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

Index ....................................................................................... 32

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.
collapse_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()
common_network

Examples

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

---

**common_network**  
*Find the common network of two addresses*

**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**

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

**Arguments**

- `address1`  
  An `ip_address` vector

- `address2`  
  An `ip_address` vector

**Value**

An `ip_network` vector

**See Also**

`summarize_address_range()`

**Examples**

```r
# 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

Country-level IP networks

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 collapse_networks().

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

```r
## 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  Remove networks from others

Description

eclude_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

eclude_networks(include, exclude)

Arguments

include An ip_network vector
exclude  An ip_network 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 IPv4 address space allocation

Description

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

Usage

iana_ipv4
iana_ipv6

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

<table>
<thead>
<tr>
<th>iana_ipv6</th>
<th>IPv6 address space allocation</th>
</tr>
</thead>
</table>

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

---

### 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

```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`

<table>
<thead>
<tr>
<th>Vector of IP addresses</th>
</tr>
</thead>
</table>

**Description**

Construct a vector of IP addresses.

**Usage**

```r
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

```r
# 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"))
```

Description

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

Usage

```r
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"))

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)
**Description**

Construct a vector of IP interfaces. This hybrid class stores both the host address and the network it is on.

**Usage**

```r
ip_interface(...) 
```

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

### S3 method for class 'ip_address'
```r
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 `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()`.

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)

---

**ip_network**  
*Vector of IP networks*

**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**

...  
Arguments passed to methods.

x  
A character vector of IP networks, in CIDR notation (IPv4 or IPv6)

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 is returned. If you need to retain the host bits, consider using `ip_interface()` instead.

address  
An `ip_address` vector

prefix_length  
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

```r
# 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.
\textit{ip_to_bytes}

\textbf{Value}

- \texttt{ip_to_binary()}: A character vector
- \texttt{binary_to_ip()}: An \texttt{ip_address} vector

\textbf{See Also}

Other address representations: \texttt{ip_to_bytes()}, \texttt{ip_to_hex()}, \texttt{ip_to_integer()}

\textbf{Examples}

\begin{verbatim}
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))
\end{verbatim}

---

\textit{ip_to_bytes} \hspace{1cm} \textit{Represent address as raw bytes}

\textbf{Description}

Encode or decode an \texttt{ip_address} as a list of raw bytes.

\textbf{Usage}

\begin{verbatim}
 ip_to_bytes(x)
 bytes_to_ip(x)
\end{verbatim}

\textbf{Arguments}

\begin{verbatim}
x
 - \texttt{ip_to_bytes()}: An \texttt{ip_address} vector
 - \texttt{bytes_to_ip()}: A list of raw vectors or a \texttt{blob::blob} object
\end{verbatim}

\textbf{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 \texttt{NULL}.

\textbf{Value}

- \texttt{ip_to_bytes()}: A list of raw vectors
- \texttt{bytes_to_ip()}: An \texttt{ip_address} vector

\textbf{See Also}

Use \texttt{blob::as_blob()} to cast result to a blob object

Other address representations: \texttt{ip_to_binary()}, \texttt{ip_to_hex()}, \texttt{ip_to_integer()}
Examples

```r
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,
    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**

```r
ip_to_hex(x)

hex_to_ip(x, is_ipv6 = NULL)
```

**Arguments**

- `x` • `ip_to_hex()`: An `ip_address` vector
  • `hex_to_ip()`: A character vector containing hexadecimal strings
- `is_ipv6` 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 `x` 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()`
\textit{ip\_to\_hostname}

\textbf{Examples}

\begin{verbatim}
  x <- ip_address(c("192.168.0.1", "2001:db8::82e:3734", NA))
  ip_to_hex(x)
  hex_to_ip(ip_to_hex(x))
\end{verbatim}

\textbf{Description}

Perform reverse and forward DNS resolution.

\textbf{Note:} These functions are significantly slower than others in the ipaddress package.

\textbf{Usage}

\begin{verbatim}
  ip_to_hostname(x)
  ip_to_hostname_all(x)
  hostname_to_ip(x)
  hostname_to_ip_all(x)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{x} \hspace{1cm} \textbf{ip\_to\_hostname()}: An \texttt{ip\_address} vector
  \item \texttt{x} \hspace{1cm} \textbf{hostname\_to\_ip()}: A character vector of hostnames
\end{itemize}

\textbf{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 \texttt{NA} is returned for that input. If an error occurs during DNS lookup, then a warning is emitted and \texttt{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.

\textbf{Value}

\begin{itemize}
  \item \texttt{ip\_to\_hostname()}: A character vector
  \item \texttt{ip\_to\_hostname\_all()}: A list of character vectors
  \item \texttt{hostname\_to\_ip()}: An \texttt{ip\_address} vector
  \item \texttt{hostname\_to\_ip\_all()}: A list of \texttt{ip\_address} vectors
\end{itemize}
See Also

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

Examples

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

#### Usage

```r
ip_to_integer(x)
integer_to_ip(x, is_ipv6 = NULL)
```

#### Arguments

- `x` • `ip_to_integer()`: An `ip_address` vector
  • `integer_to_ip()`: A `bignum::biginteger` vector
- `is_ipv6` 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 `x` 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
is_ipv6

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

---

Reserved addresses

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", ":/128")))

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

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

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)
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

```r
prefix_length(x)

netmask(x, ...)
```

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

```r
hostmask(x, ...)
```

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

Arguments

- **x**
  - An `ip_network` vector.
  - An `ip_interface` vector.
  - `prefix_length()`: An `ip_address` vector of netmasks and/or hostmasks. Ambiguous cases (all zeros, all ones) are treated as netmasks.
  - `netmask()` and `hostmask()`: An integer vector of prefix lengths.

- **...**
  - Arguments passed to methods.

- **is_ipv6**
  - 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 `x` 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()`
**network_in_network**

**Examples**

```r
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 `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`.
- Use `supernet()` and `subnets()` to traverse the network hierarchy.
Examples

```r
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)
```

<table>
<thead>
<tr>
<th>network_size</th>
<th>Network size</th>
</tr>
</thead>
</table>

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()`: 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

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

network_address(x)
broadcast_address(x)
num_addresses(x)
```

---

**reverse_pointer**  
*Reverse DNS pointer*

Description

Returns the PTR record used by reverse DNS.

Usage

```r
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

```r
reverse_pointer(ip_address("127.0.0.1"))
reverse_pointer(ip_address("2001:db8::1"))
```
Sampling 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
- `...`: These dots are for future extensions and must be empty.
- `replace`: Should sampling be with replacement? (default: `FALSE`)
- `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)
```
sequence  List addresses within a network

Description

seq() returns all hosts
hosts() returns only usable hosts

Usage

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

hosts(x)

Arguments

x  An ip_network 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.
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 ip_address vector
address2 An ip_address 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

Traverse the network hierarchy

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

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

Arguments

- `x`: An `ip_network` vector
- `new_prefix`: 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). The 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

```r
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
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_ipv4_mapped(ipv6-transition)</td>
<td>8</td>
</tr>
<tr>
<td>is_ipv4_mapped()</td>
<td>10</td>
</tr>
<tr>
<td>is_ipv6</td>
<td>21</td>
</tr>
<tr>
<td>is_ipv6()</td>
<td>23</td>
</tr>
<tr>
<td>is_link_local(is_reserved)</td>
<td>22</td>
</tr>
<tr>
<td>is_loopback(is_reserved)</td>
<td>22</td>
</tr>
<tr>
<td>is_multicast(is_reserved)</td>
<td>22</td>
</tr>
<tr>
<td>is_private(is_reserved)</td>
<td>22</td>
</tr>
<tr>
<td>is_reserved</td>
<td>22</td>
</tr>
<tr>
<td>is_reserved()</td>
<td>7, 8</td>
</tr>
<tr>
<td>is_site_local(is_reserved)</td>
<td>22</td>
</tr>
<tr>
<td>is_subnet(network_in_network)</td>
<td>25</td>
</tr>
<tr>
<td>is_subnet()</td>
<td>3, 31</td>
</tr>
<tr>
<td>is_supernet(network_in_network)</td>
<td>25</td>
</tr>
<tr>
<td>is_supernet()</td>
<td>31</td>
</tr>
<tr>
<td>is_teredo(ipv6-transition)</td>
<td>8</td>
</tr>
<tr>
<td>is_unspecified(is_reserved)</td>
<td>22</td>
</tr>
<tr>
<td>is_within(address_in_network)</td>
<td>2</td>
</tr>
<tr>
<td>is_within()</td>
<td>25</td>
</tr>
<tr>
<td>is_within(any(address_in_network)</td>
<td>2</td>
</tr>
<tr>
<td>long vectors</td>
<td>29, 31</td>
</tr>
<tr>
<td>max_prefix_length</td>
<td>23</td>
</tr>
<tr>
<td>max_prefix_length()</td>
<td>21, 24</td>
</tr>
<tr>
<td>netmask</td>
<td>24</td>
</tr>
<tr>
<td>netmask()</td>
<td>12, 14</td>
</tr>
<tr>
<td>network_address(network_size)</td>
<td>26</td>
</tr>
<tr>
<td>network_address()</td>
<td>14, 29</td>
</tr>
<tr>
<td>network_in_network</td>
<td>25</td>
</tr>
<tr>
<td>network_size</td>
<td>26</td>
</tr>
<tr>
<td>num_addresses(network_size)</td>
<td>26</td>
</tr>
<tr>
<td>operators</td>
<td>10</td>
</tr>
<tr>
<td>overlaps(network_in_network)</td>
<td>25</td>
</tr>
<tr>
<td>prefix_length(netmask)</td>
<td>24</td>
</tr>
<tr>
<td>prefix_length()</td>
<td>12, 14, 23</td>
</tr>
<tr>
<td>reverse_pointer</td>
<td>27</td>
</tr>
<tr>
<td>sample</td>
<td>28</td>
</tr>
<tr>
<td>sample_ipv4(sample)</td>
<td>28</td>
</tr>
<tr>
<td>sample_ipv6(sample)</td>
<td>28</td>
</tr>
<tr>
<td>sample_network(sample)</td>
<td>28</td>
</tr>
<tr>
<td>sample_network()</td>
<td>29</td>
</tr>
<tr>
<td>seq.ip_network(sequence)</td>
<td>29</td>
</tr>
<tr>
<td>seq.ip_network()</td>
<td>26, 28, 31</td>
</tr>
<tr>
<td>sequence</td>
<td>29</td>
</tr>
</tbody>
</table>