentially return multiple values for each element of the input vector. The multiple argument control whether all values are returned (a vector for each input), or just the first value (a scalar for each input).

Value

• For ip_to_hostname(): A character vector (multiple = FALSE) or a list of character vectors (multiple = TRUE)
• For hostname_to_ip(): A ip_address vector (multiple = FALSE) or a list of ip_address vectors (multiple = TRUE)

See Also

The base function nsl() provides forward DNS resolution to IPv4 addresses, but only on Unix-like systems.
ip_to_integer

Examples

```r
## Not run:
hostname_to_ip("r-project.org")

ip_toHostname(hostname_to_ip("r-project.org"))

## End(Not run)
```

### ip_to_integer

**Represent address as integer**

#### Description

Encode or decode an `ip_address` as an integer.

**Note:** The result is a character vector (see below for why). This can be converted using `as.numeric()` for IPv4 addresses.

#### Usage

```r
ip_to_integer(x)

integer_to_ip(x, is_ipv6 = NULL)
```

#### Arguments

- `x`: 
  - For `ip_to_integer()`: An `ip_address` vector
  - For `integer_to_ip()`: A character vector

- `is_ipv6`: A logical vector indicating whether to construct an IPv4 or IPv6 address. If `NULL` (the default), then integers less than $2^{32}$ will construct an IPv4 address and anything larger will construct an IPv6 address.

#### Details

It is common to represent an IP address as an integer, by reinterpreting the bit sequence as a big-endian unsigned integer. This means IPv4 and IPv6 addresses can be represented by 32-bit and 128-bit unsigned integers. In this way, the IPv4 addresses 0.0.0.0 and 255.255.255.255 would be represented as 0 and 4,294,967,295.

#### Value

- For `ip_to_integer()`: A character vector
- For `integer_to_ip()`: An `ip_address` vector
Why is a character vector returned?

Base R provides two data types that can store integers: integer and double (also known as numeric). The former is a fixed-point data format (32-bit) and the latter is a floating-point data format (64-bit). Both types are signed, and R does not offer unsigned variants.

For the purpose of storing IP addresses as integers, we need to be able to store a large range of positive integers without losing integer precision. Under these circumstances, the integer type can store integers up to \(2^{31} - 1\) and the double type can store integers up to \(2^{53}\). However, for IPv4 and IPv6 addresses we need to store integers up to \(2^{32} - 1\) and \(2^{128} - 1\), respectively. This means an IPv4 address can be stored in a double, but an IPv6 address cannot be stored in either R data type.

Although the integer representation of an IPv6 address cannot be stored in a numeric R data type, it can be stored as a character string instead. This allows the integer to be written to disk or used by other software that does support 128-bit unsigned integers. To treat IPv4 and IPv6 equally, ip_to_integer() will always return a character vector. With IPv4 addresses, this output can be converted to a double vector using as.double() or as.numeric(). With IPv6 addresses, this conversion loses the distinction between individual addresses because integer precision is lost.

See Also

- ip_to_bytes() and bytes_to_ip()
- ip_to_binary() and binary_to_ip()

Examples

```r
x <- ip_address(c("192.168.0.1", "2001:db8::8a2e:370:7334", NA))
ip_to_integer(x)
integer_to_ip(ip_to_integer(x))

# with IPv4 only, we can use numeric data type
as.numeric(ip_to_integer(ip_address("192.168.0.1")))
integer_to_ip(as.character(3232235521))
```

Description

Version of the address space

Usage

```r
is_ipv4(x)
is_ipv6(x)
```
Arguments

x  An ip_address or ip_network vector

Value

A logical vector

See Also

max_prefix_length()

Examples

ip <- ip_address(c("192.168.0.1", "2001:db8::7334"))

is_ipv4(ip)

is_ipv6(ip)

is_reserved  Reserved addresses

Description

Check if an address or network is reserved for special use. A network is considered reserved if both the network_address() and broadcast_address() are reserved.

Usage

is_multicast(x)

is_unspecified(x)

is_loopback(x)

is_link_local(x)

Arguments

x  An ip_address or ip_network vector

Details

These special use addresses are documented in IETF documents RFC 5735 (for IPv4) and RFC 4291 (for IPv6).

Value

A logical vector
max_prefix_length

See Also

Addresses reserved by IPv6 transition mechanisms can be identified by functions described in `ipv6-transition`.

Examples

```r
# these examples show the reserved networks
is_multicast(ip_network(c("224.0.0.0/4", "ff00::/8")))

is_unspecified(ip_network(c("0.0.0.0/32", ":/128")))

is_loopback(ip_network(c("127.0.0.0/8", ":/128")))

is_link_local(ip_network(c("169.254.0.0/16", "fe80::/10")))
```

max_prefix_length

Size of the address space

Description

The total number of bits available in the address space. IPv4 uses 32-bit addresses and IPv6 uses 128-bit addresses.

Usage

```r
max_prefix_length(x)
```

Arguments

- `x` An `ip_address` or `ip_network` vector

Value

An integer vector

See Also

`is_ipv4()`, `is_ipv6()`, `prefix_length()`

Examples

```r
x <- ip_address(c("192.168.0.1", "2001:db8::7334"))

max_prefix_length(x)
```
Description

These functions yield equivalent representations of the network mask.

Usage

prefix_length(x)

netmask(...)

hostmask(...)

## S3 method for class 'ip_network'
netmask(x, ...)

## S3 method for class 'ip_network'
hostmask(x, ...)

## S3 method for class 'ip_interface'
netmask(x, ...)

## S3 method for class 'ip_interface'
hostmask(x, ...)

## Default S3 method:
netmask(prefix_length = integer(), is_ipv6 = logical(), ...)

## Default S3 method:
hostmask(prefix_length = integer(), is_ipv6 = logical(), ...)

Arguments

x An ip_network or ip_interface vector

... Arguments to be passed to other methods

prefix_length An integer vector

is_ipv6 A logical vector

Value

• prefix_length() returns an integer vector

• netmask() and hostmask() return an ip_address vector
network_in_network

See Also

max_prefix_length()

Examples

x <- ip_network(c("192.168.0.0/22", "2001:db00::0/26"))

prefix_length(x)

netmask(x)

hostmask(x)

netmask(c(22L, 26L), c(FALSE, TRUE))

hostmask(c(22L, 26L), c(FALSE, TRUE))

network_in_network  Network membership of other networks

Description

overlaps() checks for any overlap between two networks; is_subnet() and is_supernet() check if one network is a true subnet or supernet of another network.

Usage

overlaps(network, other)

is_subnet(network, other)

is_supernet(network, other)

Arguments

network       An ip_network vector
other         An ip_network vector

Value

A logical vector

See Also

Use is_within() to check if an ip_address is within an ip_network.
Examples

```r
net1 <- ip_network("192.168.1.128/30")
net2 <- ip_network("192.168.1.0/24")

overlaps(net1, net2)
is_subnet(net1, net2)
is_supernet(net1, net2)
```

---

### network_size

**Network size**

| Description |
|-------------|------------------------------------------------|
| network_address() and broadcast_address() yield the first and last addresses of the network; num_addresses() gives the total number of addresses in the network. |

### Usage

```r
network_address(x)
broadcast_address(x)
um_addresses(x)
```

### Arguments

- `x` An ip_network vector

### Details

The broadcast address is a special address at which any host connected to the network can receive messages. That is, packets sent to this address are received by all hosts on the network. In IPv4, the last address of a network is the broadcast address. Although IPv6 does not follow this approach to broadcast addresses, the broadcast_address() function still returns the last address of the network.

### Value

- network_address() and broadcast_address() return an ip_address vector
- num_addresses() returns a numeric vector

### See Also

Use `seq.ip_network()` to generate all addresses in a network.
Examples

```r
x <- ip_network(c("192.168.0.0/22", "2001:db8::/33"))

network_address(x)
broadcast_address(x)
um_addresses(x)
```

---

**Sample random addresses**

**Description**

`sample_ipv4()` and `sample_ipv6()` sample from the entire address space; `sample_network()` samples from a specific network.

**Usage**

```r
sample_ipv4(size, replace = FALSE)
sample_ipv6(size, replace = FALSE)
sample_network(x, size, replace = FALSE)
```

**Arguments**

- `size` Integer specifying the number of addresses to return
- `replace` Should sampling be with replacement?
- `x` An `ip_network` scalar

**Value**

An `ip_address` vector

**See Also**

Use `seq.ip_network()` to generate all addresses in a network.

**Examples**

```r
sample_ipv4(5)
sample_ipv6(5)
sample_network(ip_network("192.168.0.0/16"), 5)
sample_network(ip_network("2001:db8::/48"), 5)
```
**Description**

seq() returns *all* hosts

hosts() returns only *usable* hosts

**Usage**

```r
## S3 method for class 'ip_network'
seq(x, ...)

hosts(x)
```

**Arguments**

- `x` An `ip_network` scalar
- `...` Included for generic consistency

**Details**

In IPv4, the unusable hosts are the network address and the broadcast address (i.e. the first and last addresses in the network). In IPv6, the only unusable host is the subnet router anycast address (i.e. the first address in the network).

For networks with a prefix length of 31 (for IPv4) or 127 (for IPv6), the unusable hosts are included in the results of `hosts()`.

The `ipaddress` package does not support long vectors (i.e. vectors with more than $2^{31} - 1$ elements). As a result, these two functions do not support networks larger than this size. This corresponds to prefix lengths less than 2 (for IPv4) or 98 (for IPv6). However, you might find that machine memory imposes stricter limitations.

**Value**

An `ip_address` vector

**See Also**

Use `network_address()` and `broadcast_address()` to get the first and last address of a network.

Use `sample_network()` to randomly sample addresses from a network.
Examples

seq(ip_network("192.168.0.0/30"))

seq(ip_network("2001:db8::/126"))

hosts(ip_network("192.168.0.0/30"))

hosts(ip_network("2001:db8::/126"))
Index

address_in_network, 2
as.character.ip_address (ip_address), 4
as.character.ip_interface
  (ip_interface), 6
as.character.ip_network (ip_network), 8
as.double(), 14
as.numeric(), 13, 14
as_ip_address (ip_address), 4
as_ip_interface (ip_interface), 6
as_ip_network (ip_network), 8
as_ip_network(), 7
binary_to_ip (ip_to_binary), 10
binary_to_ip(), 11
blob::blob, 11
broadcast_address (network_size), 19
broadcast_address(), 21
bytes_to_ip (ip_to_bytes), 11
bytes_to_ip(), 10, 14
character, 14
double, 14
extract_6to4 (ipv6-transition), 3
extract_ipv4_mapped (ipv6-transition), 3
extract_teredo_client
  (ipv6-transition), 3
extract_teredo_server
  (ipv6-transition), 3
format.ip_address (ip_address), 4
format.ip_interface (ip_interface), 6
format.ip_network (ip_network), 8
hostmask (netmask), 17
hostmask(), 7, 9
hostname_to_ip (ip_to_hostname), 12
hosts (sequence), 21
integer, 14
integer_to_ip (ip_to_integer), 13
integer_to_ip(), 10, 11
ip_address, 2–4, 4, 7, 9–13, 15–21
ip_interface, 6, 17
ip_interface(), 9
ip_network, 2, 3, 7, 8, 15–21
ip_to_binary, 10
ip_to_binary(), 11, 14
ip_to_bytes, 11
ip_to_bytes(), 10, 14
ip_to_hostname, 12
ip_to_integer, 13
ip_to_integer(), 10, 11
ipv6-transition, 3, 16
is_6to4 (ipv6-transition), 3
is_ip_address (ip_address), 4
is_ip_interface (ip_interface), 6
is_ip_network (ip_network), 8
is_ipv4 (is_ipv6), 14
is_ipv4(), 16
is_ipv4_mapped (ipv6-transition), 3
is_ipv4_mapped(), 5
is_ipv6, 14
is_ipv6(), 16
is_link_local (is_reserved), 15
is_loopback (is_reserved), 15
is_multicast (is_reserved), 15
is_reserved, 15
is_subnet (network_in_network), 18
is_subnet(), 3
is_super_net (network_in_network), 18
is_teredo (ipv6-transition), 3
is_unspecified (is_reserved), 15
is_within (address_in_network), 2
is_within(), 18
is_within_any (address_in_network), 2
long vectors, 21
max_prefix_length, 16
max_prefix_length(), 15, 18

netmask, 17
netmask(), 7, 9
network_address (network_size), 19
network_address(), 9, 21
network_in_network, 18
network_size, 19
num_addresses (network_size), 19
numeric, 14

overlaps (network_in_network), 18

prefix_length (netmask), 17
prefix_length(), 7, 9, 16

sample, 20
sample_ipv4 (sample), 20
sample_ipv6 (sample), 20
sample_network (sample), 20
sample_network(), 21
seq.ip_network (sequence), 21
seq.ip_network(), 19, 20
sequence, 21