# py-cidr

***Release 3.12.0***

**Gene C**

**Feb 21, 2026**

# CONTENTS

CHAPTER

ONE

# PY-CIDR

## 1.1 Overview

py-cidr : python module providing network / CIDR tools

## 1.2 Key features

- Built on python's native ipaddress module
- 3 Classes : Cidr, CidrMap, CidrFile
- Cidr provides for many common operations for example:
  - Support for IPv4 and IPv6
  - compact lists of CIDRs to smallest set of CIDR blocks
  - convert an IP range to a list of CIDRs
  - Identify and validate
  - many more
- CidrFile offers common operations on files with lists of cidrs.
  - Includes atomic file writes
- CidrMap provides a class that maps CIDRs to values.
  - File cache employs locking to ensure multiple processes handle cache correctly.

See API reference documentation for more details.

## 1.3 New / Interesting

**3.12.0**

- CidrMap : New .items() method provides an Iterator over the map.

  Each iteration yields a tuple[cidr: str, value: Any]
- Add net_range_split/cidr_range_split:

  split one net/cidr into (first, mid, last) ip addresses

**3.11.0**

- Code Reorg
- Switch packaging from hatch to uv

- Testing to confirm all working on python 3.14.2
- License GPL-2.0-or-later

**Older**

- Rename *py-cidr-cache-print* (without the .py extension)
- PEP-8, PEP-257 and PEP-484 style changes
- PEP 561 type hints (improves module use for type checkers e.g. *mypy*)
- CidrMap now uses separate CidrCache for "private cache data" instead of just the "data" part. CidrCache class no longer needs it's own "private data" functionality.
- Add some tests (via pytest)
- Reorganize CidrMap and simplify/improve way we do private_cache supporing multiprocess/multithreading use-case. This is now all done in CidrMap.
- Change cache file storage to pickle format as its more flexible than json Provide simple app to show contents of cache:

```
py-cidr-cache-print <cache_directory>
```

## 1.4 Documentation:

We include pre-built versions of both html and PDF documentation, including the API reference.

The PDF file is *Docs/py-cidr.pdf* and after the package is installed it will be available:

PDF Documentation.

and a browser can be used to view:

HTML Documentation.

CHAPTER

TWO

# GETTING STARTED

All git tags are signed with arch@sapience.com key which is available via WKD or download from https://www.sapience.com/tech. Add the key to your package builder gpg keyring. The key is included in the Arch package and the source= line with *?signed* at the end can be used to verify the git tag. You can also manually verify the signature

## 2.1 py-cidr module

### 2.1.1 module functions

The library provides the following tools:

**CidrMap Class**

CidrMap provides a reasonably optimized tool to cache (cidr, value) pairs. i.e. it maps a CIDR address to some value (string). These are cached to file if a cache directory is provided when instantiating the class.

Ths will create an IPv4 and an IPv6 cache file in the given directory. The code is careful about reading and writing the cache files and uses locking as well as atomic writes. For example if application starts, reads cache, updates with new items and some time later saves the cache - the module will detect if the cache changed (by another process using same cache directory) since it was read in, and merge its own changes with the changes in the cache file before writing out the updated cache. So nothing should be lost.

This was built this originally for our firewall tool, where part of the data gathering component creates maps of CIDR blocks to geolocated country codes for all CIDRs as listed by each of registries. This process can take several minutes. Run time was cut roughly in half using CidrMap() to provide a mapping of CIDR to location.

Since parallelizing can provide siginificant speedups, the CidrMap::add_cidr() method has a mechanism to allow that by avoiding multiple threads/processes updating the in memory data at the same time. It offers the ability for each thread/subprocess to add cidr blocks to thread local data. After all the threads/processes complete, then the private data maps of each of the processes can be merged together using CidrMap::merge() method.

Additional details are available in the API reference documentation.

Methods provided:

- CidrMap.lookup
- CidrMap.add_cidr
- CidrMap.merge

Static functions:

- create_private_cache

**Cidr Class**

See the API reference in the documentation for details. This class provides a suite of tools we found ourselves using often, so we encapsulated them in this class. All methods in the class are @*staticmethod* and thus no instance of the class is needed. Just use them as functions (*Cidr.xxx()*)

- Cidr.is_valid_ip4
- Cidr.is_valid_ip6
- Cidr.is_valid_cidr
- Cidr.cidr_iptype
- Cidr.cidr_type_network
- Cidr.cidr_to_net
- Cidr.cidrs_to_nets
- Cidr.nets_to_cidrs
- Cidr.compact_cidrs
- Cidr.ip_to_address
- Cidr.ips_to_addresses
- Cidr.addresses_to_ips
- Cidr.cidr_set_prefix
- Cidr.ipaddr_cidr_from_string
- Cidr.cidr_is_subnet
- Cidr.address_iptype
- Cidr.compact_nets
- Cidr.net_exclude
- Cidr.nets_exclude
- Cidr.cidrs_exclude
- Cidr.cidrs2_minus_cidrs1
- Cidr.cidr_exclude
- Cidr.sort_cidrs
- Cidr.sort_ips
- Cidr.get_host_bits
- Cidr.clean_cidr
- Cidr.clean_cidrs
- Cidr.range_to_cidrs
- Cidr.cidr_to_range
- Cidr.fix_cidr_host_bits
- Cidr.fix_cidrs_host_bits

**CidrFile Class**

This class provides a few reader/writer tools for files with lists of CIDR strings. Readers ignores comments. All methods are @*staticmethod* and thus no instance of the class is required. Simply use them as functions (*Cidr.xxx()*)

- Cidr.read_cidr_file(file:str, verb:bool=False) -> [str]:
- Cidr.read_cidr_files(targ_dir:str, file_list:[str]) -> [str]
- Cidr.write_cidr_file(cidrs:[str], pathname:str) -> bool
- Cidr.read_cidrs(fname:str|None, verb:bool=False) -> (ipv4:[str], ipv6:[str]):
- Cidr.copy_cidr_file(src_file:str, dst_file:str) -> None

CHAPTER

THREE

# APPENDIX

## 3.1 Installation

Available on * Github * Archlinux AUR

On Arch you can build using the provided PKGBUILD in the packaging directory or from the AUR. To build manually, clone the repo and :

```
rm -f dist/*
/usr/bin/python -m build --wheel --no-isolation
root_dest="/"
./scripts/do-install $root_dest
```

When running as non-root then set root_dest a user writable directory

## 3.2 Dependencies

**Run Time** :

- python (3.13 or later)
- lockmgr

**Building Package** :

- git
- hatch (aka python-hatch)
- wheel (aka python-wheel)
- build (aka python-build)
- installer (aka python-installer)
- rsync

**Optional for building docs** :

- sphinx
- python-myst-parser
- python-sphinx-autoapi
- texlive-latexextra (archlinux packaguing of texlive tools)

Building docs is not really needed since pre-built docs are provided in the git repo.

## 3.3 Philosophy

We follow the *live at head commit* philosophy as recommended by Google's Abseil team[1]. This means we recommend using the latest commit on git master branch.

## 3.4 License

Created by Gene C. and licensed under the terms of the GPL-2.0-or-later license.

- SPDX-License-Identifier: GPL-2.0-or-later
- SPDX-FileCopyrightText: © 2024-present Gene C <arch@sapience.com>

[1] https://abseil.io/about/philosophy#upgrade-support

CHAPTER

FOUR

# LICENSE

Python module that provides IP / network / CIDR tools

CHAPTER

FIVE

# API REFERENCE

This page contains auto-generated API reference documentation[1].

## 5.1 py_cidr

Public Methods. py_cidr.

### 5.1.1 Submodules

#### py_cidr.cidr_class

Class providing some common CIDR utilities

##### Module Contents

**class Cidr**

Provides suite of CIDR tools.

All mathods are (static) and are thus called without need to instantiate the class. For example:

net = Cidr.cidr_to_net(cidr_string)

**Notation:**

- cidr means a string
- net means ipaddress network (IPv4Network or IPv6Network)
- ip means an IP address string
- addr means an ip address (IPv4Address or IPv6Address)
- address means either a IP address or a cidr network as a string

**static address_iptype**(*addr: py_cidr.cidr_types.IPvxAddress* | *py_cidr.cidr_types.IPvxNetwork*) → str

Identify address or net (IPvxNetwork) as ipv4, ipv6 or neither.

**Args:**
addr (str): ipaddress IP or network .

**Returns:**
str | None: 'ip4', 'ip6' or ''

[1] Created with sphinx-autoapi

`static address_to_net`(*addr: py_cidr.cidr_types.IPAddress | py_cidr.cidr_types.IPvxNetwork*, *strict: bool = False*) → py_cidr.cidr_types.IPvxNetwork | None

Convert an address to IPvxNetwork.

Be flexible with input address. Can be IPAddress (includes string) or even IPvxNetwork.

**Args:**

**addr (IPAddress | IPvxNetwork):**
Input address.

**strict (bool):**
If true then cidr is considered invalid if host bits are set. Defaults to False. (see ipaddress docs).

**Returns:**

**IPvxNetwork | None:**
The IPvxNetwork derived from input "addr" or None if not an address/network.

`static addresses_to_ips`(*addresses: list[py_cidr.cidr_types.IPvxAddress]*) → list[str]

From list of IPs in ipaddress format, get list of ip strings.

**Args:**
addresses (list[IPvxAddress]): list of IP addresses in ipaddress format

**Returns:**
list[str]: list of IP strings

`static cidr_exclude`(*cidr1: str*, *cidrs2: list[str]*) → list[str]

Exclude cidr1 from any of networks in cidrs2.

**Args:**
cidr1 (str): cidr to be excluded.

cidrs2 (list[str]): list fo cidrs from which cidr1 will be excluded.

**Returns:**
list[str]: Resulting list of cidrs ("cidrs2" - "cidr1")

`static cidr_iptype`(*address: Any*) → str

Determines if address string is valid ipv4 or ipv6 or not.

**Args:**

**address (Any):**
address or cidr string

**Returns:**

**str :**
'ip4' or 'ip6' or empty string, '', if address invalid. Note. Earlier versions returned None instead of empty string if unable to be converted.

`static cidr_is_subnet`(*cidr: str*, *ipa_nets: list[py_cidr.cidr_types.IPvxNetwork]*) → bool

Check if cidr is a subnet of any of the list of IPvxNetworks .

**Args:**
cidr (str): Cidr string to check.

ipa_nets (list[IPvxNetwork]): list of IPvxNetworks to check.

**Returns:**
bool: True if cidr is subnet of any of the ipa_nets, else False.

`static cidr_list_compact`(*cidrs: list[str]*, *string: bool = True*) → list[str] | list[py_cidr.cidr_types.IPvxNetwork]

Compact list of cidr networks to smallest list possible. Deprecated - use compact_cidrs(cidrs, return_nets)) instead, it is the same with the boolean flag reversed.

**Args:**
cidrs (list[str]): list of cidr strings to compact.

**string (bool):**

- If True (default), then return is a list of strings.
- If False, a list of IPvxNetworks.

**Returns:**
list[str] | list[IPvxNetwork]: Compressed list of cidrs as ipaddress networks (string=False) or list of strings when string=True

`static cidr_range_split`(*cidr: str*) → tuple[str, str, str]

Convert network to IP Range.

**Args:**
net (IPvxNetwork): The network (IPvxNetwork) to examine.

string (bool): If True then returns cidr strings instead of IPvxAddress

**Returns:**
tuple[str, str, str]: that are strings of the first IP, middle IP and last IP in the cidr block

`static cidr_set_prefix`(*cidr: str*, *prefix: int*) → str

Set new prefix for cidr and return new cidr string.

**Args:**
cidr (str): Cidr string to use

prefix (int): The new prefix to use

**Returns:**
str: Cidr string using the specified prefix

`static cidr_to_net`(*cidr: str*, *strict: bool = False*) → py_cidr.cidr_types.IPvxNetwork | None

Convert cidr string to ipaddress network.

**Args:**
cidr (str): Input cidr string

strict (bool): If true then cidr is considered invalid if host bits are set. Defaults to False. (see ipaddress docs).

**Returns:**
IPvxNetwork | None: The ipaddress network derived from cidr string as IPvxNetwork = IPv4Network or IPv6Network or None if invalid.

`static cidr_to_range`(*cidr: str*, *string: bool = False*) → tuple[py_cidr.cidr_types.IPvxAddress | str | None, py_cidr.cidr_types.IPvxAddress | str | None]

Cidr string to an IP Range.

**Args:**
cidr (str): The cidr string to examine.

string (bool): If True then returns cidr strings instead of IPvxAddress

**Returns:**
tuple[IPAddress, IPAddress]: tuple (ip0, ip1) of first and last IP address in net (ip0, ip1) are IPvxAddress or str when string is True

`static cidr_type_network`(*cidr: str*) → tuple[str, type[py_cidr.cidr_types.IPvxNetwork]]

Cidr Network Type.

**Args:**
cidr (str): Cidr string to examine

**Returns:**
tuple[str, IPvxNetwork]: tuple(ip-type, net-type). ip-type is a string ('ip4', 'ip6') while network type is IPv4Network or IPv6Network

`static cidrs2_minus_cidrs1`(*cidrs1: list[str]*, *cidrs2: list[str]*) → list[str]

Exclude all of cidrs1 from cidrs2.

i.e. return "cidrs2" - "cidrs1".

**Args:**
cidrs1 (list[str]): list of cidr strings to be excluded.

cidrs2 (list[str]): list of cidr strings from which cidrs1 are excluded.

**Returns:**
list[str]: Resulting list of cidr strings = "cidrs2" - "cidrs1".

`static cidrs_exclude`(*cidrs1: list[str]*, *cidrs2: list[str]*) → list[str]

Deprecated: replaced by cidrs2_minus_cidrs1()

`static cidrs_split_type`(*cidrs: list[str]*) → tuple[list[str], list[str], list[str]]

Split a list of cidrs into ipv4, ipv6 and other.

**Args:**

**cidrs (list[str]):**
list of cidr strings

**Returns:**
tuple[ip4: list[str], ip6: list[str], other: list[str]] Tuple of lists of ipv4, ipv6 and unknown.

`static cidrs_to_nets`(*cidrs: list[str]*, *strict: bool = False*) → list[py_cidr.cidr_types.IPvxNetwork]

Convert list of cidr strings to list of IPvxNetwork.

**Args:**
cidrs (list[str]): list of cidr strings

strict (bool): If true, cidr with host bits set is invalid. Defaults to false.

**Returns:**
list[IPvxNetwork]: list of IPvxNetworks generated from cidrs.

`static clean_cidr`(*cidr: str*) → str | None

Clean up a cidr address.

**Does:**

- fix up host bits to match the prefix
- convert old class A,B,C style IPv4 addresses to cidr.

**e.g.**
a.b.c -> a.b.c.0/24 a.b.c.23/24 -> a.b.c.0/24

**Args:**
cidr (str): Cidr string to clean up.

**Returns:**

**str | None:**

- cidr string if valid
- None if cidr is invalid.

`static clean_cidrs`(*cidrs: list[str]*) → list[str]

Clean list of cidrs.

Similar to clean_cidr() but for a list.

**Args:**
cidrs (list[str]): list of cidr strings to clean up.

**Returns:**
list[str]: list of cleaned cidrs. If input cidr is invalid then its returnded as None

`static compact`(*cidrs: list[str]*) → list[str]

Compact list of cidrs - can be mixed ipv4/ipv6. Returns 1 type, making type annotation simpler for caller.

**Args:**

**cidrs (list[str]):**
Input list of cidr strings.

**Returns:**

**list[str]:**
Compacted list of cidrs. If mixed ipv4 is before ipv6

Recommended methods are compact() or compact_nets(). Others are kept for backward compatibility.

`static compact_cidrs`(*cidrs: list[str]*, *nets: bool = False*) → list[str] | list[py_cidr.cidr_types.IPvxNetwork]

Compact a list of cidr networks as strings.

**Args:**
cidrs (list[str]): list of cidrs to compact.

nets (bool): If False, the default, the result will be list of strings else a list of IPvxNetwork's.

**Returns:**
list[str | IPvxNetwork]: A list of compacted networks whose elements are strings if return_nets is False or IPvxNetworks if True.

`static compact_cidrs_to_cidrs`(*cidrs: list[str]*) → list[str]

Compact list of cidr networks and return list of cidr strings

Same as compact_cidrs(cidrs, nets=False). With single return type this is helpful when using type annotation checkers.

`static compact_nets`(*nets: list[py_cidr.cidr_types.IPvxNetwork]*) → list[py_cidr.cidr_types.IPvxNetwork]

Compact list of IPvxNetwork.

**Args:**
nets (list[IPvxNetwork]): Input list if networks to compact.

**Returns:**
list[IPvxNetwork]: Compacted list of IPvxNetworks.

**static fix_cidr_host_bits**(*cidr: str*, *verb: bool = False*) → str

zero out any host bits.

A strictly valid cidr address must have host bits set to zero.

**Args:**

cidr (str): The cidr to "fix" if needed.

verb (bool): Some info on stdout when set True. Defaults to False.

**Returns:**

str: The cidr with any non-zero host bits now zeroed out.

**static fix_cidrs_host_bits**(*cidrs: list[str]*, *verb: bool = False*) → list[str]

zero any host bits for a list of cidrs.

Similar to fix_cidr_host_bits() but for a list of cidrs.

**Args:**

cidrs (list[str]): list of cidrs to fix up.

verb (bool): Some info on stdout when set True. Defaults to False.

**Returns:**

list[str]: The list of cidrs each with any non-zero host bits now zeroed out.

**static get_host_bits**(*ip: str*, *pfx: int = 24*) → int

Gets the host bits from an IP address given the netmask.

**Args:**

ip (str): The IP to examine.

pfx (int): The cidr prefix.

**Returns:**

int: The host bits from the IP.

**static ip_to_address**(*ip: str*) → py_cidr.cidr_types.IPvxAddress | None

Return ipaddress of given ip.

If IP has prefix or host bits set, strip the prefix and keep host bits.

**Args:**

ip (str): The IP string to convert

Rreturns (IPvxAddress | None): IPvxAddress derived from IP or None if not an IP address.

**static ipaddr_cidr_from_string**(*address: str*, *strict: bool = False*) → py_cidr.cidr_types.IPvxNetwork | None

Convert string of IP address or cidr net to IPvxNetwork

**Args:**

address: IP or CIDR network as a string.

strict (bool): If true, host bits are disallowed for cidr block.

**Returns:**

IPvxNetwork | None: An IPvxNetwork or None if invalid.

**static ips_to_addresses**(*ips: list[str]*) → list[py_cidr.cidr_types.IPvxAddress]

Convert list of IP strings to a list of ip addresses

**Args:**

ips (list[str]): list of IP strings to convert

**Returns:**
list[IPvxAddress]: list of IPvxAddress derived from input IPs.

**static is_rfc_1918**(*cidr: str*) → bool

Check if cidr is any RFC 1918.

**Args:**
cidr (str): IP or Cidr to check if RFC 1918.

**Returns:**
bool: True if cidr is an RFC 1918 address. False if not.

**static is_valid_cidr**(*address: Any*) → bool

Check if address is a valid ip or cidr network.

**Args:**
address (Any): Address to check. Host bits set is permitted for a cidr network.

**Returns:**
bool: True/False if address is valid IPv4 or IPv6 address or network.

**static is_valid_ip4**(*address: Any*) → bool

check if valid IPv4 address or cidr.

**Args:**
address (Any): Check if this is a valid IPv4 address or cidr.

**Returns:**
bool: True if valid IPv4 else False

**static is_valid_ip6**(*address: Any*) → bool

check if valid IPv6 address or cidr.

**Args:**
address (Any): Check if this is a valid IPv6 address or cidr.

**Returns:**
bool: True if valid IPv6 else False

**static net_exclude**(*net1: py_cidr.cidr_types.IPvxNetwork*, *nets2: list[py_cidr.cidr_types.IPvxNetwork]*) → list[py_cidr.cidr_types.IPvxNetwork]

Exclude net1 from any of networks in net2 and return resulting list of nets (without net1).

**Args:**
net1 (IPvxNetwork): Network to be ecluded.

nets2 (list[IPvxNetwork]): list of networks from which net1 will be excluded from.

**Returns:**
list[IPvxNetwork]: Resultant list of networks "nets2 - net1".

**static net_is_subnet**(*net1: py_cidr.cidr_types.IPvxNetwork*, *net2: py_cidr.cidr_types.IPvxNetwork | list[py_cidr.cidr_types.IPvxNetwork]*) → bool

Determines if net1 is a subnet of any of net2.

**Args:**

**net1 (IPvxNetwork):**
Network to check if is a subnet.

**net2 (IPvxNetwork | list[IPvxNetwork]):**
Network or list of networks to be checked.

**Returns:**

**bool:**
True if net1 is a subnet of any of net2.

`static net_range_split`(*net: py_cidr.cidr_types.IPvxNetwork*) → tuple[py_cidr.cidr_types.IPvxAddress, py_cidr.cidr_types.IPvxAddress, py_cidr.cidr_types.IPvxAddress]

Convert network to IP Range.

**Args:**
net (IPvxNetwork): The network (IPvxNetwork) to examine.

string (bool): If True then returns cidr strings instead of IPvxAddress

**Returns:**
tuple[, IPvxAddressIPAddress, , IPvxAddressIPAddress, IPvxAddress]: that are the first IP, middle IP and last IP in the cidr block

`static net_to_cidr`(*net: py_cidr.cidr_types.IPvxNetwork*) → str

**Net to Cidr String**
Convert an ipaddress network to a cidr string.

**Args:**

**net (IPvxNetwork):**
Ipaddress Network to convert.

**Returns:**
str: Cidr string from net. If unable to conver, then empty string is returned.

`static net_to_range`(*net: py_cidr.cidr_types.IPvxNetwork*, *string: bool = False*) → tuple[py_cidr.cidr_types.IPvxAddress | str | None, py_cidr.cidr_types.IPvxAddress | str | None]

Convert network to IP Range.

**Args:**
net (IPvxNetwork): The network (IPvxNetwork) to examine.

string (bool): If True then returns cidr strings instead of IPvxAddress

**Returns:**
tuple[IPAddress, IPAddress]: tuple (ip0, ip1) of first and last IP address in net Each (ip0, ip1) is IPvx-Address or a string if "string" == True

`static nets_exclude`(*nets1: list[py_cidr.cidr_types.IPvxNetwork]*, *nets2: list[py_cidr.cidr_types.IPvxNetwork]*) → list[py_cidr.cidr_types.IPvxNetwork]

Exclude every nets1 network from from any networks in nets2.

Similar to net_exclude() except this version has a list to be excluded instead of a single network.

**Args:**
nets1 (list[IPvxNetwork]): list of nets to be excluded.

nets2: (list[IPvxNetwork]): list of nets from which will exclude any of nets1.

**Returns:**
list[IPvxNetwork]: list of resultant networks ("nets2" - "nets1")

`static nets_to_cidrs`(*nets: list[py_cidr.cidr_types.IPvxNetwork]*) → list[str]

Convert list of ipaddress networks to list of cidr strings.

**Args:**
nets (list[IPvxNetwork]): list of nets to convert.

**Returns:**
list[str]: list of cidr strings.

`static range_to_cidrs`(*addr_start: py_cidr.cidr_types.IPAddress*, *addr_end: py_cidr.cidr_types.IPAddress*, *string: bool = False*) → list[py_cidr.cidr_types.IPvxNetwork] | list[str]

Generate a list of cidr/nets from an IP range.

**Args:**
addr_start (IPAddress): Start of IP range

addr_end (IPAddress): End of IP range

string (bool): If True then returns list of cidr strings otherwise IPvxNetwork

**Returns:**
list[IPvxNetwork] | list[str] list of cidr network blocks representing the IP range. list elements are IPvxAddress or str if parameter string=True

`static remove_rfc_1918`(*cidrs_in: str | list[str]*) → tuple[str | list[str], str | list[str]]

Given list of cidrs, return list without any rfc 1918

**Args:**
cidrs_in (str | list[str]: Cidr string or list of cidr strings.

**Returns:**
tuple[str | list[str], str | list[str]]: Returns (tuple[cidrs_cleaned, rfc_1918_cidrs_found]):

- cidrs_cleaned: list of cidrs with all rfc_1918 removed.
- rfc_1918_cidrs_found: list of any rfc 1918 found in the input.

If input cidr(s) is a list, then items in output are a (possibly empty) list If not a list then returned items will be string or None.

`static rfc_1918_cidrs`() → list[str]

Return list of rfc 1918 networks cidr strings

**Returns (list[str]):**
list of RFC 1918 networks as cidr strings

`static rfc_1918_nets`() → list[py_cidr.cidr_types.IPvxNetwork]

Return list of rfc 1918 networks

**Returns:**
list[IPv4Network]: list of RFC 1918 networks.

`static sort_cidrs`(*cidrs: list[str]*) → list[str]

Sort the list of cidr strings.

**Args:**
cidrs (list[str]): list of cidrs.

**Returns:**
list[str]: Sorted copy of cidr list

`static sort_ips`(*ips: list[str]*) → list[str]

Sort a list of IP addresses.

**Args:**
ips (list[str]): list of ips to be sorted.

**Returns:**
list[str]: Sorted copy of ips.

`static sort_nets`(*nets: list[py_cidr.cidr_types.IPvxNetwork]*) → list[py_cidr.cidr_types.IPvxNetwork]

Sort a list of networks.

**Args:**

**nets (list[IPvxNetwork]):**
list of networks to be sorted.

**Returns:**

**list[IPvxNetwork]:**
Sorted copy of networks.

`static version`() → str

**Returns**
Version of py-cidr

## py_cidr.cidr_file_class

CidrFile class: Read/write a file with list of cidr blocks as strings For reading

- comments ignored
- pname = is path to the file.
- cidr are all in column 1

### Module Contents

`class CidrFile`

Provides common CIDR string file reader/writer tools.

All methods are static so no class instance variable needed.

`static copy_cidr_file`(*src_file: str*, *dst_file: str*) → bool

Copy one file to another.

**Args:**
src_file (str): Source file to copy.

dst_file (str): Where to save copy

**Returns:**
bool: True if all okay else False

`static read_cidr_file`(*fname: str*, *verb: bool = False*) → list[str]

Read file of cidrs and return list of all IPv4 and IPv6.

See read_cidrs() which this uses.

**Args:**
fname (str): Path to file of cidrs to read.

verb (bool): More verbose output

**Returns:**
list[str]: list of all cidrs (ip4 and ip6 combined)

**static read_cidr_files**(*targ_dir: str*, *file_list: list[str]*) → list[str]

Read files in a directory and return merged list of cidr strings.

**Args:**
targ_dir (str): Directory to find each file.

file_list (list[str]): list of files in *targ_dir* to read.

**Returns:**
list[str]: list of all cidrs found in the files.

**static read_cidrs**(*fname: str | None*, *verb: bool = False*) → tuple[list[str], list[str]]

Read file of cidrs and return tuple of separate lists (ip4, ip6).

- if fname is None or sys.stdin then data is read from stdin.
- only column 1 of file is used.
- comments are ignored

**Args:**
fname (str | None): File name to read.

verb (bool): More verbose output when True.

**Returns:**
tuple[list[str], list[str]]: tuple of lists of cidrs (ip4, ip6)

**static write_cidr_file**(*cidrs: list[str]*, *pname: str*) → bool

Write list of cidrs to a file.

**Args:**
cidrs (list[str]): list of cidr strings to write.

pname (str): Path to file where cidrs are to be written.

**Returns:**
bool: True if successful otherwise False.

## py_cidr.cidr_map

Map cidr/ips to a (str) value. Requires CidrCache

Keep separate caches for ipv4 and ipv6 cidr matches cache.cidr cidr when cidr is subnet of cache.cidr.

Requires CidrCache for the actual cache management

### Module Contents

**class CidrMap**(*cache_dir: str | None = None*)

Class provides map(cidr) -> some value.

- ipv4 and ipv6 are cached separately
- built on CidrCache and Cidr classes

**Args:**
cache_dir (str): Optional directory to save cache file

todo: generalize value to be any object not just string

`add_cidr`(*cidr: str*, *result: str*, *priv_cache:* NetCache | *None* = *None*)

Add cidr to cache.

**Args:**

cidr (str): Add this cidr string and its associated result value to the map.

result (str): The result value to be associated with this cidr. i.e. map(cidr) = result

priv_data (private):

If using multiple processes/threads then provide this object where changes are kept instead of in the instance cache. This way the same instance (and its cache) can be used across multiple processes/threads.

Use CidrMap.create_private_cache() to create private_data

`static create_private_cache`() → *NetCache*

Create and Return private cache object to use with add_cidr().

This cache has no cache_dir set - memory only. Required if one CidrMap instance is used in multiple processes/threads Give each process/thread a private data cache and they can be merged into the CidrMap instance after they have all completed.

**Returns:**

(private): private_cache_data object.

`items`(*v6: bool* = *False*) → Iterator[tuple[str, Any]]

Iterator that returns oen element at a time of the map. Args:

**v6 (bool):**

Default is false, and the elements are from IPv4 map If True, then elements are taken from the IPv6 map.

**Returns:**

**tuple[cidr: str, value: str]:**

The (cidr, value) for each map element

`lookup`(*cidr: str*) → Any | None

Check if cidr is in map.

**Args:**

cidr (str): Cidr value to lookup.

**Returns:**

Any | None: Result = map(cidr) if found else None.

`lookup_both`(*cidr: str*) → tuple[str, Any | None]

Check if cidr is in map. Similar to lookup() but returns the cidr_found in the map as well as it's value

**Args:**

cidr (str): Cidr value to lookup.

**Returns:**

tuple[cidr_found: str, value: Any | None] cidr_found is the "prefix" If in map, then cidr_found <= cidr and the value is that associated with cidr_found If not in map, then ['', None] is returned

`merge`(*priv_cache:* NetCache | *None*)

Merge private cache into our internal cache.

**Args:**
priv_data (_PrivCache): The “private data” to add (cidr, result) to the map, then this merges content of priv_data into the current data. priv_data must be created by CidrMap.create_private_cache()

`print()`
Print the cache data.

`save_cache()`
Write cache to files

`class NetCache(`*cache_dir: str*`)`
holds both ipv4 and ipv6 cache

`get(`*ipt: str*`)`
Returns the cache for ipv4 or ipv6

**Args:**
ipt (str): One of ‘ipv4’ or ‘ipv6’

**Returns:**
Cache for the requested network type

`get_cache(`*iptype: str*`)` → py_cidr._cidr_cache.CidrCache
Extract the cache for the provided iptype

**Args:**
ipt (str): One of ‘ipv4’ or ‘ipv6’

**Returns:**
CidrCache: Cache for “iptype”

`set(`*ipt: str*, *cache: py_cidr._cidr_cache.CidrCache*`)`
Assigns the cache for ipv4 or ipv6

**Args:**
ipt (str): One of ‘ipv4’ or ‘ipv6’

**Returns:**
Cache for the requested network type

### py_cidr.cidr_types

Convenience types:

Provides types IPvxNetwork, IPvxAddress and IPAddress which are based on ipaddress types:

IPv4Network, IPv6Network, IPv4Address and IPv6Address.

#### Module Contents

**`type IPAddress = IPvxAddress | str`**

**`type IPv4 = IPv4Address | IPv4Network`**

**`type IPv6 = IPv6Address | IPv6Network`**

**`type IPvxAddress = IPv4Address | IPv6Address`**

**`type IPvxNetwork = IPv4Network | IPv6Network`**

# PYTHON MODULE INDEX

# INDEX