Package 'IP'

March 3, 2024

Type Package

Title Classes and Methods for 'IP' Addresses

Version 0.1.3

Author Thomas Soubiran [aut, cre]

Maintainer Thomas Soubiran <thomas.soubiran@univ-lille.fr>

Description Provides S4 classes for Internet Protocol (IP) versions 4 and 6 addresses and efficient methods for 'IP' addresses comparison, arithmetic, bit manipulation and lookup. Both 'IPv4' and 'IPv6' arbitrary ranges are also supported as well as internationalized (IDN) domain lookup with and 'whois' query.

License GPL (>= 2)

SystemRequirements libidn: libidn11-dev (deb)

NeedsCompilation yes

Depends R (>= 3.5.0), methods

Imports stringi

Suggests knitr

VignetteBuilder knitr

Encoding UTF-8

Repository CRAN

Date/Publication 2024-03-03 17:20:02 UTC

R topics documented:

`IP-package` *Classes and methods for IP addresses*

Description

Classes and methods for IP addresses

Details

The IP package provides vector-like classes and methods for Internet Protocol (IP) addresses. It is based on the ip4r PostgreSQL extension available at `https://github.com/RhodiumToad/ip4r`.

An IP address is a numerical label assigned to each device connected to a computer network that uses the Internet Protocol for communication. The Internet Protocol uses those labels to identify nodes such as host or network interface for relaying datagrams between them across network boundaries.

Internet Protocol version 4 (IPv4) defines an IP address as an unsigned 32-bit number. However, because of the growth of the Internet and the depletion of available IPv4 addresses, a new version of IP (IPv6) using 128 bits for the IP address was developed from 1995 on. IPv6 deployment has been ongoing since the mid-2000s. Note that there is no IPv5 address. In addition, IPv4 and IPv6 protocols differ in many respects besides IP addresses representation.

IP addresses are usually written and displayed in human-readable notations, such as "192.168.0.1" in IPv4, and "fe80::3b13:cff7:1013:d2e7" in IPv6. Ranges can be represented using two IP addresses separated by a dash or using the Classless Inter-Domain Routing (CIDR) notation. CIDR suffixes an address with he size of the routing prefix of the address which is the number of significant bits. For instance, "192.168.0.0/16" is a private network with subnet mask "255.255.0.0" and is equivalent to "192.168.0.0-192.168.255.255".

Currently, the IP package supports the following object types implemented using S4 classes :

- the `IPv4` class stores IP version 4 addresses
- the `IPv4r` class stores IP version 4 addresses ranges
- the `IPv6` class stores IP version 6 addresses
- the `IPv6r` class stores IP version 6 addresses ranges
- the `IP` class stores both IPv4 and IPv6 addresses
- the `IPr` class stores both IPv4r and IPv6r addresses
- the (still experimental) `host` class holds the result of DNS lookup

The IP package also provides methods for arithmetic, comparison and bitwise unary and binary operations in addition to sorting and lookup and querying information about IP addresses and domain names. All operators are not available for all classes mostly by design but a few are still missing because they have not been implemented yet. IP objects can also be subseted or stored in a `data.frame` and serialized.

The IP and IPr classes are only convenience containers for instances when addresses must be created from vectors mixing both protocols. The IPv4 and IPv6 protocols and their corresponding IP

representation are indeed very different in many respects so only a subset of methods are available for them. In addition, methods for those containers tend to run slower because, at the moment, they need to make two passes (one for IPv4* and one for IPv6* objects). Use the `ipv4(IP)` (resp. `ipv4r(IPr)`) and `ipv6(IP)` (resp. `ipv6r(IP)`) getters to work with v4 and v6 objects separately.

Design considerations:

IP objects were designed to behave as much as possible like base R atomic vectors. Therefore many R base functions such as `table()` or `factor()` or merging two `data.frame` using IP objects as keys work.

But there are a few caveats when using functions or methods not provided by the IP package in which case you **may have to convert to the character representation of the addresses**.

IP objects are S4 objects that all inherit from the integer class and because of this there are instances where function calls will operate on the inherited integer `.Data` part of the object only. As of writing, this is for example the case for the `nchar` function which returns the number of characters of the .Data vector only. But `grep` works because the x argument to the function is explicitly coerced to character before further processing.

The .Data slot does not hold the addresses but an index to the addresses. When calling a non-IP method, R will first look for a method for this particular object. If none is found, it will try to find one for the class this object inherits from. Hence, the call will operate on the index, and not on the object as a hole. This is why some operations are explicitly blacklisted such as multiplication. Since there are no `` `*` `` for IP objects, multiplying an IP with a number would otherwise fall back to multiplying the index by this number, thus badly damaging the object.

Reasons for using an index are twofold. First, each IP address space use the entire 32 (resp. 128) bits integer range. Thus, no value can be used for NA. For instance, R defines `NA_integer_` as 2^{31} which a perfectly valid IP v4 address ("128.0.0.0"). Second reason is IP words size. An IPv4 address uses 32 bits and thus can be stored using an integer vector (and IPv4 address ranges uses 64 bits and could be stored using a numeric vector). But an IP v6 address uses 128 bits and an IP v6 address range uses 256 bits and currently no R built-in atomic vectors are wide enough to hold them. IP addresses other than IPv4 have to be stored in a separate matrix and the index is used to retrieve their value.

Therefore, each IP* object has an index which either points to the IP location in a table or mark the value as NA. This way R believes it is dealing with a regular vector but at the cost of increased memory consumption. The memory footprint is a function of the number of NA.

On the other end, this design makes it easy to know if there are any NA and, if none, skip NA checking which makes things faster.

SIMD support:

The IP package provide an experimental support for AVX2 vectorized operations for IP comparison and arithmetic. To enable AVX2 support, please pass the `"--enable-avx2"` `configure.args` argument to the `install.packages()` function.

Data protection:

One last caveat. In certain countries such as EU member countries, IP addresses are considered personal data (see Article 29 Working Party Opinion 4/2007 and ECJ ruling dated 19 October 2016 –ref.: C582/14). IP processing must therefore be done in accordance to the applicable laws and regulations.

`Arithmetic` *Methods for IP arithmetic*

Description

Methods for unary and binary IP arithmetic

Usage

```
-e1
e1 + e2
e1 - e2
```

Arguments

`e1`	an object of either an 'IPv4', 'IPv6' or 'IP' class
`e2`	either a corresponding object of class 'IPv4', 'IPv6', 'IP' or an integer or a numerical vector

Details

Both IPv4 and v6 sets are represented as unsigned integers and are closed under addition and subtraction. An operation resulting in a negative number (or an overflow) is marked as NA.

Operations are currently not always commutative. IP*-IP* are but those involving integers or floats are not. Thus, adding (or subtracting) an integer or a float to an IP* object will work but the reverse (adding (or subtracting) an IP* to an integer or a float) will raise an error (see example below and the caveat section in the package description). In addition multiplication and division are not implemented and will raise an error.

Arithmetic operations involving IP* are better done using methods provided. Both IPv4 and IPv6 addresses are represented as unsigned integers but R only works with 32 bits signed integers. In addition, double precision numbers cannot represent all integers in the 0-(2^128-1) range. Therefore, converting an IPv6 object to numeric may cause a loss of precision and the same applies to arithmetic operations on IPv6 represented as floating point numbers.

Value

an object of either an 'IPv4', 'IPv6' or 'IP' class

Examples

```
##
ipv4("192.0.0.1") + 1
ipv6("fd00::1") + 1
ip(c("192.0.0.1", "fd00::1")) + 1

##
## Prohibited Arith operations
```

```
##
## this raises an error
tryCatch(1L - ipv4("192.0.0.1"), error=function(e) e )
## and so will
tryCatch(1 + ipv6("fd00::1"), error=function(e) e )
## as well as
tryCatch(ipv4("192.0.0.1") * 2, error=function(e) e )

##
## Loss of precision in arithmetical operations
##
(2^52 +1)- 2^52
(2^53 +1)- 2^53
##
identical((2^64 +1)- 2^64  , 0 )
## ...and so on
( (2^64 + 2^11 ) - (2^64))
## next representable number with IEEE 754 double precision floats; mind the gap
( (2^64 + 2^12 ) - (2^64))


## OTH,
((ipv6('::1') %<<% 53L) + ipv6('::1')) - (ipv6('::1') %<<% 53L)
##
(x <- ( ( ipv6('::1') %<<% 64L ) + ( ipv6('::1') %<<% 11L ) ) - ( ipv6('::1') %<<% 64L ) )
log2(as.numeric(x))
```

`Bitwise-methods` *Bitwise operations*

Description

Methods for IP bitwise operations

`!e1`

`e1 & e2`

`e1 | e2`

`e1 %>>% e2`

`e1 %<<% e2`

`e1 ^ e2`

`ip.xor(e1 , e2 )`

`ipv4.netmask(n)`

`ipv6.netmask(n)`

`ipv4.hostmask(n)`

`ipv6.hostmask(n)`

Arguments

e1	an object of either an 'IPv4', 'IPv6' or 'IP' class
e2	an object of either an 'IPv4', 'IPv6' or 'IP' class except for shifts where e2 is like 'n'
n	an integer in the range (0,32) for IPv4 or in the (0,128) for IPv6 for masking methods

Details

The &, | and ! operators behave differently from their base R counterparts in that they perform bitwise operation much like in the C language.

- & : bitwise AND
- | : bitwise non exclusive OR
- ! : bitwise NOT

ip.xor() provides a faster alternative to base xor().

%>>% and %<<% perform left (binary division) and right shift (binary multiplication) respectively.

The *.netmask() and *.hostmask() functions return the net and host mask of specified length n.

Value

an object of either an 'IPv4', 'IPv6' or 'IP' class

Examples

```
##
private.network <- ipv4r("192.0.0.0/16")
##
(mask.len <- ceiling(log2(ip.range(private.network))))
##
ip <- ipv4("192.168.1.1")
##
(netmask <- ipv4.netmask(mask.len))
##
ip & netmask
##
(hostmask <- ipv4.hostmask(mask.len))
##
ip & hostmask
##
((ip & netmask) | (ip & hostmask) )==ip
## 2 complement
((!ip) + 1L)==-ip
##
ipv4('0.0.0.2') %>>% 1L
##
ipv4('0.0.0.2') %<<% 1L
##
```

```
## branchless swap
##
ipv4.ifelse <- function(test, yes, no){
  ##
  if( ( class(yes)!='IPv4' ) | ( class(no)!='IPv4' ) ){
    stop('both arguments should be of class IPv4')
  }
  ##
  ip.xor(
    no
    , ip.xor(
      no, yes
    ) & -(ipv4(test)) ## mask
  )
}
##
x <- ipv4('192.168.0.0') + 1:5
## recycling without warning (yet)
y <- x + c(1,-1)
##
test <- x < y
##
data.frame(
  x, y, test, res= ipv4.ifelse(test , x,y)
)
##
##
##
ip6 <- ipv6("2606:2800:220:1:248:1893:25c8:1946")
## Unicast addresses global routing prefix
ip6 & ipv6.netmask(48)
## Subnet ID
ip6 & (ipv6.hostmask(128-16) %<<% 64L)
## Interface ID
ip6 & ipv6.hostmask(64)
```

`Comparison` *Methods for IP Comparison*

Description

Methods for IP binary comparison

Arguments

`e1, e2`	objects of either class 'IPv4', 'IPv6'or 'IP'

Details

Only the == operator is supported for ranges

Value

a logical vector

Methods

x == y

x != y

x > y

x < y

x >= y

x <= y

Examples

```
##
ip1 <- ip(c("192.0.0.1", "fd00::1")) + rep(c(0:2),each=2)
##
ip2 <- ip1 + rep(c(1,-1,0), each=2)
##
data.frame(
  ip1, ip2
  , lt = ip1<ip2
  , le = ip1<=ip2
  , eq = ip1==ip2
  , ge = ip1>=ip2
  , gt = ip1>ip2
)
```

Conversion-methods *Methods for converting IP objects to other representations*

Description

Methods for converting IP objects to other representations

Arguments

x	an object of class 'IPv4', 'IPv4r', 'IPv6', 'IP' or 'IPr'
...	not used

Hostinfo *host-info*

Description

Methods for querying information about hosts (DNS) or IP (address spaces)

Usage

```
host(host,...)

host(host,...)

host.info(host,...)

localhost.ip(...)

toIdna(domain, flags)

fromIdna(domain, flags)

fqdn(hostname)

is.fqdn(hostname)

whois(domain, refer , output, verbose)

rir.names()

ipv4.rir()

ipv6.rir()

ipv4.addr.space()

ipv6.addr.space()

ipv4.reserved()

ipv6.reserved()

ipv4.recovered()

ipv6.unicast()
```

Arguments

host	a vector of either or IPv4, IPv6, IP addresses
...	further arguments. Only host.info (default:FALSE) for host() at the moment
hostname	A character vector of host names
domain	A character vector of domain names
flags	Flags for IDNA conversion. "IDNA_DEFAULT": default behavior, "IDNA_ALLOW_UNASSIGNED": allow processing of unassigned Unicode code points, "IDNA_USE_STD3_ASCII_RULES": check output to make sure it is a STD3 conforming host name.
refer	An optional referrer to be queried
output	An integer specifying whether to return the raw response from the referrer (0) or parse the response and return a key-value named vector (1). The latter is still experimental due to the heterogeneity of responses.
verbose	An integer specifying the level of verbosity

Details

Methods and functions for querying informations about hosts:

- `host()` takes a character vector of domain names as arguments for DNS lookup. Addresses can be extracted with the corresponding methods (ipv4(), ipv6(), ip()). Also takes either IPv4, IPv6 or IP objects for reverse DNS lookup and returns the corresponding domain name (or NA if not found).
- `host.info()` (depreciated) takes either IPv4, IPv6 or IP objects for reverse DNS lookup and returns the corresponding domain name (or NA if not found)
- `localhost.ip()` retrieves the host's interfaces IP adresses
- `fqdn()` extracts the fully qualified name of a domain name, -eg to query `whois` databases
- `is.fqdn()` tests whether strings in a character vector qualify as fully qualified names
- `whois()` queries `whois` databases for a vector of fully qualified domain names.

Since `localhost.ip()` needs OS specific system call —as well as `host()` and `host.info()`—, this function is only available for **POSIX compliant OS** at the moment. Support of Internationalized Domain Names (IDN) also depends on the system's libraries. For instance, glibc supports IDN starting with version 2.3.4. Use the `toIdna()` function to ensure proper domain names encoding. Note that result may differ depdending on the flag argument.

In addition, the `host()` and `host.info()` methods are still **very experimental** and might change in the future.

`whois` databases typically contain information such as registrars' names ... Note that responses are not standardized at all and may require an ad hoc parser. This is why the `whois()` function returns either a (still buggy at the moment) named vector of key-value pairs or the raw responses from the referrers.

The relevant referrer `url` can be determined automatically (default) or passed as an argument.

Functions returning IP addresses assignments from IANA registries:

- `ipv4.addr.space()` and `ipv6.addr.space()` : return the corresponding IP address space
- `ipv4.reserved()` and `ipv6.reserved()` : return the corresponding IP reserved address space

- `ipv6.unicast()` : IPv6 unicast addresses
- `ipv4.recovered()` : pool of IPv4 addresses recovered by IANA from RIRs
- `ipv4.rir()` and `ipv6.rir()` : returns the RIRs IP address spaces
- `rir.names()` : Regional Internet Registry names

The IP address spaces is divided into many ranges with specific purposes. For instance, IP addresses can be assigned to organizations. Some addresses are otherwise reserved for special purposes such as loopback, subnetting, local communications within a private network, multicasting, broadcasting,...

The IP address space is managed globally by the Internet Assigned Numbers Authority (IANA), and locally by five regional Internet registries (RIRs) :

- The African Network Information Center (AFRINIC) serves Africa
- The American Registry for Internet Numbers (ARIN) serves Antarctica, Canada, parts of the Caribbean, and the United States
- The Asia-Pacific Network Information Centre (APNIC) serves East Asia, Oceania, South Asia, and Southeast Asia
- The Latin America and Caribbean Network Information Centre (LACNIC) serves most of the Caribbean and all of Latin America
- The Réseaux IP Européens Network Coordination Centre (RIPE NCC) serves Europe, Central Asia, Russia, and West Asia

RIRs are responsible in their designated territories for assignment to end users and local Internet registries, such as Internet service providers.

Note differences in ouptut between `ipv4.addr.space()` and `ipv6.addr.space()`. RIRs IPv4 and Ipv6 assignments are stored by IANA in tables with different naming scheme (corresponding to `ipv4.addr.space()` and `ipv6.unicast()`). In the early days of IPv4 deployment, addresses were assigned directly to end user organizations. Therefore, `ipv4.addr.space()` also mixes RIR and end user organizations assignments. To find the corresponding RIR, use `ipv4.rir()` and `ipv6.rir()` instead.

Also note that some lookups may be misleading because some IPv4 ranges have been transferred from one RIR to an another (see example). For instance, some address ranges were assigned by ARIN in the 80's to European organizations such as universities before RIPE-NCC began its operations in 1992. Those ranges were later transferred to the RIPE NCC but still belong to the ARIN address space.

Likewise, some IPv4 addresses have been recovered by IANA from RIRs in order to delay IPv4 pool exhaustion and were later reassigned to other RIRs (see `ipv4.recovered`).

Value

`host`	an host object or a character vector
`host.info`	a character vector
`localhost.ip`	an IP

Examples

```
##

host(
```

```
  ipv4(
    c("127.0.0.1")
  )
)
##
h <- host(c(
  "icann.org", "iana.org"
))
##
host(ipv4(h))
##
## Domain names internationalization
##
##
## results may vary according to the (POSIX) platform
host(c("bucher.de", "Bücher.de"))

##
if( ip.capabilities()["IDN"] ){
  ##
  dn <- c(
    enc2utf8("bücher.de") ## ensure UTF-8
    ## cannot input emoji with Latex
   , "\U1f4a9" # or alternatively: rawToChar(as.raw(c(0xf0, 0x9f, 0x92, 0xa9, 0x2e, 0x6c, 0x61)))
  )
  ##
  Encoding(dn) <- "UTF-8"
  ##
  dn
  ## enforce internationalization with different options
  flags <-rep( c( "IDNA_DEFAULT" , "IDNA_ALLOW_UNASSIGNED"), each = length(dn))
  ##
  dni <- toIdna( dn, flags)
  ## convert back
  fromIdna(dni, flags)

  ##
  host(dni)

}
##

##
## French country-code top-level domains (ccTLD)
##
tld <- whois(
  c(
    "fr", "re", "tf", "wf", "pm", "yt"
    , "nc", "mq"##, "gp", "gf"
    , "pf"
  )
  , verbose = 1 ## be a little verbose
  , output = 1 ## output key-value pairs
```

```
)
##
sapply(tld, function(x) x[names(x)=="whois"])
##
## R related info
##
rhost     <- host('r-project.org')
## hostname       : "cran.wu-wien.ac.at"
rhost.hnm <- host.info(ipv4(rhost))
## primary domain : "ac.at"
fqdn(rhost.hnm)
## ARIN
ipv4.rir()[ip.match(ipv4(rhost), ipv4.rir())]
##
ip.match(ipv4(rhost), ipv4.recovered())
## domain name info
rdom.wh   <- whois('r-project.org', output=1)
## "AT"
rdom.wh[['r-project.org']]['Registrant Country']
## host
rhost.wh0 <- whois(ipv4(rhost),verbose = 2, output=1)
```

`IP classes` *IPv4, IPv6 and IP classes*

Description

classes for IPv4 and IPv6 addresses

Usage

```
ipv4(object,...)
ipv6(object,...)
ip(e1,e2,...)
```

Arguments

`object`	a vector of IPv4 or IPv6 strings. If missing, returns an empty IPv4 or IPv6 object
`e1, e2`	either e1= a vector of IPv4 or IPv6 strings (and e2 missing) or objects of class e1 = an object of class IPv4' and e2 = an object of class 'IPv6
`...`	for c, zero or more objects objects of either class 'IPv4' or 'IPv6'or 'IP' exclusively

Details

IPv4 and IPv6 objects are created either from either character strings or integer vectors through `ipv4()` and `ipv6()` calls.

IP objects store both IPv4 and IPv6 addresses. IP are created either from a character string or from IPv4 and IPv6 objects through `ip()` calls.

Since the IPv4 and IPv6 protocols use a different address representation, IP objects store both IPv4 and IPv6 addresses but do not mix them. The i-th element of an IP object can only an IPv4 or an IPv6 address but not both. So, if the i-th IPv4 is set, the corresponding i-th IPv6 must be NA and vice-versa.

in addition to object creation, he `ipv4()` and `ipv6()` methods also extract the IPv4 and IPv6 addresses from an IP object and return an object with the same length. Use the drop argument to remove all NA values.

Like atomic base R vectors, IPv4, IPv6 and IP objects elements can be subsetted (`[`) and replaced (`[<-`) and named (`name<-`). Objects can also be concatenated (`c()` or `rbind2()`) or stored in a `data.frame`.

Note that in order to avoid undesirable side-effects, `is.numeric()` returns `FALSE`

Examples

```
##
ipv4("0.0.0.0")==ipv4(0L)
##
ipv6("::")==ipv6(0L)
## create an empty object
ip0    <- ip()
## grow it
ip0[3] <- ipv4(3L)
ip0[5] <- ipv6(5L)
ip0
## same thing with NA
ip0    <- ip()
ip0[2] <- NA
ip0
## private networks
ip.strings <- c(v4 = "192.0.0.1", v6 = "fd00::1" )
##
(ip4 <- ipv4(ip.strings))
##
(ip6 <- ipv6(ip.strings))
##
(ip <- ip(ip.strings))
##
all(ip==ip(ip4, ip6))
##
pnet0 <- data.frame(
  ip
  , v = ip.version(ip)
)
##
```

```
pnet1 <- rbind(
  pnet0
  , within(pnet0, ip <- ip+1L)
)
##
pnet0==pnet1[1:2,]
## fails (why?):
identical(pnet0,pnet1[1:2,])
##
ip(ip4[1],ip6[2],append=TRUE)
##
## IPv6 transition mechanism
##
## IPv4-mapped Address
(ip6 <- ipv6("::ffff:c000:0280"))==ipv6("::ffff:192.0.2.128")
##
ipv6.reserved()[ip.index(ipv6.reserved())(ip6)]
## NAT64 IPv4-IPv6 translation
(ip6 <- ipv6("64:ff9b::c000:201") ) & ipv6.hostmask(96)
##
ipv6.reserved()[ip.index(ipv6.reserved())(ip6)]
```

`IP range classes` *IPv4, IPv6 and IP ranges classes*

Description

classes for IPv4 and IPv6 ranges addresses

Details

IPv4 and IPv6 ranges may be created from character vector using either range or Classless Inter-Domain Routing (CIDR) notation. Range notation represents ranges by using first and last address separated by a dash (`"<ipr-start/>-<ipr-end/>"`). CIDR notation uses a network prefix and a network identifier separated by a slash (`"<net-prefix/>/<identifier/>"`). The network identifier is a decimal number which counts the number of leading 1 bits in the subnet mask (see `hostmask()`).

the lo() and hi() methods extract the low and high ends of ip ranges. When extracting IPv4r or IPv6r parts from IPr objects, use the drop argument to remove all NA values.

Examples

```
##
## Range notation
##
ipv4r("192.0.0.0-192.0.0.10")
##
## CIDR notation
##
```

```
## The entire IPv4 address space
ipv4(ipv4r('0.0.0.0/0'))
## Is there life on Mars ? (Martian packets)
ipv4r("100.64.0.0/10")
##
ip4 <- ipv4("192.0.0.0")
## power of 2
ipv4r( print(sprintf("%s-%s", ip4,  ip4 + ( 2^8-1) ) ))
## not a power of 2
ipv4r( print(sprintf("%s-%s", ip4,  ip4 + 10 ) ))
##
## Network classes
##
ip.class <- data.frame(
  name = paste('class', LETTERS[1:5])
  , class = ipv4r(
  c(
        '0.0.0.0/8'                ## Class A
      , '128.0.0.0/16'             ## Class B
      , '192.0.0.0/24'             ## Class C
      , '224.0.0.0-239.255.255.255' ## Class D
      , '240.0.0.0-255.255.255.255' ## Class E
    )
  )
)
##
## extract IP range start and end
##
(class.ip <- ipv4(ip.class$class))
##
lo(ip.class$class)==class.ip$lo
##
hi(ip.class$class)==class.ip$hi
##
## # of hosts on this network
##
ip.range(ip.class$class)
## this is ok for IP v4 but may cause loss of precision for IPv6
## (please refer to the Arithmetic section)
ip.range(ip.class$class)==as.numeric(class.ip$hi - class.ip$lo)
##
##
##
ipr0 <- ipr()
##
ipr0[3] <- ipv4r(
  "0.0.0.0", "0.0.0.1"

)
ipr0[5] <- ipv6r(
  "::" , 0L
)
ipr0
```

```
##
ipr0    <- ipr()
ipr0[2] <- NA
ipr0
##
## sequences
##
seq(ipv4r('0.0.0.0/24'), by=5)
seq(ipv4r('0.0.0.0/24'), length.out=3)
##
seq(ipv6r('::/120'), b=5)
seq(ipv6r('::/120'), length.out=3)
##
## throws an error : seq(ipv6r('::/96'),by=1)
## because this would yield a 2^32 vector
```

`Varia` *Miscellaneous methods and functions for IP classes*

Description

Mostly IP counterparts of base R methods and functions for atomic vectors. Namely,

- length(), is.na(), anyNA()
- unique()
- sorting : xtfrm()
- matching : match(), ip.match(), ip.index()
- set operations : ip.setequal(), ip.union(), ip.intersect(), ip.setdiff(), ip.symdiff()

Details

Sorting:

IP object may be efficiently sorted through call to R generic functions `order()` and `sort()` thanks to the `xtfrm` generic function. The IP package also provides the `ip.order()` which falls back to the default order method at the moment.

Lookup:

This part is still **experimental** and might be subject to change in the future.

`match()` and `ip.match()` do IP lookup like base `match()` while `ip.index()` can be used for range queries. The IP package make `match()` generic to avoid unwanted effects of method dispatch in code using the package. But note that, unfortunately, this won't change the behaviour of `match()` in other packages (see caveat section in the package description).

`match()` and `ip.match()` behave differently according to their signature. When table is of class IPv4 or IPv6, `ip.match()` does a table lookup like base `match()`. But when table is an IP range and the x argument is not, both look for the range x lies into. If you want to test whether an IP range lies within another range, use the function returned by the `ip.index()` method (see example).

When arguments are of the same class, `match()` simply call base `match()` on the character representation while `ip.match()` uses hash tables. Range search uses a binary search tree. Beware that binary search can only handle non overlapping IP* ranges by default. Use `ip.index()` with `overlap=TRUE` to allow for overlap. Note that this also allows for multiple matches. As a consequence, result vector might be longer that input vector and therefore needs specialized data structures and access methods inspired by the compressed column storage of sparse matrices. See the example section for testing for overlap and lookup.

`ip.index()` returns a function. Calling this function with the value argument set to TRUE returns the matched value and the indices of the matches otherwise. When both overlap and value are TRUE, the function returns a two–columns `data.frame`with x and the matching values in the table.

Also, the incomparable argument for `match()` or `unique()` is not implemented yet.

Examples

```
##
x  <- ipv4(0L) + sample.int(10)
x[order(x)]
sort(x)

##
## matching the address space of a wifi interface on a GNU/Linux box
## that uses Predictable Network Interface Names
## notes: the name of the interface might change depending on the distribution
##       you're using among other things and the localhost.ip() function
##       only works for POSIX platforms at the moment
##

  ipv4.reserved()[match(ipv4(localhost.ip())['wlp2s0'], ipv4.reserved() )]
  ## alternatively, if tables has to be looked up several time
  m <- ip.index(ipv4.reserved())
  m(ipv4(localhost.ip())['wlp2s0'])


##
## ip.match() and ip.index() comparison
##
##
## index the table
bidx <- ip.index(ipv4.reserved())
## "169.254.0.0/16"
x <- ipv4.reserved()['Link Local']
## match
ip.match(x, ipv4.reserved() )
## match
ipv4.reserved()[bidx(x)]
## a range that lies within "169.254.0.0/16"
x <- ipv4r("169.254.0.0/24")
## no match ("169.254.0.0/24"!="169.254.0.0/16")
ip.match(x, ipv4.reserved() )
## match ("169.254.0.0/24" \in "169.254.0.0/16")
ipv4.reserved()[bidx(x)]
```

```
##
## overlap
##
## this demonstrates that ranges in ipv4.reserved() overlap
##
## range match
m <- (
  ip.index(ipv4.reserved())
)(value=TRUE)
## FALSE because there are overlapping ranges and, in this case,
## the query returns the first matching range
all(m==ipv4.reserved())
## OTH match works as expected
all(ipv4.reserved()[ip.match(ipv4.reserved(),ipv4.reserved())]==ipv4.reserved())
##
## Find overlapping IPv4 ranges (pure R)
##
ipr.overlaps <- function(x, y, rm.diag  = FALSE){
  overlaps <- function(x,y) ( lo(x) <= hi(y) ) & ( hi(x) >= lo( y))
  x <- x[!is.na(x)]
  y <- if( missing(y) ) x else y[!is.na(y)]
  rv <- outer( x , y, overlaps)
  if( rm.diag) diag(rv) <- 0
  ij <- which(rv>0,arr.ind = TRUE)
  data.frame(nm=names(x)[ij[,1]], x=x[ij[,1]], y=y[ij[,2]])
}
##
ipr.overlaps(ipv4.reserved(),rm.diag=TRUE)
##
## Find overlapping IPv4 ranges (IP package)
##
bsearch <- ip.index(ipv4.reserved(), overlap=TRUE)
##
m <- bsearch()
## get the indices
idx <- getIdx(m)
## matches indices
midx <- idx$midx
## start indices for each address in the midx vector
## (diff(ptr) gives the number of matches per address)
ptr <- idx$ptr
##
subset(
  data.frame(
       nm  = names(ipv4.reserved()[midx])
     , x   = rep(m, diff(ptr))
     , tbl = ipv4.reserved()[midx]
     , n   = rep(diff(ptr), diff(ptr))
  )
  , n>1 & x!=tbl
)
##
```

```
## Same thing for IPv6r
##
ip.index(ipv6.reserved(), overlap=TRUE)(value=TRUE)
```

Index