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
```