Package ‘ipaddress’

March 25, 2020

Title  Tidy IP Addresses

Version  0.2.0

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 (>= 3.2.0)

Imports  blob (>= 1.2.0), methods, Rcpp, rlang (>= 0.4.0), vctrs (>= 0.2.2)

Suggests  knitr, rmarkdown, testthat (>= 2.1.0)

LinkingTo  AsioHeaders, 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  2020-03-25 17:30:02 UTC
**R topics documented:**

- `address_in_network` ................................. 2
- `binary` ............................................. 3
- `hostname` ........................................... 4
- `ipv6-transition` ................................. 5
- `ip_address` ........................................ 6
- `ip_interface` ....................................... 8
- `ip_network` ........................................ 9
- `is_ipv6` ............................................. 11
- `is_reserved` ....................................... 12
- `max_prefix_length` ............................... 13
- `netmask` ........................................... 13
- `network_in_network` ............................. 15
- `network_size` ..................................... 16
- `packed` ............................................. 17
- `sample` .............................................. 18
- `sequence` .......................................... 19

**Index**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
</tr>
</tbody>
</table>

---

`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

```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")))
```

binary

Represent address as binary

Description

as_binary() and from_binary() encode and decode an ip_address vector to a character vector of bits.

Usage

```r
as_binary(ip)

from_binary(bits)
```

Arguments

- `ip` An ip_address vector
- `bits` 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

- as_binary() returns a character vector
- from_binary() returns an ip_address vector

See Also

Use as_packed() and from_packed() to encode/decode raw bytes.

Examples

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

as_binary(x)

from_binary(as_binary(x))
```
**hostname**

Translate address to/from hostname

**Description**

as_hostname() performs reverse DNS resolution (translating IP addresses to hostnames)

from_hostname() performs forward DNS resolution (translating hostnames to IP addresses)

**Usage**

as_hostname(ip, multiple = FALSE)

from_hostname(host, multiple = FALSE)

**Arguments**

- **ip** An ip_address vector
- **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.
- **host** 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, 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**

- as_hostname() returns a character vector or a list of character vectors (depending upon the multiple argument)
- from_hostname() returns a ip_address vector or a list of ip_address vectors (depending upon the multiple argument)

**See Also**

The base function nsl() provides forward DNS resolution to IPv4 addresses, but only on Unix-like systems.
ipv6-transition IPv6 transition mechanisms

Examples

```r
## Not run:
from_hostname("r-project.org")
as_hostname(from_hostname("r-project.org"))
## End(Not run)
```

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

# 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.

as_ip_address() casts an object to ip_address.

is_ip_address() checks if an object is of class ip_address.

Usage

ip_address(ip = character())

as_ip_address(x)

is_ip_address(x)

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

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

Arguments

ip  Character vector of IP addresses, in dot-decimal notation (IPv4) or hexadecimal notation (IPv6).

x  An ip_address vector

...  Arguments to be passed to other methods
**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).

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**

```r
# supports IPv4 and IPv6 simultaneously
ip_address(c("192.168.0.1", "2001:0db8:8a2e: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.
- as_ip_interface() casts an object to ip_interface.
- is_ip_interface() checks if an object is of class ip_interface.

Usage

ip_interface(...)

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

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

as_ip_interface(x)

is_ip_interface(x)

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

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

Arguments

- ... Arguments to be passed to other methods
- ip Character vector of IP interfaces, in CIDR notation (IPv4 or IPv6).
- address An ip_address vector
- prefix_length An integer vector
- x An ip_interface 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:
ip_network

- 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

# 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.
as_ip_network() casts an object to ip_network.
is_ip_network() checks if an object is of class ip_network.

Usage
ip_network(...)

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

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

is_ip_network(x)

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

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

### Arguments

- ...: Arguments to be passed to other methods
- ip: 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
- x: An ip_network 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.

### Value

An S3 vector of class ip_network

### See Also

prefix_length(), network_address(), netmask(), hostmask()
Examples

# 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)

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

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

See Also

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

Examples

# 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", "::1/128")))

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

---

**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)
```

---

netmask

---

**netmask**  
*Network mask*

### Description

These functions yield equivalent representations of the network mask.

### Usage

```r
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
...
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

See Also

`max_prefix_length()`

Examples

```r
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

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)
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)
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

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

network_address(x)
broadcast_address(x)
um_addresses(x)
Description

as_packed() and from_packed() encode and decode an ip_address vector to a blob::blob vector.

Usage

as_packed(ip)

from_packed(bytes)

Arguments

ip An ip_address vector

bytes 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

• as_packed() returns a blob::blob vector

• from_packed() returns an ip_address vector

See Also

Use as_binary() and from_binary() to encode/decode binary.

Examples

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

as_packed(x)

from_packed(as_packed(x))
Sample random addresses

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
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

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), 6
as.character.ip_interface(ip_interface), 8
as.character.ip_network(ip_network), 9
as_binary(binary), 3
as_host_ip_address(hostname), 4
as_ip_address(ip_address), 6
as_ip_interface(ip_interface), 8
as_ip_network(ip_network), 9
as_packed(packed), 17
as_6to4(ipv6-transition), 5, 12
as_ip_address(ip_address), 6
as_ip_interface(ip_interface), 8
as_ip_network(ip_network), 9
as_ipv4(ipv6), 11
as_ipv4(), 13
as_ipv4_mapped(ipv6-transition), 5
as_ipv4_mapped(), 7
as_ipv6, 11
as_ipv6(), 13
is_6to4(ipv6-transition), 5
is_ipv4(ipv6), 11
is_ipv4(), 13
is_ipv4_mapped(ipv6-transition), 5
is_ipv4_mapped(), 7
is_ipv6, 11
is_ipv6(), 13
is_link_local(is_reserved), 12
is_loopback(is_reserved), 12
is_multicast(is_reserved), 12
is_reserved, 12
is_subnet(network_in_network), 15
is_subnet(), 2
is_supernet(network_in_network), 15
is_teredo(ipv6-transition), 5
is_unspecified(is_reserved), 12
is_within(address_in_network), 2
is_within(), 15
is_within_any(address_in_network), 2
is_within(), 15
is_within(), 15
long vectors, 19
max_prefix_length, 13
max_prefix_length(), 11, 14
netmask, 13
netmask(), 9, 10
network_address(network_size), 16
network_address(), 10, 19
network_in_network, 15
network_size, 16
ns1(), 4
num_addresses(network_size), 16
overlaps(network_in_network), 15
packed, 17
prefix_length (netmask), 13
prefix_length(), 9, 10, 13

sample, 18
sample_ipv4 (sample), 18
sample_ipv6 (sample), 18
sample_network (sample), 18
sample_network(), 19
seq.ip_network (sequence), 19
seq.ip_network(), 16, 18
sequence, 19