netaddr Documentation

Release 0.7.19

David P. D. Moss

May 09, 2017

Contents

CHAPTER 1

Introduction

A Python library for representing and manipulating network addresses.

Provides support for:

Layer 3 addresses

- IPv4 and IPv6 addresses, subnets, masks, prefixes
- iterating, slicing, sorting, summarizing and classifying IP networks
- dealing with various ranges formats (CIDR, arbitrary ranges and globs, nmap)
- set based operations (unions, intersections etc) over IP addresses and subnets
- parsing a large variety of different formats and notations
- looking up IANA IP block information
- generating DNS reverse lookups
- supernetting and subnetting

Layer 2 addresses

- representation and manipulation MAC addresses and EUI-64 identifiers
- looking up IEEE organisational information (OUI, IAB)
- generating derived IPv6 addresses

Changes

For details on the latest updates and changes, see *What's new in netaddr 0.7.19*

License

This software is released under the liberal BSD license.

See the *License* and *Copyright* for full text.

Dependencies

- Python 2.5.x through 3.5.x
- IPython (for netaddr interactive shell)

Installation

See *Installing netaddr* for details.

Documentation

This library has comprehensive docstrings and a full set of project documentation (including tutorials):

- http://pythonhosted.org/netaddr/
- http://netaddr.readthedocs.io/en/latest/

Tests

netaddr requires py.test (http://pytest.org/).

To run the test suite, clone the repository and run:

```
python setup.py test
```

If any of the tests fail, *please* help the project's user base by filing bug reports on the netaddr issue tracker:

- http://github.com/drkjam/netaddr/issues

Finally...

Share and enjoy!

CHAPTER 2

Installing netaddr

netaddr is available in various formats :

- source code repository
- source distribution packages (tarball and zip formats)
- Python universal wheel packages

Various Linux distributions make it available via their package managers.

Locating the software

The netaddr project is hosted here on github

http://github.com/drkjam/netaddr/

Installing from the Python Package Index

The easiest way to install netaddr is to use pip.

Download and install the latest version from PyPI - http://pypi.python.org/pypi/pip and run the following command

```
pip install netaddr
```

Installing from a source package

Download the latest release tarball/zip file and extract it to a temporary directory or clone the repository into a local working directory.

Run the setup file from directory:

```
python setup.py install
```

This automatically places the required files in the `lib/site-packages` directory of the Python version you used to run the setup script, may also be part of a virtualenv or similar environment manager.

CHAPTER 3

Tutorial 1: IP Addresses, Subnets and Ranges

First of all you need to pull the various netaddr classes and functions into your namespace.

Note: Do this for the purpose of this tutorial only. In your own code, you should be explicit about the classes, functions and constants you import to avoid name clashes.

```
>>> from netaddr import *
```

We also import the standard library module *pprint* to help format our output.

```
>>> import pprint
```

Basic operations

The following *IPAddress* object represents a single IP address.

```
>>> ip = IPAddress('192.0.2.1')
>>> ip.version
4
```

The *repr()* call returns a Python statement that can be used to reconstruct an equivalent IP address object state from scratch when run in the Python interpreter.

```
>>> repr(ip)
"IPAddress('192.0.2.1')"
>>> ip
IPAddress('192.0.2.1')
```

Access in the string context returns the IP object as a string value.

```
>>> str(ip)
'192.0.2.1'
>>> '%s' % ip
'192.0.2.1'
>>> ip.format()  # only really useful for IPv6 addresses.
'192.0.2.1'
```

Numerical representation

You can view an IP address in various other formats.

```
>>> int(ip) == 3221225985
True
>>> hex(ip)
'0xc0000201'
>>> ip.bin
'0b11000000000000000000001000000001'
>>> ip.bits()
'11000000.00000000.00000010.00000001'
>>> ip.words == (192, 0, 2, 1)
True
```

Representing networks and subnets

IPNetwork objects are used to represent subnets, networks or VLANs that accept CIDR prefixes and netmasks.

```
>>> ip = IPNetwork('192.0.2.1')
>>> ip.ip
IPAddress('192.0.2.1')
>>> ip.network, ip.broadcast
(IPAddress('192.0.2.1'), None)
>>> ip.netmask, ip.hostmask
(IPAddress('255.255.255.255'), IPAddress('0.0.0.0'))
>>> ip.size
1
```

In this case, the network and broadcast address are the same, akin to a host route.

```
>>> ip = IPNetwork('192.0.2.0/24')
>>> ip.ip
IPAddress('192.0.2.0')
>>> ip.network, ip.broadcast
(IPAddress('192.0.2.0'), IPAddress('192.0.2.255'))
>>> ip.netmask, ip.hostmask
(IPAddress('255.255.255.0'), IPAddress('0.0.0.255'))
>>> ip.size
256
```

And finally, this IPNetwork object represents an IP address that belongs to a given IP subnet.

```
>>> ip = IPNetwork('192.0.3.112/22')
>>> ip.ip
IPAddress('192.0.3.112')
```

```
>>> ip.network, ip.broadcast
(IPAddress('192.0.0.0'), IPAddress('192.0.3.255'))
>>> ip.netmask, ip.hostmask
(IPAddress('255.255.252.0'), IPAddress('0.0.3.255'))
>>> ip.size
1024
```

Internally, each IPNetwork object only stores 3 values :-

- the IP address value as an unsigned integer
- a reference to the IP protocol module for the IP version being represented
- the CIDR prefix bitmask

All the other values are calculated on-the-fly on access.

It is possible to adjust the IP address value and the CIDR prefix after object instantiation.

```
>>> ip = IPNetwork('0.0.0.0/0')
>>> ip
IPNetwork('0.0.0.0/0')
>>> ip.value = 3221225985
>>> ip
IPNetwork('192.0.2.1/0')
>>> ip.prefixlen
0
>>> ip.prefixlen = 23
>>> ip
IPNetwork('192.0.2.1/23')
```

There is also a property that lets you access the *true* CIDR address which removes all host bits from the network address based on the CIDR subnet prefix.

```
>>> ip.cidr
IPNetwork('192.0.2.0/23')
```

This is handy for specifying some networking configurations correctly.

If you want to access information about each of the various IP addresses that form the IP subnet, this is available by performing pass through calls to sub methods of each *IPAddress* object.

For example if you want to see a binary digit representation of each address you can do the following.

```
>>> ip.ip.bits()
'11000000.00000000.00000010.00000001'
>>> ip.network.bits()
'11000000.00000000.00000010.00000000'
>>> ip.netmask.bits()
'11111111.11111111.11111110.00000000'
>>> ip.broadcast.bits()
'11000000.00000000.00000011.11111111'
```

IPv6 support

Full support for IPv6 is provided. Let's try a few examples:

```
>>> ip = IPAddress(0, 6)
>>> ip
IPAddress('::')
>>> ip = IPNetwork('fe80::dead:beef/64')
>>> str(ip), ip.prefixlen, ip.version
('fe80::dead:beef/64', 64, 6)
>>> int(ip.ip) == 338288524927261089654018896845083623151
True
>>> hex(ip.ip)
'0xfe8000000000000000000000deadbeef'
```

Bit-style output isn't as quite as friendly as hexadecimal for such a long numbers, but here the proof that it works!

```
>>> ip.ip.bits()

↪'1111111010000000:0000000000000000:0000000000000000:0000000000000000:0000000000000000:0000000000000
↪'
```

Here are some networking details for an IPv6 subnet.

```
>>> ip.network, ip.broadcast, ip.netmask, ip.hostmask
(IPAddress('fe80::'), IPAddress('fe80::ffff:ffff:ffff:ffff'), IPAddress(
↪'ffff:ffff:ffff:ffff::'), IPAddress('::ffff:ffff:ffff:ffff'))
```

Interoperability between IPv4 and IPv6

It is likely that with IPv6 becoming more prevalent, you'll want to be able to interoperate between IPv4 and IPv6 address seamlessly.

Here are a couple of methods that help achieve this.

IPv4 to IPv6 conversion

```
>>> IPAddress('192.0.2.15').ipv4()
IPAddress('192.0.2.15')
>>> ip = IPAddress('192.0.2.15').ipv6()
>>> ip
IPAddress('::ffff:192.0.2.15')
>>> ip.is_ipv4_mapped()
True
>>> ip.is_ipv4_compat()
False
```

```
>>> IPAddress('192.0.2.15').ipv6(ipv4_compatible=True)
IPAddress('::192.0.2.15')
>>> IPAddress('192.0.2.15').ipv6(ipv4_compatible=True).is_ipv4_compat()
True
>>> IPAddress('192.0.2.15').ipv6(True)
IPAddress('::192.0.2.15')
>>> ip = IPNetwork('192.0.2.1/23')
>>> ip.ipv4()
IPNetwork('192.0.2.1/23')
>>> ip.ipv6()
```

```
IPNetwork('::ffff:192.0.2.1/119')
>>> ip.ipv6(ipv4_compatible=True)
IPNetwork('::192.0.2.1/119')
```

IPv6 to IPv4 conversion

```
>>> IPNetwork('::ffff:192.0.2.1/119').ipv6()
IPNetwork('::ffff:192.0.2.1/119')
>>> IPNetwork('::ffff:192.0.2.1/119').ipv6(ipv4_compatible=True)
IPNetwork('::192.0.2.1/119')
>>> IPNetwork('::ffff:192.0.2.1/119').ipv4()
IPNetwork('192.0.2.1/23')
>>> IPNetwork('::192.0.2.1/119').ipv4()
IPNetwork('192.0.2.1/23')
```

Note that the IP object returns IPv4 "mapped" addresses by default in preference to IPv4 "compatible" ones. This has been chosen purposefully as the latter form has been deprecated (see RFC 4291 for details).

List operations

If you treat an *IPNetwork* object as if it were a standard Python list object it will give you access to a list of individual IP address objects. This of course is illusory and they are not created until you access them.

```
>>> ip = IPNetwork('192.0.2.16/29')
```

Accessing an IP object using the *list()* context invokes the default generator which returns a list of all IP objects in the range specified by the IP object's subnet.

```
>>> ip_list = list(ip)
>>> len(ip_list)
8
>>> ip_list
[IPAddress('192.0.2.16'), IPAddress('192.0.2.17'), ..., IPAddress('192.0.2.22'),␣
↪IPAddress('192.0.2.23')]
```

The length of that list is 8 individual IP addresses.

```
>>> len(ip)
8
```

Indexing

You can use standard index access to IP addresses in the subnet.

```
>>> ip[0]
IPAddress('192.0.2.16')
>>> ip[1]
IPAddress('192.0.2.17')
>>> ip[-1]
IPAddress('192.0.2.23')
```

Slicing

You can also use list slices on IP addresses in the subnet.

```
>>> ip[0:4]
<generator object ...>
```

The slice is a generator function. This was done to save time and system resources as some slices can end up being very large for certain subnets!

Here is how you'd access all elements in a slice.

```
>>> list(ip[0:4])
[IPAddress('192.0.2.16'), IPAddress('192.0.2.17'), IPAddress('192.0.2.18'), IPAddress(
↪'192.0.2.19')]
```

Extended slicing is also supported.

```
>>> list(ip[0::2])
[IPAddress('192.0.2.16'), IPAddress('192.0.2.18'), IPAddress('192.0.2.20'), IPAddress(
↪'192.0.2.22')]
```

List reversal.

```
>>> list(ip[-1::-1])
[IPAddress('192.0.2.23'), IPAddress('192.0.2.22'), ..., IPAddress('192.0.2.17'),␣
↪IPAddress('192.0.2.16')]
```

Use of generators ensures working with large IP subnets is efficient.

```
>>> for ip in IPNetwork('192.0.2.0/23'):
...     print '%s' % ip
...
192.0.2.0
192.0.2.1
192.0.2.2
192.0.2.3
...
192.0.3.252
192.0.3.253
192.0.3.254
192.0.3.255
```

In IPv4 networks you only usually assign the addresses between the network and broadcast addresses to actual host interfaces on systems.

Here is the iterator provided for accessing these IP addresses :-

```
>>> for ip in IPNetwork('192.0.2.0/23').iter_hosts():
...     print '%s' % ip
...
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
...
192.0.3.251
192.0.3.252
```

```
192.0.3.253
192.0.3.254
```

Sorting IP addresses and networks

It is fairly common and useful to be able to sort IP addresses and networks canonically.

Here is how sorting works with individual addresses.

```
>>> import random
>>> ip_list = list(IPNetwork('192.0.2.128/28'))
>>> random.shuffle(ip_list)
>>> sorted(ip_list)
[IPAddress('192.0.2.128'), IPAddress('192.0.2.129'), ..., IPAddress('192.0.2.142'),␣
↪IPAddress('192.0.2.143')]
```

For convenience, you are able to sort IP subnets at the same time as addresses and they can be combinations of IPv4 and IPv6 addresses at the same time as well (IPv4 addresses and network appear before IPv6 ones).

```
>>> ip_list = [
... IPAddress('192.0.2.130'),
... IPAddress('10.0.0.1'),
... IPNetwork('192.0.2.128/28'),
... IPNetwork('192.0.3.0/24'),
... IPNetwork('192.0.2.0/24'),
... IPNetwork('fe80::/64'),
... IPAddress('::'),
... IPNetwork('172.24/12')]
>>> random.shuffle(ip_list)
>>> ip_list.sort()
>>> pprint.pprint(ip_list)
[IPAddress('10.0.0.1'),
 IPNetwork('172.24.0.0/12'),
 IPNetwork('192.0.2.0/24'),
 IPNetwork('192.0.2.128/28'),
 IPAddress('192.0.2.130'),
 IPNetwork('192.0.3.0/24'),
 IPAddress('::'),
 IPNetwork('fe80::/64')]
```

Notice how overlapping subnets also sort in order from largest to smallest.

Summarizing list of addresses and subnets

Another useful operation is the ability to summarize groups of IP subnets and addresses, merging them together where possible to create the smallest possible list of CIDR subnets.

You do this in netaddr using the *cidr_merge()* function.

First we create a list of IP objects that contains a good mix of individual addresses and subnets, along with some string based IP address values for good measure. To make things more interesting some IPv6 addresses are thrown in as well.

```
>>> ip_list = [ip for ip in IPNetwork('fe80::/120')]
>>> ip_list.append(IPNetwork('192.0.2.0/24'))
>>> ip_list.extend([str(ip) for ip in IPNetwork('192.0.3.0/24')])
>>> ip_list.append(IPNetwork('192.0.4.0/25'))
>>> ip_list.append(IPNetwork('192.0.4.128/25'))
>>> len(ip_list)
515
>>> cidr_merge(ip_list)
[IPNetwork('192.0.2.0/23'), IPNetwork('192.0.4.0/24'), IPNetwork('fe80::/120')]
```

Useful isn't it?

Supernets and subnets

It is quite common to have a large CIDR subnet that you may want to split up into multiple smaller component blocks to better manage your network allocations, firewall rules etcc and netaddr gives you the tools required to do this.

Here we take a large /16 private class B network block and split it up into a set of smaller 512 sized blocks.

```
>>> ip = IPNetwork('172.24.0.0/16')
>>> ip.subnet(23)
<generator object ...>
```

Once again, this method produces and iterator because of the possibility for a large number of return values depending on this subnet size specified.

```
>>> subnets = list(ip.subnet(23))
>>> len(subnets)
128
>>> subnets
[IPNetwork('172.24.0.0/23'), IPNetwork('172.24.2.0/23'), IPNetwork('172.24.4.0/23'), .
↪.., IPNetwork('172.24.250.0/23'), IPNetwork('172.24.252.0/23'), IPNetwork('172.24.
↪254.0/23')]
```

It is also possible to retrieve the list of supernets that a given IP address or subnet belongs to. You can also specify an optional limit.

```
>>> ip = IPNetwork('192.0.2.114')
>>> supernets = ip.supernet(22)
>>> pprint.pprint(supernets)
[IPNetwork('192.0.0.0/22'),
 IPNetwork('192.0.2.0/23'),
 IPNetwork('192.0.2.0/24'),
 IPNetwork('192.0.2.0/25'),
 IPNetwork('192.0.2.64/26'),
 IPNetwork('192.0.2.96/27'),
 IPNetwork('192.0.2.112/28'),
 IPNetwork('192.0.2.112/29'),
 IPNetwork('192.0.2.112/30'),
 IPNetwork('192.0.2.114/31')]
```

Here, we return a list rather than a generator because the potential list of values is of a predictable size (no more than 31 subnets for an IPv4 address and 127 for IPv6).

Support for non-standard address ranges

While CIDR is a useful way to describe networks succinctly, it is often necessary (particularly with IPv4 which predates the CIDR specification) to be able to generate lists of IP addresses that have an arbitrary start and end address that do not fall on strict bit mask boundaries.

The *iter_iprange()* function allow you to do just this.

```
>>> ip_list = list(iter_iprange('192.0.2.1', '192.0.2.14'))
>>> len(ip_list)
14
>>> ip_list
[IPAddress('192.0.2.1'), IPAddress('192.0.2.2'), ..., IPAddress('192.0.2.13'),␣
↪IPAddress('192.0.2.14')]
```

It is equally nice to know what the actual list of CIDR subnets is that would correctly cover this non-aligned range of addresses.

Here *cidr_merge()* comes to the rescue once more.

```
>>> cidr_merge(ip_list)
[IPNetwork('192.0.2.1/32'), IPNetwork('192.0.2.2/31'), IPNetwork('192.0.2.4/30'),␣
↪IPNetwork('192.0.2.8/30'), IPNetwork('192.0.2.12/31'), IPNetwork('192.0.2.14/32')]
```

Dealing with older IP network specifications

Until the advent of the CIDR specification it was common to infer the netmask of an IPv4 address based on its first octet using an set of classful rules (first defined in RFC 791).

You frequently come across reference to them in various RFCs and they are well supported by a number of software libraries. For completeness, rather than leave out this important (but now somewhat historical) set of rules, they are supported via the cryptically named *cidr_abbrev_to_verbose()* function.

Here is an example of these rules for the whole of the IPv4 address space.

```
>>> cidrs = [cidr_abbrev_to_verbose(octet) for octet in range(0, 256)]
>>> pprint.pprint(cidrs)
['0.0.0.0/8',
...
 '127.0.0.0/8',
 '128.0.0.0/16',
...
 '191.0.0.0/16',
 '192.0.0.0/24',
...
 '223.0.0.0/24',
 '224.0.0.0/4',
...
 '239.0.0.0/4',
 '240.0.0.0/32',
...
 '255.0.0.0/32']
>>> len(cidrs)
256
```

IP address categorisation

IP addresses fall into several categories, not all of which are suitable for assignment as host addresses.

Unicast

```
>>> IPAddress('192.0.2.1').is_unicast()
True
>>> IPAddress('fe80::1').is_unicast()
True
```

Multicast

Used to indentify multicast groups (see RFC 2365 and 3171 for more info).

```
>>> IPAddress('239.192.0.1').is_multicast()
True
>>> IPAddress('ff00::1').is_multicast()
True
```

Private

Found on intranets and used behind NAT routers.

```
>>> IPAddress('172.24.0.1').is_private()
True
>>> IPAddress('10.0.0.1').is_private()
True
>>> IPAddress('192.168.0.1').is_private()
True
>>> IPAddress('fc00::1').is_private()
True
```

Reserved

Addresses in reserved ranges are not available for general use.

```
>>> IPAddress('253.0.0.1').is_reserved()
True
```

Public

Addresses accessible via the Internet.

Note: circa the end of 2011 all IPv4 addresses had been allocated to the Regional Internet Registrars. A booming after market in IPv4 addresses has started. There is still plenty of life left in this protocol version yet :)

```
>>> ip = IPAddress('62.125.24.5')
>>> ip.is_unicast() and not ip.is_private()
True
```

Netmasks

A bitmask used to divide an IP address into its network address and host address.

```
>>> IPAddress('255.255.254.0').is_netmask()
True
```

Hostmasks

Similar to a netmask but with the all the bits flipped the opposite way.

```
>>> IPAddress('0.0.1.255').is_hostmask()
True
```

Loopback

These addresses are used internally within an IP network stack and packets sent to these addresses are not distributed via a physical network connection.

```
>>> IPAddress('127.0.0.1').is_loopback()
True
>>> IPAddress('::1').is_loopback()
True
```

Comparing IP addresses

IPAddress objects can be compared with each other. As an *IPAddress* object can represent both an individual IP address and an implicit network, it pays to get both sides of your comparison into the same terms before you compare them to avoid odd results.

Here are some comparisons of individual IP address to get the ball rolling.

```
>>> IPAddress('192.0.2.1') == IPAddress('192.0.2.1')
True
>>> IPAddress('192.0.2.1') < IPAddress('192.0.2.2')
True
>>> IPAddress('192.0.2.2') > IPAddress('192.0.2.1')
True
>>> IPAddress('192.0.2.1') != IPAddress('192.0.2.1')
False
>>> IPAddress('192.0.2.1') >= IPAddress('192.0.2.1')
True
>>> IPAddress('192.0.2.2') >= IPAddress('192.0.2.1')
True
>>> IPAddress('192.0.2.1') <= IPAddress('192.0.2.1')
True
```

```
>>> IPAddress('192.0.2.1') <= IPAddress('192.0.2.2')
True
```

Now, lets try something a little more interesting.

```
>>> IPNetwork('192.0.2.0/24') == IPNetwork('192.0.2.112/24')
True
```

Hmmmmmmmm... looks a bit odd doesn't it? That's because by default, IP objects compare their subnets (or lower and upper boundaries) rather than their individual IP address values.

The solution to this situation is very simple. Knowing this default behaviour, just be explicit about exactly which portion of each IP object you'd like to compare using pass-through properties.

```
>>> IPNetwork('192.0.2.0/24').ip == IPNetwork('192.0.2.112/24').ip
False
>>> IPNetwork('192.0.2.0/24').ip < IPNetwork('192.0.2.112/24').ip
True
```

That's more like it. You can also be explicit about comparing networks in this way if you so wish (although it is not strictly necessary).

```
>>> IPNetwork('192.0.2.0/24').cidr == IPNetwork('192.0.2.112/24').cidr
True
```

Armed with this information here are some examples of network comparisons.

```
>>> IPNetwork('192.0.2.0/24') == IPNetwork('192.0.3.0/24')
False
>>> IPNetwork('192.0.2.0/24') < IPNetwork('192.0.3.0/24')
True
```

This will inevitably raise questions about comparing IPAddress (scalar) objects and IPNetwork (vector) objects with each other (or at least it should).

Here is how netaddr chooses to address this situation.

```
>>> IPAddress('192.0.2.0') == IPNetwork('192.0.2.0/32')
False
>>> IPAddress('192.0.2.0') != IPNetwork('192.0.2.0/32')
True
```

An IP network or subnet is different from an individual IP address and therefore cannot be (directly) compared.

If you want to compare them successfully, you must be explicit about which aspect of the IP network you wish to match against the IP address in question.

You can use the index of the first or last address if it is a /32 like so :-

```
>>> IPAddress('192.0.2.0') == IPNetwork('192.0.2.0/32')[0]
True
>>> IPAddress('192.0.2.0') == IPNetwork('192.0.2.0/32')[-1]
True
>>> IPAddress('192.0.2.0') != IPNetwork('192.0.2.0/32')[0]
False
```

You can also use the base address if this is what you wish to compare :-

```
>>> IPAddress('192.0.2.0') == IPNetwork('192.0.2.0/32').ip
True
>>> IPAddress('192.0.2.0') != IPNetwork('192.0.2.0/32').ip
False
```

While this may seem a bit pointless at first, netaddr strives to keep IP addresses and network separate from one another while still allowing reasonable interoperability.

DNS support

It is a common administrative task to generate reverse IP lookups for DNS. This is particularly arduous for IPv6 addresses.

Here is how you do this using an IPAddress object's *reverse_dns()* method.

```
>>> IPAddress('172.24.0.13').reverse_dns
'13.0.24.172.in-addr.arpa.'
>>> IPAddress('fe80::feeb:daed').reverse_dns
'd.e.a.d.b.e.e.f.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.e.f.ip6.arpa.'
```

Note that `ip6.int` is not used as this has been deprecated (see RFC 3152 for details).

Non standard address ranges

As CIDR is a relative newcomer given the long history of IP version 4 you are quite likely to come across systems and documentation which make reference to IP address ranges in formats other than CIDR. Converting from these arbitrary range types to CIDR and back again isn't a particularly fun task. Fortunately, netaddr tries to make this job easy for you with two purpose built classes.

Arbitrary IP address ranges

You can represent an arbitrary IP address range using a lower and upper bound address in the form of an IPRange object.

```
>>> r1 = IPRange('192.0.2.1', '192.0.2.15')
>>> r1
IPRange('192.0.2.1', '192.0.2.15')
```

You can iterate across and index these ranges just like and IPNetwork object.

Importantly, you can also convert it to it's CIDR equivalent.

```
>>> r1.cidrs()
[IPNetwork('192.0.2.1/32'), IPNetwork('192.0.2.2/31'), IPNetwork('192.0.2.4/30'),␣
↪IPNetwork('192.0.2.8/29')]
```

Here is how individual IPRange and IPNetwork compare.

```
>>> IPRange('192.0.2.0', '192.0.2.255') != IPNetwork('192.0.2.0/24')
False
>>> IPRange('192.0.2.0', '192.0.2.255') == IPNetwork('192.0.2.0/24')
True
```

You may wish to compare an IP range against a list of IPAddress and IPNetwork objects.

```
>>> r1 = IPRange('192.0.2.1', '192.0.2.15')
>>> addrs = list(r1)
>>> addrs
[IPAddress('192.0.2.1'), IPAddress('192.0.2.2'), IPAddress('192.0.2.3'), IPAddress(
↪'192.0.2.4'), IPAddress('192.0.2.5'), IPAddress('192.0.2.6'), IPAddress('192.0.2.7
↪'), IPAddress('192.0.2.8'), IPAddress('192.0.2.9'), IPAddress('192.0.2.10'),␣
↪IPAddress('192.0.2.11'), IPAddress('192.0.2.12'), IPAddress('192.0.2.13'),␣
↪IPAddress('192.0.2.14'), IPAddress('192.0.2.15')]
>>> r1 == addrs
False
```

Oops! Not quite what we were looking for or expecting.

The way to do this is to get either side of the comparison operation into the same terms.

```
>>> list(r1) == addrs
True
```

That's more like it.

The same goes for IPNetwork objects.

```
>>> subnets = r1.cidrs()
>>> subnets
[IPNetwork('192.0.2.1/32'), IPNetwork('192.0.2.2/31'), IPNetwork('192.0.2.4/30'),␣
↪IPNetwork('192.0.2.8/29')]
>>> r1 == subnets
False
>>> r1.cidrs() == subnets
True
```

The above works if the list you are comparing contains one type or the other, but what if you have a mixed list of *IPAddress*, *IPNetwork* and string addresses?

Time for some slightly more powerful operations. Let's make use of a new class for dealing with groups of IP addresses and subnets. The IPSet class.

```
>>> ips = [IPAddress('192.0.2.1'), '192.0.2.2/31', IPNetwork('192.0.2.4/31'),␣
↪IPAddress('192.0.2.6'), IPAddress('192.0.2.7'), '192.0.2.8', '192.0.2.9', IPAddress(
↪'192.0.2.10'), IPAddress('192.0.2.11'), IPNetwork('192.0.2.12/30')]
>>> s1 = IPSet(r1.cidrs())
>>> s2 = IPSet(ips)
>>> s2
IPSet(['192.0.2.1/32', '192.0.2.2/31', '192.0.2.4/30', '192.0.2.8/29'])
>>> s1 == s2
True
```

Let's remove one of the element from one of the IPSet objects and see what happens.

```
>>> s2.pop()
IPNetwork('192.0.2.4/30')
>>> s1 == s2
False
```

This is perhaps a somewhat contrived example but it just shows you some of the capabilities on offer.

See the IPSet tutorial *Tutorial 3: Working with IP sets* for more details on that class.

IP Glob ranges

netaddr also supports a user friendly form of specifying IP address ranges using a "glob" style syntax.

Note: At present only IPv4 globs are supported.

```
>>> IPGlob('192.0.2.*') == IPNetwork('192.0.2.0/24')
True
```

IPGlob('192.0.2.*') != IPNetwork('192.0.2.0/24') False

As *IPGlob* is a subclass of *IPRange*, all of the same operations apply.

CHAPTER 4

Tutorial 2: MAC addresses

First of all you need to pull the various MAC related classes and functions into your namespace.

Note: Do this for the purpose of this tutorial only. In your own code, you should be explicit about the classes, functions and constants you import to avoid name clashes.

```
>>> from netaddr import *
```

You can reasonably safely import everything from the netaddr namespace as care has been taken to only export the necessary classes, functions and constants.

Always hand pick your imports if you are unsure about possible name clashes.

Basic operations

Instances of the EUI class are used to represent MAC addresses.

```
>>> mac = EUI('00-1B-77-49-54-FD')
```

Standard repr() access returns a Python statement that can reconstruct the MAC address object from scratch if executed in the Python interpreter.

```
>>> mac
EUI('00-1B-77-49-54-FD')
```

Accessing the EUI object in the string context.

```
>>> str(mac)
'00-1B-77-49-54-FD'
>>> '%s' % mac
'00-1B-77-49-54-FD'
```

Here are a few other common properties.

```
>>> str(mac), str(mac.oui), mac.ei, mac.version
('00-1B-77-49-54-FD', '00-1B-77', '49-54-FD', 48)
```

Numerical representations

You can view an individual IP address in various other formats.

```
>>> int(mac) == 117965411581
True
>>> hex(mac)
'0x1b774954fd'
>>> oct(mac)
'01556722252375'
>>> mac.bits()
'00000000-00011011-01110111-01001001-01010100-11111101'
>>> mac.bin
'0b110110111011101001001010100111111101'
```

Formatting

It is very common to see MAC address in many different formats other than the standard IEEE EUI-48.

The EUI class constructor handles all these common forms.

```
>>> EUI('00-1B-77-49-54-FD')
EUI('00-1B-77-49-54-FD')
```

IEEE EUI-48 lowercase format

```
>>> EUI('00-1b-77-49-54-fd')
EUI('00-1B-77-49-54-FD')
```

Common UNIX format

```
>>> EUI('0:1b:77:49:54:fd')
EUI('00-1B-77-49-54-FD')
```

Cisco triple hextet format

```
>>> EUI('001b:7749:54fd')
EUI('00-1B-77-49-54-FD')
>>> EUI('1b:7749:54fd')
EUI('00-1B-77-49-54-FD')
>>> EUI('1B:7749:54FD')
EUI('00-1B-77-49-54-FD')
```

Bare MAC addresses (no delimiters)

```
>>> EUI('001b774954fd')
EUI('00-1B-77-49-54-FD')
>>> EUI('01B774954FD')
EUI('00-1B-77-49-54-FD')
```

PostreSQL format (found in documentation)

```
>>> EUI('001B77:4954FD')
EUI('00-1B-77-49-54-FD')
```

It is equally possible to specify a selected format for your MAC string output in the form of a 'dialect' class. Its use is similar to the dialect class used in the Python standard library csv module.

```
>>> mac = EUI('00-1B-77-49-54-FD')
>>> mac
EUI('00-1B-77-49-54-FD')
>>> mac.dialect = mac_unix
>>> mac
EUI('0:1b:77:49:54:fd')
>>> mac.dialect = mac_unix_expanded
>>> mac
EUI('00:1b:77:49:54:fd')
>>> mac.dialect = mac_cisco
>>> mac
EUI('001b.7749.54fd')
>>> mac.dialect = mac_bare
>>> mac
EUI('001B774954FD')
>>> mac.dialect = mac_pgsql
>>> mac
EUI('001b77:4954fd')
```

You can, of course, create your own dialect classes to customise the MAC formatting if the standard ones do not suit your needs.

Here's a tweaked UNIX MAC dialect that generates uppercase, zero-filled octets.

```
>>> class mac_custom(mac_unix): pass
>>> mac_custom.word_fmt = '%.2X'
>>> mac = EUI('00-1B-77-49-54-FD', dialect=mac_custom)
>>> mac
EUI('00:1B:77:49:54:FD')
```

Querying organisational information

EUI objects provide an interface to the OUI (Organisationally Unique Identifier) and IAB (Individual Address Block) registration databases available from the IEEE.

Here is how you query an OUI with the EUI interface.

```
>>> mac = EUI('00-1B-77-49-54-FD')
>>> oui = mac.oui
>>> oui
OUI('00-1B-77')
>>> oui.registration().address
[u'Lot 8, Jalan Hi-Tech 2/3', u'Kulim  Kedah  09000', u'MY']
>>> oui.registration().org
u'Intel Corporate'
```

You can also use OUI objects directly without going through the EUI interface.

A few OUI records have multiple registrations against them. I'm not sure if this is recording historical information or just a quirk of the IEEE reigstration process.

This example shows you how you access them individually by specifying an index number.

```
>>> oui = OUI(524336)    #   OUI constructor accepts integer values, too.
>>> oui
OUI('08-00-30')
>>> oui.registration(0).address
[u'2380 N. ROSE AVENUE', u'OXNARD  CA  93010', u'US']
>>> oui.registration(0).org
u'NETWORK RESEARCH CORPORATION'
>>> oui.registration(0).oui
'08-00-30'
>>> oui.registration(1).address
[u'GPO BOX 2476V', u'MELBOURNE  VIC  3001', u'AU']
>>> oui.registration(1).org
u'ROYAL MELBOURNE INST OF TECH'
>>> oui.registration(1).oui
'08-00-30'
>>> oui.registration(2).address
[u'CH-1211 GENEVE 23', u'SUISSE/SWITZ', u'CH']
>>> oui.registration(2).org
u'CERN'
>>> oui.registration(2).oui
'08-00-30'
>>> for i in range(oui.reg_count):
...     str(oui), oui.registration(i).org
...
('08-00-30', u'NETWORK RESEARCH CORPORATION')
('08-00-30', u'ROYAL MELBOURNE INST OF TECH')
('08-00-30', u'CERN')
```

Here is how you query an IAB with the EUI interface.

```
>>> mac = EUI('00-50-C2-00-0F-01')
>>> mac.is_iab()
True
>>> iab = mac.iab
>>> iab
IAB('00-50-C2-00-00-00')
>>> iab.registration()
{'address': [u'1241 Superieor Ave E', u'Cleveland  OH  44114', u'US'],
 'iab': '00-50-C2-00-00-00',
 'idx': 84680704,
 ...
 'org': u'T.L.S. Corp.',
 'size': 537}
```

CHAPTER 5

Tutorial 3: Working with IP sets

First of all you need to pull the various netaddr classes and functions into your namespace.

Note: Do this for the purpose of this tutorial only. In your own code, you should be explicit about the classes, functions and constants you import to avoid name clashes.

```
>>> from netaddr import *
```

Creating IP sets

Here how to create IP sets.

An empty set.

```
>>> IPSet()
IPSet([])
>>> IPSet([])
IPSet([])
>>> len(IPSet([]))
0
```

You can specify either IP addresses and networks as strings. Alternatively, you can use IPAddress, IPNetwork, IPRange or other IPSet objects.

```
>>> IPSet(['192.0.2.0'])
IPSet(['192.0.2.0/32'])
>>> IPSet([IPAddress('192.0.2.0')])
IPSet(['192.0.2.0/32'])
>>> IPSet([IPNetwork('192.0.2.0')])
IPSet(['192.0.2.0/32'])
>>> IPSet(IPNetwork('1234::/32'))
IPSet(['1234::/32'])
```

```
>>> IPSet([IPNetwork('192.0.2.0/24')])
IPSet(['192.0.2.0/24'])
>>> IPSet(IPSet(['192.0.2.0/32']))
IPSet(['192.0.2.0/32'])
>>> IPSet(IPRange("10.0.0.0", "10.0.1.31"))
IPSet(['10.0.0.0/24', '10.0.1.0/27'])
>>> IPSet(IPRange('0.0.0.0', '255.255.255.255'))
IPSet(['0.0.0.0/0'])
```

You can interate over all the IP addresses that are members of the IP set.

```
>>> for ip in IPSet(['192.0.2.0/28', '::192.0.2.0/124']):
...     print ip
192.0.2.0
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
192.0.2.5
192.0.2.6
192.0.2.7
192.0.2.8
192.0.2.9
192.0.2.10
192.0.2.11
192.0.2.12
192.0.2.13
192.0.2.14
192.0.2.15
::192.0.2.0
::192.0.2.1
::192.0.2.2
::192.0.2.3
::192.0.2.4
::192.0.2.5
::192.0.2.6
::192.0.2.7
::192.0.2.8
::192.0.2.9
::192.0.2.10
::192.0.2.11
::192.0.2.12
::192.0.2.13
::192.0.2.14
::192.0.2.15
```

Adding and removing set elements

```
>>> s1 = IPSet()
>>> s1.add('192.0.2.0')
>>> s1
IPSet(['192.0.2.0/32'])
>>> s1.remove('192.0.2.0')
>>> s1
IPSet([])
```

```
>>> s1.add(IPRange("10.0.0.0", "10.0.0.255"))
>>> s1
IPSet(['10.0.0.0/24'])
>>> s1.remove(IPRange("10.0.0.128", "10.10.10.10"))
>>> s1
IPSet(['10.0.0.0/25'])
```

Set membership

Here is a simple arbitrary IP address range.

```
>>> iprange = IPRange('192.0.1.255', '192.0.2.16')
```

We can see the CIDR networks that can existing with this defined range.

```
>>> iprange.cidrs()
[IPNetwork('192.0.1.255/32'), IPNetwork('192.0.2.0/28'), IPNetwork('192.0.2.16/32')]
```

Here's an IP set.

```
>>> ipset = IPSet(['192.0.2.0/28'])
```

Now, let's iterate over the IP addresses in the arbitrary IP address range and see if they are found within the IP set.

```
>>> for ip in iprange:
...     print ip, ip in ipset
192.0.1.255 False
192.0.2.0 True
192.0.2.1 True
192.0.2.2 True
192.0.2.3 True
192.0.2.4 True
192.0.2.5 True
192.0.2.6 True
192.0.2.7 True
192.0.2.8 True
192.0.2.9 True
192.0.2.10 True
192.0.2.11 True
192.0.2.12 True
192.0.2.13 True
192.0.2.14 True
192.0.2.15 True
192.0.2.16 False
```

More exotic IPSets

```
>>> bigone = IPSet(['0.0.0.0/0'])
>>> IPAddress("10.0.0.1") in bigone
True
>>> IPAddress("0.0.0.0") in bigone
True
>>> IPAddress("255.255.255") in bigone
True
>>> IPNetwork("10.0.0.0/24") in bigone
```

```
True
>>> IPAddress("::1") in bigone
False
```

```
>>> smallone = IPSet(["10.0.0.42/32"])
>>> IPAddress("10.0.0.42") in smallone
True
>>> IPAddress("10.0.0.41") in smallone
False
>>> IPAddress("10.0.0.43") in smallone
False
>>> IPNetwork("10.0.0.42/32") in smallone
True
>>> IPNetwork("10.0.0.42/31") in smallone
False
```

Unions, intersections and differences

Here are some examples of union operations performed on *IPSet* objects.

```
>>> IPSet(['192.0.2.0'])
IPSet(['192.0.2.0/32'])
```

```
>>> IPSet(['192.0.2.0']) | IPSet(['192.0.2.1'])
IPSet(['192.0.2.0/31'])
```

```
>>> IPSet(['192.0.2.0']) | IPSet(['192.0.2.1']) | IPSet(['192.0.2.3'])
IPSet(['192.0.2.0/31', '192.0.2.3/32'])
```

```
>>> IPSet(['192.0.2.0']) | IPSet(['192.0.2.1']) | IPSet(['192.0.2.3/30'])
IPSet(['192.0.2.0/30'])
```

```
>>> IPSet(['192.0.2.0']) | IPSet(['192.0.2.1']) | IPSet(['192.0.2.3/31'])
IPSet(['192.0.2.0/30'])
```

```
>>> IPSet(['192.0.2.0/24']) | IPSet(['192.0.3.0/24']) | IPSet(['192.0.4.0/24'])
IPSet(['192.0.2.0/23', '192.0.4.0/24'])
```

Here is an example of the union, intersection and symmetric difference operations all in play at the same time.

```
>>> adj_cidrs = list(IPNetwork('192.0.2.0/24').subnet(28))
>>> even_cidrs = adj_cidrs[::2]
>>> evens = IPSet(even_cidrs)
>>> evens
IPSet(['192.0.2.0/28', '192.0.2.32/28', '192.0.2.64/28', '192.0.2.96/28', '192.0.2.
↪128/28', '192.0.2.160/28', '192.0.2.192/28', '192.0.2.224/28'])
>>> IPSet(['192.0.2.0/24']) & evens
IPSet(['192.0.2.0/28', '192.0.2.32/28', '192.0.2.64/28', '192.0.2.96/28', '192.0.2.
↪128/28', '192.0.2.160/28', '192.0.2.192/28', '192.0.2.224/28'])
>>> odds = IPSet(['192.0.2.0/24']) ^ evens
>>> odds
IPSet(['192.0.2.16/28', '192.0.2.48/28', '192.0.2.80/28', '192.0.2.112/28', '192.0.2.
↪144/28', '192.0.2.176/28', '192.0.2.208/28', '192.0.2.240/28'])
```

```
>>> evens | odds
IPSet(['192.0.2.0/24'])
>>> evens & odds
IPSet([])
>>> evens ^ odds
IPSet(['192.0.2.0/24'])
```

Supersets and subsets

IP sets provide the ability to test whether a group of addresses ranges fit within the set of another group of address ranges.

```
>>> s1 = IPSet(['192.0.2.0/24', '192.0.4.0/24'])
>>> s2 = IPSet(['192.0.2.0', '192.0.4.0'])
>>> s1
IPSet(['192.0.2.0/24', '192.0.4.0/24'])
>>> s2
IPSet(['192.0.2.0/32', '192.0.4.0/32'])
>>> s1.issuperset(s2)
True
>>> s2.issubset(s1)
True
>>> s2.issuperset(s1)
False
>>> s1.issubset(s2)
False
```

Here's a more complete example using various well known IPv4 address ranges.

```
>>> ipv4_addr_space = IPSet(['0.0.0.0/0'])
>>> private = IPSet(['10.0.0.0/8', '172.16.0.0/12', '192.0.2.0/24', '192.168.0.0/16',
↪'239.192.0.0/14'])
>>> reserved = IPSet(['225.0.0.0/8', '226.0.0.0/7', '228.0.0.0/6', '234.0.0.0/7',
↪'236.0.0.0/7', '238.0.0.0/8', '240.0.0.0/4'])
>>> unavailable = reserved | private
>>> available = ipv4_addr_space ^ unavailable
```

Let's see what we've got:

```
>>> for cidr in available.iter_cidrs():
...     print cidr, cidr[0], cidr[-1]
0.0.0.0/5 0.0.0.0 7.255.255.255
8.0.0.0/7 8.0.0.0 9.255.255.255
11.0.0.0/8 11.0.0.0 11.255.255.255
12.0.0.0/6 12.0.0.0 15.255.255.255
16.0.0.0/4 16.0.0.0 31.255.255.255
32.0.0.0/3 32.0.0.0 63.255.255.255
64.0.0.0/2 64.0.0.0 127.255.255.255
128.0.0.0/3 128.0.0.0 159.255.255.255
160.0.0.0/5 160.0.0.0 167.255.255.255
168.0.0.0/6 168.0.0.0 171.255.255.255
172.0.0.0/12 172.0.0.0 172.15.255.255
172.32.0.0/11 172.32.0.0 172.63.255.255
172.64.0.0/10 172.64.0.0 172.127.255.255
172.128.0.0/9 172.128.0.0 172.255.255.255
```

```
173.0.0.0/8 173.0.0.0 173.255.255.255
174.0.0.0/7 174.0.0.0 175.255.255.255
176.0.0.0/4 176.0.0.0 191.255.255.255
192.0.0.0/23 192.0.0.0 192.0.1.255
192.0.3.0/24 192.0.3.0 192.0.3.255
192.0.4.0/22 192.0.4.0 192.0.7.255
192.0.8.0/21 192.0.8.0 192.0.15.255
192.0.16.0/20 192.0.16.0 192.0.31.255
192.0.32.0/19 192.0.32.0 192.0.63.255
192.0.64.0/18 192.0.64.0 192.0.127.255
192.0.128.0/17 192.0.128.0 192.0.255.255
192.1.0.0/16 192.1.0.0 192.1.255.255
192.2.0.0/15 192.2.0.0 192.3.255.255
192.4.0.0/14 192.4.0.0 192.7.255.255
192.8.0.0/13 192.8.0.0 192.15.255.255
192.16.0.0/12 192.16.0.0 192.31.255.255
192.32.0.0/11 192.32.0.0 192.63.255.255
192.64.0.0/10 192.64.0.0 192.127.255.255
192.128.0.0/11 192.128.0.0 192.159.255.255
192.160.0.0/13 192.160.0.0 192.167.255.255
192.169.0.0/16 192.169.0.0 192.169.255.255
192.170.0.0/15 192.170.0.0 192.171.255.255
192.172.0.0/14 192.172.0.0 192.175.255.255
192.176.0.0/12 192.176.0.0 192.191.255.255
192.192.0.0/10 192.192.0.0 192.255.255.255
193.0.0.0/8 193.0.0.0 193.255.255.255
194.0.0.0/7 194.0.0.0 195.255.255.255
196.0.0.0/6 196.0.0.0 199.255.255.255
200.0.0.0/5 200.0.0.0 207.255.255.255
208.0.0.0/4 208.0.0.0 223.255.255.255
224.0.0.0/8 224.0.0.0 224.255.255.255
232.0.0.0/7 232.0.0.0 233.255.255.255
239.0.0.0/9 239.0.0.0 239.127.255.255
239.128.0.0/10 239.128.0.0 239.191.255.255
239.196.0.0/14 239.196.0.0 239.199.255.255
239.200.0.0/13 239.200.0.0 239.207.255.255
239.208.0.0/12 239.208.0.0 239.223.255.255
239.224.0.0/11 239.224.0.0 239.255.255.255
```

```
>>> ipv4_addr_space ^ available
IPSet(['10.0.0.0/8', '172.16.0.0/12', '192.0.2.0/24', '192.168.0.0/16', '225.0.0.0/8',
↪ '226.0.0.0/7', '228.0.0.0/6', '234.0.0.0/7', '236.0.0.0/7', '238.0.0.0/8', '239.
↪192.0.0/14', '240.0.0.0/4'])
```

Combined IPv4 and IPv6 support

In keeping with netaddr's pragmatic approach, you are free to mix and match IPv4 and IPv6 within the same data structure.

```
>>> s1 = IPSet(['192.0.2.0', '::192.0.2.0', '192.0.2.2', '::192.0.2.2'])
>>> s2 = IPSet(['192.0.2.2', '::192.0.2.2', '192.0.2.4', '::192.0.2.4'])
```

```
>>> s1
IPSet(['192.0.2.0/32', '192.0.2.2/32', '::192.0.2.0/128', '::192.0.2.2/128'])
```

```
>>> s2
IPSet(['192.0.2.2/32', '192.0.2.4/32', '::192.0.2.2/128', '::192.0.2.4/128'])
```

IPv4 and IPv6 set union

```
>>> s1 | s2
IPSet(['192.0.2.0/32', '192.0.2.2/32', '192.0.2.4/32', '::192.0.2.0/128', '::192.0.2.
↪2/128', '::192.0.2.4/128'])
>>> s2 | s1
IPSet(['192.0.2.0/32', '192.0.2.2/32', '192.0.2.4/32', '::192.0.2.0/128', '::192.0.2.
↪2/128', '::192.0.2.4/128'])
```

set intersection

```
>>> s1 & s2
IPSet(['192.0.2.2/32', '::192.0.2.2/128'])
```

set difference

```
>>> s1 - s2
IPSet(['192.0.2.0/32', '::192.0.2.0/128'])
>>> s2 - s1
IPSet(['192.0.2.4/32', '::192.0.2.4/128'])
```

set symmetric difference

```
>>> s1 ^ s2
IPSet(['192.0.2.0/32', '192.0.2.4/32', '::192.0.2.0/128', '::192.0.2.4/128'])
```

Disjointed IP sets

```
>>> s1 = IPSet(['192.0.2.0', '192.0.2.1', '192.0.2.2'])
>>> s2 = IPSet(['192.0.2.2', '192.0.2.3', '192.0.2.4'])
>>> s1 & s2
IPSet(['192.0.2.2/32'])
>>> s1.isdisjoint(s2)
False
>>> s1 = IPSet(['192.0.2.0', '192.0.2.1'])
>>> s2 = IPSet(['192.0.2.3', '192.0.2.4'])
>>> s1 & s2
IPSet([])
>>> s1.isdisjoint(s2)
True
```

Updating an IP set

As with a normal Python set you can also update one IP set with the contents of another.

```
>>> s1 = IPSet(['192.0.2.0/25'])
>>> s1
IPSet(['192.0.2.0/25'])
>>> s2 = IPSet(['192.0.2.128/25'])
>>> s2
IPSet(['192.0.2.128/25'])
>>> s1.update(s2)
>>> s1
IPSet(['192.0.2.0/24'])
>>> s1.update(['192.0.0.0/24', '192.0.1.0/24', '192.0.3.0/24'])
>>> s1
IPSet(['192.0.0.0/22'])
```

```
>>> s2 = IPSet(['10.0.0.0/16'])
>>> s2.update(IPRange('10.1.0.0', '10.1.255.255'))
>>> s2
IPSet(['10.0.0.0/15'])
```

```
>>> s2.clear()
>>> s2
IPSet([])
```

Removing elements from an IP set

Removing an IP address from an IPSet will split the CIDR subnets within it into their constituent parts.

Here we create a set representing the entire IPv4 address space.

```
>>> s1 = IPSet(['0.0.0.0/0'])
>>> s1
IPSet(['0.0.0.0/0'])
```

Then we strip off the last address.

```
>>> s1.remove('255.255.255.255')
```

Leaving us with:

```
>>> s1
IPSet(['0.0.0.0/1', '128.0.0.0/2', ..., '255.255.255.252/31', '255.255.255.254/32'])
>>> list(s1.iter_cidrs())
[IPNetwork('0.0.0.0/1'), IPNetwork('128.0.0.0/2'), ..., IPNetwork('255.255.255.252/31
↪'), IPNetwork('255.255.255.254/32')]
>>> len(list(s1.iter_cidrs()))
32
```

Let's check the result using the *cidr_exclude* function.

```
>>> list(s1.iter_cidrs()) == cidr_exclude('0.0.0.0/0', '255.255.255.255')
True
```

Next, let's remove the first address from the original range.

```
>>> s1.remove('0.0.0.0')
```

This fractures the CIDR subnets further.

```
>>> s1
IPSet(['0.0.0.1/32', '0.0.0.2/31', ..., '255.255.255.252/31', '255.255.255.254/32'])
>>> len(list(s1.iter_cidrs()))
62
```

You can keep doing this but be aware that large IP sets can take up a lot of memory if they contain many thousands of entries.

Adding elements to an IP set

Let's fix up the fractured IP set from the previous section by re-adding the IP addresses we removed.

```
>>> s1.add('255.255.255.255')
>>> s1
IPSet(['0.0.0.1/32', '0.0.0.2/31', ..., '64.0.0.0/2', '128.0.0.0/1'])
```

Getting better.

```
>>> list(s1.iter_cidrs())
[IPNetwork('0.0.0.1/32'), IPNetwork('0.0.0.2/31'), ..., IPNetwork('64.0.0.0/2'),␣
↪IPNetwork('128.0.0.0/1')]
```

```
>>> len(list(s1.iter_cidrs()))
32
```

Add back the other IP address.

```
>>> s1.add('0.0.0.0')
```

And we're back to our original address.

```
>>> s1
IPSet(['0.0.0.0/0'])
```

Convert an IP set to an IP Range

Sometimes you may want to convert an IPSet back to an IPRange.

```
>>> s1 = IPSet(['10.0.0.0/25', '10.0.0.128/25'])
>>> s1.iprange()
IPRange('10.0.0.0', '10.0.0.255')
```

This only works if the IPSet is contiguous

```
>>> s1.iscontiguous()
True
>>> s1.remove('10.0.0.16')
```

```
>>> s1
IPSet(['10.0.0.0/28', '10.0.0.17/32', '10.0.0.18/31', '10.0.0.20/30', '10.0.0.24/29',
↪'10.0.0.32/27', '10.0.0.64/26', '10.0.0.128/25'])
>>> s1.iscontiguous()
False
>>> s1.iprange()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: IPSet is not contiguous
```

If it is not contiguous, you can still convert the IPSet, but you will get multiple IPRanges. >>> list(s1.iter_ipranges()) [IPRange('10.0.0.0', '10.0.0.15'), IPRange('10.0.0.17', '10.0.0.255')]

```
>>> s2 = IPSet(['0.0.0.0/0'])
>>> s2.iscontiguous()
True
>>> s2.iprange()
IPRange('0.0.0.0', '255.255.255.255')
```

```
>>> s3 = IPSet()
>>> s3.iscontiguous()
True
>>> s3.iprange()
```

```
>>> s4 = IPSet(IPRange('10.0.0.0', '10.0.0.8'))
>>> s4.iscontiguous()
True
```

Pickling IPSet objects

As with all other netaddr classes, you can use `pickle` to persist IP sets for later use.

```
>>> import pickle
>>> ip_data = IPSet(['10.0.0.0/16', 'fe80::/64'])
>>> buf = pickle.dumps(ip_data)
>>> ip_data_unpickled = pickle.loads(buf)
>>> ip_data == ip_data_unpickled
True
```

Compare IPSet objects

```
>>> x = IPSet(['fc00::/2'])
>>> y = IPSet(['fc00::/3'])
```

```
>>> x > y
True
```

```
>>> x < y
False
```

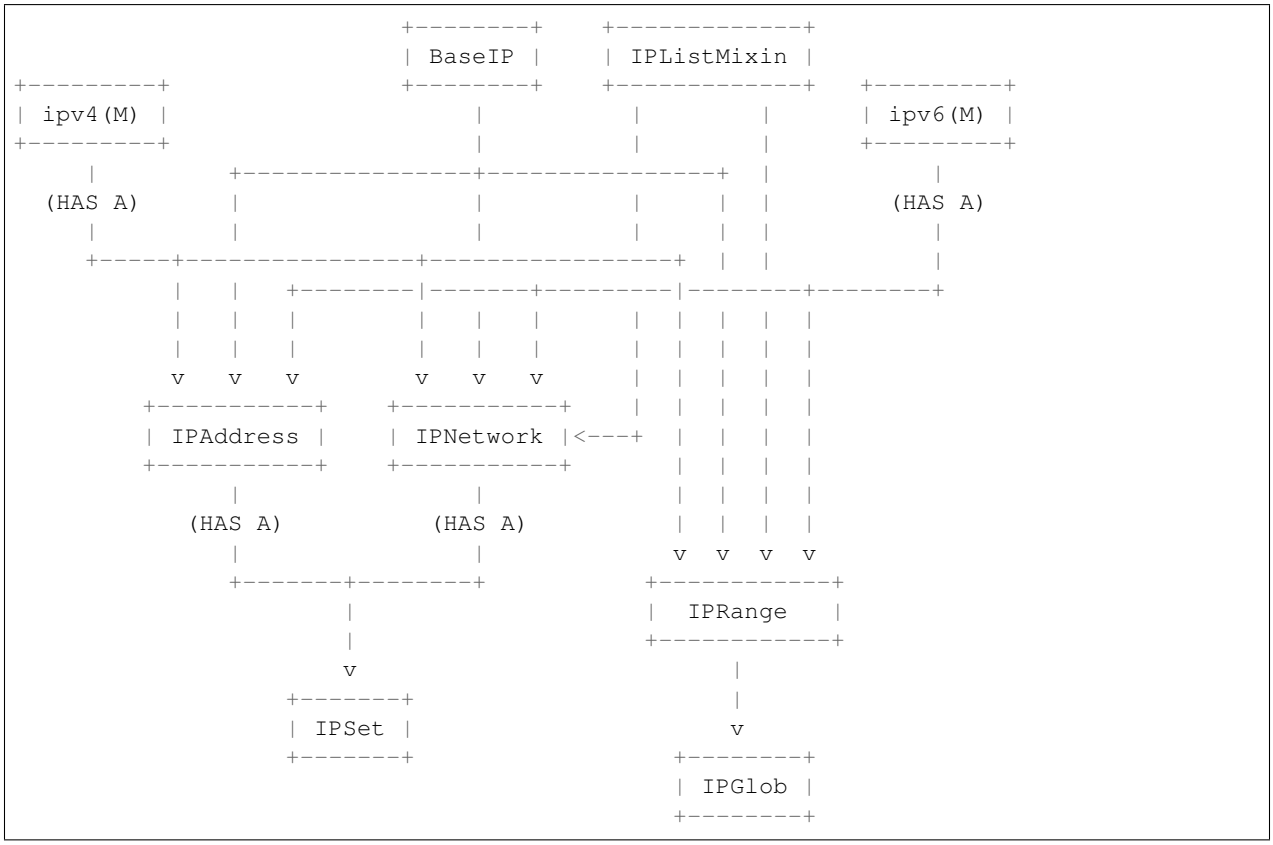
```
>>> x != y
True
```

CHAPTER 6

API Reference

IP Class Hierarchy

Here the class hierarchy for IP related classes



Constants

The following constants are used by the various *flags* arguments on netaddr class constructors.

P
INET_PTON
Use inet_pton() semantics instead of inet_aton() when parsing IPv4.

Z
ZEROFILL
Remove any preceding zeros from IPv4 address octets before parsing.

N
NOHOST
Remove any host bits found to the right of an applied CIDR prefix

Custom Exceptions

exception `netaddr.`**AddrConversionError**
An Exception indicating a failure to convert between address types or notations.

exception `netaddr.`**AddrFormatError**
An Exception indicating a network address is not correctly formatted.

exception `netaddr.`**NotRegisteredError**
An Exception indicating that an OUI or IAB was not found in the IEEE Registry.

IP addresses

An IP address is a virtual address used to identify the source and destination of (layer 3) packets being transferred between hosts in a switched network. This library fully supports both IPv4 and the new IPv6 standards.

The *IPAddress* class is used to identify individual IP addresses.

class `netaddr.`**IPAddress**(*addr*, *version=None*, *flags=0*)
An individual IPv4 or IPv6 address without a net mask or subnet prefix.

To support these and other network based operations, see *IPNetwork*.

__add__(*num*)
Add the numerical value of this IP address to num and provide the result as a new IPAddress object.

Parameters **num** – size of IP address increase.

Returns a new IPAddress object with its numerical value increased by num.

__and__(*other*)

Parameters **other** – An *IPAddress* object (or other int-like object).

Returns bitwise AND (x & y) between the integer value of this IP address and `other`.

__bool__()

Returns `True` if the numerical value of this IP address is not zero, `False` otherwise.

__getstate__()

Returns Pickled state of an *IPAddress* object.

__hex__()

Returns a hexadecimal string representation of this IP address.

__iadd__(*num*)
Increases the numerical value of this IPAddress by num.

An IndexError is raised if result exceeds maximum IP address value or is less than zero.

Parameters **num** – size of IP address increment.

__index__()

Returns return the integer value of this IP address when called by hex(), oct() or bin().

__init__(*addr*, *version=None*, *flags=0*)
Constructor.

Parameters

- **addr** – an IPv4 or IPv6 address which may be represented in an accepted string format, as an unsigned integer or as another IPAddress object (copy construction).
- **version** – (optional) optimizes version detection if specified and distinguishes between IPv4 and IPv6 for addresses with an equivalent integer value.
- **flags** – (optional) decides which rules are applied to the interpretation of the addr value. Supported constants are INET_PTON and ZEROFILL. See the netaddr.core docs for further details.

__int__()

Returns the value of this IP address as an unsigned integer

__isub__(*num*)
Decreases the numerical value of this IPAddress by num.

An IndexError is raised if result is less than zero or exceeds maximum IP address value.

Parameters **num** – size of IP address decrement.

__long__()

Returns the value of this IP address as an unsigned integer

__lshift__(*numbits*)

Parameters **numbits** – size of bitwise shift.

Returns an *IPAddress* object based on this one with its integer value left shifted by `numbits`.

__nonzero__()

Returns `True` if the numerical value of this IP address is not zero, `False` otherwise.

__oct__()

Returns an octal string representation of this IP address.

__or__(*other*)

Parameters **other** – An *IPAddress* object (or other int-like object).

Returns bitwise OR (x | y) between the integer value of this IP address and `other`.

__radd__(*num*)
Add the numerical value of this IP address to num and provide the result as a new IPAddress object.

Parameters **num** – size of IP address increase.

Returns a new IPAddress object with its numerical value increased by num.

__repr__()

Returns Python statement to create an equivalent object

__rshift__(*numbits*)

Parameters **numbits** – size of bitwise shift.

Returns an *IPAddress* object based on this one with its integer value right shifted by `numbits`.

__rsub__(*num*)
Subtract num (lvalue) from the numerical value of this IP address (rvalue) providing the result as a new IPAddress object.

Parameters **num** – size of IP address decrease.

Returns a new IPAddress object with its numerical value decreased by num.

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled *IPAddress* object.

__str__()

Returns IP address in presentational format

__sub__(*num*)
Subtract the numerical value of this IP address from num providing the result as a new IPAddress object.

Parameters **num** – size of IP address decrease.

Returns a new IPAddress object with its numerical value decreased by num.

__xor__(*other*)

Parameters **other** – An *IPAddress* object (or other int-like object).

Returns bitwise exclusive OR (x ^ y) between the integer value of this IP address and `other`.

bin
The value of this IP adddress in standard Python binary representational form (0bxxx). A back port of the format provided by the builtin bin() function found in Python 2.6.x and higher.

bits(*word_sep=None*)

Parameters **word_sep** – (optional) the separator to insert between words. Default: None - use default separator for address type.

Returns the value of this IP address as a binary digit string.

format(*dialect=None*)
Only relevant for IPv6 addresses. Has no effect for IPv4.

Parameters **dialect** – An ipv6_* dialect class.

Returns an alternate string representation for this IP address.

ipv4()
Raises an *AddrConversionError* if IPv6 address cannot be converted to IPv4.

Returns A numerically equivalent version 4 *IPAddress* object.

ipv6(*ipv4_compatible=False*)

Note: The IPv4-mapped IPv6 address format is now considered deprecated. See RFC 4291 or later for details.

Parameters **ipv4_compatible** – If `True` returns an IPv4-mapped address (::ffff:x.x.x.x), an IPv4-compatible (::x.x.x.x) address otherwise. Default: False (IPv4-mapped).

Returns A numerically equivalent version 6 *IPAddress* object.

is_hostmask()

Returns `True` if this IP address host mask, `False` otherwise.

is_netmask()

Returns `True` if this IP address network mask, `False` otherwise.

key()

Returns a key tuple that uniquely identifies this IP address.

netmask_bits()

@return: If this IP is a valid netmask, the number of non-zero bits are returned, otherwise it returns the width in bits for the IP address version.

packed
The value of this IP address as a packed binary string.

reverse_dns
The reverse DNS lookup record for this IP address

sort_key()

Returns A key tuple used to compare and sort this *IPAddress* correctly.

words
A list of unsigned integer words (octets for IPv4, hextets for IPv6) found in this IP address.

IPv6 formatting dialects

The following dialect classes can be used with the IPAddress.format method.

class netaddr.**ipv6_compact**
An IPv6 dialect class - compact form.

compact = True
Boolean flag indicating if IPv6 compaction algorithm should be used.

word_fmt = '%x'
The format string used to converting words into string values.

class netaddr.**ipv6_full**
An IPv6 dialect class - 'all zeroes' form.

compact = False
Boolean flag indicating if IPv6 compaction algorithm should be used.

class netaddr.**ipv6_verbose**
An IPv6 dialect class - extra wide 'all zeroes' form.

compact = False
Boolean flag indicating if IPv6 compaction algorithm should be used.

word_fmt = '%.4x'
The format string used to converting words into string values.

IP networks and subnets

The *IPNetwork* class is used to represent a group of IP addresses that comprise a network/subnet/VLAN containing hosts.

Nowadays, IP networks are usually specified using the CIDR format with a prefix indicating the size of the netmask. In the real world, there are a number of ways to express a "network"" and so the flexibility of the *IPNetwork* class constructor reflects this.

class netaddr.**IPNetwork**(*addr*, *implicit_prefix=False*, *version=None*, *flags=0*)
An IPv4 or IPv6 network or subnet.

A combination of an IP address and a network mask.

Accepts CIDR and several related variants :

1.Standard CIDR:

```
x.x.x.x/y -> 192.0.2.0/24
x::/y -> fe80::/10
```

2.Hybrid CIDR format (netmask address instead of prefix), where 'y' address represent a valid netmask:

```
x.x.x.x/y.y.y.y -> 192.0.2.0/255.255.255.0
x::/y:: -> fe80::/ffc0::
```

3.ACL hybrid CIDR format (hostmask address instead of prefix like Cisco's ACL bitmasks), where 'y' address represent a valid netmask:

```
x.x.x.x/y.y.y.y -> 192.0.2.0/0.0.0.255
x::/y:: -> fe80::/3f:ffff:ffff:ffff:ffff:ffff:ffff:ffff
```

4.Abbreviated CIDR format (as of netaddr 0.7.x this requires the optional constructor argument `implicit_prefix=True`):

```
x       -> 192
x/y     -> 10/8
x.x/y   -> 192.168/16
x.x.x/y -> 192.168.0/24

which are equivalent to::

x.0.0.0/y   -> 192.0.0.0/24
x.0.0.0/y   -> 10.0.0.0/8
x.x.0.0/y   -> 192.168.0.0/16
x.x.x.0/y   -> 192.168.0.0/24
```

__contains__(*other*)

Parameters **other** – an *IPAddress* or ranged IP object.

Returns `True` if other falls within the boundary of this one, `False` otherwise.

__getstate__()

Returns Pickled state of an *IPNetwork* object.

__iadd__(*num*)

Increases the value of this *IPNetwork* object by the current size multiplied by `num`.

An *IndexError* is raised if result exceeds maximum IP address value or is less than zero.

Parameters **num** – (optional) number of *IPNetwork* blocks to increment this IPNetwork's value by.

__init__(*addr*, *implicit_prefix=False*, *version=None*, *flags=0*)

Constructor.

Parameters

- **addr** – an IPv4 or IPv6 address with optional CIDR prefix, netmask or hostmask. May be an IP address in presentation (string) format, an tuple containing and integer address and a network prefix, or another IPAddress/IPNetwork object (copy construction).
- **implicit_prefix** – (optional) if True, the constructor uses classful IPv4 rules to select a default prefix when one is not provided. If False it uses the length of the IP address version. (default: False)
- **version** – (optional) optimizes version detection if specified and distinguishes between IPv4 and IPv6 for addresses with an equivalent integer value.
- **flags** – (optional) decides which rules are applied to the interpretation of the addr value. Currently only supports the NOHOST option. See the netaddr.core docs for further details.

__isub__(*num*)

Decreases the value of this *IPNetwork* object by the current size multiplied by `num`.

An *IndexError* is raised if result is less than zero or exceeds maximum IP address value.

Parameters **num** – (optional) number of *IPNetwork* blocks to decrement this IPNetwork's value by.

__repr__()

Returns Python statement to create an equivalent object

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled *IPNetwork* object.

__str__()

Returns this IPNetwork in CIDR format

broadcast

The broadcast address of this *IPNetwork* object

cidr

The true CIDR address for this *IPNetwork* object which omits any host bits to the right of the CIDR subnet prefix.

first

The integer value of first IP address found within this *IPNetwork* object.

hostmask

The host mask of this *IPNetwork* object.

ip
The IP address of this *IPNetwork* object. This is may or may not be the same as the network IP address which varies according to the value of the CIDR subnet prefix.

ipv4()

Returns A numerically equivalent version 4 *IPNetwork* object. Raises an *AddrConversionError* if IPv6 address cannot be converted to IPv4.

ipv6(*ipv4_compatible=False*)

Note: the IPv4-mapped IPv6 address format is now considered deprecated. See RFC 4291 or later for details.

Parameters **ipv4_compatible** – If `True` returns an IPv4-mapped address (::ffff:x.x.x.x), an IPv4-compatible (::x.x.x.x) address otherwise. Default: False (IPv4-mapped).

Returns A numerically equivalent version 6 *IPNetwork* object.

iter_hosts()
A generator that provides all the IP addresses that can be assigned to hosts within the range of this IP object's subnet.

- for IPv4, the network and broadcast addresses are always excluded. for subnets that contains less than 4 IP addresses /31 and /32 report in a manner per RFC 3021
- for IPv6, only the unspecified address '::' or Subnet-Router anycast address (first address in the network) is excluded.

Returns an IPAddress iterator

key()

Returns A key tuple used to uniquely identify this *IPNetwork*.

last
The integer value of last IP address found within this *IPNetwork* object.

netmask
The subnet mask of this *IPNetwork* object.

network
The network address of this *IPNetwork* object.

next(*step=1*)

Parameters **step** – the number of IP subnets between this *IPNetwork* object and the expected subnet. Default: 1 (the next IP subnet).

Returns The adjacent subnet succeeding this *IPNetwork* object.

prefixlen
size of the bitmask used to separate the network from the host bits

previous(*step=1*)

Parameters **step** – the number of IP subnets between this *IPNetwork* object and the expected subnet. Default: 1 (the previous IP subnet).

Returns The adjacent subnet preceding this *IPNetwork* object.

sort_key()

Returns A key tuple used to compare and sort this *IPNetwork* correctly.

subnet(*prefixlen*, *count=None*, *fmt=None*)
A generator that divides up this IPNetwork's subnet into smaller subnets based on a specified CIDR prefix.

Parameters

- **prefixlen** – a CIDR prefix indicating size of subnets to be returned.
- **count** – (optional) number of consecutive IP subnets to be returned.

Returns an iterator containing IPNetwork subnet objects.

supernet(*prefixlen=0*)
Provides a list of supernets for this *IPNetwork* object between the size of the current prefix and (if specified) an endpoint prefix.

Parameters **prefixlen** – (optional) a CIDR prefix for the maximum supernet. Default: 0 - returns all possible supernets.

Returns a tuple of supernet *IPNetwork* objects.

Arbitrary IP address ranges

netaddr was designed to accommodate the many different ways in which groups of IP addresses and IP networks are specified, not only in router configurations but also less standard but more human-readable forms found in, for instance, configuration files.

Here are the options currently available.

bounded ranges

A bounded range is a group of IP addresses specified using a start and end address forming a contiguous block. No bit boundaries are assumed but the end address must be numerically greater than or equal to the start address.

class netaddr.**IPRange**(*start*, *end*, *flags=0*)
An arbitrary IPv4 or IPv6 address range.

Formed from a lower and upper bound IP address. The upper bound IP cannot be numerically smaller than the lower bound and the IP version of both must match.

__getstate__()

Returns Pickled state of an *IPRange* object.

__init__(*start*, *end*, *flags=0*)
Constructor.

Parameters

- **start** – an IPv4 or IPv6 address that forms the lower boundary of this IP range.
- **end** – an IPv4 or IPv6 address that forms the upper boundary of this IP range.
- **flags** – (optional) decides which rules are applied to the interpretation of the start and end values. Supported constants are INET_PTON and ZEROFILL. See the netaddr.core docs for further details.

__repr__()

Returns Python statement to create an equivalent object

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled *IPRange* object.

__str__()

Returns this *IPRange* in a common representational format.

cidrs()
The list of CIDR addresses found within the lower and upper bound addresses of this *IPRange*.

first
The integer value of first IP address in this *IPRange* object.

key()

Returns A key tuple used to uniquely identify this *IPRange*.

last
The integer value of last IP address in this *IPRange* object.

sort_key()

Returns A key tuple used to compare and sort this *IPRange* correctly.

IP glob ranges

A very useful way to represent IP network in configuration files and on the command line for those who do not speak CIDR.

The *IPGlob* class is used to represent individual glob ranges.

class netaddr.**IPGlob**(*ipglob*)
Represents an IP address range using a glob-style syntax `x.x.x-y.*`

Individual octets can be represented using the following shortcuts :

1. `*` - the asterisk octet (represents values `0` through `255`)
2. `x-y` - the hyphenated octet (represents values `x` through `y`)

A few basic rules also apply :

1. `x` must always be greater than `y`, therefore :
   - `x` can only be `0` through `254`
   - `y` can only be `1` through `255`
2. only one hyphenated octet per IP glob is allowed
3. only asterisks are permitted after a hyphenated octet

Examples:

IP glob	Description
`192.0.2.1`	a single address
`192.0.2.0-31`	32 addresses
`192.0.2.*`	256 addresses
`192.0.2-3.*`	512 addresses
`192.0-1.*.*`	131,072 addresses
`*.*.*.*`	the whole IPv4 address space

Note: IP glob ranges are not directly equivalent to CIDR blocks. They can represent address ranges that do not fall on strict bit mask boundaries. They are suitable for use in configuration files, being more obvious and readable than their CIDR counterparts, especially for admins and end users with little or no networking knowledge or experience. All CIDR addresses can always be represented as IP globs but the reverse is not always true.

__getstate__()

Returns Pickled state of an *IPGlob* object.

__repr__()

Returns Python statement to create an equivalent object

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled *IPGlob* object.

__str__()

Returns IP glob in common representational format.

glob
an arbitrary IP address range in glob format.

globbing functions

It is also very useful to be able to convert between glob ranges and CIDR and IP ranges. The following function enable these various conversions.

netaddr.**cidr_to_glob**(*cidr*)
A function that accepts an IP subnet in a glob-style format and returns a list of CIDR subnets that exactly matches the specified glob.

Parameters **cidr** – an IP object CIDR subnet.

Returns a list of one or more IP addresses and subnets.

netaddr.**glob_to_cidrs**(*ipglob*)
A function that accepts a glob-style IP range and returns a list of one or more IP CIDRs that exactly matches it.

Parameters **ipglob** – an IP address range in a glob-style format.

Returns a list of one or more IP objects.

netaddr.**glob_to_iprange**(*ipglob*)
A function that accepts a glob-style IP range and returns the equivalent IP range.

Parameters **ipglob** – an IP address range in a glob-style format.

Returns an IPRange object.

netaddr.**glob_to_iptuple**(*ipglob*)
A function that accepts a glob-style IP range and returns the component lower and upper bound IP address.

Parameters **ipglob** – an IP address range in a glob-style format.

Returns a tuple contain lower and upper bound IP objects.

netaddr.**iprange_to_globs**(*start*, *end*)
A function that accepts an arbitrary start and end IP address or subnet and returns one or more glob-style IP ranges.

Parameters

- **start** – the start IP address or subnet.
- **end** – the end IP address or subnet.

Returns a list containing one or more IP globs.

nmap ranges

nmap is a well known network security tool. It has a particularly flexible way of specifying IP address groupings.

Functions are provided that allow the verification and enumeration of IP address specified in this format.

netaddr.**valid_nmap_range**(*target_spec*)

Parameters **target_spec** – an nmap-style IP range target specification.

Returns True if IP range target spec is valid, False otherwise.

netaddr.**iter_nmap_range**(**nmap_target_spec*)
An generator that yields IPAddress objects from defined by nmap target specifications.

See https://nmap.org/book/man-target-specification.html for details.

Parameters ***nmap_target_spec** – one or more nmap IP range target specification.

Returns an iterator producing IPAddress objects for each IP in the target spec(s).

IP sets

When dealing with large numbers of IP addresses and ranges it is often useful to manipulate them as sets so you can calculate intersections, unions and differences between various groups of them.

The *IPSet* class was built specifically for this purpose.

class netaddr.**IPSet**(*iterable=None*, *flags=0*)
Represents an unordered collection (set) of unique IP addresses and subnets.

__and__(*other*)

Parameters **other** – an IP set.

Returns the intersection of this IP set and another as a new IP set. (IP addresses and subnets common to both sets).

__bool__()
Return True if IPSet contains at least one IP, else False

__contains__(*ip*)

Parameters **ip** – An IP address or subnet.

Returns `True` if IP address or subnet is a member of this IP set.

__eq__(*other*)

Parameters **other** – an IP set

Returns `True` if this IP set is equivalent to the `other` IP set, `False` otherwise.

__ge__(*other*)

Parameters **other** – an IP set.

Returns `True` if every IP address and subnet in other IP set is found within this one.

__getstate__()

Returns Pickled state of an `IPSet` object.

__gt__(*other*)

Parameters **other** – an IP set.

Returns `True` if this IP set is greater than the `other` IP set, `False` otherwise.

__hash__()
Raises `TypeError` if this method is called.

Note: IPSet objects are not hashable and cannot be used as dictionary keys or as members of other sets.

__init__(*iterable=None*, *flags=0*)
Constructor.

Parameters

- **iterable** – (optional) an iterable containing IP addresses and subnets.
- **flags** – decides which rules are applied to the interpretation of the addr value. See the netaddr.core namespace documentation for supported constant values.

__iter__()

Returns an iterator over the IP addresses within this IP set.

__le__(*other*)

Parameters **other** – an IP set.

Returns `True` if every IP address and subnet in this IP set is found within `other`.

__len__()

Returns the cardinality of this IP set (i.e. sum of individual IP addresses). Raises `IndexError` if size > maxint (a Python limitation). Use the .size property for subnets of any size.

__lt__(*other*)

Parameters **other** – an IP set

Returns `True` if this IP set is less than the `other` IP set, `False` otherwise.

__ne__(*other*)

Parameters **other** – an IP set

Returns `False` if this IP set is equivalent to the `other` IP set, `True` otherwise.

__nonzero__()
Return True if IPSet contains at least one IP, else False

__or__(*other*)

Parameters **other** – an IP set.

Returns the union of this IP set and another as a new IP set (combines IP addresses and subnets from both sets).

__repr__()

Returns Python statement to create an equivalent object

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled `IPSet` object.

__str__()

Returns Python statement to create an equivalent object

__sub__(*other*)

Parameters **other** – an IP set.

Returns the difference between this IP set and another as a new IP set (all IP addresses and subnets that are in this IP set but not found in the other.)

__xor__(*other*)

Parameters **other** – an IP set.

Returns the symmetric difference of this IP set and another as a new IP set (all IP addresses and subnets that are in exactly one of the sets).

add(*addr*, *flags=0*)
Adds an IP address or subnet or IPRange to this IP set. Has no effect if it is already present.

Note that where possible the IP address or subnet is merged with other members of the set to form more concise CIDR blocks.

Parameters

- **addr** – An IP address or subnet in either string or object form, or an IPRange object.
- **flags** – decides which rules are applied to the interpretation of the addr value. See the netaddr.core namespace documentation for supported constant values.

clear()
Remove all IP addresses and subnets from this IP set.

compact()
Compact internal list of *IPNetwork* objects using a CIDR merge.

copy()

Returns a shallow copy of this IP set.

difference(*other*)

Parameters **other** – an IP set.

Returns the difference between this IP set and another as a new IP set (all IP addresses and subnets that are in this IP set but not found in the other.)

intersection(*other*)

Parameters **other** – an IP set.

Returns the intersection of this IP set and another as a new IP set. (IP addresses and subnets common to both sets).

iprange()
Generates an IPRange for this IPSet, if all its members form a single contiguous sequence.

Raises `ValueError` if the set is not contiguous.

Returns An `IPRange` for all IPs in the IPSet.

iscontiguous()
Returns True if the members of the set form a contiguous IP address range (with no gaps), False otherwise.

Returns `True` if the `IPSet` object is contiguous.

isdisjoint(*other*)

Parameters **other** – an IP set.

Returns `True` if this IP set has no elements (IP addresses or subnets) in common with other. Intersection *must* be an empty set.

issubset(*other*)

Parameters **other** – an IP set.

Returns `True` if every IP address and subnet in this IP set is found within `other`.

issuperset(*other*)

Parameters **other** – an IP set.

Returns `True` if every IP address and subnet in other IP set is found within this one.

iter_cidrs()

Returns an iterator over individual IP subnets within this IP set.

iter_ipranges()
Generate the merged IPRanges for this IPSet.

In contrast to self.iprange(), this will work even when the IPSet is not contiguous. Adjacent IPRanges will be merged together, so you get the minimal number of IPRanges.

pop()
Removes and returns an arbitrary IP address or subnet from this IP set.

Returns An IP address or subnet.

remove(*addr*, *flags=0*)
Removes an IP address or subnet or IPRange from this IP set. Does nothing if it is not already a member.

Note that this method behaves more like discard() found in regular Python sets because it doesn't raise KeyError exceptions if the IP address or subnet is question does not exist. It doesn't make sense to fully emulate that behaviour here as IP sets contain groups of individual IP addresses as individual set members using IPNetwork objects.

Parameters

- **addr** – An IP address or subnet, or an IPRange.
- **flags** – decides which rules are applied to the interpretation of the addr value. See the netaddr.core namespace documentation for supported constant values.

size
The cardinality of this IP set (based on the number of individual IP addresses including those implicitly defined in subnets).

symmetric_difference(*other*)

Parameters **other** – an IP set.

Returns the symmetric difference of this IP set and another as a new IP set (all IP addresses and subnets that are in exactly one of the sets).

union(*other*)

Parameters **other** – an IP set.

Returns the union of this IP set and another as a new IP set (combines IP addresses and subnets from both sets).

update(*iterable*, *flags=0*)
Update the contents of this IP set with the union of itself and other IP set.

Parameters

- **iterable** – an iterable containing IP addresses and subnets.
- **flags** – decides which rules are applied to the interpretation of the addr value. See the netaddr.core namespace documentation for supported constant values.

IP functions and generators

The following are a set of useful helper functions related to the various format supported in this library.

netaddr.**all_matching_cidrs**(*ip*, *cidrs*)
Matches an IP address or subnet against a given sequence of IP addresses and subnets.

Parameters

- **ip** – a single IP address.
- **cidrs** – a sequence of IP addresses and/or subnets.

Returns all matching IPAddress and/or IPNetwork objects from the provided sequence, an empty list if there was no match.

netaddr.**cidr_abbrev_to_verbose**(*abbrev_cidr*)
A function that converts abbreviated IPv4 CIDRs to their more verbose equivalent.

Parameters **abbrev_cidr** – an abbreviated CIDR.

Uses the old-style classful IP address rules to decide on a default subnet prefix if one is not explicitly provided.

Only supports IPv4 addresses.

Examples

```
10                  - 10.0.0.0/8
10/16               - 10.0.0.0/16
128                 - 128.0.0.0/16
128/8               - 128.0.0.0/8
192.168             - 192.168.0.0/16
```

Returns A verbose CIDR from an abbreviated CIDR or old-style classful network address. The original value if it was not recognised as a supported abbreviation.

netaddr.**cidr_exclude**(*target*, *exclude*)

Removes an exclude IP address or subnet from target IP subnet.

Parameters

- **target** – the target IP address or subnet to be divided up.
- **exclude** – the IP address or subnet to be removed from target.

Returns list of *IPNetwork* objects remaining after exclusion.

netaddr.**cidr_merge**(*ip_addrs*)

A function that accepts an iterable sequence of IP addresses and subnets merging them into the smallest possible list of CIDRs. It merges adjacent subnets where possible, those contained within others and also removes any duplicates.

Parameters **ip_addrs** – an iterable sequence of IP addresses and subnets.

Returns a summarized list of *IPNetwork* objects.

netaddr.**iprange_to_cidrs**(*start*, *end*)

A function that accepts an arbitrary start and end IP address or subnet and returns a list of CIDR subnets that fit exactly between the boundaries of the two with no overlap.

Parameters

- **start** – the start IP address or subnet.
- **end** – the end IP address or subnet.

Returns a list of one or more IP addresses and subnets.

netaddr.**iter_iprange**(*start*, *end*, *step=1*)

A generator that produces IPAddress objects between an arbitrary start and stop IP address with intervals of step between them. Sequences produce are inclusive of boundary IPs.

Parameters

- **start** – start IP address.
- **end** – end IP address.
- **step** – (optional) size of step between IP addresses. Default: 1

Returns an iterator of one or more *IPAddress* objects.

netaddr.**iter_unique_ips**(**args*)

Parameters **args** – A list of IP addresses and subnets passed in as arguments.

Returns A generator that flattens out IP subnets, yielding unique individual IP addresses (no duplicates).

netaddr.**largest_matching_cidr**(*ip*, *cidrs*)

Matches an IP address or subnet against a given sequence of IP addresses and subnets.

Parameters

- **ip** – a single IP address or subnet.
- **cidrs** – a sequence of IP addresses and/or subnets.

Returns the largest (least specific) matching IPAddress or IPNetwork object from the provided sequence, None if there was no match.

netaddr.**smallest_matching_cidr**(*ip*, *cidrs*)
Matches an IP address or subnet against a given sequence of IP addresses and subnets.

Parameters

- **ip** – a single IP address or subnet.
- **cidrs** – a sequence of IP addresses and/or subnets.

Returns the smallest (most specific) matching IPAddress or IPNetwork object from the provided sequence, None if there was no match.

netaddr.**spanning_cidr**(*ip_addrs*)
Function that accepts a sequence of IP addresses and subnets returning a single *IPNetwork* subnet that is large enough to span the lower and upper bound IP addresses with a possible overlap on either end.

Parameters **ip_addrs** – sequence of IP addresses and subnets.

Returns a single spanning *IPNetwork* subnet.

MAC addresses and the IEEE EUI standard

A MAC address is the 48-bit hardware address associated with a particular physical interface on a networked host. They are found in all networked devices and serve to identify (layer 2) frames in the networking stack.

The *EUI* class is used to represents MACs (as well as their larger and less common 64-bit cousins).

class netaddr.**EUI**(*addr*, *version=None*, *dialect=None*)
An IEEE EUI (Extended Unique Identifier).

Both EUI-48 (used for layer 2 MAC addresses) and EUI-64 are supported.

Input parsing for EUI-48 addresses is flexible, supporting many MAC variants.

__eq__(*other*)

Returns `True` if this EUI object is numerically the same as other, `False` otherwise.

__ge__(*other*)

Returns `True` if this EUI object is numerically greater or equal in value to other, `False` otherwise.

__getitem__(*idx*)

Returns The integer value of the word referenced by index (both positive and negative). Raises `IndexError` if index is out of bounds. Also supports Python list slices for accessing word groups.

__getstate__()

Returns Pickled state of an *EUI* object.

__gt__(*other*)

Returns `True` if this EUI object is numerically greater in value than other, `False` otherwise.

__hash__()

Returns hash of this EUI object suitable for dict keys, sets etc

__init__(*addr*, *version=None*, *dialect=None*)
Constructor.

Parameters

- **addr** – an EUI-48 (MAC) or EUI-64 address in string format or an unsigned integer. May also be another EUI object (copy construction).
- **version** – (optional) the explicit EUI address version, either 48 or 64. Mainly used to distinguish EUI-48 and EUI-64 identifiers specified as integers which may be numerically equivalent.
- **dialect** – (optional) the mac_* dialect to be used to configure the formatting of EUI-48 (MAC) addresses.

__le__(*other*)

Returns `True` if this EUI object is numerically lower or equal in value to other, `False` otherwise.

__lt__(*other*)

Returns `True` if this EUI object is numerically lower in value than other, `False` otherwise.

__ne__(*other*)

Returns `True` if this EUI object is numerically the same as other, `False` otherwise.

__repr__()

Returns executable Python string to recreate equivalent object.

__setitem__(*idx*, *value*)
Set the value of the word referenced by index in this address

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled *EUI* object.

__str__()

Returns EUI in representational format

bin
The value of this EUI adddress in standard Python binary representational form (0bxxx). A back port of the format provided by the builtin bin() function found in Python 2.6.x and higher.

bits(*word_sep=None*)

Parameters **word_sep** – (optional) the separator to insert between words. Default: None - use default separator for address type.

Returns human-readable binary digit string of this address.

dialect
a Python class providing support for the interpretation of various MAC address formats.

ei
The EI (Extension Identifier) for this EUI

eui64()

•If this object represents an EUI-48 it is converted to EUI-64 as per the standard.

•If this object is already an EUI-64, a new, numerically equivalent object is returned instead.

Returns The value of this EUI object as a new 64-bit EUI object.

iab
If is_iab() is True, the IAB (Individual Address Block) is returned, `None` otherwise.

info
A record dict containing IEEE registration details for this EUI (MAC-48) if available, None otherwise.

ipv6(*prefix*)

Note: This poses security risks in certain scenarios. Please read RFC 4941 for details. Reference: RFCs 4291 and 4941.

Parameters **prefix** – ipv6 prefix

Returns new IPv6 *IPAddress* object based on this *EUI* using the technique described in RFC 4291.

ipv6_link_local()

Note: This poses security risks in certain scenarios. Please read RFC 4941 for details. Reference: RFCs 4291 and 4941.

Returns new link local IPv6 *IPAddress* object based on this *EUI* using the technique described in RFC 4291.

is_iab()

Returns True if this EUI is an IAB address, False otherwise

modified_eui64()

•create a new EUI object with a modified EUI-64 as described in RFC 4291 section 2.5.1

Returns a new and modified 64-bit EUI object.

oui
The OUI (Organisationally Unique Identifier) for this EUI.

packed
The value of this EUI address as a packed binary string.

value
a positive integer representing the value of this EUI indentifier.

version
The EUI version represented by this EUI object.

words
A list of unsigned integer octets found in this EUI address.

class netaddr.**OUI**(*oui*)
An individual IEEE OUI (Organisationally Unique Identifier).

For online details see - http://standards.ieee.org/regauth/oui/

__getstate__()

Returns Pickled state of an *OUI* object.

__init__(*oui*)
Constructor

Parameters **oui** – an OUI string `XX-XX-XX` or an unsigned integer. Also accepts and parses full MAC/EUI-48 address strings (but not MAC/EUI-48 integers)!

__repr__()

Returns executable Python string to recreate equivalent object.

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled *OUI* object.

__str__()

Returns string representation of this OUI

reg_count
Number of registered organisations with this OUI

registration(*index=0*)
The IEEE registration details for this OUI.

Parameters **index** – the index of record (may contain multiple registrations) (Default: 0 - first registration)

Returns Objectified Python data structure containing registration details.

class `netaddr.`**IAB**(*iab*, *strict=False*)
An individual IEEE IAB (Individual Address Block) identifier.

For online details see - http://standards.ieee.org/regauth/oui/

__getstate__()

Returns Pickled state of an *IAB* object.

__init__(*iab*, *strict=False*)
Constructor

Parameters

- **iab** – an IAB string `00-50-C2-XX-X0-00` or an unsigned integer. This address looks like an EUI-48 but it should not have any non-zero bits in the last 3 bytes.
- **strict** – If True, raises a ValueError if the last 12 bits of IAB MAC/EUI-48 address are non-zero, ignores them otherwise. (Default: False)

__repr__()

Returns executable Python string to recreate equivalent object.

__setstate__(*state*)

Parameters **state** – data used to unpickle a pickled *IAB* object.

__str__()

Returns string representation of this IAB

registration()
The IEEE registration details for this IAB

static **split_iab_mac**(*eui_int*, *strict=False*)

Parameters

- **eui_int** – a MAC IAB as an unsigned integer.
- **strict** – If True, raises a ValueError if the last 12 bits of IAB MAC/EUI-48 address are non-zero, ignores them otherwise. (Default: False)

MAC formatting dialects

The following dialects are used to specify the formatting of MAC addresses.

class netaddr.**mac_bare**
A bare (no delimiters) MAC address dialect class.

class netaddr.**mac_cisco**
A Cisco 'triple hextet' MAC address dialect class.

class netaddr.**mac_eui48**
A standard IEEE EUI-48 dialect class.

max_word = 255
The maximum integer value for an individual word in this address type.

num_words = 6
The number of words in this address type.

word_base = 16
The number base to be used when interpreting word values as integers.

word_fmt = '%.2X'
The format string to be used when converting words to string values.

word_sep = '-'
The separator character used between each word.

word_size = 8
The individual word size (in bits) of this address type.

class netaddr.**mac_pgsql**
A PostgreSQL style (2 x 24-bit words) MAC address dialect class.

class netaddr.**mac_unix**
A UNIX-style MAC address dialect class.

Validation functions

netaddr.**valid_ipv4**(*addr*, *flags=0*)

Parameters

- **addr** – An IPv4 address in presentation (string) format.
- **flags** – decides which rules are applied to the interpretation of the addr value. Supported constants are INET_PTON and ZEROFILL. See the netaddr.core docs for details.

Returns `True` if IPv4 address is valid, `False` otherwise.

netaddr.**valid_ipv6**(*addr*, *flags=0*)

Parameters

- **addr** – An IPv6 address in presentation (string) format.

- **flags** – decides which rules are applied to the interpretation of the addr value. Future use - currently has no effect.

Returns `True` if IPv6 address is valid, `False` otherwise.

`netaddr.`**`valid_glob`**(*ipglob*)

Parameters **ipglob** – An IP address range in a glob-style format.

Returns `True` if IP range glob is valid, `False` otherwise.

`netaddr.`**`valid_mac`**(*addr*)

Parameters **addr** – An IEEE EUI-48 (MAC) address in string form.

Returns `True` if MAC address string is valid, `False` otherwise.

A bit of fun

Who said networking was all about being serious? It's always good to lighten up and have a bit of fun.

Let's face it, no networking library worth its salt would be complete without support for RFC 1924 - http://www.ietf.org/rfc/rfc1924 `:-)`

`netaddr.`**`base85_to_ipv6`**(*addr*)
Convert a base 85 IPv6 address to its hexadecimal format.

`netaddr.`**`ipv6_to_base85`**(*addr*)
Convert a regular IPv6 address to base 85.

CHAPTER 7

What's new in netaddr 0.7.19

Release: 0.7.19

Date: 11 Jan 2017

Changes since 0.7.18

- added a new SubnetSplitter class for those looking to divide up subnets. Thanks alanwill and RyPeck and those on (Stack Overflow discussion).
- removed bundled pytest dependency code for "python setup.py test".
- setup.py now uses setuptools only (no more distutils) and setup_egg.py removed.
- cleaned up INSTALL docs so they accurately reflect current Python packaging.
- fixed broken parsing, generating and reading of IEEE index files when switching between Python 2.x and 3.x.

Specific bug fixes addressed in this release

FIXED Issue 133: https://github.com/drkjam/netaddr/issues/133

- Splitting a single network into multiple prefixed networks

FIXED Issue 129: https://github.com/drkjam/netaddr/issues/129

- fix IPAddress().netmask_bits to return 0 for 0.0.0.0 and [::] addresses

FIXED Issue 117: https://github.com/drkjam/netaddr/issues/117

- (python setup.py test) failing with python3 >= 3.5

FIXED Issue 137: https://github.com/drkjam/netaddr/issues/137

- API reference is broken on ReadTheDocs

FIXED Issue 143: https://github.com/drkjam/netaddr/issues/143

- Please refresh the bundled IANA and IEEE databases

Miscellanea

- Goodbye to NYSE Euronext (good times), hello Intercontinental Exchange ...

Release: 0.7.18

Date: 4 Sep 2015

Changes since 0.7.17

- cidr_merge() algorithm is now O(n) and much faster. Thanks to Anand Buddhdev (aabdnn) and Stefan Nordhausen (snordhausen).
- nmap target specification now fully supported including IPv4 CIDR prefixes and IPv6 addresses.

Specific bug fixes addressed in this release

FIXED Issue 100: https://github.com/drkjam/netaddr/issues/100

- nmap.py - CIDR targets

FIXED Issue 112: https://github.com/drkjam/netaddr/issues/112

- Observation: netaddr slower under pypy

Release: 0.7.17

Date: 31 Aug 2015

Changes since 0.7.16

- Fixed a regression with valid_mac due to shadow import in the netaddr module.

Specific bug fixes addressed in this release

FIXED Issue 114: https://github.com/drkjam/netaddr/issues/114

- netaddr.valid_mac('00-B0-D0-86-BB-F7')==False for 0.7.16 but True for 0.7.15

Release: 0.7.16

Date: 30 Aug 2015

Changes since 0.7.15

- **IPv4 networks with /31 and /32 netmasks are now treated according to** RFC 3021. Thanks to kalombos and braaen.

Specific bug fixes addressed in this release

FIXED Issue 109: https://github.com/drkjam/netaddr/issues/109

- Identify registry of global IPv6 unicast allocations

FIXED Issue 108: https://github.com/drkjam/netaddr/issues/108

- One part of docs unclear?

FIXED Issue 106: https://github.com/drkjam/netaddr/issues/106

- Eui64 Updated (pull request for Issue 105)

FIXED Issue 105: https://github.com/drkjam/netaddr/issues/105

- Support dialects for EUI-64 addresses

FIXED Issue 102: https://github.com/drkjam/netaddr/issues/102

- 0.7.15 tarball is missing tests.

FIXED Issue 96: https://github.com/drkjam/netaddr/issues/96

- Wrong hosts and broadcasts for /31 and /32 networks.

Release: 0.7.15

Date: 29 Jun 2015

Changes since 0.7.14

- Fix slowness in IPSet.__contains__. Thanks to novas0x2a for noticing.
- Normalize IPNetworks when they are added to an IPSet
- Converted test suite to py.test

Specific bug fixes addressed in this release

FIXED Issue 98: https://github.com/drkjam/netaddr/issues/98

- Convert test suite to py.test

FIXED Issue 94: https://github.com/drkjam/netaddr/issues/94

- IPSet.__contains__ is about 40 times slower than the equivalent IPRange

FIXED Issue 95: https://github.com/drkjam/netaddr/issues/95

- Inconsistent Address Handling in IPSet

Release: 0.7.14

Date: 31st Mar 2015

Changes since 0.7.13

- Fix weird build breakage in 0.7.13 (wrong Python path, incorrect OUI DB).
- **EUI, OUI, and IAB objects can now be compared with strings. You can do** my_mac = EUI("11:22:33:44:55:66") my_mac == "11:22:33:44:55:66" and Python will return True on the "==" operator.
- **Implement the "!=" operator for OUI and IAB under Python2. It was already** working under Python3.
- **64 bit EUIs could only be created from strings with "-" as a separator.** Now, ":" and no seperator are supported, which already worked for 48 bit EUIs.

Specific bug fixes addressed in this release

FIXED Issue 80: https://github.com/drkjam/netaddr/pull/80

- Compare L2 addresses with their representations

FIXED Issue 81: https://github.com/drkjam/netaddr/issues/81

- OUI database tests fail in 0.7.13

FIXED Issue 84: https://github.com/drkjam/netaddr/issues/84

- Incorrect python executable path in netaddr-0.7.13-py2.py3-none-any.whl

FIXED Issue 87: https://github.com/drkjam/netaddr/pull/87

- Handle eui64 addresses with colon as a delimiter and without delimeter.

Release: 0.7.13

Date: 31st Dec 2014

Changes since 0.7.12

- IPAddress objects can now be added to/subtracted from each other

Specific bug fixes addressed in this release

FIXED Issue 73: https://github.com/drkjam/netaddr/issues/73

- Adding IP Addresses

FIXED Issue 74: https://github.com/drkjam/netaddr/issues/74

- compute static global ipv6 addr from the net prefix and mac address

FIXED Issue 75: https://github.com/drkjam/netaddr/issues/75

- add classifiers for python 3.3 and 3.4 support

Release: 0.7.12

Date: 6th Jul 2014

Changes since 0.7.11

- Added method IPSet.iter_ipranges().
- bool(IPSet()) works now for large IPSets, e.g. IPSet(['2405:8100::/32']).
- IPNetwork.iter_hosts now skips the subnet-router anycast address for IPv6.
- Removed function fbsocket.inet_aton because it is unused and unnecessary

Specific bug fixes addressed in this release

FIXED Issue 69: https://github.com/drkjam/netaddr/issues/69

- Add __nonzero__ method to IPSet

FIXED Pull Request 68: https://github.com/drkjam/netaddr/pull/68

- Fixed a bug related to allowing ::0 during iter_hosts for v6

FIXED Issue 67: https://github.com/drkjam/netaddr/issues/67

- Remove function fbsocket.inet_aton

FIXED Pull Request 66: https://github.com/drkjam/netaddr/pull/66

- Added Function to create list of IPRange for non-contiguous IPSet

Release: 0.7.11

Date: 19th Mar 2014

Changes since 0.7.10

- **Performance of IPSet increased dramatically, implemented by** Stefan Nordhausen and Martijn van Oosterhout. As a side effect, IPSet(IPNetwork("10.0.0.0/8")) is now as fast as you'd expect.
- Various performance improvements all over the place.
- netaddr is now hosted on PyPI and can be installed via pip.
- Doing "10.0.0.42" in IPNetwork("10.0.0.0/24") works now.
- IPSet has two new methods: iscontiguous() and iprange(), thanks to Louis des Landes.
- Re-added the IPAddress.netmask_bits() method that was accidently removed.
- **Networks 128.0.0.0/16, 191.255.0.0/16, and 223.255.255.0/24 are not marked as** reserved IPv4 addresses any more. Thanks to marnickv for pointing that out.
- Various bug fixes contributed by Wilfred Hughes, 2*yo and Adam Goodman.

Specific bug fixes addressed in this release

FIXED Issue 58: https://github.com/drkjam/netaddr/issues/58

- foo.bar doesn't throw AddrFormatError

FIXED Issue 57: https://github.com/drkjam/netaddr/issues/57

- netaddr packages not hosted on PyPI

FIXED Issue 56: https://github.com/drkjam/netaddr/issues/56

- Fix comparison with large IPSet()

FIXED Issue 55: https://github.com/drkjam/netaddr/pull/55

- Fix smallest_matching_cidr and all_matching_cidrs

FIXED Issue 53: https://github.com/drkjam/netaddr/issues/53

- Exclude 128.0.0.0/16 and possibly others from reserved range set?

FIXED Issue 51: https://github.com/drkjam/netaddr/issues/51

- Encoding errors in netaddr/eui/oui.txt

FIXED Issue 46: https://github.com/drkjam/netaddr/issues/46

- len(IPSet()) fails on python3

FIXED Issue 43: https://github.com/drkjam/netaddr/issues/43

- Method to check if IPSet is contiguous

FIXED Issue 38: https://github.com/drkjam/netaddr/issues/38

- netmask_bits is missing from the IPAddress

FIXED Issue 37: https://github.com/drkjam/netaddr/issues/37

- Test failures with Python 3.3

Release: 0.7.10

Date: 6th Sep 2012

Changes since 0.7.9

- A bunch of Python 3.x bug fixes. Thanks Arfrever.
- Extended nmap support to cover full target specification.

Specific bug fixes addressed in this release

FIXED Issue 36 - http://github.com/drkjam/netaddr/issues/36

- ResourceWarnings with Python >=3.2

FIXED Issue 35 - http://github.com/drkjam/netaddr/issues/35

- netaddr-0.7.9: Test failure with Python 3

FIXED Issue 34 - http://github.com/drkjam/netaddr/issues/34

- netaddr.ip.iana.SaxRecordParser.endElement() incompatible with Python 3.1

FIXED Issue 33 - http://github.com/drkjam/netaddr/issues/33

- netaddr script not installed with Python 3

FIXED Issue 23 - http://github.com/drkjam/netaddr/issues/23

- valid_nmap_range() does not validate nmap format case.

FIXED Issue 22 - http://github.com/drkjam/netaddr/issues/22

- all_matching_cidrs: documentation incorrect

Release: 0.7.9

Date: 28th Aug 2012

Changes since 0.7.8

- Re-release to fix build removing Sphinx dependency.

Release: 0.7.8

Date: 28th Aug 2012

Changes since 0.7.7

- New SAX parser for IANA data source files (contributed by Andrew Stromnov)
- Fixed pickling failures with EUI, OUI and IAB classes.

Specific bug fixes addressed in this release

FIXED Issue 31 - http://github.com/drkjam/netaddr/issues/31

- Exclude '39.0.0.0/8' network from reserved set. Thanks Andrew Stromnov

FIXED Issue 28 - http://github.com/drkjam/netaddr/issues/28

- Fix algorithm in ipv6_link_local to fully conform to rfc4291. Thanks Philipp Wollermann

FIXED Issue 25 - http://github.com/drkjam/netaddr/issues/25

- install_requires is too aggressive? Thanks Adam Lindsay and commenters.

FIXED Issue 21 - http://github.com/drkjam/netaddr/issues/21

- deepcopy for EUI fails. Thanks Ryan Nowakowski.

Release: 0.7.7

Date: 30th May 2012

Changes since 0.7.6

- Comprehensive documentation update! It's only taken 4 years to get around to using Sphinx and I can confirm it is **TOTALLY AWESOME!**
- Various bug fixes
- Refreshed IEEE OUI and IAB data

Specific bug fixes addressed in this release

FIXED Issue 24 - http://github.com/drkjam/netaddr/issues/24

- Fixed TypeError when comparing BaseIP instance with non-BaseIP objects. Thanks pvaret

FIXED Issue 17 - http://github.com/drkjam/netaddr/issues/17

- For large ipv6 networks the .subnet() method fails. Thanks daveyss

FIXED Issue 20 - http://github.com/drkjam/netaddr/issues/20

- Test failure with Python 3. Thanks Arfrever

Release: 0.7.6

Date: 13th Sep 2011

Changes since 0.7.5

- A bug fix point release
- Refreshed 3rd party data caches
- Tested against Python 3.2.x and PyPy 1.6.x
- Fixed unit tests under for Mac OSX

Specific bug fixes addressed in this release

FIXED Issue 15 - http://github.com/drkjam/netaddr/issues/15

- Incorrect and invalid glob produced when last octet is not *

FIXED Issue 13 - http://github.com/drkjam/netaddr/issues/13

- Added support for IPython 0.11 API changes. Thanks juliantaylor

FIXED Issue 11 - http://github.com/drkjam/netaddr/issues/11

- Calling valid_glob on cidr raises ValueError. Thanks radicand

FIXED Issue 7 - http://github.com/drkjam/netaddr/issues/7

- Unpickling Bug in IPSet. Thanks LuizOz and labeneator

FIXED Issue 2 - http://github.com/drkjam/netaddr/issues/2

- UnboundLocalError raised in IPNetwork constructor. Thanks keesbos

Miscellanea

- Has a famous soft drink company started making it own NICs?

Release: 0.7.5

Date: 5th Oct 2010

Changes since 0.7.4

- Python 3.x is now fully supported. The paint is still drying on this so please help with testing and raise bug tickets when you find any issues! New Issue Tracker - http://github.com/drkjam/netaddr/issues
- Moved code hosting to github. History ported thanks to svn2git. - (http://github.com/nirvdrum/svn2git)
- All netaddr objects now use approx. 65% less memory due to the use of __slots__ in classes throughout the codebase. Thanks to Stefan Nordhausen and his Python guru for this suggestion!
- Applied many optimisations and speedups throughout the codebase.
- Fixed the behaviour of the IPNetwork constructor so it now behaves in a much more sensible and expected way (i.e. no longer uses inet_aton semantics which is just plain odd for network addresses).
- One minor change to behaviour in this version is that the .value property on IPAddress and IPNetwork objects no longer support assignment using a string IP address. Only integer value assignments are now valid. The impact of this change should be minimal for the majority of users.

Specific bug fixes addressed in this release

FIXED Issue 49 - http://code.google.com/p/netaddr/issues/detail?id=49

- Incorrect IP range recognition on IPs with leading zeros

FIXED Issue 50 - http://code.google.com/p/netaddr/issues/detail?id=50

- CIDR block parsing

FIXED Issue 52 - http://code.google.com/p/netaddr/issues/detail?id=52

- ipv6 cidr matches incorrectly match ipv4 [sic]

FIXED Issue 53 - http://code.google.com/p/netaddr/issues/detail?id=53

- Error in online documentation

FIXED Issue 54 - http://code.google.com/p/netaddr/issues/detail?id=54

- IP recognition failure

FIXED Issue 55 - http://code.google.com/p/netaddr/issues/detail?id=55

- Support for Python 3.x

FIXED Issue 56 - http://code.google.com/p/netaddr/issues/detail?id=56

- checking IPAddress in IPNetwork

FIXED Issue 57 - http://code.google.com/p/netaddr/issues/detail?id=57

- netaddr objects can't pickle

FIXED Issue 58 - http://code.google.com/p/netaddr/issues/detail?id=58

- IPSet operations should accept the same arguments as IPAddress

FIXED Issue 59 - http://code.google.com/p/netaddr/issues/detail?id=59

- netaddr fails to load when imported by a PowerDNS coprocess

Miscellanea

- Welcome back to standards.ieee.org which seems to have been down for weeks!
- Goodbye Sun Microsystems + Merrill Lynch, hello Oracle + Bank of America ...

Release: 0.7.4

Date: 2nd Dec 2009

Changes since 0.7.3

- Applied speed patches by S. Nordhausen
- Fixed an inconsistency between EUI and IPAddress interfaces. Made EUI.packed and EUI.bin properties (previously methods) and added a words() property.

Release: 0.7.3

Date: 14th Sep 2009

Changes since 0.7.2

- Added __add__, __radd__, __sub__, __rsub__ operators to the IPAddress class.
- Added support for validation and iteration of simple nmap style IPv4 ranges (raised in Issue 46).
- Removed some unused constants from fallback socket module.

Specific bug fixes addressed in this release

FIXED Issue 44 - http://code.google.com/p/netaddr/issues/detail?id=44

- int/long type error

FIXED Issue 46 - http://code.google.com/p/netaddr/issues/detail?id=46

- Question about IPv4 ranges

FIXED Issue 47 - http://code.google.com/p/netaddr/issues/detail?id=47

- IPNetwork cannot be evaluated as a boolean when it has a large size

Release: 0.7.2

Date: 20th Aug 2009

Changes since 0.7.1

FIXED a boundary problem with the iter_iprange() generator function and all associated calls to it throughout the codebase, including unit test coverage and adjustments.

- Replaced regular expressions in cidr_merge() with pre-compiled equivalents for a small speed boost.
- Adjustments to README raised by John Eckersberg.

Specific bug fixes addressed in this release

FIXED Issue 43 - http://code.google.com/p/netaddr/issues/detail?id=43

- IPNetwork('0.0.0.0/0') not usable in for loop

Release: 0.7.1

Date: 14th Aug 2009

Changes since 0.7

- Renamed the netaddr shell script from 'nash' to plain 'netaddr'. This is to avoid a potentially nasty clash with an important Linux tool with the same name.

  Thanks to John Eckersberg for spotting this one early!

- Updated IANA and IEEE data files with latest versions.

Specific bug fixes addressed in this release

FIXED Issue 42 - http://code.google.com/p/netaddr/issues/detail?id=42

- Bug in cidr_merge() function when passed the CIDRs 0.0.0.0/0 and/or ::/0

Release: 0.7

Date: 11th Aug 2009

Changes since 0.6.x

Please Note - This release represents a major overhaul of netaddr. It breaks backward compatibility with previous releases. See the API documentation for full details of what is available.

Some highlights of what has changed :-

- Internal module hierarchy has been completely overhauled and redesigned. This fixes up a lot of inconsistencies and problems with interdependent imports. All public classes, objects, functions and constants are still published via the main netaddr module namespace as in previous releases.
- No more AT_* and ST_* 'constants'.
- The Addr base class is gone. This removes the link between EUI and IP functionality so the library is can now easily be split into distinct units without many interdependencies between layer 2 and layer 3 functionality.
- The use of custom descriptor classes has been completely discontinued.
- Strategy classes and singleton objects have been replaced with a group of strategy modules in their own netaddr.strategy namespace. Each IP or EUI address object now holds a reference to a module rather than a singleton object.
- Many operations that were previously static class methods are now presented as functions in the relevant modules. See the API documentation for details.
- The IP and CIDR classes have been replaced with two new classes called IPAddress and IPNetwork respectively. This name change is important as the IP part of netaddr has been completed redesigned. The notion of an individual IP address and an IP network or subnet has been made more obvious. IPAddress objects are now true scalars and do not evaluate in a list or tuple context. They also do not support any notion of a netmask or CIDR prefix; this is the primary function of an IPNetwork object.
- Abritrary IP ranges and are still supported but a lot of their functionality has also been exposed via handy functions.
- IP globbing routines (previous known as Wildcards) have been moved into their own submodule.
- Added a new IPSet class which fully emulates mutable Python sets. This replaces a lot of half-baked experimental classes found in 0.5.x and 0.6.x such as IPRangeSet and CIDRGroup. See documentation for details.
- All methods and properties that previously used or supported the 'fmt' formatting property no longer do so. In all cases, objects are now returned to correctly support pass through calls without side effects. It is up to the user to extract data in the right format from the objects IPAddress objects returned as required.
- Unit tests have been completed re-written to support docstring style tests bundled into test suites. These are handy as they double up as documentation being combined with wiki syntax. Implemented code coverage checking using coverage 3.x.
- nash - a nascent shell like tool for the netaddr library (requires IPython).
- Support for RFC 1924 added ;-)

Specific bug fixes addressed in this release

FIXED Issue 13 - http://code.google.com/p/netaddr/issues/detail?id=13

- Searching for a match in a list of CIDR objects

FIXED Issue 26 - http://code.google.com/p/netaddr/issues/detail?id=26

- Refactor out use of isinstance()

FIXED Issue 28 - http://code.google.com/p/netaddr/issues/detail?id=28

- Add support for network block operations

FIXED Issue 34 - http://code.google.com/p/netaddr/issues/detail?id=34

- Addition issue?

Release: 0.6.4

Date: 11th Aug 2009

Specific bug fixes addressed in this release

FIXED Issue 40 - http://code.google.com/p/netaddr/issues/detail?id=40

- Building RPM wth "python setup.py bdist_rpm" fails, multiple errors

Release: 0.6.3

Date: 23rd Jun 2009

Changes since 0.6.2

- Fixed line endings in a number of new files created under Windows.
- Tweaked the ordering of values in tuple passed into the hash() function in the __hash__ method of the IP and IPRange classes to make it the same as the values used for comparisons implemented in the __eq__ method (Python best practice).
- Added a number of unit tests to improve code coverage.

Specific bug fixes addressed in this release

FIXED Issue 33 - http://code.google.com/p/netaddr/issues/detail?id=33

- CIDR subtraction is broken for out-of-range CIDR objects

FIXED Issue 35 - http://code.google.com/p/netaddr/issues/detail?id=35

- install error (on Python interpreters where socket.has_ipv6 is False)

FIXED Issue 36 - http://code.google.com/p/netaddr/issues/detail?id=36

- netaddr.CIDR fails to parse default route CIDR

FIXED Issue 37 - http://code.google.com/p/netaddr/issues/detail?id=37

- Bug in bitwise AND operator for IP addresses

FIXED Issue 38 - http://code.google.com/p/netaddr/issues/detail?id=38

- Feature request: Addr.__nonzero__

FIXED Issue 39 - http://code.google.com/p/netaddr/issues/detail?id=39

- CIDR.abbrev_to_verbose() not applying implicit classful netmask rules consistently

Release: 0.6.2

Date: 13th Apr 2009

Changes since 0.6.1

- Refreshed IEEE and IANA data files with latest revisions from their respective URLs.
  - IANA IPv4 Address Space Registry (last updated 2009-03-11)
  - Internet Multicast Addresses (last updated 2009-03-17)
  - IEEE OUI and IAB files (last updated 2009-04-13)
- Added get_latest_files() functions to both the netaddr.eui and netaddr.ip modules to assist in automating release builds.

Specific bug fixes addressed in this release

FIXED Issue 32 - http://code.google.com/p/netaddr/issues/detail?id=32

- Addr.__ne__ returns wrong answer

Release: 0.6.1

Date: 6th Apr 2009

Changes since 0.6

- Added COPYRIGHT file with details and attribution for all 3rd party files bundled with netaddr.
- Minimum Python version required is now 2.4.x changed from 2.3.x.
  - Python 2.3 compatibility code in many sections of code have been removed.
  - the @property and @staticmethod decorators are now used throughout the code along with the reversed() and sorted() builtin iterators.
  - A specific version check has also been added that will raise RuntimeError exceptions if you run netaddr on a Python interpreter version < 2.4.x.
- Integer addresses passed to the IP() and EUI() constructors no longer require a mandatory second address type (AT_*) argument in most cases. This is now only really required to disambiguate between IPv4/IPv6 addresses with the same numerical value. The same behaviour applies to EUI-48/EUI-64 identifiers. A small speed boost is achieved if the 2nd address type argument is explicitly provided.
- IPv6 addresses returned by EUI.ipv6_link_local() now always have a subnet prefix of /64.
- Default sort order of aggregate classes (IPRange, CIDR and Wildcard) has been changed (again). They now sort initially by first address and then by network block size from largest to smallest which feels more natural.
- Fixed a bug in the CIDR.abbrev_to_verbose() static method where IPv4 addresses with 4 octets (i.e. non-partial addresses) were being assigned subnet prefixes using abbreviated rules. All complete IPv4 addresses should always get a /32 prefix where it is not explicitly provided.

- Abbreviated address expansion in the CIDR constructor is now optional and can be controlled by a new 'expand_abbrev' boolean argument.
- Added the new CIDR.summarize() static method which transforms lists of IP addresses and CIDRs into their most compact forms. Great for trimming down large ad hoc address lists!
- Added the previous() and next() methods to the CIDR classes which return the CIDR subnets either side of a given CIDR that are of the same size. For the CIDR 192.0.2.0/24, previous will return 192.0.1.0/24 and next will return 192.0.3.0/24. Also accepts and optional step size (default is 1).
- Added the supernet() method to the CIDR class which returns a generator of all the subnets that contain the current CIDR found by decrementing the prefixlen value for each step until it reaches zero.
- Changed the way the fallback code works when the socket module is missing important constants and functions.
- Removed the uppercase options from the Strategy constructors and internals as this behaviour can be easily replicated using the word_fmt option instead and requires less code (word_fmt='%X').

Specific bug fixes addressed in this release

FIXED Issue 23 - http://code.google.com/p/netaddr/issues/detail?id=23

- Improve IPv6 IPv4 mapped/compatible address formatting

FIXED Issue 24 - http://code.google.com/p/netaddr/issues/detail?id=24

- bug in CIDR.subnet() when using the fmt argument

FIXED Issue 29 - http://code.google.com/p/netaddr/issues/detail?id=29

- CIDR.subnet method's count argument isn't working as documented

FIXED Issue 30 - http://code.google.com/p/netaddr/issues/detail?id=30

- not compatible with Python 2.3

FIXED Issue 31 - http://code.google.com/p/netaddr/issues/detail?id=31

- byte order in documentation confusing or wrong

Release: 0.6

Date: 20th Jan 2009

Changes since 0.5.x

- Namespace changes

  3 new sub namespaces have been added :-

  - netaddr.eui

  Currently contains IEEE OUI and IAB classes and lookup code.

  - netaddr.ip

  Currently contains IANA IPv4, IPv6 and IPv4 multicast lookup code.

  - netaddr.core

Currently contains only a couple of classes that are shared between code in netaddr.eui and netaddr.ip.

Please Note: This change is part of a two stage internal restructuring of netaddr. In future releases, layer-2 MAC/EUI functionality will be separated from and layer-3 IP, CIDR and Wildcard functionality. All shared code will be moved to netaddr.core. When the migration is complete (expected in 0.7) the netaddr.address and netaddr.strategy namespaces will be removed. Please endeavour to access everything you need via the top-level netaddr namespace from this release onwards. See netaddr.__all__ for details of constants, objects, classes and functions intended for the public interface.

- Addition of IEEE and IANA informational lookups
  - the IP() and EUI() classes now have an additional info() method through which contextual information about your addresses can be accessed. This data is published by IANA and the IEEE respectively and sourced directly from text files bundled with netaddr that are available for download publically online. Details are available in the docstring of the relevant parsing classes. Subsequent netaddr releases will endeavour to keep up-to-date with any updates to these files.
  - the EUI() class has been updated with the addition of the OUI() and IAB() classes. They provide object based access to returned via the EUI.info() method. Please see API docs included with netaddr for details.
  - added new NotRegisteredError exception that is raised when an EUI doesn't match any currently registration entries in the IEEE registry files.
- Addr() class removed from the public interface
  - This class is only ever meant to be used internally and its usage may soon be deprecated in favour converting it into an abstract base class in future releases.
- Deletion of AddrRange() class
  - replaced with the more specific IPRange() class. AddrRange() wasn't very useful in practice. Too much time has been spent explaining its theoretical merits over its actual practicality for every day use.
- Addition of new IPRange() class
  - the new base class for CIDR() and Wildcard().
  - a 'killer feature' of this new class are the new methods iprange(), cidrs() and wildcard() which allow you to use and switch between all 3 formats easily. IPRange('x', 'y').cidrs() is particularly useful returning all the intervening CIDRs between 2 arbitrary IP addresses.
  - IPRange() is a great place to expose several new methods available to sub classes. They are issupernet(), issubnet(), adjacent() and overlaps().
  - previous method called data_flavour() has been renamed (again) to a more suitable format().
- IP() class updates
  - is_netmask() and is_hostmask() methods have been optimised and are now both approximately 4 times faster than previously!
  - added wildcard() and iprange() methods that return pre-initialised objects of those classes based on the current netmask / subnet prefix.
  - copy constructor methods ipv4() and ipv6() now preserve the value of the prefixlen property now also support IPv6 options for returning IPv4-mapped or IPv4-compatible IPv6 addresses.
  - added new methods is_loopback(), is_private(), is_link_local(), is_ipv4_mapped() and is_ipv4_compat() which are all self explanatory.
  - added a bin() method which provides an IP address in the same format as the standard Python bin() builtin type ('0bxxx') now available in Python 2.6.x and higher.

  - added a packed() method which provides an IP address in packed binary string format, suitable for passing directly to Python socket calls.

- nrange() generator function updates
  - by default this now returns IP() objects instead of Addr() objects.
- CIDR() class updates
  - the 'strict_bitmask' option in the CIDR class constructor has been had a name change and is now just 'strict' (less typing).
  - support for Cisco ACL-style (hostmask) prefixes. Also available to the IP() class. They are converted to their netmask equivalents before being applied to the base address.
  - added a new subnet() generator method that returns iterators to subnet CIDRs found within the current CIDR object's boundaries e.g. a /24 CIDR can provide address with subnet prefixes between a /25 and /32.
  - added a new span() method which takes a list of IP, IPRange, CIDR and/or Wildcards returning a single CIDR that 'spans' the lowest and highest boundary addresses. An important property of this class is that only a single CIDR is returned and that it (potentially) overlaps the start and end addresses. The most important aspect of this method is that it identifies the left-most set of bits that are common to all supplied addresses. It is the plumbing that makes a lot of other features function correctly.
  - although IPv6 doesn't support the concept of a broadcast address, after some pondering I've decide to add network() and broadcast() methods to the CIDR class. It is an interface quirk that users expect so it has been added for ease of use.
  - the methods network(), broadcast(), hostmask() and netmask() have been wrapped in property() builtin calls to make them appear as read-only properties.
- Many more MAC and IPv4 string address representation are now supported
  - Improvements to both EUI and IP classes. They now accept many more valid address formats than previously. Thanks for all the bugs tickets raised.
- `__repr__()` method behaviour change
  - Using `repr()` now assume that you have performed a `from netaddr import *` before you execute them. They no longer specify the originating namespace of objects which is a bit unnecessary and a lot to read on-screen.They will also be moving around within the namespace shortly anyway so its best not to think of them as being anywhere other than directly below netaddr itself.
- 'klass' property renamed to 'fmt' (format)
  - now referred to as the 'format callable' property. An unfortunately but necessary change. 'klass' was a bad initial name choice as it most often doesn't even reference a class object also supporting references to Python types, builtin functions and user defined callables.
- Complete re-work and consolidation of unit tests.
  - now over 100 tests covering all aspects of the API and library functionality.
  - Moved all tests into a single file. Lots of additional tests have been added along with interface checks to ensure netaddr's always presents a predictable set of properties and methods across releases.
- Nascent support for Python eggs and setuptools.
  - Help is need to test this as it is not something I use personally.

Specific bug fixes addressed in this release

- Finally fixed the IPv6 string address compression algorithm so that it is now compliant with the socket modules inet_ntop() and inet_pton() calls. (not available on all platforms).

Experimental Features

- added bitwise operators to the IP class
  - does what it says on the tin. Does not effect that value of the IP object itself but rather, returns a new IP after the operation has been applied.
- IPRangeSet() class added (EXPERIMENTAL).
  - the intention with this class is to allows you to create collections of unique IP(), IPRange(), CIDR() and Wildcard() objects. It provides iteration over IPs in the collection as well as several membership based operations such as any_match() all_matches(), min_match() and max_match().
  - lots more work to do here. Please raise bugs and feature requests against this as you find them. Improvements to this are coming in 0.7.

Release: 0.5.2

Date: 29th Sep 2008

Specific bug fixes addressed in this release

- Fixed Issue 15 in bug tracker. Bad validation and conversion of IPv4 mapped IPv6 address values in IPv6Strategy class. Covered with unit test cases.
- Updated PrefixLenDescriptor() class so that modifications to the property CIDR.prefixlen also update CIDR.first and CIDR.last keeping them in sync. Covered by unit test cases.
- IP.hostname() method returns None when DNS lookup fails.

Release: 0.5.1

Date: 23rd Sep 2008

Specific bug fixes addressed in this release

- CIDR constructor was throwing a TypeError for valid unicode string addresses which worked in previous releases. Fixed and covered with a unit test case.
- The methods CIDR.netmask() and CIDR.hostmask() contained code errors that were causing them to fail. Problem fixed and covered with unit test case.

Release: 0.5

Date: 19th Sep 2008

Changes since 0.4.x

General

- Access to all important object attributes in all netaddr classes now takes place via custom Python descriptor protocol classes. This has greatly simplified internal class logic and made external attributes changes much safer and less error prone. It has also made aggregate classes such as CIDR and Wildcard effectively read-write rather than read-only which they have been up until this release.
- Ammended the way sort order is calculated for Addr and AddrRange (sub)class instances so that the address type is taken into account as well as as the numerical value of the address or address range. The ascending sort order is IPv4, IPv6, EUI-48 and EUI-64. Sequences of AddrRange (sub)class instances now sort correctly!
- Comparisons between instances of Addr and AddrRange (sub)classes now return False, rather than raising an AttributeError.
- Added checks and workaround code for Python runtime environments that suffer from the infamous socket module inet_aton('255.255.255.255') bug. This was discovered recently in Python 2.4.x on PowerPC under MacOS X. The fix also applies in cases where the socket module is not available (e.g. on Google App Engine).
- All general Exception raising in the strategy module has now been replaced with more specific exceptions, mainly ValueError (these were unintentionally missed out of the 0.4 release).
- Implemented __hash__() operations for the Addr and AddrStrategy classes. This allows you to use IP, CIDR and Wildcard objects as keys in dictionaries and as elements in sets. Please note - this is currently an experimental feature which may change in future releases.
- Added __ne__() operation to Addr and AddrRange classes.
- Obeying the 'Law of Demeter', the address type of Addr and AddrRange (sub)class instances can be accessed using the property directly :-

  obj.addr_type # 0.5 onwards

  rather than having to go via the strategy object :-

  obj.strategy.addr_type # 0.4 and earlier
- Renamed the AT_DESCR lookup dictionary to AT_NAMES. Removed invalid and duplicated imports from all modules.

Addr class changes

- Removed the setvalue() method from the Addr class and replaced all uses of __setattr__() replaced by custom descriptors throughout.

IP class changes

- Removed the ambiguity with masklen and prefixlen attributes in the IP class. prefixlen now denotes the number of bits that define the netmask for an IP address. The new method netmask_bits() returns the number of non-zero bits in an IP object if the is_netmask() method returns True. A prefixlen value other than /32 for an address where is_netmask() returns True is invalid and will raise a ValueError exception.
- Removed the family() method from the IP class. It duplicates information now provided by the prefixlen property.
- IP class has several new methods. is_multicast() and is_unicast() quickly tell you what category of IP address you have and while ipv4() and ipv6() act as IPv4 <-> IPv6 conversions or copy constructors depending on context.
- Reverse DNS lookup entries now contain a trailing, top-level period (.) character appended to them.
- Added the hostname() method to IP instances which performs a reverse DNS

- The IP class __str__() method now omits the subnet prefix is now implicit for IPv4 addresses that are /32 and IPv6 addresses that are /128. Subnet prefix is maintained in return value for all other values.

AddrRange class changes

- The AddrRange class no longer stores instances of Addr (sub)classes for the first and last address in the range. The instance variables self.start_addr and self.stop_addr have been renamed to self.first and self.last and the methods obj.first() and obj.last() have been removed.

  Instead, self.first and self.last contain integer values and a reference to a strategy object is stored. Doing this is a lot more useful and cleaner for implementing internal logic.

  To get Addr (sub)class objects (or strings, hex etc when manipulating the the klass property) use the index values obj[0] and obj[-1] as a substitute for obj.first() and obj.last() respectively.

- AddrRange (sub)class instances now define the increment, __iadd__(), and decrement, __isub__(), operators. This allows you to 'slide' CIDRs and Wildcards upwards and downwards based on their block sizes.

- The _retval() method has now been renamed data_flavour() - yes, the UK spelling ;-) You shouldn't really care much about this as it mostly for internal use. I gave it a decent name as I didn't see any real need to hide the functionality if users wanted it.

CIDR class changes

- The strictness of the CIDR class constructor in relation to non-zero bits once the prefix bitmask has been applied can be disabled use the optional argument strict_bitmask=False. It is True (strictness enabled) by default.

- Fixed a bug in abbreviated CIDR conversion. Subnet prefix for multicast address 224.0.0.0 is now /4 instead of /8.

- The CIDR class now supports subtraction between two CIDR objects, returning a list of the remainder. Please note that the bigger of the two CIDR objects must be on the left hand side of the the expression, otherwise an empty list is return. Sorry, you are not allowed to create negative CIDRs ;-)

- The function abbrev_to_cidr() has been renamed to and turned into the static method CIDR.abbrev_to_verbose(). No major changes to the logic have been made.

Wildcard class changes

- The Wildcard class now defines a static method Wildcard.is_valid() that allows you to perform validity tests on wildcard strings without fully instantiation a Wildcard object.

Release: 0.4

Date: 7th Aug 2008

Changes since 0.3.x

- All general Exception raising has been replaced with more specific exceptions such as TypeError and ValueError and with the addition of two custom exception classes, AddrFormatError and AddrConversionError.

- The IP class now accepts a subnet prefix. It is *NOT* strict about non-zero bits to the right of implied subnet mask, unlike the CIDR class (see below).

- The CIDR class is now completely strict about non-zero bits to the right of the implied subnet netmask and raises a ValueError if they exist, with a handy hint as to the correct CIDR to be used based on the supplied subnet prefix.

- The CIDR class now also supports abbreviated CIDR ranges and uses older classful network address rules to decided on a subnet prefix if one is not explicitly provided. Supported forms now include 10, 10/8 and 192.168/16. Currently only supports these options for IPv4 CIDR address ranges.
- __repr__() methods have been defined for all classes in the netaddr module producing executable Python statements that can be used to re-create the state of any object.
- CIDR and Wildcard classes now have methods that support conversions between these two aggregate types :-
  - CIDR -> Wildcard
  - Wildcard -> CIDR

Housekeeping Changes

- Massive docstring review and tidy up with the inclusino of epydoc specific syntax to spruce up auto-generated API documentation.
- Thorough review of code using pylint.
- Netaddr module now has the special __version__ variable defined which is also referenced by setup.py.
- Some minor changes to setup.py and MANIFEST.in.
- Constants and custom Exception classes have been moved to __init__.py from strategy.py
- An import * friendly __all__ has been defined for the netaddr namespace which should remove the need to delve too much into the address and strategy submodules.
- Fixed a number of line-ending issues in several files.

CHAPTER 8

Standards and References

The following references are applicable to the netaddr library.

RFCs

The following RFCs have guided netaddr's feature set and capabilities.

IPv4

RFC 791 - Internet Protocol

- http://www.ietf.org/rfc/rfc791

RFC 1918 - Address Allocation for Private Internets

- http://www.ietf.org/rfc/rfc1918

RFC 3330 - Special-Use IPv4 Addresses

- http://www.ietf.org/rfc/rfc3330

RFC 3927 - Dynamic Configuration of IPv4 Link-Local Addresses

- http://www.ietf.org/rfc/rfc3927

Multicast (IPv4)

RFC 2365 - Administratively Scoped IP Multicast

- http://www.ietf.org/rfc/rfc2365

RFC 3171 - IANA IPv4 Multicast Guidelines

- http://www.ietf.org/rfc/rfc3171

RFC 3927 - Dynamic Configuration of IPv4 Link-Local Addresses

- http://www.ietf.org/rfc/rfc3927

IPv6

RFC 3330 - Special-Use IPv4 Addresses

- http://www.ietf.org/rfc/rfc3330

RFC 4291 - IPv6 Addressing Architecture

- http://www.ietf.org/rfc/rfc4291

RFC 3306 - Unicast-Prefix-based IPv6 Multicast

- http://www.ietf.org/rfc/rfc3306

RFC 3956 - The RP Address in IPv6 Multicast Address

- http://www.ietf.org/rfc/rfc3956

RFC 3879 - Deprecating Site Local Addresses

- http://www.ietf.org/rfc/rfc3879

RFC 4193 - Unique Local IPv6 Unicast Addresses

- http://www.ietf.org/rfc/rfc4193

RFC 4941 - Privacy Extensions for Stateless Address

- http://www.ietf.org/rfc/rfc4941

RFC 1924 - A Compact Representation of IPv6 Addresses

- http://www.ietf.org/rfc/rfc1924

Classless Inter-Domain Routing (CIDR)

RFC 1338 - Supernetting: an Address Assignment and Aggregation Strategy

- http://www.ietf.org/rfc/rfc1338

RFC 4632 - Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan

- http://www.ietf.org/rfc/rfc4632

Data Sources

Data from the following sources is exposed via the netaddr API.

Internet Assigned Numbers Authority (IANA)

IANA Protocol Registry

- http://www.iana.org/protocols/

IPv4 Address Space

- http://www.iana.org/assignments/ipv4-address-space

IPv6 Address Space

- http://www.iana.org/assignments/ipv6-address-space

Multicast Registrations

- http://www.iana.org/assignments/multicast-addresses

Institute of Electrical and Electronics Engineers (IEEE)

IEEE Organisation Registry

- http://standards.ieee.org/regauth/oui/index.shtml

OUI (Organisationally Unique Identifier) Registrations

- http://standards.ieee.org/regauth/oui/oui.txt

IAB (Individual Address Block) Registrations

- http://standards.ieee.org/regauth/oui/iab.txt

CHAPTER 9

Authors

- David P. D. Moss (author, maintainer) drkjam@gmail.com
- Stefan Nordhausen (maintainer) stefan.nordhausen@immobilienscout24.de

CHAPTER 10

Contributors

netaddr is written and maintained by David P. D. Moss

It is released under the BSD License.

Many people further contributed to netaddr by reporting problems, suggesting various improvements or submitting actual code. Here is a list of these people (in alphabetical order). Help me keep it complete and free of errors.

Vincent Bernat <bernat AT debian.org>

Sebastien Douche <sdouche AT gmail.com>

John Eckersberg <john DOT eckersberg AT gmail.com>

Yi-Jheng Lin <yzlin AT cs.nctu.edu.tw>

Clay McClure <clay AT daemons.net>

Duncan McGreggor <duncan DOT mcgreggor AT gmail.com>

Stefan Nordhausen <stefan DOT nordhausen AT axiros.com>

Brian F. Peters <brianfpeters AT gmail.com>

James William Pye <jwp AT gmail.com>

Chaitan Rogers <chaitan DOT rogers AT gmail.com>

Victor Stinner <victor DOT stinner AT haypocalc.com>

Andrew Stromnov <stromnov AT gmail.com>

Thanks to everyone on the netaddr mailing list, those who raised bug reports and to all those who have, directly and indirectly, guided and influenced the development and distribution of this library.

Thanks also for the use of the following code contributions :-

1. Python Cookbook recipe 18.11: “Formatting Integers as Binary Strings”

   Python Cookbook 2d ed. (O’Reilly Media 2005) ISBN 0596-00797-3 Alex Martelli, Anna Martelli Ravenscroft and David Ascher

2. ASPN Cookbook Recipe 466286: “Integer set type” by Heiko Wundram

http://code.activestate.com/recipes/466286/

And last but not least, thanks to Guido van Rossum for his encouraging words and for giving us all Python.

CHAPTER 11

Copyright

Here are the copyright notices applicable to the netaddr library.

netaddr

IANA (Internet Assigned Numbers Authority)

netaddr is not sponsored nor endorsed by IANA.

Use of data from IANA (Internet Assigned Numbers Authority) is subject to copyright and is provided with prior written permission.

IANA data files included with netaddr are not modified in any way but are parsed and made available to end users through an API.

See README file and source code for URLs to latest copies of the relevant files.

IEEE (Institution of Electrical Engineers)

netaddr is not sponsored nor endorsed by the IEEE.

Use of data from the IEEE (Institute of Electrical and Electronics Engineers) is subject to copyright. See the following URL for details :-

http://www.ieee.org/web/publications/rights/legal.html

IEEE data files included with netaddr are not modified in any way but are parsed and made available to end users through an API. There is no guarantee that referenced files are not out of date.

See README file and source code for URLs to latest copies of the relevant files.

CHAPTER 12

License

Here are the licenses applicable to the use of the netaddr library.

netaddr

CHAPTER 13

Indices and tables

- genindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L