

# Package ‘ipaddress’

April 4, 2023

**Title** Data Analysis for IP Addresses and Networks

**Version** 1.0.1

**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 (>= 3.5)

**Imports** cli (>= 3.4.0), Rcpp, rlang (>= 1.0.3), vctrs

**Suggests** bignum, blob, crayon, dplyr, fuzzyjoin, knitr, pillar, rmarkdown, testthat, tibble, tidyr

**LinkingTo** AsioHeaders, Rcpp

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 7.2.3

**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** 2023-04-04 05:40:02 UTC

**R topics documented:**

address_in_network . . . . .	2
collapse_networks . . . . .	3
common_network . . . . .	4
country_networks . . . . .	5
exclude_networks . . . . .	6
iana_ipv4 . . . . .	6
iana_ipv6 . . . . .	7
ipv6-transition . . . . .	8
ip_address . . . . .	9
ip_cast . . . . .	10
ip_format . . . . .	11
ip_interface . . . . .	12
ip_network . . . . .	13
ip_operators . . . . .	15
ip_test . . . . .	15
ip_to_binary . . . . .	16
ip_to_bytes . . . . .	17
ip_to_hex . . . . .	18
ip_to_hostname . . . . .	19
ip_to_integer . . . . .	20
is_ipv6 . . . . .	21
is_reserved . . . . .	22
max_prefix_length . . . . .	23
netmask . . . . .	24
network_in_network . . . . .	25
network_size . . . . .	26
reverse_pointer . . . . .	27
sample . . . . .	28
sequence . . . . .	29
summarize_address_range . . . . .	30
traverse_hierarchy . . . . .	31
<b>Index</b>	<b>32</b>

---

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**

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

```
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**

```
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")))
```

---

collapse\_networks       *Collapse contiguous and overlapping networks*

---

**Description**

Given a vector of networks, this returns the minimal set of networks required to represent the same range of addresses.

**Usage**

```
collapse_networks(network)
```

**Arguments**

network           An [ip\\_network](#) vector

**Value**

An [ip\\_network](#) vector (potentially shorter than the input)

**See Also**

[exclude\\_networks\(\)](#)

## Examples

```
collapse_networks(ip_network(c("192.168.0.0/24", "192.168.1.0/24")))
```

---

common_network	<i>Find the common network of two addresses</i>
----------------	---

---

## Description

Returns the smallest network that contains both addresses.

This can construct a network from its first and last addresses. However, if the address range does not match the network boundaries, then the result extends beyond the original address range. Use [summarize\\_address\\_range\(\)](#) to receive a list of networks that exactly match the address range.

## Usage

```
common_network(address1, address2)
```

## Arguments

address1	An <a href="#">ip_address</a> vector
address2	An <a href="#">ip_address</a> vector

## Value

An [ip\\_network](#) vector

## See Also

[summarize\\_address\\_range\(\)](#)

## Examples

```
# address range matches network boundaries
common_network(ip_address("192.168.0.0"), ip_address("192.168.0.15"))

# address range does not match network boundaries
common_network(ip_address("192.167.255.255"), ip_address("192.168.0.16"))
```

---

country_networks	<i>Country-level IP networks</i>
------------------	----------------------------------

---

### Description

Retrieve lists of IP networks registered to specific countries.

### Usage

```
country_networks(country, ..., collapse = TRUE)
```

### Arguments

country	Character vector of two-letter country codes (ISO 3166-1 alpha-2)
...	These dots are for future extensions and must be empty.
collapse	If TRUE (the default), contiguous networks are collapsed. See <a href="#">collapse_networks()</a> .

### Details

This function requires an internet connection to download network lists.

### Value

A data frame with 2 variables:

- country: A character vector
- network: A list of [ip\\_network](#) vectors

Each row represents a single country associated with a vector of IP networks.

### Source

<https://www.iwik.org/ipcountry/> (updated daily)

### Examples

```
## Not run:
country_networks(c("GB", "US"))

country_networks(c("GB", "US"), collapse = FALSE)

# retrieve networks for a single country
country_networks("TO")$networks[[1]]

# expand networks for multiple countries
tidyr::unchop(country_networks(c("GB", "US")), networks)

## End(Not run)
```

---

exclude_networks	<i>Remove networks from others</i>
------------------	------------------------------------

---

### Description

exclude\_networks() takes lists of networks to include and exclude. It then calculates the address ranges that are included but not excluded (similar to [setdiff\(\)](#)), and finally returns the minimal set of networks needed to describe the remaining address ranges.

### Usage

```
exclude_networks(include, exclude)
```

### Arguments

include	An <a href="#">ip_network</a> vector
exclude	An <a href="#">ip_network</a> vector

### Value

An [ip\\_network](#) vector

### See Also

[collapse\\_networks\(\)](#), [setdiff\(\)](#)

### Examples

```
exclude_networks(ip_network("192.0.2.0/28"), ip_network("192.0.2.1/32"))  
exclude_networks(ip_network("192.0.2.0/28"), ip_network("192.0.2.15/32"))
```

---

iana_ipv4	<i>IPv4 address space allocation</i>
-----------	--------------------------------------

---

### Description

A dataset containing the registry of allocated blocks in IPv4 address space.

### Usage

```
iana_ipv4
```

**Format**

A data frame with 121 rows and 3 variables:

**network** Address block (an [ip\\_network](#) vector)

**allocation** There are three types of allocation:

- reserved
- managed by regional Internet registry (RIR)
- assigned to organization

**label** The RIR, organization or purpose for reservation

**Note**

Last updated 2022-12-12

**Source**

<https://www.iana.org/assignments/ipv4-address-space>

**See Also**

[is\\_reserved\(\)](#)

**Examples**

```
iana_ipv4
```

---

iana\_ipv6

*IPv6 address space allocation*

---

**Description**

A dataset containing the registry of allocated blocks in IPv6 address space.

**Usage**

```
iana_ipv6
```

**Format**

A data frame with 47 rows and 3 variables:

**network** Address block (an [ip\\_network](#) vector)

**allocation** There are two types of allocation:

- reserved
- managed by regional Internet registry (RIR)

**label** The RIR or purpose for reservation

**Note**

Last updated 2020-08-18

**Source**

<https://www.iana.org/assignments/ipv6-address-space>

<https://www.iana.org/assignments/ipv6-unicast-address-assignments>

**See Also**

[is\\_reserved\(\)](#)

**Examples**

```
iana_ipv6
```

---

ipv6-transition	<i>IPv6 transition mechanisms</i>
-----------------	-----------------------------------

---

**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**

```
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(): A logical vector
- extract\_xxx(): An [ip\\_address](#) vector

**Examples**

```
# 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::"))

extract_teredo_server(ip_address("2001:0000:4136:e378:8000:63bf:3fff:fdd2"))

extract_teredo_client(ip_address("2001:0000:4136:e378:8000:63bf:3fff:fdd2"))
```

---

ip_address	<i>Vector of IP addresses</i>
------------	-------------------------------

---

**Description**

Construct a vector of IP addresses.

**Usage**

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

**Arguments**

x	A character vector of IP addresses, in dot-decimal notation (IPv4) or hexadecimal notation (IPv6)
---	---

## 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. 2001:0db8:85a3:0000:0000:8a2e:0370:7334). This representation can also be compressed by removing leading zeros and replacing consecutive groups of zeros with double-colon (e.g. 2001:db8:85a3::8a2e:370:7334). Finally, there is also the dual representation. This expresses the final two groups as an IPv4 address (e.g. 2001:db8:85a3::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`). `ip_address` vectors support a number of [operators](#).

## Value

An S3 vector of class `ip_address`

## See Also

[ip\\_operators](#), `vignette("ip-data")`

## 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"))
```

---

ip\_cast

*Cast to IP vector*

---

## Description

Methods for converting character vectors and `ip_interface` vectors to `ip_address` and `ip_network` vectors.

## Usage

```
as_ip_address(x)
```

```
as_ip_interface(x)
```

```
as_ip_network(x)
```

**Arguments**

x                    An object to cast

**Value**

- as\_ip\_address(): An [ip\\_address](#) vector
- as\_ip\_network(): An [ip\\_network](#) vector
- as\_ip\_interface(): An [ip\\_interface](#) vector

**Examples**

```
as_ip_address(ip_interface("192.168.0.1/10"))
```

```
as_ip_network(ip_interface("192.168.0.1/10"))
```

---

ip\_format                    *Format IP vector*

---

**Description**

Format vector of IP data using compressed or exploded representation.

**Usage**

```
## S3 method for class 'ip_address'
format(x, ..., exploded = FALSE)
```

```
## S3 method for class 'ip_interface'
format(x, ..., exploded = FALSE)
```

```
## S3 method for class 'ip_network'
format(x, ..., exploded = FALSE)
```

**Arguments**

x                    An object to format

...                  These dots are for future extensions and must be empty.

exploded            Logical scalar. Should IPv6 addresses display leading zeros? (default: FALSE)

**Value**

A character vector

**Examples**

```
format(ip_address("2001:db8::8a2e:370:7334"))
```

```
format(ip_address("2001:db8::8a2e:370:7334"), exploded = TRUE)
```

---

ip_interface	<i>Vector of IP interfaces</i>
--------------	--------------------------------

---

### Description

Construct a vector of IP interfaces.

This hybrid class stores both the host address and the network it is on.

### Usage

```
ip_interface(...)

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

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

### Arguments

...	Arguments passed to methods.
x	A character vector of IP interfaces, in CIDR notation (IPv4 or IPv6)
address	An <a href="#">ip_address</a> 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\(\)](#).

When comparing and sorting `ip_interface` vectors, the network is compared before the host address.

### Value

An S3 vector of class `ip_interface`

**See Also**

```
vignette("ip-data")
```

**Examples**

```
# 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)
```

---

<code>ip_network</code>	<i>Vector of IP networks</i>
-------------------------	------------------------------

---

**Description**

Construct a vector of IP networks.

**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)
```

**Arguments**

<code>...</code>	Arguments passed to methods.
<code>x</code>	A character vector of IP networks, in CIDR notation (IPv4 or IPv6)
<code>strict</code>	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 is returned. If you need to retain the host bits, consider using <a href="#">ip_interface()</a> instead.
<code>address</code>	An <a href="#">ip_address</a> vector
<code>prefix_length</code>	An integer vector

## Details

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 `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 `ip_network` object. If the input is invalid, a warning is emitted and NA is stored instead.

An alternative constructor accepts an `ip_address` vector and an integer vector containing the network address and prefix length, respectively.

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

When comparing and sorting `ip_network` vectors, the network address is compared before the prefix length.

## Value

An S3 vector of class `ip_network`

## See Also

[prefix\\_length\(\)](#), [network\\_address\(\)](#), [netmask\(\)](#), [hostmask\(\)](#)  
[vignette\("ip-data"\)](#)

## Examples

```
# construct from character vector
ip_network(c("192.168.0.0/24", "2001:db8::/48"))

# 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)

# construct from address + prefix length
ip_network(ip_address("192.168.0.0"), 24L)

# construct from address + netmask
ip_network(ip_address("192.168.0.0"), prefix_length(ip_address("255.255.255.0")))

# construct from address + hostmask
ip_network(ip_address("192.168.0.0"), prefix_length(ip_address("0.0.0.255")))
```

---

ip\_operators

*Operators for IP addresses*

---

### Description

`ip_address` vectors support the following operators:

- bitwise logic operators: ! (NOT), & (AND), | (OR), ^ (XOR)
- bitwise shift operators: %<<% (left shift), %>>% (right shift)
- arithmetic operators: + (addition), - (subtraction)

### Examples

```
# use ip_to_binary() to understand these examples better

# 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")

# bitwise shift left
ip_address("192.168.0.1") %<<% 1

# bitwise shift right
ip_address("192.168.0.1") %>>% 1

# addition of integers
ip_address("192.168.0.1") + 10

# subtraction of integers
ip_address("192.168.0.1") - 10
```

---

ip\_test

*Test for IP vector*

---

### Description

Check if an object is an `ip_address`, `ip_network` or `ip_interface` vector.

**Usage**

```
is_ip_address(x)
```

```
is_ip_interface(x)
```

```
is_ip_network(x)
```

**Arguments**

x                    An object to test

**Value**

A logical scalar

**Examples**

```
is_ip_address(ip_address("192.168.0.1"))
is_ip_interface(ip_interface("192.168.0.1/10"))
is_ip_network(ip_network("192.168.0.0/24"))
```

---

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
- ip\_to\_binary(): An [ip\\_address](#) vector
  - 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**

- ip\_to\_binary(): A character vector
- binary\_to\_ip(): An [ip\\_address](#) vector

**See Also**

Other address representations: [ip\\_to\\_bytes\(\)](#), [ip\\_to\\_hex\(\)](#), [ip\\_to\\_integer\(\)](#)

**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 list of raw bytes.

**Usage**

```
ip_to_bytes(x)

bytes_to_ip(x)
```

**Arguments**

- x
- ip\_to\_bytes(): An [ip\\_address](#) vector
  - bytes\_to\_ip(): A list of raw vectors or a [blob::blob](#) object

**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**

- ip\_to\_bytes(): A list of raw vectors
- bytes\_to\_ip(): An [ip\\_address](#) vector

**See Also**

Use [blob::as\\_blob\(\)](#) to cast result to a blob object

Other address representations: [ip\\_to\\_binary\(\)](#), [ip\\_to\\_hex\(\)](#), [ip\\_to\\_integer\(\)](#)

**Examples**

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

bytes <- list(
  as.raw(c(0xc0, 0xa8, 0x00, 0x01)),
  as.raw(c(
    0x20, 0x01, 0x0d, 0xb8, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x8a, 0x2e, 0x03, 0x70, 0x73, 0x34
  )),
  NULL
)
bytes_to_ip(bytes)
```

---

**ip\_to\_hex***Represent address as hexadecimal*

---

**Description**

Encode or decode an [ip\\_address](#) as a hexadecimal string.

**Usage**

```
ip_to_hex(x)
```

```
hex_to_ip(x, is_ipv6 = NULL)
```

**Arguments**

<code>x</code>	<ul style="list-style-type: none"> <li>• <code>ip_to_hex()</code>: An <a href="#">ip_address</a> vector</li> <li>• <code>hex_to_ip()</code>: A character vector containing hexadecimal strings</li> </ul>
<code>is_ipv6</code>	A logical vector indicating whether to construct an IPv4 or IPv6 address. If NULL (the default), then IPv4 is preferred but an IPv6 address is constructed when <code>x</code> is too large for the IPv4 address space.

**Value**

- `ip_to_hex()`: A character vector
- `hex_to_ip()`: An [ip\\_address](#) vector

**See Also**

Other address representations: [ip\\_to\\_binary\(\)](#), [ip\\_to\\_bytes\(\)](#), [ip\\_to\\_integer\(\)](#)

### Examples

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

hex_to_ip(ip_to_hex(x))
```

---

ip\_to\_hostname

*Translate address to/from hostname*

---

### Description

Perform reverse and forward DNS resolution.

**Note:** These functions are significantly slower than others in the ipaddress package.

### Usage

```
ip_to_hostname(x)

ip_to_hostname_all(x)

hostname_to_ip(x)

hostname_to_ip_all(x)
```

### Arguments

- x
- ip\_to\_hostname(): An [ip\\_address](#) vector
  - hostname\_to\_ip(): A character vector of hostnames

### 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, there are two versions of each function. The regular version returns just the first value and the `_all()` suffix version returns all values.

### Value

- ip\_to\_hostname(): A character vector
- ip\_to\_hostname\_all(): A list of character vectors
- hostname\_to\_ip(): An [ip\\_address](#) vector
- hostname\_to\_ip\_all(): A list of [ip\\_address](#) vectors

**See Also**

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

**Examples**

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

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

## End(Not run)
```

---

<code>ip_to_integer</code>	<i>Represent address as integer</i>
----------------------------	-------------------------------------

---

**Description**

Encode or decode an `ip_address` as an integer.

**Usage**

```
ip_to_integer(x)

integer_to_ip(x, is_ipv6 = NULL)
```

**Arguments**

<code>x</code>	<ul style="list-style-type: none"> <li>• <code>ip_to_integer()</code>: An <code>ip_address</code> vector</li> <li>• <code>integer_to_ip()</code>: A <code>bignum::biginteger</code> vector</li> </ul>
<code>is_ipv6</code>	A logical vector indicating whether to construct an IPv4 or IPv6 address. If <code>NULL</code> (the default), then IPv4 is preferred but an IPv6 address is constructed when <code>x</code> is too large for the IPv4 address space.

**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.

The numeric data types within base R (`integer` and `double`) have insufficient precision to cover the IPv6 address space. Instead we return a `bignum::biginteger` vector, which supports arbitrary precision integers.

**Value**

- `ip_to_integer()`: A `bignum::biginteger` vector
- `integer_to_ip()`: An `ip_address` vector

**See Also**

Other address representations: [ip\\_to\\_binary\(\)](#), [ip\\_to\\_bytes\(\)](#), [ip\\_to\\_hex\(\)](#)

**Examples**

```
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(3232235521)
```

---

is\_ipv6

*Version of the address space*

---

**Description**

Version of the address space

**Usage**

```
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	<i>Reserved addresses</i>
-------------	---------------------------

---

### Description

Most of these functions check if an address or network is reserved for special use. The exception is `is_global()`, which checks if it is *not* reserved.

A network is considered reserved if both the `network_address()` and `broadcast_address()` are reserved.

### Usage

`is_private(x)`

`is_global(x)`

`is_multicast(x)`

`is_unspecified(x)`

`is_reserved(x)`

`is_loopback(x)`

`is_link_local(x)`

`is_site_local(x)`

### Arguments

`x`                    An [ip\\_address](#) or [ip\\_network](#) vector

### Details

Here are hyperlinks to the IANA registries of allocated address space:

- [IPv4: allocations, special purpose](#)
- [IPv6: allocations, special purpose](#)

### Value

A logical vector

### See Also

Addresses reserved by IPv6 transition mechanisms can be identified by functions described in [ipv6-transition](#).

**Examples**

```

is_private(ip_network(c("192.168.0.0/16", "2001:db8::/32")))

is_global(ip_network(c("1.0.0.0/8", "2002::/32")))

is_multicast(ip_network(c("224.0.0.0/4", "ff00::/8")))

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

is_reserved(ip_network(c("240.0.0.0/4", "f000::/5")))

is_loopback(ip_network(c("127.0.0.0/8", "::1/128")))

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

is_site_local(ip_network("fec0::/10"))

```

---

max_prefix_length	<i>Size of the address space</i>
-------------------	----------------------------------

---

**Description**

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

**Usage**

```
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**

```

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

max_prefix_length(x)

```

---

netmask	<i>Network mask</i>
---------	---------------------

---

### Description

`prefix_length()`, `netmask()` and `hostmask()` extract different (but equivalent) representations of the network mask. They accept an `ip_network` or `ip_interface` vector.

The functions can also convert between these alternative representations. For example, `prefix_length()` can infer the prefix length from an `ip_address` vector of netmasks and/or hostmasks, while `netmask()` and `hostmask()` can accept a vector of prefix lengths.

### Usage

```
prefix_length(x)

netmask(x, ...)

## S3 method for class 'numeric'
netmask(x, is_ipv6 = NULL, ...)

hostmask(x, ...)

## S3 method for class 'numeric'
hostmask(x, is_ipv6 = NULL, ...)
```

### Arguments

<code>x</code>	<ul style="list-style-type: none"> <li>An <code>ip_network</code> vector.</li> <li>An <code>ip_interface</code> vector.</li> <li><code>prefix_length()</code>: An <code>ip_address</code> vector of netmasks and/or hostmasks. Ambiguous cases (all zeros, all ones) are treated as netmasks.</li> <li><code>netmask()</code> and <code>hostmask()</code>: An integer vector of prefix lengths.</li> </ul>
<code>...</code>	Arguments passed to methods.
<code>is_ipv6</code>	A logical vector indicating whether to construct an IPv4 or IPv6 address. If <code>NULL</code> (the default), then IPv4 is preferred but an IPv6 address is constructed when <code>x</code> is too large for the IPv4 address space.

### Value

- `prefix_length()`: An integer vector
- `netmask()`: An `ip_address` vector
- `hostmask()`: An `ip_address` vector

### 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)

# construct netmask/hostmask from prefix length
netmask(c(22L, 26L), c(FALSE, TRUE))

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

# extract prefix length from netmask/hostmask
prefix_length(ip_address(c("255.255.255.0", "0.255.255.255")))

# invalid netmask/hostmask raise a warning and return NA
prefix_length(ip_address("255.255.255.1"))
```

---

network\_in\_network      *Network membership of other networks*

---

**Description**

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

**Usage**

```
is_supernet(network, other)
```

```
is_subnet(network, other)
```

```
overlaps(network, other)
```

**Arguments**

network	An <a href="#">ip_network</a> vector
other	An <a href="#">ip_network</a> vector

**Value**

A logical vector

**See Also**

Use [is\\_within\(\)](#) to check if an [ip\\_address](#) is within an [ip\\_network](#).

Use [supernet\(\)](#) and [subnets\(\)](#) to traverse the network hierarchy.

### Examples

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

is_supernet(net1, net2)

is_subnet(net1, net2)

overlaps(net1, net2)
```

---

network_size	<i>Network size</i>
--------------	---------------------

---

### 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

```
network_address(x)

broadcast_address(x)

num_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()`: An [ip\\_address](#) vector
- `broadcast_address()`: An [ip\\_address](#) vector
- `num_addresses()`: A numeric vector

### See Also

Use [seq.ip\\_network\(\)](#) to generate all addresses in a network.

**Examples**

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

---

reverse_pointer	<i>Reverse DNS pointer</i>
-----------------	----------------------------

---

**Description**

Returns the PTR record used by reverse DNS.

**Usage**

```
reverse_pointer(x)
```

**Arguments**

x                   An [ip\\_address](#) vector

**Details**

These documents describe reverse DNS lookup in more detail:

- **IPv4:** [RFC-1035 Section 3.5](#)
- **IPv6:** [RFC-3596 Section 2.5](#)

**Value**

A character vector

**Examples**

```
reverse_pointer(ip_address("127.0.0.1"))  
  
reverse_pointer(ip_address("2001:db8::1"))
```

---

sample	<i>Sample random addresses</i>
--------	--------------------------------

---

### Description

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

### Usage

```
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
...	These dots are for future extensions and must be empty.
replace	Should sampling be with replacement? (default: FALSE)
x	An <code>ip_network</code> scalar

### Value

An `ip_address` vector

### See Also

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

### Examples

```
sample_ipv4(5)
```

```
sample_ipv6(5)
```

```
sample_network(ip_network("192.168.0.0/16"), 5)
```

```
sample_network(ip_network("2001:db8::/48"), 5)
```

---

sequence	<i>List addresses within a network</i>
----------	--

---

### Description

`seq()` returns *all* hosts

`hosts()` returns only *usable* hosts

### Usage

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

```
hosts(x)
```

### Arguments

x	An <a href="#">ip_network</a> scalar
...	These dots are for future extensions and must be empty.

### 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.

Use [subnets\(\)](#) to list the subnetworks within 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"))
```

---

summarize\_address\_range

*List constituent networks of an address range*

---

## Description

Given an address range, this returns the list of constituent networks.

If you know the address range matches the boundaries of a single network, it might be preferable to use [common\\_network\(\)](#). This returns an [ip\\_network](#) vector instead of a list of [ip\\_network](#) vectors.

## Usage

```
summarize_address_range(address1, address2)
```

## Arguments

address1	An <a href="#">ip_address</a> vector
address2	An <a href="#">ip_address</a> vector

## Value

A list of [ip\\_network](#) vectors

## See Also

[common\\_network\(\)](#)

## Examples

```
# address range matches network boundaries  
summarize_address_range(ip_address("192.168.0.0"), ip_address("192.168.0.15"))  
  
# address range does not match network boundaries  
summarize_address_range(ip_address("192.167.255.255"), ip_address("192.168.0.16"))
```

---

traverse_hierarchy	<i>Traverse the network hierarchy</i>
--------------------	---------------------------------------

---

### Description

These functions step up and down the network hierarchy. `supernet()` returns the supernet containing the given network. `subnets()` returns the list of subnetworks which join to make the given network.

### Usage

```
supernet(x, new_prefix = prefix_length(x) - 1L)
```

```
subnets(x, new_prefix = prefix_length(x) + 1L)
```

### Arguments

<code>x</code>	An <code>ip_network</code> vector
<code>new_prefix</code>	An integer vector indicating the desired prefix length. By default, this steps a single level through the hierarchy.

### Details

The `ipaddress` package does not support [long vectors](#) (i.e. vectors with more than  $2^{31} - 1$  elements). This limits the number of subnetworks that `subnets()` can return. However, you might find that machine memory imposes stricter limitations.

### Value

- `supernet()`: An `ip_network` vector
- `subnets()`: A list of `ip_network` vectors

### See Also

Use [seq.ip\\_network\(\)](#) to list the addresses within a network.

Use [is\\_supernet\(\)](#) and [is\\_subnet\(\)](#) to check if one network is contained within another.

### Examples

```
supernet(ip_network("192.168.0.0/24"))
```

```
supernet(ip_network("192.168.0.0/24"), new_prefix = 10L)
```

```
subnets(ip_network("192.168.0.0/24"))
```

```
subnets(ip_network("192.168.0.0/24"), new_prefix = 27L)
```

# Index

- \* **address representations**
  - ip\_to\_binary, 16
  - ip\_to\_bytes, 17
  - ip\_to\_hex, 18
  - ip\_to\_integer, 20
- \* **datasets**
  - iana\_ipv4, 6
  - iana\_ipv6, 7
- %<<% (ip\_operators), 15
- %>>% (ip\_operators), 15
- address\_in\_network, 2
- as\_ip\_address (ip\_cast), 10
- as\_ip\_address(), 12
- as\_ip\_interface (ip\_cast), 10
- as\_ip\_network (ip\_cast), 10
- as\_ip\_network(), 12
- bignum::biginteger, 20
- binary\_to\_ip (ip\_to\_binary), 16
- blob::as\_blob(), 17
- blob::blob, 17
- broadcast\_address (network\_size), 26
- broadcast\_address(), 29
- bytes\_to\_ip (ip\_to\_bytes), 17
- collapse\_networks, 3
- collapse\_networks(), 5, 6
- common\_network, 4
- common\_network(), 30
- country\_networks, 5
- double, 20
- exclude\_networks, 6
- exclude\_networks(), 3
- extract\_6to4 (ipv6-transition), 8
- extract\_ipv4\_mapped (ipv6-transition), 8
- extract\_teredo\_client (ipv6-transition), 8
- extract\_teredo\_server (ipv6-transition), 8
- format.ip\_address (ip\_format), 11
- format.ip\_interface (ip\_format), 11
- format.ip\_network (ip\_format), 11
- hex\_to\_ip (ip\_to\_hex), 18
- hostmask (netmask), 24
- hostmask(), 12, 14
- hostname\_to\_ip (ip\_to\_hostname), 19
- hostname\_to\_ip\_all (ip\_to\_hostname), 19
- hosts (sequence), 29
- iana\_ipv4, 6
- iana\_ipv6, 7
- integer, 20
- integer\_to\_ip (ip\_to\_integer), 20
- ip\_address, 3, 4, 8, 9, 9, 10–30
- ip\_cast, 10
- ip\_format, 11
- ip\_interface, 10, 11, 12, 15, 24
- ip\_interface(), 13
- ip\_network, 3–7, 10–12, 13, 15, 21–26, 28–31
- ip\_operators, 10, 15
- ip\_test, 15
- ip\_to\_binary, 16, 17, 18, 21
- ip\_to\_bytes, 17, 17, 18, 21
- ip\_to\_hex, 17, 18, 21
- ip\_to\_hostname, 19
- ip\_to\_hostname\_all (ip\_to\_hostname), 19
- ip\_to\_integer, 17, 18, 20
- ipv6-transition, 8, 22
- is\_6to4 (ipv6-transition), 8
- is\_global (is\_reserved), 22
- is\_ip\_address (ip\_test), 15
- is\_ip\_interface (ip\_test), 15
- is\_ip\_network (ip\_test), 15
- is\_ipv4 (is\_ipv6), 21
- is\_ipv4(), 23



`is_ipv4_mapped(ipv6-transition)`, 8  
`is_ipv4_mapped()`, 10  
`is_ipv6`, 21  
`is_ipv6()`, 23  
`is_link_local(is_reserved)`, 22  
`is_loopback(is_reserved)`, 22  
`is_multicast(is_reserved)`, 22  
`is_private(is_reserved)`, 22  
`is_reserved`, 22  
`is_reserved()`, 7, 8  
`is_site_local(is_reserved)`, 22  
`is_subnet(network_in_network)`, 25  
`is_subnet()`, 3, 31  
`is_supernet(network_in_network)`, 25  
`is_supernet()`, 31  
`is_teredo(ipv6-transition)`, 8  
`is_unspecified(is_reserved)`, 22  
`is_within(address_in_network)`, 2  
`is_within()`, 25  
`is_within_any(address_in_network)`, 2

long vectors, 29, 31

`max_prefix_length`, 23  
`max_prefix_length()`, 21, 24

`netmask`, 24  
`netmask()`, 12, 14  
`network_address(network_size)`, 26  
`network_address()`, 14, 29  
`network_in_network`, 25  
`network_size`, 26  
`num_addresses(network_size)`, 26

operators, 10  
`overlaps(network_in_network)`, 25

`prefix_length(netmask)`, 24  
`prefix_length()`, 12, 14, 23

`reverse_pointer`, 27

`sample`, 28  
`sample_ipv4(sample)`, 28  
`sample_ipv6(sample)`, 28  
`sample_network(sample)`, 28  
`sample_network()`, 29  
`seq.ip_network(sequence)`, 29  
`seq.ip_network()`, 26, 28, 31  
`sequence`, 29  
`setdiff()`, 6  
`subnets(traverse_hierarchy)`, 31  
`subnets()`, 25, 29  
`summarize_address_range`, 30  
`summarize_address_range()`, 4  
`supernet(traverse_hierarchy)`, 31  
`supernet()`, 25  
`traverse_hierarchy`, 31