cyberpandas Documentation

Release 0.1.0

ToTom Augspurger

Jun 12, 2018

Contents:

cyberpandas is a library for working with arrays of IP Addresses. It's specifically designed to work well with pandas.

CHAPTER 1

Key Concepts

1.1 IPType

This is a data type (like `numpy.dtype('int64')` or `pandas.api.types.CategoricalDtype()`. For the most part, you won't interact with `IPType` directly. It will be the value of the `.dtype` attribute on your arrays.

1.2 IPArray

This is the container for your IPAddress data.

1.3 Usage

```
In [1]: from cyberpandas import IPArray

In [2]: import pandas as pd

In [3]: arr = IPArray(['192.168.1.1',
   ...:                '2001:0db8:85a3:0000:0000:8a2e:0370:7334'])
   ...:

In [4]: arr
Out[4]: IPArray(['192.168.1.1', '2001:db8:85a3::8a2e:370:7334'])
```

`IPArray` is a container for both IPv4 and IPv6 addresses. It can in turn be stored in pandas' containers:

```
In [5]: pd.Series(arr)
Out[5]:
0                    192.168.1.1
1    2001:db8:85a3::8a2e:370:7334
```

(continues on next page)

(continued from previous page)

```
dtype: ip

In [6]: pd.DataFrame({"addresses": arr})
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[6]:␣
↪
                      addresses
0                   192.168.1.1
1   2001:db8:85a3::8a2e:370:7334
```

See *Usage* for more.

CHAPTER 2

API

2.1 Install

cyberpandas requires pandas 0.23 or newer. On Python 2, the 3rd party *ipaddress* module is required (it's built into the standard library in Python 3).

Once pandas is installed, cyberpandas can be installed from conda-forge:

```
conda install -c conda-forge cyberpandas
```

Or PyPI:

```
pip install cyberpandas
```

2.2 Usage

This document describes how to use the methods and classes provided by `cyberpandas`.

We'll assume that the following imports have been performed.

```
In [1]: import ipaddress

In [2]: import pandas as pd

In [3]: from cyberpandas import IPArray, to_ipaddress
```

2.2.1 Parsing

First, you'll need some IP Address data. Much like pandas' `pandas.to_datetime()`, `cyberpandas` provides `to_ipaddress()` for converting sequences of anything to a specialized array, `IPArray` in this case.

From Strings

`to_ipaddress()` can parse a sequence strings where each element represents an IP address.

```
In [4]: to_ipaddress([
   ...:     '192.168.1.1',
   ...:     '2001:0db8:85a3:0000:0000:8a2e:0370:7334',
   ...: ])
   ...:
Out[4]: IPArray(['192.168.1.1', '2001:db8:85a3::8a2e:370:7334'])
```

You can also parse a *container* of bytes (Python 2 parlance).

```
In [5]: to_ipaddress([
   ...:     b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xc0\xa8\x01\x01',
   ...:     b' \x01\r\xb8\x85\xa3\x00\x00\x00\x00\x8a.\x03ps4',
   ...: ])
   ...:
Out[5]: IPArray(['192.168.1.1', '2001:db8:85a3::8a2e:370:7334'])
```

If you have a buffer / bytestring, see *From Bytes*.

From Integers

IP Addresses are just integers, and `to_ipaddress()` can parse a sequence of them.

```
In [6]: to_ipaddress([
   ...:    3232235777,
   ...:    42540766452641154071740215577757643572
   ...: ])
   ...:
Out[6]: IPArray(['192.168.1.1', '2001:db8:85a3::8a2e:370:7334'])
```

There's also the `IPArray.from_pyints()` method that does the same thing.

From Bytes

If you have a correctly structured buffer of bytes or bytestring, you can directly construct an `IPArray` without any intermediate copies.

```
In [7]: stream = (b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xc0\xa8\x01'
   ...:           b'\x01 \x01\r\xb8\x85\xa3\x00\x00\x00\x00\x8a.\x03ps4')
   ...:

In [8]: IPArray.from_bytes(stream)
Out[8]: IPArray(['192.168.1.1', '2001:db8:85a3::8a2e:370:7334'])
```

`stream` is expected to be a sequence of bytes representing IP Addresses (note that it's just a bytestring that's be split across two lines for readability). Each IP Address should be 128 bits, left padded with 0s for IPv4 addresses. In particular, `IPArray.to_bytes()` produces such a sequence of bytes.

2.2.2 Pandas Integration

`IPArray` satisfies pandas extension array interface, which means that it can safely be stored inside pandas' Series and DataFrame.

```
In [9]: values = to_ipaddress([
   ...:     0,
   ...:     3232235777,
   ...:     42540766452641154071740215577757643572
   ...: ])
   ...:

In [10]: values
Out[10]: IPArray(['0.0.0.0', '192.168.1.1', '2001:db8:85a3::8a2e:370:7334'])

In [11]: ser = pd.Series(values)

In [12]: ser
Out[12]:
0                         0.0.0.0
1                     192.168.1.1
2    2001:db8:85a3::8a2e:370:7334
dtype: ip

In [13]: df = pd.DataFrame({"addresses": values})

In [14]: df
Out[14]:
                     addresses
0                      0.0.0.0
1                  192.168.1.1
2  2001:db8:85a3::8a2e:370:7334
```

Most pandas methods that make sense should work. The following section will call out points of interest.

Indexing

If your selection returns a scalar, you get back an `ipaddress.IPv4Address` or `ipaddress.IPv6Address`.

```
In [15]: ser[0]
Out[15]: IPv4Address('0.0.0.0')

In [16]: df.loc[2, 'addresses']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[16]: IPv6Address('2001:db8:85a3::8a2e:370:7334')
```

Missing Data

The address 0 (`0.0.0.0`) is used to represent missing values.

```
In [17]: ser.isna()
Out[17]:
0     True
1    False
2    False
dtype: bool

In [18]: ser.dropna()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[18]:
1                     192.168.1.1
```

(continues on next page)

(continued from previous page)

```
2    2001:db8:85a3::8a2e:370:7334
dtype: ip
```

2.2.3 IP Accessor

`cyberpandas` offers an accessor for IP-specific methods.

```
In [19]: ser.ip.isna
Out[19]:
0     True
1    False
2    False
dtype: bool

In [20]: df['addresses'].ip.is_ipv6
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[20]:
0    False
1    False
2     True
Name: addresses, dtype: bool
```

2.3 API

Cyberpandas provides two extension types, *IPArray* and `MACArray`.

2.3.1 IP Array

class cyberpandas.**IPArray**(*values*)

Holder for IP Addresses.

IPArray is a container for IPv4 or IPv6 addresses. It satisfies pandas' extension array interface, and so can be stored inside `pandas.Series` and `pandas.DataFrame`.

See *Usage* for more.

Constructors

The class constructor is flexible, and accepts integers, strings, or bytes. Dedicated alternate constructors are also available.

classmethod IPArray.**from_pyints**(*values*)

Construct an IPArray from a sequence of Python integers.

This can be useful for representing IPv6 addresses, which may be larger than 2**64.

Parameters

values [Sequence] Sequence of Python integers.

Examples

```
>>> IPArray.from_pyints([0, 10, 2 ** 64 + 1])
IPArray(['0.0.0.1', '0.0.0.2', '0.0.0.3', '0:0:0:1::'])
```

classmethod IPArray.**from_bytes**(*bytestring*)

Create an IPArray from a bytestring.

Parameters

bytestring [bytes] Note that bytestring is a Python 3-style string of bytes, not a sequences of bytes where each element represents an IPAddress.

Returns

IPArray

See also:

to_bytes, *from_pyints*

Examples

```
>>> arr = IPArray([10, 20])
>>> buf = arr.to_bytes()
>>> buf
b'\\\...x00}'
>>> IPArray.from_bytes(buf)
IPArray(['0.0.0.10', '0.0.0.20'])
```

Finally, the top-level `ip_range` method can be used.

cyberpandas.**ip_range**(*start=None*, *stop=None*, *step=None*)

Generate a range of IP Addresses

Parameters

start [int, str, IPv4Address, or IPv6Address, optional] Start of interval. The interval includes this value. The default start value is 0.

start [int, str, IPv4Address, or IPv6Address, optional] End of interval. The interval does not include this value.

step [int, optional] Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.

Returns

IPArray

Notes

Performance will worsen if either of *start* or *stop* are larger than 2**64.

Examples

From integers

```
>>> ip_range(1, 5)
IPArray(['0.0.0.1', '0.0.0.2', '0.0.0.3', '0.0.0.4'])
```

Or strings

```
>>> ip_range('0.0.0.1', '0.0.0.5')
IPArray(['0.0.0.1', '0.0.0.2', '0.0.0.3', '0.0.0.4'])
```

Or *ipaddress* objects

```
>>> ip_range(ipaddress.IPv4Address(1), ipaddress.IPv4Address(5))
IPArray(['0.0.0.1', '0.0.0.2', '0.0.0.3', '0.0.0.4'])
```

Serialization

Convert the IPArray to various formats.

IPArray.**to_pyipaddress**()

Convert the array to a list of scalar IP Adress objects.

Returns

addresses [List] Each element of the list will be an `ipaddress.IPv4Address` or `ipaddress.IPv6Address`, depending on the size of that element.

See also:

IPArray.to_pyints

Examples

```
>>> IPArray(['192.168.1.1', '2001:db8::1000']).to_pyipaddress()
[IPv4Address('192.168.1.1'), IPv6Address('2001:db8::1000')]
```

IPArray.**to_pyints**()

Convert the array to a list of Python integers.

Returns

addresses [List[int]] These will be Python integers (not NumPy), which are unbounded in size.

See also:

IPArray.to_pyipaddresses, *IPArray.from_pyints*

Examples

```
>>> IPArray(['192.168.1.1', '2001:db8::1000']).to_pyints()
[3232235777, 42540766411282592856903984951653830656]
```

IPArray.**to_bytes**()
Serialize the IPArray as a Python bytestring.

This and :meth:IPArray.from_bytes is the fastest way to roundtrip serialize and de-serialize an IPArray.

See also:

IPArray.from_bytes

Examples

```
>>> arr = IPArray([10, 20])
>>> arr.to_bytes()
b'\\\...x00}'
```

Methods

Various methods that are useful for pandas. When a Series contains an IPArray, calling the Series method will dispatch to these methods.

IPArray.**take**(*indices*, *allow_fill=False*, *fill_value=None*)
Take elements from an array.

Parameters

indices [sequence of integers] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if nescessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`, `pandas.api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignemnt, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result)
```

`IPArray.`**`unique`**`()`
Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

`IPArray.`**`isin`**`(`*`other`*`)`
Check whether elements of *self* are in *other*.

Comparison is done elementwise.

Parameters

other [str or sequences] For `str` *other*, the argument is attempted to be converted to an `ipaddress.IPv4Network` or a `ipaddress.IPv6Network` or an *`IPArray`*. If all those conversions fail, a TypeError is raised.

For a sequence of strings, the same conversion is attempted. You should not mix networks with addresses.

Finally, other may be an `IPArray` of addresses to compare to.

Returns

contained [ndarray] A 1-D boolean ndarray with the same length as self.

Examples

Comparison to a single network

```
>>> s = IPArray(['192.168.1.1', '255.255.255.255'])
>>> s.isin('192.168.1.0/24')
array([ True, False])
```

Comparison to many networks >>> s.isin([‘192.168.1.0/24’, ‘192.168.2.0/24’]) array([True, False])

Comparison to many IP Addresses

```
>>> s.isin(['192.168.1.1', '192.168.1.2', '255.255.255.1']])
array([ True, False])
```

IPArray.**isna**()
Indicator for whether each element is missing.

The IPAddress 0 is used to indecate missing values.

Examples

```
>>> IPArray(['0.0.0.0', '192.168.1.1']).isna()
array([ True, False])
```

IP Address Attributes

IP addresss-specific attributes.

IPArray.**is_ipv4**
Indicator for whether each address fits in the IPv4 space.

IPArray.**is_ipv6**
Indicator for whether each address requires IPv6.

IPArray.**version**
IP version (4 or 6).

IPArray.**is_multicast**
Indiciator for whether each address is multicast.

IPArray.**is_private**
Indiciator for whether each address is private.

IPArray.**is_global**
Indiciator for whether each address is global.

IPArray.**is_unspecified**
Indiciator for whether each address is unspecified.

IPArray.**is_reserved**
Indiciator for whether each address is reserved.

IPArray.**is_loopback**
Indiciator for whether each address is loopback.

IPArray.**is_link_local**
Indiciator for whether each address is link local.

IPArray.**netmask**(*v4_prefixlen=32*, *v6_prefixlen=128*)

Compute an array of netmasks for an array of IP addresses.

Note that this is a method, rather than a property, to support taking *v4_prefixlen* and *v6_prefixlen* as arguments.

Parameters

v4_prefixlen [int, default 32] Length of the network prefix, in bits, for IPv4 addresses

v6_prefixlen [int, default 128] Lnegth of the network prefix, in bits, for IPv6 addresses

Returns

IPArray

See also:

IPArray.hostmask

Examples

```
>>> arr = ip.IPArray(['192.0.0.0', '1:1::'])
>>> arr.netmask(v4_prefixlen=16, v6_prefixlen=32)
IPArray(['255.255.0.0', 'ffff:ffff::'])
```

IPArray.**hostmask**(*v4_prefixlen=32*, *v6_prefixlen=128*)

Compute an array of hostmasks for an array of IP addresses.

Parameters

v4_prefixlen [int, default 32] Length of the network prefix, in bits, for IPv4 addresses

v6_prefixlen [int, default 128] Lnegth of the network prefix, in bits, for IPv6 addresses

Returns

IPArray

See also:

IPArray.netmask

Examples

```
>>> arr = ip.IPArray(['192.0.0.0', '1:1::'])
>>> arr.netmask(v4_prefixlen=16, v6_prefixlen=32)
IPArray(['0.0.255.255', '::ffff:ffff:ffff:ffff:ffff:ffff'])
```

IPArray.**mask**(*mask*)

Apply a host or subnet mask.

Parameters

mask [IPArray] The host or subnet mask to be applied

Returns

masked [IPArray]

See also:

netmask, *hostmask*

Examples

```
>>> arr = IPArray(['216.003.128.12', '192.168.100.1'])
>>> mask = arr.netmask(v4_prefixlen=24)
>>> mask
IPArray(['255.255.255.0', '255.255.255.0'])
>>> arr.mask(mask)
IPArray(['216.3.128.0', '192.168.100.0'])
```

2.3.2 MACArray

utofun .. autoclass:: MACArray

2.4 Changelog

2.4.1 Version 1.1.1

- Added `IPArray.mask()` to apply net and host masks to an array of IP addresses (GH#36).

2.4.2 Version 1.1.0

- Added `ip_range()` for generating an array of regularly-spaced IP addresses (GH#27).
- Added `IPArray.netmask()` and `IPArray.hostmask()` (GH#30).
- Fixed Python 2 dependencies so that the *ipaddress* backport is installed automatically when install cyberpandas from PyPI (GH#29).

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index