dnspython

Dnspython is a DNS toolkit for Python. It can be used for queries, zone transfers, dynamic updates, nameserver testing, and many other things.

Dnspython provides both high and low level access to the DNS. The high level classes perform queries for data of a given name, type, and class, and return an answer set. The low level classes allow direct manipulation of DNS zones, messages, names, and records. Almost all RR types are supported.

dnspython originated at Nominum where it was developed for testing DNS nameservers.

Contents:

Indices and tables

What's New in dnspython

2.6.1

- The Tudoor fix ate legitimate Truncated exceptions, preventing the resolver from failing over to TCP and causing the query to timeout [#1053].

2.6.0

- As mentioned in the "TuDoor" paper and the associated CVE-2023-29483, the dnspython stub resolver is vulnerable to a potential DoS if a bad-in-some-way response from the right address and port forged by an attacker arrives before a legitimate one on the UDP port dnspython is using for that query.

  This release addresses the issue by adopting the recommended mitigation, which is ignoring the bad packets and continuing to listen for a legitimate response until the timeout for the query has expired.

- Added support for the NSID EDNS option.
- Dnspython now looks for version metadata for optional packages and will not use them if they are too old. This prevents possible exceptions when a feature like DoH is not desired in dnspython, but an old httpx is installed along with dnspython for some other purpose.
- The DoHNameserver class now allows GET to be used instead of the default POST, and also passes source and source_port correctly to the underlying query methods.

2.5.0

- Dnspython now uses hatchling for builds.
- Asynchronous destinationless sockets now work on Windows.
- Cython is no longer supported due to various typing issues.
- Dnspython now explicitly canonicalizes IPv4 and IPv6 addresses. Previously it was possible for non-canonical IPv6 forms to be stored in a AAAA address, which would work correctly but possibly cause problmes if the address were used as a key in a dictionary.
- The number of messages in a section can be retrieved with section_count().
- Truncation preferences for messages can be specified.
- The length of a message can be automatically prepended when rendering.
- dns.message.create_response() automatically adds padding when required by RFC 8467.
- The TLS verify parameter is now supported by dns.query.tls(), and the DoH and DoT Nameserver subclasses.

- The MutableMapping used to store content in a zone may now be specified by a factory when subclassing. Factories may also be provided for writable verisons and immutable versions.
- dns.name.Name now has predecessor() and successor() methods implementing RFC 4471.
- QUIC has had a number of bug fixes and also now supports session tickets for faster session resumption.
- The NSEC3 class now has a next_name() method for retrieving the next name as a dns.name.Name.
- Windows WMI interface detection should be more robust.

2.4.2

- Async queries could wait forever instead of respecting the timeout if the timeout was 0 and a packet was lost. The timeout is now respected.
- Restore HTTP/2 support which was accidentally broken during the https refactoring done as part of 2.4.0.
- When an inception time and lifetime are specified, the signer now sets the expiration to the inception time plus lifetime, instead of the current time plus the lifetime.

2.4.1

- Importing dns.dnssecalgs without the cryptography module installed no longer causes an ImportError.
- A number of timeout bugs with the asyncio backend have been fixed.
- DNS-over-QUIC for the asyncio backend now works for IPv6.
- Dnspython now enforces that the candidate DNSKEYs for DNSSEC signatures have protocol 3 and have the ZONE flag set. This is a standards compliance issue more than a security issue as the legitimate authority would have to have published the non-compliant keys as well as updated their DS record in order for the records to validate (the DS digest includes both flags and protocol). Dnspython will not make invalid keys by default, but does allow them to be created and used for testing purposes.
- Dependency specifications for optional features in the package metadata have been improved.

2.4.0

- Python 3.8 or newer is required.
- The stub resolver now uses instances of `dns.nameserver.Nameserver` to represent remote recursive resolvers, and can communicate using DNS over UDP/TCP, HTTPS, TLS, and QUIC. In additional to being able to specify an IPv4, IPv6, or HTTPS URL as a nameserver, instances of `dns.nameserver.Nameserver` are now permitted.
- The DNS-over-HTTPS bootstrap address no longer causes URL rewriting.
- DNS-over-HTTPS now only uses httpx; support for requests has been dropped. A source port may now be supplied when using httpx.

- DNSSEC zone signing with NSEC records is now supported. Thank you very much (again!) Jakob Schlyter!
- The resolver and async resolver now have the `try_ddr()` method, which will try to use Discovery of Designated Resolvers (DDR) to upgrade the connection from the stub resolver to the recursive server so that it uses DNS-over-HTTPS, DNS-over-TLS, or DNS-over-QUIC. This feature is currently experimental as the standard is still in draft stage.
- The resolver and async resolver now have the `make_resolver_at()` and `resolve_at()` functions, as a convenience for making queries to specific recursive servers.
- Curio support has been removed.

2.3.0

- Python 3.7 or newer is required.
- Type annotations are now integrated with the source code and cover far more of the library.
- The get_soa() method has been added to dns.zone.Zone.
- The minimum TLS version is now 1.2.
- EDNS padding is now supported. Messages with EDNS enabled and with a non-zero pad option will be automatically padded appropriately when converted to wire format.
- `dns.zone.from_text()` and `dns.zone.from_file()` now have an `allow_directives` parameter to allow finer control over how directives in zonefiles are processed.
- A preliminary implementation of DNS-over-QUIC has been added, and will be available if the aioquic library is present. See `dns.query.quic()`, `dns.asyncquery.quic()`, and examples/doq.py for more info. This API is subject to change in future releases. For asynchronous I/O, both asyncio and Trio are supported, but Curio is not.
- DNSSEC signing support has been added to the `dns.dnssec` module, along with a number of functions to help generate DS, CDS, and CDNSKEY RRsets. Thank you very much Jakob Schlyter!
- Curio asynchronous I/O support is deprecated as of this release and will be removed in a future release.
- The resolver object's `nameserver` field is planned to become a property in dnspython 2.4. Writing to this field other than by direct assignment is deprecated, and so is depending on the mutability and form of the iterable returned when it is read.

2.2.1

This release has no new features, but fixes the following issues:

- dns.zone.from_text failed if relativize was False and an origin was specified in the parameters.
- A number of types permitted an empty "rest of the rdata".
- L32, L64, LP, and NID were missing from dns/rdtypes/ANY/__init__.py
- The type definition for dns.resolver.resolve_address() was incorrect.
- dns/win32util.py erroneously had the executable bit set.

- The type definition for a number of asynchronous query routines was missing the default of None for the backend parameter.
- dns/tsigkeyring.py didn't import dns.tsig.
- A number of rdata types that have a "rest of the line" behavior for the last field of the rdata erroneously permitted an empty string.
- Timeout intervals are no longer reported with absurd precision in exception text.

2.2.0

- SVCB and HTTPS records have been updated to track the evolving draft standard.
- The ZONEMD type has been added.
- The resolver now returns a LifetimeTimeout exception which includes an error trace like the NoNameservers exception. This class is a subclass of dns.exception.Timeout for backwards compatibility.
- DNS-over-HTTPS will try to use HTTP/2 if the httpx and h2 packages are installed.
- DNS-over-HTTPS is now supported for asynchronous queries and resolutions.
- `dns.zonefile.read_rrsets()` has been added, which allows rrsets in zonefile format, or a restrition of it, to be read. This function is useful for applications that want to read DNS data in text format, but do not want to use a Zone.
- On Windows systems, if the WMI module is available, the resolver will retrieve the nameserver from WMI instead of trying to figure it out by reading the registry. This may lead to more accurate results in some cases.
- The CERT rdatatype now supports certificate types IPKIX, ISPKI, IPGP, ACPKIX, and IACPKIX.
- The CDS rdatatype now allows digest type 0.
- Dnspython zones now enforces that a node is either a CNAME node or an "other data" node. A CNAME node contains only CNAME, RRSIG(CNAME), NSEC, RRSIG(NSEC), NSEC3, or RRSIG(NSEC3) rdatasets. An "other data" node contains any rdataset other than a CNAME or RRSIG(CNAME) rdataset. The enforcement is "last update wins". For example, if you have a node which contains a CNAME rdataset, and then add an MX rdataset to it, then the CNAME rdataset will be deleted. Likewise if you have a node containing an MX rdataset and add a CNAME rdataset, the MX rdataset will be deleted.
- Extended DNS Errors, as specified in RFC 8914, are now supported.

2.1.0

- End-of-line comments are now associated with rdata when read from text. For backwards compatibility with prior versions of dnspython, they are only emitted in to_text() when requested.
- Synchronous I/O is a bit more efficient, as we now try the I/O and only use poll() or select() if the I/O would block.
- The resolver cache classes now offer basic hit and miss statistics, and the LRUCache can also provide hits for every cache key.
- The resolver has a canonical_name() method.
- There is now a registration mechanism for EDNS option types.

- The default EDNS payload size has changed from 1280 to 1232.
- The SVCB, HTTPS, and SMIMEA RR types are now supported.
- TSIG has been enhanced with TKEY and GSS-TSIG support. Thanks to Nick Hall for writing this.
- Zones now can be updated via transactions.
- A new zone subclass, dns.versioned.Zone is available which has a thread-safe transaction implementation and support for keeping many versions of a zone.
- The zone file reading code has been adapted to use transactions, and is now a public API.
- Inbound zone transfer support has been rewritten and is available as dns.query.inbound_xfr() and dns.asyncquery.inbound_xfr(). It uses the transaction mechanism, and fully supports IXFR and AXFR.

2.0.0

- Python 3.6 or newer is required.
- The license is now the ISC license.
- Rdata is now immutable. Use `dns.rdata.Rdata.replace()` to make a new Rdata based on an existing one.
- dns.resolver.resolve() has been added, allowing control of whether search lists are used. dns.resolver.query() is retained for backwards compatibility, but deprecated. The default for search list behavior can be set at in the resolver object with the `use_search_by_default` parameter. The default is False.
- DNS-over-TLS is supported with `dns.query.tls()`.
- DNS-over-HTTPS is supported with `dns.query.https()`, and the resolver will use DNS-over-HTTPS for a nameserver which is an HTTPS URL.
- Basic query and resolver support for the Trio, Curio, and asyncio asynchronous I/O libraries has been added in `dns.asyncquery` and `dns.asyncresolver`. This API should be viewed as experimental as asynchronous I/O support in dnspython is still evolving.
- TSIG now defaults to using SHA-256.
- Basic type info has been added to some functions. Future releases will have comprehensive type info.
- from_text() functions now have a `relativize_to` parameter.
- python-cryptography is now used for DNSSEC.
- Ed25519 and Ed448 signatures are now supported.
- A helper for NSEC3 generating hashes has been added.
- SHA384 DS records are supported.
- Rdatasets and RRsets are much faster.
- dns.resolver.resolve_address() has been added, allowing easy address-to-name lookups.
- dns.reversename functions now allow an alternate origin to be specified.
- The `repr` form of Rdatasets and RRsets now includes the rdata.
- A number of standard resolv.conf options are now parsed.
- The nameserver and port used to get a response are now part of the resolver's `Answer` object.
- The NINFO record is supported.
- The `dns.hash` module has been removed; just use Python's native `hashlib` module.

- Rounding is done in the standard python 3 fashion; dnspython 1.x rounded in the python 2 style on both python 2 and 3.
- The resolver will now do negative caching if a cache has been configured.
- TSIG and OPT now have rdata types.
- The class for query messages is now QueryMessage. Class Message is now a base class, and is also used for messages for which we don't have a better class. Update messages are now class UpdateMessage, though class Update is retained for compatibility.
- Support for Windows 95, 98, and ME has been removed.

Community

Bugs and Feature Requests

Bugs and feature requests can be made using the github issues system at github.

Mailing Lists

dnspython-announce
dnspython-users
dnspython-dev

Installation

Requirements

Python 3.7 or newer.

Installation

Many free operating system distributions have dnspython packaged for you, so you should check there first.

The next easiest option is to use `pip`:

```
pip install dnspython
```

If `pip` is not available, you can download the latest zip file from PyPI, unzip it.

On a UNIX-like system, you then run:

```
sudo python setup.py install
```

while on a Windows system you would run:

```
python setup.py install
```

Finally, you have the option of cloning the dnspython source from github and building it:

```
git clone https://github.com/rthalley/dnspython.git
```

And then run `setup.py` as above.

Please be aware that the master branch of dnspython on github is under active development and may not always be stable.

Optional Modules

The following modules are optional, but recommended for full functionality.

If `httpx` is installed, then DNS-over-HTTPS will be available.

If `cryptography` is installed, then dnspython will be able to do low-level DNSSEC signature generation and validation.

If `idna` is installed, then IDNA 2008 will be available.

If `aioquic` is installed, the DNS-over-QUIC will be available.

Dnspython Manual

Contents:

DNS RFC Reference

The DNS is defined by a large number of RFCs, many of which have been extensively updated or obsoleted. This chapter aims to provide a roadmap and reference for this confusing space. The chapter does not aim to be encyclopedically complete, however, as the key information would then be lost in the noise. The curious are encouraged to click on the "Updated by" links on the IETF pages to see the finer points, or the "Obsoletes" links to go spelunking into the history of the DNS.

DNSSEC gets its own section instead of being included in the "Core" list because there are many DNSSEC related RFCs and it's helpful to group them together. It's not a statement that DNSSEC isn't part of the "Core" of the DNS.

The IANA DNS Parameters registry is the official reference site for all DNS constants.

Core RFCs

RFC 1034

Introduction to the DNS and description of basic behavior.

RFC 1035

The core DNS wire protocol and master file format.

RFC 1995

Incremental zone transfer (IXFR).

RFC 1996

The NOTIFY protocol.

RFC 2181

Clarifications to the specification.

RFC 2308

Negative Caching.

RFC 2845

Transaction Signatures (TSIG)

RFC 3007

Dynamic Updates

RFC 3645

GSS-TSIG.

RFC 5936

Zone transfers (AXFR).

RFC 6891

EDNS (version 0)

RFC 7830

The EDNS(0) Padding Option

RFC 8020

Clarification on the meaning of NXDOMAIN.

RFC 8467

Padding Policies for Extension Mechanisms for DNS (EDNS(0))

RFC 8914

Extended DNS Errors

DNSSEC RFCs

RFC 4033

Introduction and requirements.

RFC 4034

Resource records.

RFC 4035

Protocol.

RFC 4470

Minimally covering NSEC records and On-line Signing.

RFC 4471

Derivation of DNS Name Predecessor and Successor.

RFC 5155

DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. [NSEC3]

RFC 5702

Use of SHA-2 Algorithms with RSA in DNSKEY and RRSIG Resource Records for DNSSEC.

RFC 6605

Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC.

RFC 6781

Operational Practices, Version 2.

RFC 6840

Clarifications and Implementation Notes.

RFC 7583

Key Rollover Timing Considerations.

RFC 8080

Edwards-Curve Digital Security Algorithm (EdDSA) for DNSSEC.

RFC 8624

Algorithm Implementation Requirements and Usage Guidance for DNSSEC.

RFC 9157

Revised IANA Considerations for DNSSEC.

Misc RFCs

RFC 1101

Reverse mapping name form for IPv4.

RFC 1982

Serial number arithmetic.

RFC 4343

Case-sensitivity clarification.

RFC 8499

DNS Terminology.

Additional Transport RFCs

RFC 7858

Specification for DNS over Transport Layer Security (TLS).

RFC 8484

DNS Queries over HTTPS (DoH).

RFC 9250

DNS over Dedicated QUIC Connections.

RFCs for RR types

There are many more RR types than are listed here; if a type is not listed it means it is obsolete, deprecated, or rare “in the wild”. Some types that are currently rare are listed because they may well be more heavily used in the not-to-distant future. See the IANA DNS Parameters registry for a complete list.

A

RFC 1035

AAAA

RFC 3596

CAA

RFC 8659

CDNSKEY

RFC 7344

CDS

RFC 7344

CNAME

RFC 1035

CSYNC

RFC 7477

DNAME

RFC 6672

DNSKEY

RFC 4034

DS

RFC 4034

HTTPS

RFC 9460

LOC

RFC 1876

MX

RFC 1035

NAPTR

RFC 3403

NS

RFC 1035

NSEC

RFC 4034

NSEC3

RFC 5155

NSEC3PARAM

RFC 5155

OPENPGPKEY

RFC 7929

PTR

RFC 1035

RRSIG

RFC 4034

SMIMEA

RFC 8162

SOA

RFC 1035

SPF

RFC 7208

SRV

RFC 2782

SSHFP

RFC 4255

SVCB

RFC 9460

TLSA

RFC 6698

TXT

RFC 1035

ZONEMD

Dnspython License

ISC License

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Python Module Index

d

d

- dns
  - dns.asyncquery
  - dns.asyncresolver
  - dns.dnssec
  - dns.exception
  - dns.inet
  - dns.ipv4
  - dns.ipv6
  - dns.message
  - dns.name
  - dns.query
  - dns.rdataclass
  - dns.rdatatype
  - dns.resolver
  - dns.set
  - dns.version
  - dns.zone

dnspython

Dnspython is a DNS toolkit for Python. It can be used for queries, zone transfers, dynamic updates, nameserver testing, and many other things.

Dnspython provides both high and low level access to the DNS. The high level classes perform queries for data of a given name, type, and class, and return an answer set. The low level classes allow direct manipulation of DNS zones, messages, names, and records. Almost all RR types are supported.

dnspython originated at Nominum where it was developed for testing DNS nameservers.

Contents:

Indices and tables

DNS Names

Objects of the dns.name.Name class represent an immutable domain name. The representation is a tuple of labels, with each label being a `bytes` object in the DNS wire format. Typically names are not created by supplying the labels tuple directly, but rather by converting from DNS text format or the DNS wire format.

Labels are in the same order as in the DNS textual form, e.g. the labels value for `www.dnspython.org.` is `(b'www', b'dnspython', b'org', b'')`.

Names may be *absolute* or *relative*. Absolute names end in the root label, which is an empty `bytes`. Relative names do not end in the root label. To convert a relative name to an absolute name requires specifying an *origin*. Typically the origin is known by context. Dnspython provides tools to relativize and derelativize names. It's a good idea not to mix relative and absolute names, other than in the context of a zone. Names encoded in the DNS wire protocol are always absolute. Dnspython's functions to make names from text also default to an origin of the root name, and thus to make a relative name using them you must specify an origin of None or `dns.name.empty`.

Names are compared and ordered according to the rules of the DNS. The order is the DNSSEC canonical ordering. Relative names always sort before absolute names.

Names may also be compared according to the DNS tree hierarchy with the **`dns.name.Name.fullcompare()`** method. For example `www.dnspython.org.` is a subdomain of `dnspython.org.`. See the method description for full details.

        - `Name.is_wild()`
        - `Name.parent()`
        - `Name.predecessor()`
        - `Name.relativize()`
        - `Name.split()`
        - `Name.successor()`
        - `Name.to_digestable()`
        - `Name.to_text()`
        - `Name.to_unicode()`
        - `Name.to_wire()`
    - `dns.name.root`
    - `dns.name.empty`
    - `NameRelation`
        - `NameRelation.COMMONANCESTOR`
        - `NameRelation.EQUAL`
        - `NameRelation.NONE`
        - `NameRelation.SUBDOMAIN`
        - `NameRelation.SUPERDOMAIN`
- Making DNS Names
    - `from_text()`
    - `from_unicode()`
    - `from_wire_parser()`
    - `from_wire()`
- Name Dictionary
    - `NameDict`
        - `NameDict.get_deepest_match()`
        - `NameDict.max_depth`
        - `NameDict.max_depth_items`
- Name Helpers
    - `from_address()`
    - `to_address()`
    - `from_e164()`
    - `to_e164()`
- International Domain Name CODECs
    - `IDNACodec`
    - `IDNA2003Codec`
        - `IDNA2003Codec.decode()`

        - `IDNA2003Codec.encode()`
    - `IDNA2008Codec`
    - `dns.name.IDNA_2003_Practical`
    - `dns.name.IDNA_2003_Strict`
    - `dns.name.IDNA_2003`
    - `dns.name.IDNA_2008_Practical`
    - `dns.name.IDNA_2008_Strict`
    - `dns.name.IDNA_2008_UTS_46`
    - `dns.name.IDNA_2008_Transitional`
    - `dns.name.IDNA_2008`

DNS Rdata

An Rdata is typed data in one of the known DNS datatypes, for example type `A`, the IPv4 address of a host, or type `MX`, how to route mail. Unlike the DNS RFC concept of RR, an Rdata is not bound to an owner name. Rdata is immutable.

Rdata of the same type can be grouped into an unnamed set, an Rdataset, or into a named set, an RRset.

- `dns.rdatatype.AFSDB`
- `dns.rdatatype.AMTRELAY`
- `dns.rdatatype.ANY`
- `dns.rdatatype.APL`
- `dns.rdatatype.AVC`
- `dns.rdatatype.AXFR`
- `dns.rdatatype.CAA`
- `dns.rdatatype.CDNSKEY`
- `dns.rdatatype.CDS`
- `dns.rdatatype.CERT`
- `dns.rdatatype.CNAME`
- `dns.rdatatype.CSYNC`
- `dns.rdatatype.DHCID`
- `dns.rdatatype.DLV`
- `dns.rdatatype.DNAME`
- `dns.rdatatype.DNSKEY`
- `dns.rdatatype.DS`
- `dns.rdatatype.EUI48`
- `dns.rdatatype.EUI64`
- `dns.rdatatype.GPOS`
- `dns.rdatatype.HINFO`
- `dns.rdatatype.HIP`
- `dns.rdatatype.HTTPS`
- `dns.rdatatype.IPSECKEY`
- `dns.rdatatype.ISDN`
- `dns.rdatatype.IXFR`
- `dns.rdatatype.KEY`
- `dns.rdatatype.KX`
- `dns.rdatatype.L32`
- `dns.rdatatype.L64`
- `dns.rdatatype.LOC`
- `dns.rdatatype.LP`
- `dns.rdatatype.MAILA`
- `dns.rdatatype.MAILB`
- `dns.rdatatype.MB`
- `dns.rdatatype.MD`
- `dns.rdatatype.MF`
- `dns.rdatatype.MG`
- `dns.rdatatype.MINFO`
- `dns.rdatatype.MR`
- `dns.rdatatype.MX`

        - `dns.rdatatype.NAPTR`
        - `dns.rdatatype.NID`
        - `dns.rdatatype.NINFO`
        - `dns.rdatatype.NS`
        - `dns.rdatatype.NSAP`
        - `dns.rdatatype.NSAP_PTR`
        - `dns.rdatatype.NSEC`
        - `dns.rdatatype.NSEC3`
        - `dns.rdatatype.NSEC3PARAM`
        - `dns.rdatatype.NULL`
        - `dns.rdatatype.NXT`
        - `dns.rdatatype.OPENPGPKEY`
        - `dns.rdatatype.OPT`
        - `dns.rdatatype.PTR`
        - `dns.rdatatype.PX`
        - `dns.rdatatype.RP`
        - `dns.rdatatype.RRSIG`
        - `dns.rdatatype.RT`
        - `dns.rdatatype.SIG`
        - `dns.rdatatype.SMIMEA`
        - `dns.rdatatype.SOA`
        - `dns.rdatatype.SPF`
        - `dns.rdatatype.SRV`
        - `dns.rdatatype.SSHFP`
        - `dns.rdatatype.SVCB`
        - `dns.rdatatype.TA`
        - `dns.rdatatype.TKEY`
        - `dns.rdatatype.TLSA`
        - `dns.rdatatype.TSIG`
        - `dns.rdatatype.TXT`
        - `dns.rdatatype.TYPE0`
        - `dns.rdatatype.UNSPEC`
        - `dns.rdatatype.URI`
        - `dns.rdatatype.WKS`
        - `dns.rdatatype.X25`
        - `dns.rdatatype.ZONEMD`
- DNS Rdata Base Class
    - `Rdata`
        - `Rdata.covers()`
        - `Rdata.extended_rdatatype()`
        - `Rdata.replace()`

    - `CDS.algorithm`
    - `CDS.digest_type`
    - `CDS.digest`
- `CERT`
    - `CERT.certificate_type`
    - `CERT.key_tag`
    - `CERT.algorithm`
    - `CERT.certificate`
    - `CERT.to_text()`
- `CNAME`
    - `CNAME.target`
- `CSYNC`
    - `CSYNC.serial`
    - `CSYNC.flags`
    - `CSYNC.windows`
    - `CSYNC.to_text()`
- `DLV`
    - `DLV.key_tag`
    - `DLV.algorithm`
    - `DLV.digest_type`
    - `DLV.digest`
- `DNAME`
    - `DNAME.target`
- `DNSKEY`
    - `DNSKEY.flags`
    - `DNSKEY.protocol`
    - `DNSKEY.key`
- `DS`
    - `DS.key_tag`
    - `DS.algorithm`
    - `DS.digest_type`
    - `DS.digest`
- `EUI48`
    - `EUI48.eui`

- `EUI64`
  - `EUI64.eui`
- `GPOS`
  - `GPOS.latitude`
  - `GPOS.longitude`
  - `GPOS.altitude`
  - `GPOS.float_altitude`
  - `GPOS.float_latitude`
  - `GPOS.float_longitude`
  - `GPOS.to_text()`
- `HINFO`
  - `HINFO.cpu`
  - `HINFO.os`
  - `HINFO.to_text()`
- `HIP`
  - `HIP.hit`
  - `HIP.algorithm`
  - `HIP.key`
  - `HIP.servers`
  - `HIP.to_text()`
- `ISDN`
  - `ISDN.address`
  - `ISDN.subaddress`
  - `ISDN.to_text()`
- `L32`
  - `L32.preference`
  - `L32.locator32`
  - `L32.to_text()`
- `L64`
  - `L64.preference`
  - `L64.locator64`
  - `L64.to_text()`
- `LOC`
  - `LOC.latitude`

    - `LOC.longitude`
    - `LOC.altitude`
    - `LOC.size`
    - `LOC.horizontal_precision`
    - `LOC.vertical_precision`
    - `LOC.float_latitude`
    - `LOC.float_longitude`
    - `LOC.to_text()`
- `LP`
    - `LP.preference`
    - `LP.fqdn`
    - `LP.to_text()`
- `MX`
    - `MX.preference`
    - `MX.exchange`
- `NID`
    - `NID.preference`
    - `NID.nodeid`
    - `NID.to_text()`
- `NINFO`
    - `NINFO.strings`
- `NS`
    - `NS.target`
- `NSEC`
    - `NSEC.next`
    - `NSEC.windows`
    - `NSEC.to_text()`
- `NSEC3`
    - `NSEC3.salt`
    - `NSEC3.next`
    - `NSEC3.windows`
    - `NSEC3.to_text()`
- `NSEC3PARAM`
    - `NSEC3PARAM.salt`

    - `NSEC3PARAM.to_text()`
- `OPENPGPKEY`
    - `OPENPGPKEY.key`
    - `OPENPGPKEY.to_text()`
- `PTR`
    - `PTR.target`
- `RP`
    - `RP.mbox`
    - `RP.txt`
    - `RP.to_text()`
- `RRSIG`
    - `RRSIG.type_covered`
    - `RRSIG.algorithm`
    - `RRSIG.labels`
    - `RRSIG.original_ttl`
    - `RRSIG.expiration`
    - `RRSIG.inception`
    - `RRSIG.key_tag`
    - `RRSIG.signer`
    - `RRSIG.signature`
    - `RRSIG.covers()`
    - `RRSIG.to_text()`
- `RT`
    - `RT.preference`
    - `RT.exchange`
- `SMIMEA`
    - `SMIMEA.usage`
    - `SMIMEA.selector`
    - `SMIMEA.mtype`
    - `SMIMEA.cert`
- `SOA`
    - `SOA.mname`
    - `SOA.rname`
    - `SOA.serial`
    - `SOA.refresh`

        - `SOA.retry`
        - `SOA.expire`
        - `SOA.minimum`
        - `SOA.to_text()`
    - `SPF`
        - `SPF.strings`
    - `SSHFP`
        - `SSHFP.algorithm`
        - `SSHFP.fp_type`
        - `SSHFP.fingerprint`
        - `SSHFP.to_text()`
    - `TLSA`
        - `TLSA.usage`
        - `TLSA.selector`
        - `TLSA.mtype`
        - `TLSA.cert`
    - `TXT`
        - `TXT.strings`
    - `URI`
        - `URI.priority`
        - `URI.weight`
        - `URI.target`
        - `URI.to_text()`
    - `X25`
        - `X25.address`
        - `X25.to_text()`
- Types specific to class IN
    - `A`
        - `A.address`
        - `A.to_text()`
    - `AAAA`
        - `AAAA.address`
        - `AAAA.to_text()`
    - `APLItem`

    - `APLItem.family`
    - `APLItem.negation`
    - `APLItem.address`
    - `APLItem.prefix`
- `APL`
    - `APL.items`
    - `APL.to_text()`
- `DHCID`
    - `DHCID.data`
    - `DHCID.to_text()`
- `HTTPS`
    - `HTTPS.priority`
    - `HTTPS.target`
    - `HTTPS.params`
- `IPSECKEY`
    - `IPSECKEY.precedence`
    - `IPSECKEY.prefix`
    - `IPSECKEY.gateway_type`
    - `IPSECKEY.algorithm`
    - `IPSECKEY.gateway`
    - `IPSECKEY.key`
    - `IPSECKEY.to_text()`
- `KX`
    - `KX.preference`
    - `KX.exchange`
- `NAPTR`
    - `NAPTR.order`
    - `NAPTR.preference`
    - `NAPTR.flags`
    - `NAPTR.service`
    - `NAPTR.regexp`
    - `NAPTR.replacement`
    - `NAPTR.to_text()`
- `NSAP`
    - `NSAP.address`

        - `NSAP.to_text()`
    - `NSAP_PTR`
        - `NSAP_PTR.target`
    - `PX`
        - `PX.preference`
        - `PX.map822`
        - `PX.mapx400`
        - `PX.to_text()`
    - `SRV`
        - `SRV.priority`
        - `SRV.weight`
        - `SRV.port`
        - `SRV.target`
        - `SRV.to_text()`
    - `SVCB`
        - `SVCB.priority`
        - `SVCB.target`
        - `SVCB.params`
    - `WKS`
        - `WKS.address`
        - `WKS.protocol`
        - `WKS.bitmap`
        - `WKS.to_text()`
- SVCB and HTTPS Parameter Classes
    - `ParamKey`
        - `ParamKey.ALPN`
        - `ParamKey.ECH`
        - `ParamKey.IPV4HINT`
        - `ParamKey.IPV6HINT`
        - `ParamKey.MANDATORY`
        - `ParamKey.NO_DEFAULT_ALPN`
        - `ParamKey.PORT`
    - `Param`
    - `GenericParam`
        - `GenericParam.value`

        - `MandatoryParam`
            - `MandatoryParam.keys`
        - `ALPNParam`
            - `ALPNParam.ids`
        - `PortParam`
            - `PortParam.port`
        - `IPv4HintParam`
            - `IPv4HintParam.addresses`
        - `IPv6HintParam`
            - `IPv6HintParam.addresses`
        - `ECHParam`
            - `ECHParam.ech`
- Rdataset, RRset and Node Classes
    - `Rdataset`
        - `Rdataset.add()`
        - `Rdataset.intersection_update()`
        - `Rdataset.match()`
        - `Rdataset.processing_order()`
        - `Rdataset.to_text()`
        - `Rdataset.to_wire()`
        - `Rdataset.union_update()`
        - `Rdataset.update()`
        - `Rdataset.update_ttl()`
    - `RRset`
        - `RRset.full_match()`
        - `RRset.match()`
        - `RRset.to_rdataset()`
        - `RRset.to_text()`
        - `RRset.to_wire()`
    - `Node`
        - `Node.classify()`
        - `Node.delete_rdataset()`
        - `Node.find_rdataset()`
        - `Node.get_rdataset()`

DNS Messages

Objects of the dns.message.Message class and its subclasses represent a single DNS message, as defined by RFC 1035 and its many updates and extensions.

The module provides tools for constructing and manipulating messages. TSIG signatures and EDNS are also supported. Messages can be dumped to a textual form, and also read from that form.

Dnspython has also GSS-TSIG support, but the current API is low-level. See this discussion for the details.

        - `to_text()`
    - EDNS Flags
        - `DO`
        - `edns_from_text()`
        - `edns_to_text()`
- Message Opcodes
    - `QUERY`
    - `IQUERY`
    - `STATUS`
    - `NOTIFY`
    - `UPDATE`
    - `from_text()`
    - `to_text()`
    - `from_flags()`
    - `to_flags()`
    - `is_update()`
- Message Rcodes
    - `NOERROR`
    - `FORMERR`
    - `SERVFAIL`
    - `NXDOMAIN`
    - `NOTIMP`
    - `REFUSED`
    - `YXDOMAIN`
    - `YXRRSET`
    - `NXRRSET`
    - `NOTAUTH`
    - `NOTZONE`
    - `BADVERS`
    - `from_text()`
    - `to_text()`
    - `from_flags()`
    - `to_flags()`
- Message EDNS Options
    - `NSID`
    - `DAU`
    - `DHU`
    - `N3U`
    - `ECS`

- `ZONE`
- `PREREQ`
- `UPDATE`
- `ADDITIONAL`

DNS Query Support

The `dns.query` module is for sending messages to DNS servers, and processing their responses. If you want "stub resolver" behavior, then you should use the higher level `dns.resolver` module; see Stub Resolver.

For UDP and TCP, the module provides a single "do everything" query function, and also provides the send and receive halves of this function individually for situations where more sophisticated I/O handling is being used by the application.

UDP

dns.query.udp(*q: Message, where: str, timeout: float | None = None, port: int = 53, source: str | None = None, source_port: int = 0, ignore_unexpected: bool = False, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, raise_on_truncation: bool = False, sock: Any | None = None, ignore_errors: bool = False*)→ Message [source]

Return the response obtained after sending a query via UDP.

q, a `dns.message.Message`, the query to send

where, a `str` containing an IPv4 or IPv6 address, where to send the message.

timeout, a `float` or `None`, the number of seconds to wait before the query times out. If `None`, the default, wait forever.

port, an `int`, the port send the message to. The default is 53.

source, a `str` containing an IPv4 or IPv6 address, specifying the source address. The default is the wildcard address.

source_port, an `int`, the port from which to send the message. The default is 0.

ignore_unexpected, a `bool`. If `True`, ignore responses from unexpected sources.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the received message.

raise_on_truncation, a `bool`. If `True`, raise an exception if the TC bit is set.

sock, a `socket.socket`, or `None`, the socket to use for the query. If `None`, the default, a socket is created. Note that if a socket is provided, it must be a nonblocking datagram socket, and the *source* and *source_port* are ignored.

ignore_errors, a `bool`. If various format errors or response mismatches occur, ignore them and keep listening for a valid response. The default is `False`.

Returns a `dns.message.Message`.

`dns.query.udp_with_fallback`(*q: Message, where: str, timeout: float | None = None, port: int = 53, source: str | None = None, source_port: int = 0, ignore_unexpected: bool = False, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, udp_sock: Any | None = None, tcp_sock: Any | None = None, ignore_errors: bool = False*) → Tuple[Message, bool] [source]

Return the response to the query, trying UDP first and falling back to TCP if UDP results in a truncated response.

q, a `dns.message.Message`, the query to send

where, a `str` containing an IPv4 or IPv6 address, where to send the message.

timeout, a `float` or `None`, the number of seconds to wait before the query times out. If `None`, the default, wait forever.

port, an `int`, the port send the message to. The default is 53.

source, a `str` containing an IPv4 or IPv6 address, specifying the source address. The default is the wildcard address.

source_port, an `int`, the port from which to send the message. The default is 0.

ignore_unexpected, a `bool`. If `True`, ignore responses from unexpected sources.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the received message.

udp_sock, a `socket.socket`, or `None`, the socket to use for the UDP query. If `None`, the default, a socket is created. Note that if a socket is provided, it must be a nonblocking datagram socket, and the *source* and *source_port* are ignored for the UDP query.

tcp_sock, a `socket.socket`, or `None`, the connected socket to use for the TCP query. If `None`, the default, a socket is created. Note that if a socket is provided, it must be a nonblocking connected stream socket, and *where*, *source* and *source_port* are ignored for the TCP query.

ignore_errors, a `bool`. If various format errors or response mismatches occur while listening for UDP, ignore them and keep listening for a valid response. The default is `False`.

Returns a (`dns.message.Message`, tcp) tuple where tcp is `True` if and only if TCP was used.

`dns.query.send_udp`(*sock: Any, what: Message | bytes, destination: Any, expiration: float | None = None*) → Tuple[int, float] [source]

Send a DNS message to the specified UDP socket.

sock, a `socket`.

what, a `bytes` or `dns.message.Message`, the message to send.

destination, a destination tuple appropriate for the address family of the socket, specifying where to send the query.

expiration, a `float` or `None`, the absolute time at which a timeout exception should be raised. If `None`, no timeout will occur.

Returns an `(int, float)` tuple of bytes sent and the sent time.

`dns.query.receive_udp`(*sock: Any, destination: Any | None = None, expiration: float | None = None, ignore_unexpected: bool = False, one_rr_per_rrset: bool = False, keyring: Dict[Name, Key] | None = None, request_mac: bytes | None = b'', ignore_trailing: bool = False, raise_on_truncation: bool = False, ignore_errors: bool = False, query: Message | None = None*)→ Any [source]

Read a DNS message from a UDP socket.

sock, a `socket`.

destination, a destination tuple appropriate for the address family of the socket, specifying where the message is expected to arrive from. When receiving a response, this would be where the associated query was sent.

expiration, a `float` or `None`, the absolute time at which a timeout exception should be raised. If `None`, no timeout will occur.

ignore_unexpected, a `bool`. If `True`, ignore responses from unexpected sources.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

keyring, a `dict`, the keyring to use for TSIG.

request_mac, a `bytes` or `None`, the MAC of the request (for TSIG).

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the received message.

raise_on_truncation, a `bool`. If `True`, raise an exception if the TC bit is set.

Raises if the message is malformed, if network errors occur, of if there is a timeout.

If *destination* is not `None`, returns a `(dns.message.Message, float)` tuple of the received message and the received time.

If *destination* is `None`, returns a `(dns.message.Message, float, tuple)` tuple of the received message, the received time, and the address where the message arrived from.

ignore_errors, a `bool`. If various format errors or response mismatches occur, ignore them and keep listening for a valid response. The default is `False`.

query, a `dns.message.Message` or `None`. If not `None` and *ignore_errors* is `True`, check that the received message is a response to this query, and if not keep listening for a valid response.

TCP

dns.query.tcp(*q: Message, where: str, timeout: float | None = None, port: int = 53, source: str | None = None, source_port: int = 0, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, sock: Any | None = None*)→ Message [source]

Return the response obtained after sending a query via TCP.

q, a `dns.message.Message`, the query to send

where, a `str` containing an IPv4 or IPv6 address, where to send the message.

timeout, a `float` or `None`, the number of seconds to wait before the query times out. If `None`, the default, wait forever.

port, an `int`, the port send the message to. The default is 53.

source, a `str` containing an IPv4 or IPv6 address, specifying the source address. The default is the wildcard address.

source_port, an `int`, the port from which to send the message. The default is 0.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the received message.

sock, a `socket.socket`, or `None`, the connected socket to use for the query. If `None`, the default, a socket is created. Note that if a socket is provided, it must be a nonblocking connected stream socket, and *where*, *port*, *source* and *source_port* are ignored.

Returns a `dns.message.Message`.

dns.query.send_tcp(*sock: Any, what: Message | bytes, expiration: float | None = None*)→ Tuple[int, float] [source]

Send a DNS message to the specified TCP socket.

sock, a `socket`.

what, a `bytes` or `dns.message.Message`, the message to send.

expiration, a `float` or `None`, the absolute time at which a timeout exception should be raised. If `None`, no timeout will occur.

Returns an `(int, float)` tuple of bytes sent and the sent time.

dns.query.receive_tcp(*sock: Any, expiration: float | None = None, one_rr_per_rrset: bool = False, keyring: Dict[Name, Key] | None = None, request_mac: bytes | None = b'', ignore_trailing: bool = False*)→ Tuple[Message, float] [source]

Read a DNS message from a TCP socket.

sock, a `socket`.

expiration, a `float` or `None`, the absolute time at which a timeout exception should be raised. If `None`, no timeout will occur.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

keyring, a `dict`, the keyring to use for TSIG.

request_mac, a `bytes` or `None`, the MAC of the request (for TSIG).

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the received message.

Raises if the message is malformed, if network errors occur, of if there is a timeout.

Returns a `(dns.message.Message, float)` tuple of the received message and the received time.

TLS

`dns.query.tls`(*q: Message, where: str, timeout: float | None = None, port: int = 853, source: str | None = None, source_port: int = 0, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, sock: SSLSocket | None = None, ssl_context: SSLContext | None = None, server_hostname: str | None = None, verify: bool | str = True*) → Message [source]

Return the response obtained after sending a query via TLS.

q, a `dns.message.Message`, the query to send

where, a `str` containing an IPv4 or IPv6 address, where to send the message.

timeout, a `float` or `None`, the number of seconds to wait before the query times out. If `None`, the default, wait forever.

port, an `int`, the port send the message to. The default is 853.

source, a `str` containing an IPv4 or IPv6 address, specifying the source address. The default is the wildcard address.

source_port, an `int`, the port from which to send the message. The default is 0.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the received message.

sock, an `ssl.SSLSocket`, or `None`, the socket to use for the query. If `None`, the default, a socket is created. Note that if a socket is provided, it must be a nonblocking connected SSL stream socket, and *where*, *port*, *source*, *source_port*, and *ssl_context* are ignored.

ssl_context, an `ssl.SSLContext`, the context to use when establishing a TLS connection. If `None`, the default, creates one with the default configuration.

server_hostname, a `str` containing the server's hostname. The default is `None`, which means that no hostname is known, and if an SSL context is created, hostname checking will be disabled.

verify, a `bool` or `str`. If a `True`, then TLS certificate verification of the server is done using the default CA bundle; if `False`, then no verification is done; if a *str* then it specifies the path to a certificate file or directory which will be used for verification.

Returns a `dns.message.Message`.

HTTPS

dns.query.https(*q: Message, where: str, timeout: float | None = None, port: int = 443, source: str | None = None, source_port: int = 0, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, session: Any | None = None, path: str = '/dns-query', post: bool = True, bootstrap_address: str | None = None, verify: bool | str = True, resolver: Resolver | None = None, family: int | None = AddressFamily.AF_UNSPEC*) → Message
[source]

Return the response obtained after sending a query via DNS-over-HTTPS.

q, a `dns.message.Message`, the query to send.

where, a `str`, the nameserver IP address or the full URL. If an IP address is given, the URL will be constructed using the following schema: https://<IP-address>:<port>/<path>.

timeout, a `float` or `None`, the number of seconds to wait before the query times out. If `None`, the default, wait forever.

port, a `int`, the port to send the query to. The default is 443.

source, a `str` containing an IPv4 or IPv6 address, specifying the source address. The default is the wildcard address.

source_port, an `int`, the port from which to send the message. The default is 0.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the received message.

session, an `httpx.Client`. If provided, the client session to use to send the queries.

path, a `str`. If *where* is an IP address, then *path* will be used to construct the URL to send the DNS query to.

post, a `bool`. If `True`, the default, POST method will be used.

bootstrap_address, a `str`, the IP address to use to bypass resolution.

verify, a `bool` or `str`. If a `True`, then TLS certificate verification of the server is done using the default CA bundle; if `False`, then no verification is done; if a *str* then it specifies the path to a certificate file or directory which will be used for verification.

resolver, a `dns.resolver.Resolver` or `None`, the resolver to use for resolution of hostnames in URLs. If not specified, a new resolver with a default configuration will be used; note this is *not* the default resolver as that resolver might have been configured to use DoH causing a chicken-and-egg problem. This parameter only has an effect if the HTTP library is httpx.

family, an `int`, the address family. If socket.AF_UNSPEC (the default), both A and AAAA records will be retrieved.

Returns a `dns.message.Message`.

Zone Transfers

As of dnspython 2.1, `dns.query.xfr()` is deprecated. Please use `dns.query.inbound_xfr()` instead.

class **dns.query.UDPMode**(*value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*) [source]

How should UDP be used in an IXFR from `inbound_xfr()`?

NEVER means "never use UDP; always use TCP" TRY_FIRST means "try to use UDP but fall back to TCP if needed" ONLY means "raise `dns.xfr.UseTCP` if trying UDP does not succeed"

dns.query.inbound_xfr(*where: str, txn_manager: TransactionManager, query: Message | None = None, port: int = 53, timeout: float | None = None, lifetime: float | None = None, source: str | None = None, source_port: int = 0, udp_mode: UDPMode = UDPMode.NEVER*) → None [source]

Conduct an inbound transfer and apply it via a transaction from the txn_manager.

where, a `str` containing an IPv4 or IPv6 address, where to send the message.

txn_manager, a `dns.transaction.TransactionManager`, the txn_manager for this transfer (typically a `dns.zone.Zone`).

query, the query to send. If not supplied, a default query is constructed using information from the *txn_manager*.

port, an `int`, the port send the message to. The default is 53.

timeout, a `float`, the number of seconds to wait for each response message. If None, the default, wait forever.

lifetime, a `float`, the total number of seconds to spend doing the transfer. If `None`, the default, then there is no limit on the time the transfer may take.

source, a `str` containing an IPv4 or IPv6 address, specifying the source address. The default is the wildcard address.

source_port, an `int`, the port from which to send the message. The default is 0.

udp_mode, a `dns.query.UDPMode`, determines how UDP is used for IXFRs. The default is `dns.UDPMode.NEVER`, i.e. only use TCP. Other possibilities are `dns.UDPMode.TRY_FIRST`, which means "try UDP but fallback to TCP if needed", and `dns.UDPMode.ONLY`, which means "try UDP and raise `dns.xfr.UseTCP` if it does not succeed.

Raises on errors.

`dns.query.xfr`(*where: str, zone: ~dns.name.Name | str, rdtype: ~dns.rdatatype.RdataType | str = RdataType.AXFR, rdclass: ~dns.rdataclass.RdataClass | str = RdataClass.IN, timeout: float | None = None, port: int = 53, keyring: ~typing.Dict[~dns.name.Name, ~dns.tsig.Key] | None = None, keyname: ~dns.name.Name | str | None = None, relativize: bool = True, lifetime: float | None = None, source: str | None = None, source_port: int = 0, serial: int = 0, use_udp: bool = False, keyalgorithm: ~dns.name.Name | str = <DNS name hmac-sha256.>*)→ Any [source]

Return a generator for the responses to a zone transfer.

where, a `str` containing an IPv4 or IPv6 address, where to send the message.

zone, a `dns.name.Name` or `str`, the name of the zone to transfer.

rdtype, an `int` or `str`, the type of zone transfer. The default is `dns.rdatatype.AXFR`. `dns.rdatatype.IXFR` can be used to do an incremental transfer instead.

rdclass, an `int` or `str`, the class of the zone transfer. The default is `dns.rdataclass.IN`.

timeout, a `float`, the number of seconds to wait for each response message. If None, the default, wait forever.

port, an `int`, the port send the message to. The default is 53.

keyring, a `dict`, the keyring to use for TSIG.

keyname, a `dns.name.Name` or `str`, the name of the TSIG key to use.

relativize, a `bool`. If `True`, all names in the zone will be relativized to the zone origin. It is essential that the relativize setting matches the one specified to `dns.zone.from_xfr()` if using this generator to make a zone.

lifetime, a `float`, the total number of seconds to spend doing the transfer. If `None`, the default, then there is no limit on the time the transfer may take.

source, a `str` containing an IPv4 or IPv6 address, specifying the source address. The default is the wildcard address.

source_port, an `int`, the port from which to send the message. The default is 0.

serial, an `int`, the SOA serial number to use as the base for an IXFR diff sequence (only meaningful if *rdtype* is `dns.rdatatype.IXFR`).

use_udp, a `bool`. If `True`, use UDP (only meaningful for IXFR).

keyalgorithm, a `dns.name.Name` or `str`, the TSIG algorithm to use.

Raises on errors, and so does the generator.

Returns a generator of `dns.message.Message` objects.

Stub Resolver

Dnspython's resolver module implements a "stub resolver", which does DNS recursion with the aid of a remote "full resolver" provided by an ISP or other service provider. By default, dnspython will use the full resolver specified by its host system, but another resolver can easily be used simply by setting the *nameservers* attribute.

DNS Zones

- `Inbound`
  - `Inbound.process_message()`
- `make_query()`

The RRSet Reader

`dns.zonefile.read_rrsets()` reads one or more RRsets from text format. It is designed to be used in situations where you are processing DNS data in text format, but do not want or need a valid zone. For example, a DNS registry web application might want to allow the user to input RRs.

dns.zonefile.read_rrsets(*text: ~typing.Any, name: ~dns.name.Name | str | None = None, ttl: int | None = None, rdclass: ~dns.rdataclass.RdataClass | str | None = RdataClass.IN, default_rdclass: ~dns.rdataclass.RdataClass | str = RdataClass.IN, rdtype: ~dns.rdatatype.RdataType | str | None = None, default_ttl: int | str | None = None, idna_codec: ~dns.name.IDNACodec | None = None, origin: ~dns.name.Name | str | None = <DNS name .>, relativize: bool = False*) → List[RRset] [source]

Read one or more rrsets from the specified text, possibly subject to restrictions.

text, a file object or a string, is the input to process.

name, a string, `dns.name.Name`, or `None`, is the owner name of the rrset. If not `None`, then the owner name is “forced”, and the input must not specify an owner name. If `None`, then any owner names are allowed and must be present in the input.

ttl, an `int`, string, or None. If not `None`, the the TTL is forced to be the specified value and the input must not specify a TTL. If `None`, then a TTL may be specified in the input. If it is not specified, then the *default_ttl* will be used.

rdclass, a `dns.rdataclass.RdataClass`, string, or `None`. If not `None`, then the class is forced to the specified value, and the input must not specify a class. If `None`, then the input may specify a class that matches *default_rdclass*. Note that it is not possible to return rrsets with differing classes; specifying `None` for the class simply allows the user to optionally type a class as that may be convenient when cutting and pasting.

default_rdclass, a `dns.rdataclass.RdataClass` or string. The class of the returned rrsets.

rdtype, a `dns.rdatatype.RdataType`, string, or `None`. If not `None`, then the type is forced to the specified value, and the input must not specify a type. If `None`, then a type must be present for each RR.

default_ttl, an `int`, string, or `None`. If not `None`, then if the TTL is not forced and is not specified, then this value will be used. if `None`, then if the TTL is not forced an error will occur if the TTL is not specified.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used. Note that codecs only apply to the owner name; dnspython does not do IDNA for names in rdata, as there is no IDNA zonefile

format.

origin, a string, `dns.name.Name`, or `None`, is the origin for any relative names in the input, and also the origin to relativize to if *relativize* is `True`.

relativize, a bool. If `True`, names are relativized to the *origin*; if `False` then any relative names in the input are made absolute by appending the *origin*.

Examples

Read RRSets with name, TTL, and rdclass forced:

```
input = '''
mx 10 a
mx 20 b
ns ns1
'''
rrsets = dns.read_rrsets(input, name='name', ttl=300)
```

Read RRSets with name, TTL, rdclass, and rdtype forced:

```
input = '''
10 a
20 b
'''
rrsets = dns.read_rrsets(input, name='name', ttl=300, rdtype='mx')
```

Note that in this case the length of rrsets will always be one.

Read relativized RRsets with unforced rdclass (but which must match default_rdclass):

```
input = '''
name1 20 MX 10 a.example.
name2 30 IN MX 20 b
'''
rrsets = dns.read_rrsets(input, origin='example', relativize=True,
                         rdclass=None)
```

The dns.zonefile.Reader Class

The `Reader` class reads data in DNS zonefile format, or various restrictions of that format, and converts it to a sequence of operations in a transaction.

This class is primarily used by `dns.zone.Zone.from_text()` and `dns.zonefile.read_rrsets`, but may be useful for other software which needs to process the zonefile format.

class **dns.zonefile.Reader**(*tok: Tokenizer, rdclass: RdataClass, txn: Transaction, allow_include: bool = False, allow_directives: bool | Iterable[str] = True, force_name: Name | None = None, force_ttl: int | None = None, force_rdclass: RdataClass | None = None, force_rdtype: RdataType | None = None, default_ttl: int | None = None*) [source]

Read a DNS zone file into a transaction.

read()→ None [source]

Read a DNS zone file and build a zone object.

@raises dns.zone.NoSOA: No SOA RR was found at the zone origin @raises dns.zone.NoNS: No NS RRset was found at the zone origin

DNSSEC

Dnspython can do simple DNSSEC signature validation and signing. In order to use DNSSEC functions, you must have `python cryptography` installed.

DNSSEC Functions

dns.dnssec.algorithm_from_text(*text: str*)→ Algorithm [source]

Convert text into a DNSSEC algorithm value.

text, a `str`, the text to convert to into an algorithm value.

Returns an `int`.

dns.dnssec.algorithm_to_text(*value: Algorithm | int*)→ str [source]

Convert a DNSSEC algorithm value to text

value, a `dns.dnssec.Algorithm`.

Returns a `str`, the name of a DNSSEC algorithm.

dns.dnssec.key_id(*key: DNSKEY | CDNSKEY*)→ int [source]

Return the key id (a 16-bit number) for the specified key.

key, a `dns.rdtypes.ANY.DNSKEY.DNSKEY`

Returns an `int` between 0 and 65535

dns.dnssec.make_ds(*name: Name | str, key: Rdata, algorithm: DSDigest | str, origin: Name | None = None, policy: Policy | None = None, validating: bool = False*)→ DS [source]

Create a DS record for a DNSSEC key.

name, a `dns.name.Name` or `str`, the owner name of the DS record.

key, a `dns.rdtypes.ANY.DNSKEY.DNSKEY` or `dns.rdtypes.ANY.DNSKEY.CDNSKEY`, the key the DS is about.

algorithm, a `str` or `int` specifying the hash algorithm. The currently supported hashes are “SHA1”, “SHA256”, and “SHA384”. Case does not matter for these strings.

origin, a `dns.name.Name` or `None`. If *key* is a relative name, then it will be made absolute using the specified origin.

policy, a `dns.dnssec.Policy` or `None`. If `None`, the default policy, `dns.dnssec.default_policy` is used; this policy defaults to that of RFC 8624.

validating, a `bool`. If `True`, then policy is checked in validating mode, i.e. "Is it ok to validate using this digest algorithm?". Otherwise the policy is checked in creating mode, i.e. "Is it ok to create a DS with this digest algorithm?".

Raises `UnsupportedAlgorithm` if the algorithm is unknown.

Raises `DeniedByPolicy` if the algorithm is denied by policy.

Returns a `dns.rdtypes.ANY.DS.DS`

dns.dnssec.make_cds(*name: Name | str, key: Rdata, algorithm: DSDigest | str, origin: Name | None = None*)→ CDS [source]

Create a CDS record for a DNSSEC key.

name, a `dns.name.Name` or `str`, the owner name of the DS record.

key, a `dns.rdtypes.ANY.DNSKEY.DNSKEY` or `dns.rdtypes.ANY.DNSKEY.CDNSKEY`, the key the DS is about.

algorithm, a `str` or `int` specifying the hash algorithm. The currently supported hashes are "SHA1", "SHA256", and "SHA384". Case does not matter for these strings.

origin, a `dns.name.Name` or `None`. If *key* is a relative name, then it will be made absolute using the specified origin.

Raises `UnsupportedAlgorithm` if the algorithm is unknown.

Returns a `dns.rdtypes.ANY.DS.CDS`

dns.dnssec.make_dnskey(*public_key: GenericPublicKey | RSAPublicKey | EllipticCurvePublicKey | Ed25519PublicKey | Ed448PublicKey, algorithm: int | str, flags: int = Flag.ZONE, protocol: int = 3*)→ DNSKEY

Convert a public key to DNSKEY Rdata

public_key, a `PublicKey` (`GenericPublicKey` or `cryptography.hazmat.primitives.asymmetric`) to convert.

algorithm, a `str` or `int` specifying the DNSKEY algorithm.

flags: DNSKEY flags field as an integer.

protocol: DNSKEY protocol field as an integer.

Raises `ValueError` if the specified key algorithm parameters are not unsupported, `TypeError` if the key type is unsupported, *UnsupportedAlgorithm* if the algorithm is unknown and *AlgorithmKeyMismatch* if the algorithm does not match the key type.

Return DNSKEY `Rdata`.

dns.dnssec.make_cdnskey()

Convert a public key to CDNSKEY Rdata

public_key, the public key to convert, a `cryptography.hazmat.primitives.asymmetric` public key class applicable for DNSSEC.

algorithm, a `str` or `int` specifying the DNSKEY algorithm.

flags: DNSKEY flags field as an integer.

protocol: DNSKEY protocol field as an integer.

Raises `ValueError` if the specified key algorithm parameters are not unsupported, `TypeError` if the key type is unsupported, *UnsupportedAlgorithm* if the algorithm is unknown and *AlgorithmKeyMismatch* if the algorithm does not match the key type.

Return CDNSKEY `Rdata`.

dns.dnssec.sign(*rrset: RRset | Tuple[Name, Rdataset], private_key: GenericPrivateKey | RSAPrivateKey | EllipticCurvePrivateKey | Ed25519PrivateKey | Ed448PrivateKey, signer: Name, dnskey: DNSKEY, inception: datetime | str | int | float | None = None, expiration: datetime | str | int | float | None = None, lifetime: int | None = None, verify: bool = False, policy: Policy | None = None, origin: Name | None = None*)→ RRSIG

Sign RRset using private key.

rrset, the RRset to validate. This can be a `dns.rrset.RRset` or a (`dns.name.Name`, `dns.rdataset.Rdataset`) tuple.

private_key, the private key to use for signing, a `cryptography.hazmat.primitives.asymmetric` private key class applicable for DNSSEC.

signer, a `dns.name.Name`, the Signer's name.

dnskey, a `DNSKEY` matching `private_key`.

inception, a `datetime`, `str`, `int`, `float` or `None`, the signature inception time. If `None`, the current time is used. If a `str`, the format is "YYYYMMDDHHMMSS" or alternatively the number of seconds since the UNIX epoch in text form; this is the same the RRSIG rdata's text form. Values of type *int* or *float* are interpreted as seconds since the UNIX epoch.

expiration, a `datetime`, `str`, `int`, `float` or `None`, the signature expiration time. If `None`, the expiration time will be the inception time plus the value of the *lifetime* parameter. See the description of *inception* above for how the various parameter types are interpreted.

lifetime, an `int` or `None`, the signature lifetime in seconds. This parameter is only meaningful if *expiration* is `None`.

verify, a `bool`. If set to `True`, the signer will verify signatures after they are created; the default is `False`.

policy, a `dns.dnssec.Policy` or `None`. If `None`, the default policy, `dns.dnssec.default_policy` is used; this policy defaults to that of RFC 8624.

origin, a `dns.name.Name` or `None`. If `None`, the default, then all names in the rrset (including its owner name) must be absolute; otherwise the specified origin will be used to make names absolute when signing.

Raises `DeniedByPolicy` if the signature is denied by policy.

`dns.dnssec.validate`(*rrset: RRset | Tuple[Name, Rdataset], rrsigset: RRset | Tuple[Name, Rdataset], keys: Dict[Name, Rdataset | Node], origin: Name | None = None, now: float | None = None, policy: Policy | None = None*)→ None

Validate an RRset against a signature RRset, throwing an exception if none of the signatures validate.

rrset, the RRset to validate. This can be a `dns.rrset.RRset` or a (`dns.name.Name`, `dns.rdataset.Rdataset`) tuple.

rrsigset, the signature RRset. This can be a `dns.rrset.RRset` or a (`dns.name.Name`, `dns.rdataset.Rdataset`) tuple.

keys, the key dictionary, used to find the DNSKEY associated with a given name. The dictionary is keyed by a `dns.name.Name`, and has `dns.node.Node` or `dns.rdataset.Rdataset` values.

origin, a `dns.name.Name`, the origin to use for relative names; defaults to None.

now, an `int` or `None`, the time, in seconds since the epoch, to use as the current time when validating. If `None`, the actual current time is used.

policy, a `dns.dnssec.Policy` or `None`. If `None`, the default policy, `dns.dnssec.default_policy` is used; this policy defaults to that of RFC 8624.

Raises `ValidationFailure` if the signature is expired, not yet valid, the public key is invalid, the algorithm is unknown, the verification fails, etc.

`dns.dnssec.validate_rrsig`(*rrset: RRset | Tuple[Name, Rdataset], rrsig: RRSIG, keys: Dict[Name, Rdataset | Node], origin: Name | None = None, now: float | None = None, policy: Policy | None = None*)→ None

Validate an RRset against a single signature rdata, throwing an exception if validation is not successful.

rrset, the RRset to validate. This can be a `dns.rrset.RRset` or a (`dns.name.Name`, `dns.rdataset.Rdataset`) tuple.

rrsig, a `dns.rdata.Rdata`, the signature to validate.

keys, the key dictionary, used to find the DNSKEY associated with a given name. The dictionary is keyed by a `dns.name.Name`, and has `dns.node.Node` or `dns.rdataset.Rdataset` values.

origin, a `dns.name.Name` or `None`, the origin to use for relative names.

now, a `float` or `None`, the time, in seconds since the epoch, to use as the current time when validating. If `None`, the actual current time is used.

policy, a `dns.dnssec.Policy` or `None`. If `None`, the default policy, `dns.dnssec.default_policy` is used; this policy defaults to that of RFC 8624.

Raises `ValidationFailure` if the signature is expired, not yet valid, the public key is invalid, the algorithm is unknown, the verification fails, etc.

Raises `UnsupportedAlgorithm` if the algorithm is recognized by dnspython but not implemented.

dns.dnssec.nsec3_hash(*domain: Name | str, salt: str | bytes | None, iterations: int, algorithm: int | str*)→ str [source]

Calculate the NSEC3 hash, according to https://tools.ietf.org/html/rfc5155#section-5

domain, a `dns.name.Name` or `str`, the name to hash.

salt, a `str`, `bytes`, or `None`, the hash salt. If a string, it is decoded as a hex string.

iterations, an `int`, the number of iterations.

algorithm, a `str` or `int`, the hash algorithm. The only defined algorithm is SHA1.

Returns a `str`, the encoded NSEC3 hash.

dns.dnssec.make_ds_rdataset() [source]

Create a DS record from DNSKEY/CDNSKEY/CDS.

rrset, the RRset to create DS Rdataset for. This can be a `dns.rrset.RRset` or a (`dns.name.Name`, `dns.rdataset.Rdataset`) tuple.

algorithms, a set of `str` or `int` specifying the hash algorithms. The currently supported hashes are “SHA1”, “SHA256”, and “SHA384”. Case does not matter for these strings. If the RRset is a CDS, only digest algorithms matching algorithms are accepted.

origin, a `dns.name.Name` or `None`. If *key* is a relative name, then it will be made absolute using the specified origin.

Raises `UnsupportedAlgorithm` if any of the algorithms are unknown and `ValueError` if the given RRset is not usable.

Returns a `dns.rdataset.Rdataset`

dns.dnssec.cds_rdataset_to_ds_rdataset() [source]

Create a CDS record from DS.

rdataset, a `dns.rdataset.Rdataset`, to create DS Rdataset for.

Raises `ValueError` if the rdataset is not CDS.

Returns a `dns.rdataset.Rdataset`

dns.dnssec.dnskey_rdataset_to_cds_rdataset() [source]

Create a CDS record from DNSKEY/CDNSKEY.

name, a `dns.name.Name` or `str`, the owner name of the CDS record.

rdataset, a `dns.rdataset.Rdataset`, to create DS Rdataset for.

algorithm, a `str` or `int` specifying the hash algorithm. The currently supported hashes are "SHA1", "SHA256", and "SHA384". Case does not matter for these strings.

origin, a `dns.name.Name` or `None`. If *key* is a relative name, then it will be made absolute using the specified origin.

Raises `UnsupportedAlgorithm` if the algorithm is unknown or `ValueError` if the rdataset is not DNSKEY/CDNSKEY.

Returns a `dns.rdataset.Rdataset`

dns.dnssec.dnskey_rdataset_to_cdnskey_rdataset() [source]

Create a CDNSKEY record from DNSKEY.

rdataset, a `dns.rdataset.Rdataset`, to create CDNSKEY Rdataset for.

Returns a `dns.rdataset.Rdataset`

dns.dnssec.default_rrset_signer() [source]

Default RRset signer

dns.dnssec.sign_zone() [source]

Sign zone.

zone, a `dns.zone.Zone`, the zone to sign.

txn, a `dns.transaction.Transaction`, an optional transaction to use for signing.

keys, a list of (`PrivateKey`, `DNSKEY`) tuples, to use for signing. KSK/ZSK roles are assigned automatically if the SEP flag is used, otherwise all RRsets are signed by all keys.

add_dnskey, a `bool`. If `True`, the default, all specified DNSKEYs are automatically added to the zone on signing.

dnskey_ttl, a``int``, specifies the TTL for DNSKEY RRs. If not specified the TTL of the existing DNSKEY RRset used or the TTL of the SOA RRset.

inception, a `datetime`, `str`, `int`, `float` or `None`, the signature inception time. If `None`, the current time is used. If a `str`, the format is "YYYYMMDDHHMMSS" or alternatively the number of seconds since the UNIX epoch in text form; this is the same the RRSIG rdata's text form. Values of type *int* or *float* are interpreted as seconds since the UNIX epoch.

expiration, a `datetime`, `str`, `int`, `float` or `None`, the signature expiration time. If `None`, the expiration time will be the inception time plus the value of the *lifetime* parameter. See the description of *inception* above for how the various parameter types are interpreted.

lifetime, an `int` or `None`, the signature lifetime in seconds. This parameter is only meaningful if *expiration* is `None`.

nsec3, a `NSEC3PARAM` Rdata, configures signing using NSEC3. Not yet implemented.

rrset_signer, a `Callable`, an optional function for signing RRsets. The function requires two arguments: transaction and RRset. If the not specified, `dns.dnssec.default_rrset_signer` will be used.

Returns `None`.

DNSSEC Algorithms

dns.dnssec.RSAMD5 = *Algorithm.RSAMD5*

dns.dnssec.DH = *Algorithm.DH*

dns.dnssec.DSA = *Algorithm.DSA*

dns.dnssec.ECC = *Algorithm.ECC*

dns.dnssec.RSASHA1 = *Algorithm.RSASHA1*

dns.dnssec.DSANSEC3SHA1 = *Algorithm.DSANSEC3SHA1*

dns.dnssec.RSASHA1NSEC3SHA1 = *Algorithm.RSASHA1NSEC3SHA1*

dns.dnssec.RSASHA256 = *Algorithm.RSASHA256*

dns.dnssec.RSASHA512 = *Algorithm.RSASHA512*

dns.dnssec.ECDSAP256SHA256 = *Algorithm.ECDSAP256SHA256*

dns.dnssec.ECDSAP384SHA384 = *Algorithm.ECDSAP384SHA384*

dns.dnssec.INDIRECT = *Algorithm.INDIRECT*

dns.dnssec.PRIVATEDNS = *Algorithm.PRIVATEDNS*

dns.dnssec.PRIVATEOID = *Algorithm.PRIVATEOID*

Asynchronous I/O Support

The `dns.asyncquery` and `dns.asyncresolver` modules offer asynchronous APIs equivalent to those of `dns.query` and `dns.resolver`.

Dnspython presents a uniform API, but offers three different backend implementations, to support the Trio, Curio, and asyncio libraries. Dnspython attempts to detect which library is in use by using the `sniffio` library if it is available. It's also possible to explicitly select a "backend" library, or to pass a backend to a particular call, allowing for use in mixed library situations.

Note that Curio is not supported for DNS-over-HTTPS, due to a lack of support in the anyio library used by httpx.

Exceptions

Common Exceptions

Common DNS Exceptions.

Dnspython modules may also define their own exceptions, which will always be subclasses of `DNSException`.

exception **dns.exception.AlgorithmKeyMismatch**(**args*, ***kwargs*) [source]

The DNSSEC algorithm is not supported for the given key type.

exception **dns.exception.DNSException**(**args*, ***kwargs*) [source]

Abstract base class shared by all dnspython exceptions.

It supports two basic modes of operation:

a) Old/compatible mode is used if `__init__` was called with empty *kwargs*. In compatible mode all *args* are passed to the standard Python Exception class as before and all *args* are printed by the standard `__str__` implementation. Class variable `msg` (or doc string if `msg` is `None`) is returned from `str()` if *args* is empty.

b) New/parametrized mode is used if `__init__` was called with non-empty *kwargs*. In the new mode *args* must be empty and all kwargs must match those set in class variable `supp_kwargs`. All kwargs are stored inside `self.kwargs` and used in a new `__str__` implementation to construct a formatted message based on the `fmt` class variable, a `string`.

In the simplest case it is enough to override the `supp_kwargs` and `fmt` class variables to get nice parametrized messages.

exception **dns.exception.DeniedByPolicy**(**args*, ***kwargs*) [source]

Denied by DNSSEC policy.

exception **dns.exception.FormError**(**args*, ***kwargs*) [source]

DNS message is malformed.

exception **dns.exception.SyntaxError**(**args*, ***kwargs*) [source]

Text input is malformed.

exception **dns.exception.Timeout**(**args*, ***kwargs*) [source]

The DNS operation timed out.

exception **dns.exception.TooBig**(**args*, ***kwargs*) [source]

The DNS message is too big.

exception **dns.exception.UnexpectedEnd**(**args*, ***kwargs*) [source]

Text input ended unexpectedly.

exception **dns.exception.UnsupportedAlgorithm**(**args*, ***kwargs*) [source]

The DNSSEC algorithm is not supported.

exception **dns.exception.ValidationFailure**(**args*, ***kwargs*) [source]

The DNSSEC signature is invalid.

dns.dnssec Exceptions

exception **dns.dnssec.UnsupportedAlgorithm**(**args*, ***kwargs*) [source]

The DNSSEC algorithm is not supported.

exception **dns.dnssec.ValidationFailure**(**args*, ***kwargs*) [source]

The DNSSEC signature is invalid.

dns.message Exceptions

exception **dns.message.BadEDNS**(**args*, ***kwargs*) [source]

An OPT record occurred somewhere other than the additional data section.

exception **dns.message.BadTSIG**(**args*, ***kwargs*) [source]

A TSIG record occurred somewhere other than the end of the additional data section.

exception **dns.message.ShortHeader**(**args*, ***kwargs*) [source]

The DNS packet passed to from_wire() is too short.

exception **dns.message.TrailingJunk**(**args*, ***kwargs*) [source]

The DNS packet passed to from_wire() has extra junk at the end of it.

exception **dns.message.UnknownHeaderField**(**args*, ***kwargs*) [source]

The header field name was not recognized when converting from text into a message.

exception **dns.message.UnknownTSIGKey**(**args*, ***kwargs*) [source]

A TSIG with an unknown key was received.

dns.name Exceptions

exception **dns.name.AbsoluteConcatenation**(**args*, ***kwargs*) [source]

An attempt was made to append anything other than the empty name to an absolute DNS name.

exception **dns.name.BadEscape**(**args*, ***kwargs*) [source]

An escaped code in a text format of DNS name is invalid.

exception **dns.name.BadLabelType**(**args*, ***kwargs*) [source]

The label type in DNS name wire format is unknown.

exception **dns.name.BadPointer**(**args*, ***kwargs*) [source]

A DNS compression pointer points forward instead of backward.

exception **dns.name.EmptyLabel**(**args*, ***kwargs*) [source]

A DNS label is empty.

exception **dns.name.IDNAException**(**args*, ***kwargs*) [source]

IDNA processing raised an exception.

exception **dns.name.LabelTooLong**(**args*, ***kwargs*) [source]

A DNS label is > 63 octets long.

exception **dns.name.NameTooLong**(**args*, ***kwargs*) [source]

A DNS name is > 255 octets long.

exception **dns.name.NeedAbsoluteNameOrOrigin**(**args*, ***kwargs*) [source]

An attempt was made to convert a non-absolute name to wire when there was also a non-absolute (or missing) origin.

exception **dns.name.NoIDNA2008**(**args*, ***kwargs*) [source]

IDNA 2008 processing was requested but the idna module is not available.

exception **dns.name.NoParent**(**args*, ***kwargs*) [source]

An attempt was made to get the parent of the root name or the empty name.

dns.opcode Exceptions

exception **dns.opcode.UnknownOpcode**(**args*, ***kwargs*) [source]

An DNS opcode is unknown.

dns.query Exceptions

exception **dns.query.BadResponse**(**args*, ***kwargs*) [source]

A DNS query response does not respond to the question asked.

exception **dns.query.NoDOH**(**args*, ***kwargs*) [source]

DNS over HTTPS (DOH) was requested but the httpx module is not available.

exception **dns.query.UnexpectedSource**(**args*, ***kwargs*) [source]

A DNS query response came from an unexpected address or port.

exception **dns.query.TransferError**(*rcode*) [source]

A zone transfer response got a non-zero rcode.

dns.rcode Exceptions

exception **dns.rcode.UnknownRcode**(**args*, ***kwargs*) [source]

A DNS rcode is unknown.

dns.rdataset Exceptions

exception **dns.rdataset.DifferingCovers**(**args*, ***kwargs*) [source]

An attempt was made to add a DNS SIG/RRSIG whose covered type is not the same as that of the other rdatas in the rdataset.

exception **dns.rdataset.IncompatibleTypes**(**args*, ***kwargs*) [source]

An attempt was made to add DNS RR data of an incompatible type.

dns.resolver Exceptions

exception **dns.resolver.NoAnswer**(**args*, ***kwargs*) [source]

The DNS response does not contain an answer to the question.

exception **dns.resolver.NoMetaqueries**(**args*, ***kwargs*) [source]

DNS metaqueries are not allowed.

exception **dns.resolver.NoNameservers**(**args*, ***kwargs*) [source]

All nameservers failed to answer the query.

errors: list of servers and respective errors The type of errors is [(server IP address, any object convertible to string)]. Non-empty errors list will add explanatory message ()

exception **dns.resolver.NoRootSOA**(**args*, ***kwargs*) [source]

There is no SOA RR at the DNS root name. This should never happen!

exception **dns.resolver.NotAbsolute**(**args*, ***kwargs*) [source]

An absolute domain name is required but a relative name was provided.

exception **dns.resolver.NXDOMAIN**(**args*, ***kwargs*) [source]

The DNS query name does not exist.

exception **dns.resolver.YXDOMAIN**(**args*, ***kwargs*) [source]

The DNS query name is too long after DNAME substitution.

dns.tokenizer Exceptions

exception **dns.tokenizer.UngetBufferFull**(**args*, ***kwargs*) [source]

An attempt was made to unget a token when the unget buffer was full.

dns.ttl Exceptions

exception **dns.ttl.BadTTL**(**args*, ***kwargs*) [source]

DNS TTL value is not well-formed.

dns.zone Exceptions

exception **dns.zone.BadZone**(**args*, ***kwargs*) [source]

The DNS zone is malformed.

exception **dns.zone.NoSOA**(**args*, ***kwargs*) [source]

The DNS zone has no SOA RR at its origin.

exception **dns.zone.NoNS**(**args*, ***kwargs*) [source]

The DNS zone has no NS RRset at its origin.

exception **dns.zone.UnknownOrigin**(**args*, ***kwargs*) [source]

The DNS zone's origin is unknown.

Miscellaneous Utilities

Generic Internet address helper functions.

dns.inet.af_for_address(*text: str*)→ int [source]

Determine the address family of a textual-form network address.

text, a `str`, the textual address.

Raises `ValueError` if the address family cannot be determined from the input.

Returns an `int`.

dns.inet.any_for_af(*af*) [source]

Return the ‘any’ address for the specified address family.

dns.inet.canonicalize(*text: str*)→ str [source]

Verify that *address* is a valid text form IPv4 or IPv6 address and return its canonical text form. IPv6 addresses with scopes are rejected.

text, a `str`, the address in textual form.

Raises `ValueError` if the text is not valid.

dns.inet.inet_ntop(*family: int, address: bytes*)→ str [source]

Convert the binary form of a network address into its textual form.

family is an `int`, the address family.

address is a `bytes`, the network address in binary form.

Raises `NotImplementedError` if the address family specified is not implemented.

Returns a `str`.

dns.inet.inet_pton(*family: int, text: str*)→ bytes [source]

Convert the textual form of a network address into its binary form.

family is an `int`, the address family.

text is a `str`, the textual address.

Raises `NotImplementedError` if the address family specified is not implemented.

Returns a `bytes`.

`dns.inet.is_address`(*text: str*)→ bool [source]

Is the specified string an IPv4 or IPv6 address?

text, a `str`, the textual address.

Returns a `bool`.

`dns.inet.is_multicast`(*text: str*)→ bool [source]

Is the textual-form network address a multicast address?

text, a `str`, the textual address.

Raises `ValueError` if the address family cannot be determined from the input.

Returns a `bool`.

`dns.inet.low_level_address_tuple`(*high_tuple: Tuple[str, int]*, *af: int | None = None*)→ Any [source]

Given a "high-level" address tuple, i.e. an (address, port) return the appropriate "low-level" address tuple suitable for use in socket calls.

If an *af* other than `None` is provided, it is assumed the address in the high-level tuple is valid and has that af. If af is `None`, then af_for_address will be called.

IPv4 helper functions.

`dns.ipv4.canonicalize`(*text: str | bytes*)→ str [source]

Verify that *address* is a valid text form IPv4 address and return its canonical text form.

text, a `str` or `bytes`, the IPv4 address in textual form.

Raises `dns.exception.SyntaxError` if the text is not valid.

`dns.ipv4.inet_aton`(*text: str | bytes*)→ bytes [source]

Convert an IPv4 address in text form to binary form.

text, a `str` or `bytes`, the IPv4 address in textual form.

Returns a `bytes`.

`dns.ipv4.inet_ntoa`(*address: bytes*)→ str [source]

Convert an IPv4 address in binary form to text form.

address, a `bytes`, the IPv4 address in binary form.

Returns a `str`.

IPv6 helper functions.

`dns.ipv6.canonicalize`(*text: str | bytes*)→ str [source]

Verify that *address* is a valid text form IPv6 address and return its canonical text form. Addresses with scopes are rejected.

text, a `str` or `bytes`, the IPv6 address in textual form.

Raises `dns.exception.SyntaxError` if the text is not valid.

`dns.ipv6.inet_aton`(*text: str | bytes, ignore_scope: bool = False*)→ bytes [source]

Convert an IPv6 address in text form to binary form.

text, a `str` or `bytes`, the IPv6 address in textual form.

ignore_scope, a `bool`. If `True`, a scope will be ignored. If `False`, the default, it is an error for a scope to be present.

Returns a `bytes`.

`dns.ipv6.inet_ntoa`(*address: bytes*)→ str [source]

Convert an IPv6 address in binary form to text form.

address, a `bytes`, the IPv6 address in binary form.

Raises `ValueError` if the address isn't 16 bytes long. Returns a `str`.

`dns.ipv6.is_mapped`(*address: bytes*)→ bool [source]

Is the specified address a mapped IPv4 address?

address, a `bytes` is an IPv6 address in binary form.

Returns a `bool`.

`dns.ttl.from_text`(*text: str*)→ int [source]

Convert the text form of a TTL to an integer.

The BIND 8 units syntax for TTLs (e.g. '1w6d4h3m10s') is supported.

text, a `str`, the textual TTL.

Raises `dns.ttl.BadTTL` if the TTL is not well-formed.

Returns an `int`.

class **dns.set.Set**(*items=None*) [source]

A simple set class.

This class was originally used to deal with sets being missing in ancient versions of python, but dnspython will continue to use it as these sets are based on lists and are thus indexable, and this ability is widely used in dnspython applications.

Initialize the set.

items, an iterable or `None`, the initial set of items.

add(*item*) [source]

Add an item to the set.

clear() [source]

Make the set empty.

copy() [source]

Make a (shallow) copy of the set.

difference(*other*) [source]

Return a new set which `self` - `other`, i.e. the items in `self` which are not also in `other`.

Returns the same Set type as this set.

difference_update(*other*) [source]

Update the set, removing any elements from other which are in the set.

discard(*item*) [source]

Remove an item from the set if present.

intersection(*other*) [source]

Return a new set which is the intersection of `self` and `other`.

Returns the same Set type as this set.

intersection_update(*other*) [source]

Update the set, removing any elements from other which are not in both sets.

issubset(*other*) [source]

Is this set a subset of *other*?

Returns a `bool`.

issuperset(*other*) [source]

Is this set a superset of *other*?

Returns a `bool`.

pop() [source]

Remove an arbitrary item from the set.

remove(*item*) [source]

Remove an item from the set.

symmetric_difference(*other*) [source]

Return a new set which (`self` - `other`) | (`other` - `self), ie: the items in either ``self` or `other` which are not contained in their intersection.

Returns the same Set type as this set.

symmetric_difference_update(*other*) [source]

Update the set, retaining only elements unique to both sets.

union(*other*) [source]

Return a new set which is the union of `self` and `other`.

Returns the same Set type as this set.

union_update(*other*) [source]

Update the set, adding any elements from other which are not already in the set.

update(*other*) [source]

Update the set, adding any elements from other which are not already in the set.

other, the collection of items with which to update the set, which may be any iterable type.

dnspython release version information.

dns.version.MAJOR = *2*

MAJOR

dns.version.MICRO = *1*

MICRO

dns.version.MINOR = *6*

MINOR

dns.version.RELEASELEVEL = *15*

RELEASELEVEL

dns.version.SERIAL = *0*

SERIAL

dns.version.hexversion = *33948144*

hexversion

dns.version.version = *'2.6.1'*

version

Using Dnspython with Threads

The dnspython `Name` and `Rdata` types are immutable, and thus thread-safe.

Container objects like `Message`, `Node`, `Rdataset`, `RRset`, and `Zone` are not thread-safe, as they are mutable and not locked. It is up to the caller to ensure safety if they are shared between threads.

The `VersionedZone`, however, is thread-safe. VersionedZones offer read-only and read-write transactions. Read-only transactions access an immutable version, and all the objects returned, including containers, are immutable. Read-write transactions are only visible to their creator until they are committed. Transaction creation and commit are thread-safe. Transaction objects should not be shared between threads.

The `Resolver` is not thread-safe with regards to configuration, but it is safe for many threads to call the `resolve()` method of a resolver. The cache implementations for the resolver are also thread-safe, so if a web-crawling application associates an `LRUCache` with a Resolver, it will be safe to have many crawler threads doing resolutions.

The `dns.query` methods are also thread-safe. One caveat with these functions is that if a socket or other context (e.g. a Requests session or an SSL context) is passed to the function instead of allowing the function to create it, then it is up to the application to ensure thread safety if the context could be used by multiple threads.

Examples

The dnspython source comes with example programs that show how to use dnspython in practice. You can clone the dnspython source from GitHub:

```
git clone https://github.com/rthalley/dnspython.git
```

The example programs are in the `examples/` directory.

The dns.name.Name Class and Predefined Names

class **dns.name.Name**(**args*, ***kwargs*) [source]

labels

A tuple of `bytes` in DNS wire format specifying the DNS labels in the name, in order from least-significant label (i.e. farthest from the origin) to most-significant label.

__init__(*labels*)

Initialize a name using *labels*, an iterable of `bytes` or `str`.

canonicalize()→ Name [source]

Return a name which is equal to the current name, but is in DNSSEC canonical form.

choose_relativity(*origin: Name | None = None*, *relativize: bool = True*)→ Name [source]

Return a name with the relativity desired by the caller.

If *origin* is `None`, then the name is returned. Otherwise, if *relativize* is `True` the name is relativized, and if *relativize* is `False` the name is derelativized.

Returns a `dns.name.Name`.

concatenate(*other: Name*)→ Name [source]

Return a new name which is the concatenation of self and other.

Raises `dns.name.AbsoluteConcatenation` if the name is absolute and *other* is not the empty name.

Returns a `dns.name.Name`.

derelativize(*origin: Name*)→ Name [source]

If the name is a relative name, return a new name which is the concatenation of the name and origin. Otherwise return the name.

For example, derelativizing `www` to origin `dnspython.org.` returns the name `www.dnspython.org.`. Derelativizing `example.` to origin `dnspython.org.` returns `example.`.

Returns a `dns.name.Name`.

`fullcompare`(*other: Name*)→ Tuple[NameRelation, int, int] [source]

Compare two names, returning a 3-tuple `(relation, order, nlabels)`.

relation describes the relation ship between the names, and is one of: `dns.name.NameRelation.NONE`, `dns.name.NameRelation.SUPERDOMAIN`, `dns.name.NameRelation.SUBDOMAIN`, `dns.name.NameRelation.EQUAL`, or `dns.name.NameRelation.COMMONANCESTOR`.

order is < 0 if *self* < *other*, > 0 if *self* > *other*, and == 0 if *self* == *other*. A relative name is always less than an absolute name. If both names have the same relativity, then the DNSSEC order relation is used to order them.

nlabels is the number of significant labels that the two names have in common.

Here are some examples. Names ending in "." are absolute names, those not ending in "." are relative names.

self	other	relation	order	nlabels
www.example.	www.example.	equal	0	3
www.example.	example.	subdomain	> 0	2
example.	www.example.	superdomain	< 0	2
example1.com.	example2.com.	common anc.	< 0	2
example1	example2.	none	< 0	0
example1.	example2	none	> 0	0

`is_absolute`()→ bool [source]

Is the most significant label of this name the root label?

Returns a `bool`.

`is_subdomain`(*other: Name*)→ bool [source]

Is self a subdomain of other?

Note that the notion of subdomain includes equality, e.g. "dnspython.org" is a subdomain of itself.

Returns a `bool`.

`is_superdomain`(*other: Name*)→ bool [source]

Is self a superdomain of other?

Note that the notion of superdomain includes equality, e.g. “dnspython.org” is a superdomain of itself.

Returns a `bool`.

`is_wild`()→ bool [source]

Is this name wild? (I.e. Is the least significant label ‘*’?)

Returns a `bool`.

`parent`()→ Name [source]

Return the parent of the name.

For example, the parent of `www.dnspython.org.` is `dnspython.org`.

Raises `dns.name.NoParent` if the name is either the root name or the empty name, and thus has no parent.

Returns a `dns.name.Name`.

`predecessor`(*origin: Name, prefix_ok: bool = True*)→ Name [source]

Return the maximal predecessor of *name* in the DNSSEC ordering in the zone whose origin is *origin*, or return the longest name under *origin* if the name is origin (i.e. wrap around to the longest name, which may still be *origin* due to length considerations.

The relativity of the name is preserved, so if this name is relative then the method will return a relative name, and likewise if this name is absolute then the predecessor will be absolute.

prefix_ok indicates if prefixing labels is allowed, and defaults to `True`. Normally it is good to allow this, but if computing a maximal predecessor at a zone cut point then `False` must be specified.

`relativize`(*origin: Name*)→ Name [source]

If the name is a subdomain of *origin*, return a new name which is the name relative to origin. Otherwise return the name.

For example, relativizing `www.dnspython.org.` to origin `dnspython.org.` returns the name `www`. Relativizing `example.` to origin `dnspython.org.` returns `example.`.

Returns a `dns.name.Name`.

`split`(*depth: int*)→ Tuple[Name, Name] [source]

Split a name into a prefix and suffix names at the specified depth.

depth is an `int` specifying the number of labels in the suffix

Raises `ValueError` if *depth* was not >= 0 and <= the length of the name.

Returns the tuple `(prefix, suffix)`.

`successor`(*origin: Name, prefix_ok: bool = True*)→ Name [source]

Return the minimal successor of *name* in the DNSSEC ordering in the zone whose origin is *origin*, or return *origin* if the successor cannot be computed due to name length limitations.

Note that *origin* is returned in the "too long" cases because wrapping around to the origin is how NSEC records express "end of the zone".

The relativity of the name is preserved, so if this name is relative then the method will return a relative name, and likewise if this name is absolute then the successor will be absolute.

prefix_ok indicates if prefixing a new minimal label is allowed, and defaults to `True`. Normally it is good to allow this, but if computing a minimal successor at a zone cut point then `False` must be specified.

`to_digestable`(*origin: Name | None = None*)→ bytes [source]

Convert name to a format suitable for digesting in hashes.

The name is canonicalized and converted to uncompressed wire format. All names in wire format are absolute. If the name is a relative name, then an origin must be supplied.

origin is a `dns.name.Name` or `None`. If the name is relative and origin is not `None`, then origin will be appended to the name.

Raises `dns.name.NeedAbsoluteNameOrOrigin` if the name is relative and no origin was provided.

Returns a `bytes`.

`to_text`(*omit_final_dot: bool = False*)→ str [source]

Convert name to DNS text format.

omit_final_dot is a `bool`. If True, don't emit the final dot (denoting the root label) for absolute names. The default is False.

Returns a `str`.

`to_unicode`(*omit_final_dot: bool = False, idna_codec: IDNACodec | None = None*)→ str [source]

Convert name to Unicode text format.

IDN ACE labels are converted to Unicode.

omit_final_dot is a `bool`. If True, don't emit the final dot (denoting the root label) for absolute names. The default is False. *idna_codec* specifies the IDNA encoder/decoder. If None, the dns.name.IDNA_2003_Practical encoder/decoder is used. The IDNA_2003_Practical decoder does not impose any policy, it just decodes punycode, so if you don't want checking for compliance, you can use this decoder for IDNA2008 as well.

Returns a `str`.

to_wire(*file: Any | None = None, compress: Dict[Name, int] | None = None, origin: Name | None = None, canonicalize: bool = False*)→ bytes | None [source]

Convert name to wire format, possibly compressing it.

file is the file where the name is emitted (typically an io.BytesIO file). If `None` (the default), a `bytes` containing the wire name will be returned.

compress, a `dict`, is the compression table to use. If `None` (the default), names will not be compressed. Note that the compression code assumes that compression offset 0 is the start of *file*, and thus compression will not be correct if this is not the case.

origin is a `dns.name.Name` or `None`. If the name is relative and origin is not `None`, then *origin* will be appended to it.

canonicalize, a `bool`, indicates whether the name should be canonicalized; that is, converted to a format suitable for digesting in hashes.

Raises `dns.name.NeedAbsoluteNameOrOrigin` if the name is relative and no origin was provided.

Returns a `bytes` or `None`.

dns.name.root

The root name, i.e. `dns.name.Name([b''])`.

dns.name.empty

The empty name, i.e. `dns.name.Name([])`.

class **dns.name.NameRelation**(*value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*) [source]

Name relation result from fullcompare().

COMMONANCESTOR = *4*

The compared names have a common ancestor.

EQUAL = *3*

The compared names are equal.

NONE= *0*

The compared names have no relationship to each other.

SUBDOMAIN= *2*

The first name is a subdomain of the second.

SUPERDOMAIN= *1*

the first name is a superdomain of the second.

Making DNS Names

dns.name.from_text(*text: bytes | str, origin: ~dns.name.Name | None = <DNS name .>, idna_codec: ~dns.name.IDNACodec | None = None*)→ Name [source]

Convert text into a Name object.

text, a `bytes` or `str`, is the text to convert into a name.

origin, a `dns.name.Name`, specifies the origin to append to non-absolute names. The default is the root name.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

Returns a `dns.name.Name`.

dns.name.from_unicode(*text: str, origin: ~dns.name.Name | None = <DNS name .>, idna_codec: ~dns.name.IDNACodec | None = None*)→ Name [source]

Convert unicode text into a Name object.

Labels are encoded in IDN ACE form according to rules specified by the IDNA codec.

text, a `str`, is the text to convert into a name.

origin, a `dns.name.Name`, specifies the origin to append to non-absolute names. The default is the root name.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

Returns a `dns.name.Name`.

dns.name.from_wire_parser(*parser: Parser*)→ Name [source]

Convert possibly compressed wire format into a Name.

parser is a dns.wire.Parser.

Raises `dns.name.BadPointer` if a compression pointer did not point backwards in the message.

Raises `dns.name.BadLabelType` if an invalid label type was encountered.

Returns a `dns.name.Name`

dns.name.from_wire(*message: bytes, current: int*) → Tuple[Name, int] [source]

Convert possibly compressed wire format into a Name.

message is a `bytes` containing an entire DNS message in DNS wire form.

current, an `int`, is the offset of the beginning of the name from the start of the message

Raises `dns.name.BadPointer` if a compression pointer did not point backwards in the message.

Raises `dns.name.BadLabelType` if an invalid label type was encountered.

Returns a `(dns.name.Name, int)` tuple consisting of the name that was read and the number of bytes of the wire format message which were consumed reading it.

Name Dictionary

class **dns.namedict.NameDict**(**args*, ***kwargs*) [source]

A dictionary whose keys are dns.name.Name objects.

In addition to being like a regular Python dictionary, this dictionary can also get the deepest match for a given key.

get_deepest_match(*name*) [source]

Find the deepest match to *name* in the dictionary.

The deepest match is the longest name in the dictionary which is a superdomain of *name*. Note that *superdomain* includes matching *name* itself.

name, a `dns.name.Name`, the name to find.

Returns a `(key, value)` where *key* is the deepest `dns.name.Name`, and *value* is the value associated with *key*.

max_depth

the maximum depth of the keys that have ever been added

max_depth_items

the number of items of maximum depth

Name Helpers

Sometimes you want to look up an address in the DNS instead of a name. Dnspython provides a helper functions for converting between addresses and their “reverse map” form in the DNS.

For example:

Address	DNS Reverse Name
127.0.0.1	1.0.0.127.in-addr.arpa.
::1	1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa.

dns.reversename.from_address(*text: str, v4_origin: ~dns.name.Name = <DNS name in-addr.arpa.>, v6_origin: ~dns.name.Name = <DNS name ip6.arpa.>*)→ Name [source]

Convert an IPv4 or IPv6 address in textual form into a Name object whose value is the reverse-map domain name of the address.

text, a `str`, is an IPv4 or IPv6 address in textual form (e.g. ‘127.0.0.1’, ‘::1’)

v4_origin, a `dns.name.Name` to append to the labels corresponding to the address if the address is an IPv4 address, instead of the default (in-addr.arpa.)

v6_origin, a `dns.name.Name` to append to the labels corresponding to the address if the address is an IPv6 address, instead of the default (ip6.arpa.)

Raises `dns.exception.SyntaxError` if the address is badly formed.

Returns a `dns.name.Name`.

dns.reversename.to_address(*name: ~dns.name.Name, v4_origin: ~dns.name.Name = <DNS name in-addr.arpa.>, v6_origin: ~dns.name.Name = <DNS name ip6.arpa.>*)→ str [source]

Convert a reverse map domain name into textual address form.

name, a `dns.name.Name`, an IPv4 or IPv6 address in reverse-map name form.

v4_origin, a `dns.name.Name` representing the top-level domain for IPv4 addresses, instead of the default (in-addr.arpa.)

v6_origin, a `dns.name.Name` representing the top-level domain for IPv4 addresses, instead of the default (ip6.arpa.)

Raises `dns.exception.SyntaxError` if the name does not have a reverse-map form.

Returns a `str`.

Dnspython also provides helpers for converting E.164 numbers (i.e. telephone numbers) into the names used for them in the DNS.

For example:

Number	DNS E.164 Name
+1.650.555.1212	2.1.2.1.5.5.5.0.5.6.1.e164.arpa.
+44 20 7946 0123	3.2.1.0.6.4.9.7.0.2.4.4.e164.arpa.

`dns.e164.from_e164`(*text: str, origin: ~dns.name.Name | None = <DNS name e164.arpa.>*)→ Name [source]

Convert an E.164 number in textual form into a Name object whose value is the ENUM domain name for that number.

Non-digits in the text are ignored, i.e. “16505551212”, “+1.650.555.1212” and “1 (650) 555-1212” are all the same.

text, a `str`, is an E.164 number in textual form.

origin, a `dns.name.Name`, the domain in which the number should be constructed. The default is `e164.arpa.`.

Returns a `dns.name.Name`.

`dns.e164.to_e164`(*name: ~dns.name.Name, origin: ~dns.name.Name | None = <DNS name e164.arpa.>, want_plus_prefix: bool = True*)→ str [source]

Convert an ENUM domain name into an E.164 number.

Note that dnspython does not have any information about preferred number formats within national numbering plans, so all numbers are emitted as a simple string of digits, prefixed by a ‘+’ (unless *want_plus_prefix* is `False`).

name is a `dns.name.Name`, the ENUM domain name.

origin is a `dns.name.Name`, a domain containing the ENUM domain name. The name is relativized to this domain before being converted to text. If `None`, no relativization is done.

want_plus_prefix is a `bool`. If True, add a ‘+’ to the beginning of the returned number.

Returns a `str`.

International Domain Name CODECs

Representing non-ASCII text in the DNS is a complex and evolving topic. Generally speaking, Unicode is converted into an ASCII-only, case-insensitive form called "Punycode" by complex rules. There are two standard specifications for this process, "IDNA 2003", which is widely used, and the revised and not fully compatible standard "IDNA 2008". There are also varying degrees of strictness that can be applied in encoding and decoding. Explaining the standards in detail is out of scope for this document; Unicode Technical Standard #46 https://unicode.org/reports/tr46/ is a good place to start learning more.

Dnspython provides "codecs" to implement International Domain Name policy according to the user's desire.

class **dns.name.IDNACodec** [source]

Abstract base class for IDNA encoder/decoders.

class **dns.name.IDNA2003Codec**(*strict_decode: bool = False*) [source]

IDNA 2003 encoder/decoder.

Initialize the IDNA 2003 encoder/decoder.

strict_decode is a `bool`. If *True*, then IDNA2003 checking is done when decoding. This can cause failures if the name was encoded with IDNA2008. The default is *False*.

decode(*label: bytes*)→ str [source]

Decode *label*.

encode(*label: str*)→ bytes [source]

Encode *label*.

class **dns.name.IDNA2008Codec**(*uts_46: bool = False, transitional: bool = False, allow_pure_ascii: bool = False, strict_decode: bool = False*) [source]

IDNA 2008 encoder/decoder.

Initialize the IDNA 2008 encoder/decoder.

uts_46 is a `bool`. If True, apply Unicode IDNA compatibility processing as described in Unicode Technical Standard #46 (https://unicode.org/reports/tr46/). If False, do not apply the mapping. The default is False.

transitional is a `bool`: If True, use the "transitional" mode described in Unicode Technical Standard #46. The default is False.

allow_pure_ascii is a `bool`. If True, then a label which consists of only ASCII characters is allowed. This is less strict than regular IDNA 2008, but is also necessary for mixed names, e.g. a name with starting with "_sip._tcp." and ending in an IDN suffix which would otherwise be disallowed. The default is False.

strict_decode is a `bool`: If True, then IDNA2008 checking is done when decoding. This can cause failures if the name was encoded with IDNA2003. The default is False.

dns.name.IDNA_2003_Practical

The "practical" codec encodes using IDNA 2003 rules and decodes punycode without checking for strict IDNA 2003 compliance.

dns.name.IDNA_2003_Strict

The "strict" codec encodes using IDNA 2003 rules and decodes punycode checking for IDNA 2003 compliance.

dns.name.IDNA_2003

A synonym for `dns.name.IDNA_2003_Practical`.

dns.name.IDNA_2008_Practical

The "practical" codec encodes using IDNA 2008 rules with UTS 46 compatibility processing, and allowing pure ASCII labels. It decodes punycode without checking for strict IDNA 2008 compliance.

dns.name.IDNA_2008_Strict

The "strict" codec encodes using IDNA 2008 rules and decodes punycode checking for IDNA 2008 compliance.

dns.name.IDNA_2008_UTS_46

The "UTS 46" codec encodes using IDNA 2008 rules with UTS 46 compatibility processing and decodes punycode without checking for IDNA 2008 compliance.

dns.name.IDNA_2008_Transitional

The "UTS 46" codec encodes using IDNA 2008 rules with UTS 46 compatibility processing in the "transitional mode" and decodes punycode without checking for IDNA 2008 compliance.

dns.name.IDNA_2008

A synonym for `dns.name.IDNA_2008_Practical`.

Rdata classes and types

Sets of typed data can be associated with a given name. A single typed datum is called an *rdata*. The type of an rdata is specified by its *rdataclass* and *rdatatype*. The class is almost always *IN*, the Internet class, and may often be omitted in the dnspython APIs.

The `dns.rdataclass` module provides constants for each defined rdata class, as well as some helpful functions. The `dns.rdatatype` module does the same for rdata types. Examples of the constants are:

```
dns.rdataclass.IN
dns.rdatatype.AAAA
```

DNS Rdata Classes.

class **dns.rdataclass.RdataClass**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*) [source]

DNS Rdata Class

exception **dns.rdataclass.UnknownRdataclass**(**args*, ***kwargs*) [source]

A DNS class is unknown.

dns.rdataclass.from_text(*text: str*) → RdataClass [source]

Convert text into a DNS rdata class value.

The input text can be a defined DNS RR class mnemonic or instance of the DNS generic class syntax.

For example, “IN” and “CLASS1” will both result in a value of 1.

Raises `dns.rdatatype.UnknownRdataclass` if the class is unknown.

Raises `ValueError` if the rdata class value is not >= 0 and <= 65535.

Returns a `dns.rdataclass.RdataClass`.

dns.rdataclass.is_metaclass(*rdclass: RdataClass*) → bool [source]

True if the specified class is a metaclass.

The currently defined metaclasses are ANY and NONE.

rdclass is a `dns.rdataclass.RdataClass`.

`dns.rdataclass.to_text`(*value: RdataClass*)→ str [source]

Convert a DNS rdata class value to text.

If the value has a known mnemonic, it will be used, otherwise the DNS generic class syntax will be used.

Raises `ValueError` if the rdata class value is not >= 0 and <= 65535.

Returns a `str`.

DNS Rdata Types.

class **`dns.rdatatype.RdataType`**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*) [source]

DNS Rdata Type

exception **`dns.rdatatype.UnknownRdatatype`**(**args*, ***kwargs*) [source]

DNS resource record type is unknown.

`dns.rdatatype.from_text`(*text: str*)→ RdataType [source]

Convert text into a DNS rdata type value.

The input text can be a defined DNS RR type mnemonic or instance of the DNS generic type syntax.

For example, “NS” and “TYPE2” will both result in a value of 2.

Raises `dns.rdatatype.UnknownRdatatype` if the type is unknown.

Raises `ValueError` if the rdata type value is not >= 0 and <= 65535.

Returns a `dns.rdatatype.RdataType`.

`dns.rdatatype.is_metatype`(*rdtype: RdataType*)→ bool [source]

True if the specified type is a metatype.

rdtype is a `dns.rdatatype.RdataType`.

The currently defined metatypes are TKEY, TSIG, IXFR, AXFR, MAILA, MAILB, ANY, and OPT.

Returns a `bool`.

dns.rdatatype.is_singleton(*rdtype: RdataType*) → bool [source]

Is the specified type a singleton type?

Singleton types can only have a single rdata in an rdataset, or a single RR in an RRset.

The currently defined singleton types are CNAME, DNAME, NSEC, NXT, and SOA.

rdtype is an `int`.

Returns a `bool`.

dns.rdatatype.register_type(*rdtype: RdataType, rdtype_text: str, is_singleton: bool = False*) → None [source]

Dynamically register an rdatatype.

rdtype, a `dns.rdatatype.RdataType`, the rdatatype to register.

rdtype_text, a `str`, the textual form of the rdatatype.

is_singleton, a `bool`, indicating if the type is a singleton (i.e. RRsets of the type can have only one member.)

dns.rdatatype.to_text(*value: RdataType*) → str [source]

Convert a DNS rdata type value to text.

If the value has a known mnemonic, it will be used, otherwise the DNS generic type syntax will be used.

Raises `ValueError` if the rdata type value is not >= 0 and <= 65535.

Returns a `str`.

- Rdataclasses
  - `dns.rdataclass.ANY`
  - `dns.rdataclass.CH`
  - `dns.rdataclass.CHAOS`
  - `dns.rdataclass.HESIOD`
  - `dns.rdataclass.HS`
  - `dns.rdataclass.IN`
  - `dns.rdataclass.INTERNET`
  - `dns.rdataclass.NONE`
  - `dns.rdataclass.RESERVED0`
- Rdatatypes
  - `dns.rdatatype.A`
  - `dns.rdatatype.A6`

- `dns.rdatatype.AAAA`
- `dns.rdatatype.AFSDB`
- `dns.rdatatype.AMTRELAY`
- `dns.rdatatype.ANY`
- `dns.rdatatype.APL`
- `dns.rdatatype.AVC`
- `dns.rdatatype.AXFR`
- `dns.rdatatype.CAA`
- `dns.rdatatype.CDNSKEY`
- `dns.rdatatype.CDS`
- `dns.rdatatype.CERT`
- `dns.rdatatype.CNAME`
- `dns.rdatatype.CSYNC`
- `dns.rdatatype.DHCID`
- `dns.rdatatype.DLV`
- `dns.rdatatype.DNAME`
- `dns.rdatatype.DNSKEY`
- `dns.rdatatype.DS`
- `dns.rdatatype.EUI48`
- `dns.rdatatype.EUI64`
- `dns.rdatatype.GPOS`
- `dns.rdatatype.HINFO`
- `dns.rdatatype.HIP`
- `dns.rdatatype.HTTPS`
- `dns.rdatatype.IPSECKEY`
- `dns.rdatatype.ISDN`
- `dns.rdatatype.IXFR`
- `dns.rdatatype.KEY`
- `dns.rdatatype.KX`
- `dns.rdatatype.L32`
- `dns.rdatatype.L64`
- `dns.rdatatype.LOC`
- `dns.rdatatype.LP`
- `dns.rdatatype.MAILA`
- `dns.rdatatype.MAILB`
- `dns.rdatatype.MB`
- `dns.rdatatype.MD`
- `dns.rdatatype.MF`
- `dns.rdatatype.MG`
- `dns.rdatatype.MINFO`
- `dns.rdatatype.MR`

- `dns.rdatatype.MX`
- `dns.rdatatype.NAPTR`
- `dns.rdatatype.NID`
- `dns.rdatatype.NINFO`
- `dns.rdatatype.NS`
- `dns.rdatatype.NSAP`
- `dns.rdatatype.NSAP_PTR`
- `dns.rdatatype.NSEC`
- `dns.rdatatype.NSEC3`
- `dns.rdatatype.NSEC3PARAM`
- `dns.rdatatype.NULL`
- `dns.rdatatype.NXT`
- `dns.rdatatype.OPENPGPKEY`
- `dns.rdatatype.OPT`
- `dns.rdatatype.PTR`
- `dns.rdatatype.PX`
- `dns.rdatatype.RP`
- `dns.rdatatype.RRSIG`
- `dns.rdatatype.RT`
- `dns.rdatatype.SIG`
- `dns.rdatatype.SMIMEA`
- `dns.rdatatype.SOA`
- `dns.rdatatype.SPF`
- `dns.rdatatype.SRV`
- `dns.rdatatype.SSHFP`
- `dns.rdatatype.SVCB`
- `dns.rdatatype.TA`
- `dns.rdatatype.TKEY`
- `dns.rdatatype.TLSA`
- `dns.rdatatype.TSIG`
- `dns.rdatatype.TXT`
- `dns.rdatatype.TYPE0`
- `dns.rdatatype.UNSPEC`
- `dns.rdatatype.URI`
- `dns.rdatatype.WKS`
- `dns.rdatatype.X25`
- `dns.rdatatype.ZONEMD`

DNS Rdata Base Class

All Rdata objects are instances of some subclass of `dns.rdata.Rdata`, and are immutable. The Rdata factory functions described in Making DNS Rdata will create objects which are instances of the most appropriate subclass. For example, a AAAA record will be an instance of the `dns.rdtypes.IN.AAAA` class, but a record of TYPE12345, which we don't know anything specific about, will be an instance of `dns.rdata.GenericRdata`.

Rdata of the same type and class are ordered. For rdata that do not contain domain names, or which contain absolute domain names, the order is the same as the DNSSEC ordering. For rdata containing at least one relative name, that rdata will sort before any rdata with an absolute name. This makes comparison well defined (compared to earlier versions of dnspython), but is a stop-gap measure for backwards compatibility. We want to disallow this type of comparison because it easily leads to bugs. Consider this rdataset:

```
$ORIGIN example.
name 300 IN NS a    ; 1
            NS a.   ; 2
```

In this case the record marked "2" sorts before the one marked "1" when all the names are made absolute and the DNSSEC ordering is used. But when relative comparisons are allowed, "1" sorts before "2". This isn't merely cosmetic, as code making a DNSSEC signature or computing a zone checksum would get different answers for the same content if it failed to make all names absolute before sorting.

Comparing relative rdata with absolute is thus deprecated and will be removed in a future version of dnspython. Setting `dns.rdata._allow_relative_comparisons` to `True` will allow the future behavior to be tested with existing code.

class **dns.rdata.Rdata**(**args*, ***kwargs*) [source]

covers()→ RdataType [source]

Return the type a Rdata covers.

DNS SIG/RRSIG rdatas apply to a specific type; this type is returned by the covers() function. If the rdata type is not SIG or RRSIG, dns.rdatatype.NONE is returned. This is useful when creating rdatasets, allowing the rdataset to contain only RRSIGs of a particular type, e.g. RRSIG(NS).

Returns a `dns.rdatatype.RdataType`.

extended_rdatatype()→ int [source]

Return a 32-bit type value, the least significant 16 bits of which are the ordinary DNS type, and the upper 16 bits of which are the “covered” type, if any.

Returns an `int`.

replace(***kwargs: Any*)→ Rdata [source]

Create a new Rdata instance based on the instance replace was invoked on. It is possible to pass different parameters to override the corresponding properties of the base Rdata.

Any field specific to the Rdata type can be replaced, but the *rdtype* and *rdclass* fields cannot.

Returns an instance of the same Rdata subclass as *self*.

to_digestable(*origin: Name | None = None*)→ bytes [source]

Convert rdata to a format suitable for digesting in hashes. This is also the DNSSEC canonical form.

Returns a `bytes`.

to_generic(*origin: Name | None = None*)→ GenericRdata [source]

Creates a dns.rdata.GenericRdata equivalent of this rdata.

Returns a `dns.rdata.GenericRdata`.

to_text(*origin: Name | None = None, relativize: bool = True, **kw: Dict[str, Any]*)→ str [source]

Convert an rdata to text format.

Returns a `str`.

to_wire(*file: Any | None = None, compress: Dict[Name, int] | None = None, origin: Name | None = None, canonicalize: bool = False*)→ bytes [source]

Convert an rdata to wire format.

Returns a `bytes` or `None`.

Making DNS Rdata

dns.rdata.from_text(*rdclass: RdataClass | str, rdtype: RdataType | str, tok: Tokenizer | str, origin: Name | None = None, relativize: bool = True, relativize_to: Name | None = None, idna_codec: IDNACodec | None = None*)→ Rdata [source]

Build an rdata object from text format.

This function attempts to dynamically load a class which implements the specified rdata class and type. If there is no class-and-type-specific implementation, the GenericRdata class is used.

Once a class is chosen, its from_text() class method is called with the parameters to this function.

If *tok* is a `str`, then a tokenizer is created and the string is used as its input.

rdclass, a `dns.rdataclass.RdataClass` or `str`, the rdataclass.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdatatype.

tok, a `dns.tokenizer.Tokenizer` or a `str`.

origin, a `dns.name.Name` (or `None`), the origin to use for relative names.

relativize, a `bool`. If true, name will be relativized.

relativize_to, a `dns.name.Name` (or `None`), the origin to use when relativizing names. If not set, the *origin* value will be used.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder to use if a tokenizer needs to be created. If `None`, the default IDNA 2003 encoder/decoder is used. If a tokenizer is not created, then the codec associated with the tokenizer is the one that is used.

Returns an instance of the chosen Rdata subclass.

dns.rdata.from_wire_parser(*rdclass: RdataClass | str, rdtype: RdataType | str, parser: Parser, origin: Name | None = None*)→ Rdata [source]

Build an rdata object from wire format

This function attempts to dynamically load a class which implements the specified rdata class and type. If there is no class-and-type-specific implementation, the GenericRdata class is used.

Once a class is chosen, its from_wire() class method is called with the parameters to this function.

rdclass, a `dns.rdataclass.RdataClass` or `str`, the rdataclass.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdatatype.

parser, a `dns.wire.Parser`, the parser, which should be restricted to the rdata length.

origin, a `dns.name.Name` (or `None`). If not `None`, then names will be relativized to this origin.

Returns an instance of the chosen Rdata subclass.

`dns.rdata.from_wire`(*rdclass: RdataClass | str, rdtype: RdataType | str, wire: bytes, current: int, rdlen: int, origin: Name | None = None*) → Rdata [source]

Build an rdata object from wire format

This function attempts to dynamically load a class which implements the specified rdata class and type. If there is no class-and-type-specific implementation, the GenericRdata class is used.

Once a class is chosen, its from_wire() class method is called with the parameters to this function.

rdclass, an `int`, the rdataclass.

rdtype, an `int`, the rdatatype.

wire, a `bytes`, the wire-format message.

current, an `int`, the offset in wire of the beginning of the rdata.

rdlen, an `int`, the length of the wire-format rdata

origin, a `dns.name.Name` (or `None`). If not `None`, then names will be relativized to this origin.

Returns an instance of the chosen Rdata subclass.

Rdata Subclass Reference

Universal Types

class **dns.rdata.GenericRdata**(*rdclass*, *rdtype*, *data*) [source]

Generic Rdata Class

This class is used for rdata types for which we have no better implementation. It implements the DNS “unknown RRs” scheme.

data

A `bytes` containing the rdata’s value.

class **dns.rdtypes.ANY.AFSDB.AFSDB**(*rdclass*, *rdtype*, *preference*, *exchange*) [source]

AFSDB record

property **hostname**

the AFSDB hostname

property **subtype**

the AFSDB subtype

class **dns.rdtypes.ANY.AMTRELAY.AMTRELAY**(*rdclass*, *rdtype*, *precedence*, *discovery_optional*, *relay_type*, *relay*) [source]

AMTRELAY record

precedence

An `int`, the 8-bit unsigned integer preference.

discovery_optional

A `bool`, specifying whether discovery is optional or not.

relay_type

An `int`, the 8-bit unsigned integer relay type.

relay

A `dns.rdtypes.ANY.AMTRELAY.Relay` instance, the relay.

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.AVC.AVC**(*rdclass: RdataClass, rdtype: RdataType, strings: Iterable[bytes | str]*) [source]

AVC record

strings

A tuple of `bytes`, the list of strings.

class **dns.rdtypes.ANY.CAA.CAA**(*rdclass, rdtype, flags, tag, value*) [source]

CAA (Certification Authority Authorization) record

flags

An `int`, the flags

tag

A `bytes`, the tag

value

A `bytes`, the value

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.CDNSKEY.CDNSKEY**(*rdclass, rdtype, flags, protocol, algorithm, key*) [source]

CDNSKEY record

flags

An `int`, the key's flags.

protocol

An `int`, the protocol for which this key may be used.

algorithm:

An `int`, the algorithm used for the key.

key

A `bytes`, the public key.

class **dns.rdtypes.ANY.CDS.CDS**(*rdclass*, *rdtype*, *key_tag*, *algorithm*, *digest_type*, *digest*) [source]

CDS record

key_tag

An `int`, the key tag.

algorithm

An `int`, the algorithm used for the key.

digest_type

An `int`, the digest type.

digest

A `bytes`, the digest of the key.

class **dns.rdtypes.ANY.CERT.CERT**(*rdclass*, *rdtype*, *certificate_type*, *key_tag*, *algorithm*, *certificate*) [source]

CERT record

certificate_type

An `int`, the certificate type.

key_tag

An `int`, the key tag.

algorithm

An `int`, the algorithm.

certificate

A `bytes`, the certificate or CRL.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.CNAME.CNAME**(*rdclass*, *rdtype*, *target*) [source]

CNAME record

Note: although CNAME is officially a singleton type, dnspython allows non-singleton CNAME rdatasets because such sets have been commonly used by BIND and other nameservers for load balancing.

target

A `dns.name.Name`, the target name.

class **dns.rdtypes.ANY.CSYNC.CSYNC**(*rdclass*, *rdtype*, *serial*, *flags*, *windows*) [source]

CSYNC record

serial

An `int`, the SOA serial number.

flags

An `int`, the CSYNC flags.

windows

A tuple of `(int, bytes)` tuples.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.DLV.DLV**(*rdclass*, *rdtype*, *key_tag*, *algorithm*, *digest_type*, *digest*) [source]

DLV record

key_tag

An `int`, the key tag.

algorithm

An `int`, the algorithm used for the key.

digest_type

An `int`, the digest type.

digest

A `bytes`, the digest of the key.

class **dns.rdtypes.ANY.DNAME.DNAME**(*rdclass*, *rdtype*, *target*) [source]

DNAME record

target

A `dns.name.Name`, the target name.

class **dns.rdtypes.ANY.DNSKEY.DNSKEY**(*rdclass*, *rdtype*, *flags*, *protocol*, *algorithm*, *key*) [source]

DNSKEY record

flags

An `int`, the key's flags.

protocol

An `int`, the protocol for which this key may be used.

algorithm:

An `int`, the algorithm used for the key.

key

A `bytes`, the public key.

class **dns.rdtypes.ANY.DS.DS**(*rdclass*, *rdtype*, *key_tag*, *algorithm*, *digest_type*, *digest*) [source]

DS record

key_tag

An `int`, the key tag.

algorithm

An `int`, the algorithm used for the key.

digest_type

An `int`, the digest type.

digest

A `bytes`, the digest of the key.

class **dns.rdtypes.ANY.EUI48.EUI48**(*rdclass*, *rdtype*, *eui*) [source]

EUI48 record

eui

A `bytes`, 48-bit Extended Unique Identifier (EUI-48).

class **dns.rdtypes.ANY.EUI64.EUI64**(*rdclass*, *rdtype*, *eui*) [source]

EUI64 record

eui

A `bytes`, 64-bit Extended Unique Identifier (EUI-64).

class **dns.rdtypes.ANY.GPOS.GPOS**(*rdclass*, *rdtype*, *latitude*, *longitude*, *altitude*) [source]

GPOS record

latitude

A `bytes`, the latitude

longitude

A `bytes`, the longitude

altitude

A `bytes`, the altitude

property **float_altitude**

altitude as a floating point value

property **float_latitude**

latitude as a floating point value

property **float_longitude**

longitude as a floating point value

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.HINFO.HINFO**(*rdclass*, *rdtype*, *cpu*, *os*) [source]

HINFO record

cpu

A `bytes`, the CPU type.

os

A `bytes`, the OS type.

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.HIP.HIP**(*rdclass, rdtype, hit, algorithm, key, servers*) [source]

HIP record

hit

A `bytes`, the host identity tag.

algorithm

An `int`, the public key cryptographic algorithm.

key

A `bytes`, the public key.

servers

A tuple of `dns.name.Name` objects, the rendezvous servers.

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.ISDN.ISDN**(*rdclass, rdtype, address, subaddress*) [source]

ISDN record

address

A `bytes`, the ISDN address.

subaddress

A `bytes` the ISDN subaddress (or `b''` if not present).

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.L32.L32**(*rdclass*, *rdtype*, *preference*, *locator32*) [source]

L32 record

preference

An `int`, the preference value.

locator32

A `string`, the 32-bit locator value in the form of an IPv4 address.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.L64.L64**(*rdclass*, *rdtype*, *preference*, *locator64*) [source]

L64 record

preference

An `int`, the preference value.

locator64

A `string`, the 64-bit locator value in colon-separated-hex form.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.LOC.LOC**(*rdclass*, *rdtype*, *latitude*, *longitude*, *altitude*, *size=100.0*, *hprec=1000000.0*, *vprec=1000.0*) [source]

LOC record

latitude

An `(int, int, int, int, int)` tuple specifying the degrees, minutes, seconds, milliseconds, and sign of the latitude.

longitude

An `(int, int, int, int, int)` tuple specifying the degrees, minutes, seconds, milliseconds, and sign of the longitude.

altitude

A `float`, the altitude, in centimeters.

size

A `float`, the size of the sphere, in centimeters.

horizontal_precision

A `float`, the horizontal precision, in centimeters.

vertical_precision

A `float`, the vertical precision, in centimeters.

property **float_latitude**

latitude as a floating point value

property **float_longitude**

longitude as a floating point value

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.LP.LP**(*rdclass*, *rdtype*, *preference*, *fqdn*) [source]

LP record

preference

An `int`, the preference value.

fqdn

A `dns.name.Name`, the domain name of a locator.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.MX.MX**(*rdclass*, *rdtype*, *preference*, *exchange*) [source]

MX record

preference

An `int`, the preference value.

exchange

A `dns.name.Name`, the exchange name.

class **dns.rdtypes.ANY.NID.NID**(*rdclass*, *rdtype*, *preference*, *nodeid*) [source]

NID record

preference

An `int`, the preference value.

nodeid

A `string`, the 64-bit nodeid value in colon-separated-hex form.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.NINFO.NINFO**(*rdclass: RdataClass*, *rdtype: RdataType*, *strings: Iterable[bytes | str]*) [source]

NINFO record

strings

A tuple of `bytes`, the list of strings.

class **dns.rdtypes.ANY.NS.NS**(*rdclass*, *rdtype*, *target*) [source]

NS record

target

A `dns.name.Name`, the target name.

class **dns.rdtypes.ANY.NSEC.NSEC**(*rdclass*, *rdtype*, *next*, *windows*) [source]

NSEC record

next

A `dns.name.Name`, the next name

windows

A tuple of `(int, bytes)` tuples.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.NSEC3.NSEC3**(*rdclass*, *rdtype*, *algorithm*, *flags*, *iterations*, *salt*, *next*, *windows*) [source]

NSEC3 record

algorithm:

An `int`, the algorithm used for the hash.

flags:

An `int`, the flags.

iterations:

An `int`, the number of iterations.

salt

A `bytes`, the salt.

next

A `bytes`, the next name hash.

windows

A tuple of `(int, bytes)` tuples.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.NSEC3PARAM.NSEC3PARAM**(*rdclass*, *rdtype*, *algorithm*, *flags*, *iterations*, *salt*) [source]

NSEC3PARAM record

algorithm:

An `int`, the algorithm used for the hash.

flags:

An `int`, the flags.

iterations:

An `int`, the number of iterations.

salt

A `bytes`, the salt.

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.OPENPGPKEY.OPENPGPKEY**(*rdclass, rdtype, key*) [source]

OPENPGPKEY record

key

A `bytes`, the key.

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.PTR.PTR**(*rdclass, rdtype, target*) [source]

PTR record

target

A `dns.name.Name`, the target name.

class **dns.rdtypes.ANY.RP.RP**(*rdclass, rdtype, mbox, txt*) [source]

RP record

mbox

A `dns.name.Name`, the responsible person's mailbox.

txt

A `dns.name.Name`, the owner name of a node with TXT records, or the root name if no TXT records are associated with this RP.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.RRSIG.RRSIG**(*rdclass*, *rdtype*, *type_covered*, *algorithm*, *labels*, *original_ttl*, *expiration*, *inception*, *key_tag*, *signer*, *signature*) [source]

RRSIG record

type_covered

An `int`, the rdata type this signature covers.

algorithm

An `int`, the algorithm used for the signature.

labels

An `int`, the number of labels.

original_ttl

An `int`, the original TTL.

expiration

An *int*, the signature expiration time.

inception

An *int*, the signature inception time.

key_tag

An *int*, the key tag.

signer

A `dns.name.Name`, the signer.

signature

A `bytes`, the signature.

covers() [source]

Return the type a Rdata covers.

DNS SIG/RRSIG rdatas apply to a specific type; this type is returned by the covers() function. If the rdata type is not SIG or RRSIG, dns.rdatatype.NONE is returned. This is useful when creating rdatasets, allowing the rdataset to contain only RRSIGs of a particular type, e.g. RRSIG(NS).

Returns a `dns.rdatatype.RdataType`.

`to_text`(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **`dns.rdtypes.ANY.RT.RT`**(*rdclass, rdtype, preference, exchange*) [source]

RT record

`preference`

An `int`, the preference value.

`exchange`

A `dns.name.Name`, the exchange name.

class **`dns.rdtypes.ANY.SMIMEA.SMIMEA`**(*rdclass, rdtype, usage, selector, mtype, cert*) [source]

SMIMEA record

`usage`

An `int`, the certificate usage.

`selector`

An `int`, the selector.

`mtype`

An `int`, the matching type.

`cert`

A `bytes`, the certificate association data.

class **`dns.rdtypes.ANY.SOA.SOA`**(*rdclass, rdtype, mname, rname, serial, refresh, retry, expire, minimum*) [source]

SOA record

`mname`

A `dns.name.Name`, the MNAME (master name).

`rname`

A `dns.name.Name`, the RNAME (responsible name).

`serial`

An `int`, the zone's serial number.

`refresh`

An `int`, the zone's refresh value (in seconds).

`retry`

An `int`, the zone's retry value (in seconds).

`expire`

An `int`, the zone's expiration value (in seconds).

`minimum`

An `int`, the zone's negative caching time (in seconds, called "minimum" for historical reasons).

`to_text`(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **`dns.rdtypes.ANY.SPF.SPF`**(*rdclass: RdataClass, rdtype: RdataType, strings: Iterable[bytes | str]*) [source]

SPF record

`strings`

A tuple of `bytes`, the list of strings.

class **`dns.rdtypes.ANY.SSHFP.SSHFP`**(*rdclass, rdtype, algorithm, fp_type, fingerprint*) [source]

SSHFP record

`algorithm`

An `int`, the algorithm.

`fp_type`

An `int`, the digest type.

`fingerprint`

A `bytes`, the fingerprint.

`to_text`(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **`dns.rdtypes.ANY.TLSA.TLSA`**(*rdclass, rdtype, usage, selector, mtype, cert*) [source]

TLSA record

`usage`

An `int`, the certificate usage.

`selector`

An `int`, the selector.

`mtype`

An `int`, the matching type.

`cert`

A `bytes`, the certificate association data.

class **`dns.rdtypes.ANY.TXT.TXT`**(*rdclass: RdataClass, rdtype: RdataType, strings: Iterable[bytes | str]*) [source]

TXT record

`strings`

A tuple of `bytes`, the list of strings.

class **`dns.rdtypes.ANY.URI.URI`**(*rdclass, rdtype, priority, weight, target*) [source]

URI record

`priority`

An `int`, the priority.

`weight`

An `int`, the weight.

target

A `dns.name.Name`, the target.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.ANY.X25.X25**(*rdclass*, *rdtype*, *address*) [source]

X25 record

address

A `bytes`, the PSDN address.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

Types specific to class IN

class **dns.rdtypes.IN.A.A**(*rdclass*, *rdtype*, *address*) [source]

A record.

address

A `str`, an IPv4 address in the standard “dotted quad” text format.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.AAAA.AAAA**(*rdclass*, *rdtype*, *address*) [source]

AAAA record.

address

A `str`, an IPv6 address in the standard text format.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.APL.APLItem**(**args*, ***kwargs*) [source]

family

An `int`, the address family (in the IANA address family registry).

negation

A `bool`, is this item negated?

address

A `str`, the address.

prefix

An `int`, the prefix length.

class **dns.rdtypes.IN.APL.APL**(*rdclass*, *rdtype*, *items*) [source]

APL record.

items

A tuple of `dns.rdtypes.IN.APL.APLItem`.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.DHCID.DHCID**(*rdclass*, *rdtype*, *data*) [source]

DHCID record

data

A `bytes`, the data (the content of the RR is opaque as far as the DNS is concerned).

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.HTTPS.HTTPS**(*rdclass*, *rdtype*, *priority*, *target*, *params*) [source]

HTTPS record

priority

An `int`, the unsigned 16-bit integer priority.

target

A `dns.name.Name`, the target name.

params

A `dict[dns.rdtypes.svcbbase.ParamKey, dns.rdtypes.svcbbase.Param]`, the parameters. See the dedicated section SVCB and HTTPS Parameter Classes below for more information on the parameter types.

class **dns.rdtypes.IN.IPSECKEY.IPSECKEY**(*rdclass*, *rdtype*, *precedence*, *gateway_type*, *algorithm*, *gateway*, *key*) [source]

IPSECKEY record

precedence

An `int`, the precedence for the key data.

prefix

An `int`, the prefix length.

gateway_type

An `int`, the gateway type.

algorithm

An `int`, the algorithm to use.

gateway

The gateway. This value may be `None`, a `str` with an IPv4 or IPV6 address, or a ``dns.name.Name`.

key

A `bytes`, the public key.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.KX.KX**(*rdclass*, *rdtype*, *preference*, *exchange*) [source]

KX record

preference

An `int`, the preference value.

exchange

A `dns.name.Name`, the exchange name.

class **dns.rdtypes.IN.NAPTR.NAPTR**(*rdclass*, *rdtype*, *order*, *preference*, *flags*, *service*, *regexp*, *replacement*) [source]

NAPTR record

order

An `int`, the order.

preference

An `int`, the preference.

flags

A `bytes`, the flags.

service

A `bytes`, the service.

regexp

A `bytes`, the regular expression.

replacement

A `dns.name.Name`, the replacement name.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.NSAP.NSAP**(*rdclass*, *rdtype*, *address*) [source]

NSAP record.

address

A `bytes`, a NSAP address.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.NSAP_PTR.NSAP_PTR**(*rdclass*, *rdtype*, *target*) [source]

NSAP-PTR record

target

A `dns.name.Name`, the target name.

class **dns.rdtypes.IN.PX.PX**(*rdclass*, *rdtype*, *preference*, *map822*, *mapx400*) [source]

PX record.

preference

An `int`, the preference value.

map822

A `dns.name.Name`, the map822 name.

mapx400

A `dns.name.Name`, the mapx400 name.

to_text(*origin=None*, *relativize=True*, ***kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.SRV.SRV**(*rdclass*, *rdtype*, *priority*, *weight*, *port*, *target*) [source]

SRV record

priority

An `int`, the priority.

weight

An `int`, the weight.

port

An `int`, the port.

target

A `dns.name.Name`, the target host.

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

class **dns.rdtypes.IN.SVCB.SVCB**(*rdclass, rdtype, priority, target, params*) [source]

SVCB record

priority

An `int`, the unsigned 16-bit integer priority.

target

A `dns.name.Name`, the target name.

params

A `dict[dns.rdtypes.svcbbase.ParamKey, dns.rdtypes.svcbbase.Param]`, the parameters. See the dedicated section SVCB and HTTPS Parameter Classes below for more information on the parameter types.

class **dns.rdtypes.IN.WKS.WKS**(*rdclass, rdtype, address, protocol, bitmap*) [source]

WKS record

address

A `str`, the address.

protocol

An `int`, the protocol.

bitmap

A `bytes`, the bitmap.

to_text(*origin=None, relativize=True, **kw*) [source]

Convert an rdata to text format.

Returns a `str`.

SVCB and HTTPS Parameter Classes

class **dns.rdtypes.svcbbase.ParamKey**(*value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*) [source]

SVCB ParamKey

ALPN

ECH

IPV4HINT

IPV6HINT

MANDATORY

NO_DEFAULT_ALPN

PORT

class **dns.rdtypes.svcbbase.Param**(**args*, ***kwargs*) [source]

class **dns.rdtypes.svcbbase.GenericParam**(*value*) [source]

Generic SVCB parameter

value

A `bytes`, the value of the parameter.

class **dns.rdtypes.svcbbase.MandatoryParam**(*keys*) [source]

keys

A tuple of `ParamKey`, the keys which are mandatory.

class **dns.rdtypes.svcbbase.ALPNParam**(*ids*) [source]

ids

A tuple of `bytes` values, the APLN ids.

class **dns.rdtypes.svcbbase.PortParam**(*port*) [source]

port

An `int`, the unsigned 16-bit integer port.

class **dns.rdtypes.svcbbase.IPv4HintParam**(*addresses*) [source]

addresses

A tuple of `string`, which each string is an IPv4 address.

class **dns.rdtypes.svcbbase.IPv6HintParam**(*addresses*) [source]

addresses

A tuple of `string`, which each string is an IPv6 address.

class **dns.rdtypes.svcbbase.ECHParam**(*ech*) [source]

ech

A `bytes`.

Rdataset, RRset and Node Classes

An `Rdataset` is a set of `Rdata` objects which all have the same rdatatype, rdataclass, and covered type. `Rdatasets` also have a `ttl` (DNS time-to-live) field. Rdatasets support the normal Python set API, but are also ordered.

An `RRset` is a subclass of `Rdataset` that also has an owner name, i.e. a `dns.name.Name` that says where in the DNS tree this set is located.

A `Node` is a set of `Rdataset` objects, the Rdatasets being interpreted as at the same place (i.e. same owner name) int the DNS tree. Nodes are primarily used in `Zone` objects.

class **dns.rdataset.Rdataset**(*rdclass: RdataClass, rdtype: RdataType, covers: RdataType = RdataType.TYPE0, ttl: int = 0*) [source]

A DNS rdataset.

Create a new rdataset of the specified class and type.

rdclass, a `dns.rdataclass.RdataClass`, the rdataclass.

rdtype, an `dns.rdatatype.RdataType`, the rdatatype.

covers, an `dns.rdatatype.RdataType`, the covered rdatatype.

ttl, an `int`, the TTL.

add(*rd: Rdata, ttl: int | None = None*)→ None [source]

Add the specified rdata to the rdataset.

If the optional *ttl* parameter is supplied, then `self.update_ttl(ttl)` will be called prior to adding the rdata.

rd, a `dns.rdata.Rdata`, the rdata

ttl, an `int`, the TTL.

Raises `dns.rdataset.IncompatibleTypes` if the type and class do not match the type and class of the rdataset.

Raises `dns.rdataset.DifferingCovers` if the type is a signature type and the covered type does not match that of the rdataset.

intersection_update(*other*) [source]

Update the set, removing any elements from other which are not in both sets.

match(*rdclass: RdataClass, rdtype: RdataType, covers: RdataType*) → bool [source]

Returns `True` if this rdataset matches the specified class, type, and covers.

processing_order() → List[Rdata] [source]

Return rdatas in a valid processing order according to the type's specification. For example, MX records are in preference order from lowest to highest preferences, with items of the same preference shuffled.

For types that do not define a processing order, the rdatas are simply shuffled.

to_text(*name: Name | None = None, origin: Name | None = None, relativize: bool = True, override_rdclass: RdataClass | None = None, want_comments: bool = False, **kw: Dict[str, Any]*) → str [source]

Convert the rdataset into DNS zone file format.

See `dns.name.Name.choose_relativity` for more information on how *origin* and *relativize* determine the way names are emitted.

Any additional keyword arguments are passed on to the rdata `to_text()` method.

name, a `dns.name.Name`. If name is not `None`, emit RRs with *name* as the owner name.

origin, a `dns.name.Name` or `None`, the origin for relative names.

relativize, a `bool`. If `True`, names will be relativized to *origin*.

override_rdclass, a `dns.rdataclass.RdataClass` or `None`. If not `None`, use this class instead of the Rdataset's class.

want_comments, a `bool`. If `True`, emit comments for rdata which have them. The default is `False`.

to_wire(*name: Name, file: Any, compress: Dict[Name, int] | None = None, origin: Name | None = None, override_rdclass: RdataClass | None = None, want_shuffle: bool = True*) → int [source]

Convert the rdataset to wire format.

name, a `dns.name.Name` is the owner name to use.

file is the file where the name is emitted (typically a BytesIO file).

compress, a `dict`, is the compression table to use. If `None` (the default), names will not be compressed.

origin is a `dns.name.Name` or `None`. If the name is relative and origin is not `None`, then *origin* will be appended to it.

override_rdclass, an `int`, is used as the class instead of the class of the rdataset. This is useful when rendering rdatasets associated with dynamic updates.

want_shuffle, a `bool`. If `True`, then the order of the Rdatas within the Rdataset will be shuffled before rendering.

Returns an `int`, the number of records emitted.

union_update(*other*) [source]

Update the set, adding any elements from other which are not already in the set.

update(*other*) [source]

Add all rdatas in other to self.

other, a `dns.rdataset.Rdataset`, the rdataset from which to update.

update_ttl(*ttl: int*) → None [source]

Perform TTL minimization.

Set the TTL of the rdataset to be the lesser of the set's current TTL or the specified TTL. If the set contains no rdatas, set the TTL to the specified TTL.

ttl, an `int` or `str`.

class **dns.rrset.RRset**(*name: Name, rdclass: RdataClass, rdtype: RdataType, covers: RdataType = RdataType.TYPE0, deleting: RdataClass | None = None*) [source]

A DNS RRset (named rdataset).

RRset inherits from Rdataset, and RRsets can be treated as Rdatasets in most cases. There are, however, a few notable exceptions. RRsets have different to_wire() and to_text() method arguments, reflecting the fact that RRsets always have an owner name.

Create a new RRset.

full_match(*name: Name, rdclass: RdataClass, rdtype: RdataType, covers: RdataType, deleting: RdataClass | None = None*) → bool [source]

Returns `True` if this rrset matches the specified name, class, type, covers, and deletion state.

match(**args: Any, **kwargs: Any*) → bool [source]

Does this rrset match the specified attributes?

Behaves as `full_match()` if the first argument is a `dns.name.Name`, and as `dns.rdataset.Rdataset.match()` otherwise.

(This behavior fixes a design mistake where the signature of this method became incompatible with that of its superclass. The fix makes RRsets matchable as Rdatasets while preserving backwards compatibility.)

to_rdataset()→ Rdataset [source]

Convert an RRset into an Rdataset.

Returns a `dns.rdataset.Rdataset`.

to_text(*origin: Name | None = None, relativize: bool = True, **kw: Dict[str, Any]*)→ str [source]

Convert the RRset into DNS zone file format.

See `dns.name.Name.choose_relativity` for more information on how *origin* and *relativize* determine the way names are emitted.

Any additional keyword arguments are passed on to the rdata `to_text()` method.

origin, a `dns.name.Name` or `None`, the origin for relative names.

relativize, a `bool`. If `True`, names will be relativized to *origin*.

to_wire(*file: Any, compress: Dict[Name, int] | None = None, origin: Name | None = None, **kw: Dict[str, Any]*)→ int [source]

Convert the RRset to wire format.

All keyword arguments are passed to `dns.rdataset.to_wire()`; see that function for details.

Returns an `int`, the number of records emitted.

class **dns.node.Node** [source]

A Node is a set of rdatasets.

A node is either a CNAME node or an “other data” node. A CNAME node contains only CNAME, KEY, NSEC, and NSEC3 rdatasets along with their covering RRSIG rdatasets. An “other data” node contains any rdataset other than a CNAME or RRSIG(CNAME) rdataset. When changes are made to a node, the CNAME or “other data” state is always consistent with the update, i.e. the most recent change wins. For example, if you have a node which contains a CNAME rdataset, and then add an MX rdataset to it, then the CNAME rdataset will be deleted. Likewise if you have a node containing an MX rdataset and add a CNAME rdataset, the MX rdataset will be deleted.

classify()→ NodeKind [source]

Classify a node.

A node which contains a CNAME or RRSIG(CNAME) is a `NodeKind.CNAME` node.

A node which contains only “neutral” types, i.e. types allowed to co-exist with a CNAME, is a `NodeKind.NEUTRAL` node. The neutral types are NSEC, NSEC3, KEY, and their associated RRSIGS. An empty node is also considered neutral.

A node which contains some rdataset which is not a CNAME, RRSIG(CNAME), or a neutral type is a a `NodeKind.REGULAR` node. Regular nodes are also commonly referred to as “other data”.

`delete_rdataset`(*rdclass: RdataClass, rdtype: RdataType, covers: RdataType = RdataType.TYPE0*) → None [source]

Delete the rdataset matching the specified properties in the current node.

If a matching rdataset does not exist, it is not an error.

rdclass, an `int`, the class of the rdataset.

rdtype, an `int`, the type of the rdataset.

covers, an `int`, the covered type.

`find_rdataset`(*rdclass: RdataClass, rdtype: RdataType, covers: RdataType = RdataType.TYPE0, create: bool = False*) → Rdataset [source]

Find an rdataset matching the specified properties in the current node.

rdclass, a `dns.rdataclass.RdataClass`, the class of the rdataset.

rdtype, a `dns.rdatatype.RdataType`, the type of the rdataset.

covers, a `dns.rdatatype.RdataType`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If True, create the rdataset if it is not found.

Raises `KeyError` if an rdataset of the desired type and class does not exist and *create* is not `True`.

Returns a `dns.rdataset.Rdataset`.

`get_rdataset`(*rdclass: RdataClass, rdtype: RdataType, covers: RdataType = RdataType.TYPE0, create: bool = False*) → Rdataset | None [source]

Get an rdataset matching the specified properties in the current node.

None is returned if an rdataset of the specified type and class does not exist and *create* is not `True`.

rdclass, an `int`, the class of the rdataset.

rdtype, an `int`, the type of the rdataset.

covers, an `int`, the covered type. Usually this value is dns.rdatatype.NONE, but if the rdtype is dns.rdatatype.SIG or dns.rdatatype.RRSIG, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If True, create the rdataset if it is not found.

Returns a `dns.rdataset.Rdataset` or `None`.

`replace_rdataset`(*replacement: Rdataset*) → None [source]

Replace an rdataset.

It is not an error if there is no rdataset matching *replacement*.

Ownership of the *replacement* object is transferred to the node; in other words, this method does not store a copy of *replacement* at the node, it stores *replacement* itself.

replacement, a `dns.rdataset.Rdataset`.

Raises `ValueError` if *replacement* is not a `dns.rdataset.Rdataset`.

`to_text`(*name: Name, **kw: Dict[str, Any]*) → str [source]

Convert a node to text format.

Each rdataset at the node is printed. Any keyword arguments to this method are passed on to the rdataset's to_text() method.

name, a `dns.name.Name`, the owner name of the rdatasets.

Returns a `str`.

Making DNS Rdatasets and RRsets

dns.rdataset.from_text(*rdclass: RdataClass | str, rdtype: RdataType | str, ttl: int, *text_rdatas: Any*) → Rdataset [source]

Create an rdataset with the specified class, type, and TTL, and with the specified rdatas in text format.

Returns a `dns.rdataset.Rdataset` object.

dns.rdataset.from_text_list(*rdclass: RdataClass | str, rdtype: RdataType | str, ttl: int, text_rdatas: Collection[str], idna_codec: IDNACodec | None = None, origin: Name | None = None, relativize: bool = True, relativize_to: Name | None = None*) → Rdataset [source]

Create an rdataset with the specified class, type, and TTL, and with the specified list of rdatas in text format.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder to use; if `None`, the default IDNA 2003 encoder/decoder is used.

origin, a `dns.name.Name` (or `None`), the origin to use for relative names.

relativize, a `bool`. If true, name will be relativized.

relativize_to, a `dns.name.Name` (or `None`), the origin to use when relativizing names. If not set, the *origin* value will be used.

Returns a `dns.rdataset.Rdataset` object.

dns.rdataset.from_rdata(*ttl: int, *rdatas: Any*) → Rdataset [source]

Create an rdataset with the specified TTL, and with the specified rdata objects.

Returns a `dns.rdataset.Rdataset` object.

dns.rdataset.from_rdata_list(*ttl: int, rdatas: Collection[Rdata]*) → Rdataset [source]

Create an rdataset with the specified TTL, and with the specified list of rdata objects.

Returns a `dns.rdataset.Rdataset` object.

dns.rrset.from_text(*name: Name | str, ttl: int, rdclass: RdataClass | str, rdtype: RdataType | str, *text_rdatas: Any*) → RRset [source]

Create an RRset with the specified name, TTL, class, and type and with the specified rdatas in text format.

Returns a `dns.rrset.RRset` object.

`dns.rrset.from_text_list`(*name: Name | str, ttl: int, rdclass: RdataClass | str, rdtype: RdataType | str, text_rdatas: Collection[str], idna_codec: IDNACodec | None = None, origin: Name | None = None, relativize: bool = True, relativize_to: Name | None = None*)→ RRset [source]

Create an RRset with the specified name, TTL, class, and type, and with the specified list of rdatas in text format.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder to use; if `None`, the default IDNA 2003 encoder/decoder is used.

origin, a `dns.name.Name` (or `None`), the origin to use for relative names.

relativize, a `bool`. If true, name will be relativized.

relativize_to, a `dns.name.Name` (or `None`), the origin to use when relativizing names. If not set, the *origin* value will be used.

Returns a `dns.rrset.RRset` object.

`dns.rrset.from_rdata`(*name: Name | str, ttl: int, *rdatas: Any*)→ RRset [source]

Create an RRset with the specified name and TTL, and with the specified rdata objects.

Returns a `dns.rrset.RRset` object.

`dns.rrset.from_rdata_list`(*name: Name | str, ttl: int, rdatas: Collection[Rdata], idna_codec: IDNACodec | None = None*)→ RRset [source]

Create an RRset with the specified name and TTL, and with the specified list of rdata objects.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder to use; if `None`, the default IDNA 2003 encoder/decoder is used.

Returns a `dns.rrset.RRset` object.

The dns.message.Message Class

This is the base class for all messages, and the class used for any DNS opcodes that do not have a more specific class.

class **dns.message.Message**(*id: int | None = None*) [source]

A DNS message.

id

An `int`, the query id; the default is a randomly chosen id.

flags

An `int`, the DNS flags of the message.

sections

A list of lists of `dns.rrset.RRset` objects.

edns

An `int`, the EDNS level to use. The default is -1, no EDNS.

ednsflags

An `int`, the EDNS flags.

payload

An `int`, the EDNS payload size. The default is 0.

options

The EDNS options, a list of `dns.edns.Option` objects. The default is the empty list.

request_payload

The associated request's EDNS payload size. This field is meaningful in response messages, and if set to a non-zero value, will limit the size of the response to the specified size. The default is 0, which means "use the default limit" which is currently 65535.

keyring

A `dns.tsig.Key`, the TSIG key. The default is None.

keyname

The TSIG keyname to use, a `dns.name.Name`. The default is `None`.

keyalgorithm

A `dns.name.Name`, the TSIG algorithm to use. Defaults to `dns.tsig.default_algorithm`. Constants for TSIG algorithms are defined the in `dns.tsig` module.

request_mac

A `bytes`, the TSIG MAC of the request message associated with this message; used when validating TSIG signatures.

fudge

An `int`, the TSIG time fudge. The default is 300 seconds.

original_id

An `int`, the TSIG original id; defaults to the message's id.

tsig_error

An `int`, the TSIG error code. The default is 0.

other_data

A `bytes`, the TSIG "other data". The default is the empty `bytes`.

mac

A `bytes`, the TSIG MAC for this message.

xfr

A `bool`. This attribute is true when the message being used for the results of a DNS zone transfer. The default is `False`.

origin

A `dns.name.Name`. The origin of the zone in messages which are used for zone transfers or for DNS dynamic updates. The default is `None`.

tsig_ctx

An `hmac.HMAC`, the TSIG signature context associated with this message. The default is `None`.

had_tsig

A `bool`, which is `True` if the message had a TSIG signature when it was decoded from wire format.

multi

A `bool`, which is `True` if this message is part of a multi-message sequence. The default is `False`. This attribute is used when validating TSIG signatures on messages which are part of a zone transfer.

first

A `bool`, which is `True` if this message is stand-alone, or the first of a multi-message sequence. The default is `True`. This variable is used when validating TSIG signatures on messages which are part of a zone transfer.

index

A `dict`, an index of RRsets in the message. The index key is `(section, name, rdclass, rdtype, covers, deleting)`. The default is `{}`. Indexing improves the performance of finding RRsets. Indexing can be disabled by setting the index to `None`.

property **additional**: *List[RRset]*

The additional data section.

property **answer**: *List[RRset]*

The answer section.

property **authority**: *List[RRset]*

The authority section.

find_rrset(*section: int | str | List[RRset], name: Name, rdclass: RdataClass, rdtype: RdataType, covers: RdataType = RdataType.TYPE0, deleting: RdataClass | None = None, create: bool = False, force_unique: bool = False, idna_codec: IDNACodec | None = None*)→ RRset [source]

Find the RRset with the given attributes in the specified section.

section, an `int` section number, a `str` section name, or one of the section attributes of this message. This specifies the the section of the message to search. For example:

```
my_message.find_rrset(my_message.answer, name, rdclass, rdtype)
my_message.find_rrset(dns.message.ANSWER, name, rdclass, rdtype)
my_message.find_rrset("ANSWER", name, rdclass, rdtype)
```

name, a `dns.name.Name` or `str`, the name of the RRset.

rdclass, an `int` or `str`, the class of the RRset.

rdtype, an `int` or `str`, the type of the RRset.

covers, an `int` or `str`, the covers value of the RRset. The default is `dns.rdatatype.NONE`.

deleting, an `int`, `str`, or `None`, the deleting value of the RRset. The default is `None`.

create, a `bool`. If `True`, create the RRset if it is not found. The created RRset is appended to *section*.

force_unique, a `bool`. If `True` and *create* is also `True`, create a new RRset regardless of whether a matching RRset exists already. The default is `False`. This is useful when creating DDNS Update messages, as order matters for them.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

Raises `KeyError` if the RRset was not found and create was `False`.

Returns a `dns.rrset.RRset object`.

`get_rrset`(*section: int | str | List[RRset], name: Name, rdclass: RdataClass, rdtype: RdataType, covers: RdataType = RdataType.TYPE0, deleting: RdataClass | None = None, create: bool = False, force_unique: bool = False, idna_codec: IDNACodec | None = None*)→ RRset | None [source]

Get the RRset with the given attributes in the specified section.

If the RRset is not found, None is returned.

section, an `int` section number, a `str` section name, or one of the section attributes of this message. This specifies the the section of the message to search. For example:

```
my_message.get_rrset(my_message.answer, name, rdclass, rdtype)
my_message.get_rrset(dns.message.ANSWER, name, rdclass, rdtype)
my_message.get_rrset("ANSWER", name, rdclass, rdtype)
```

name, a `dns.name.Name` or `str`, the name of the RRset.

rdclass, an `int` or `str`, the class of the RRset.

rdtype, an `int` or `str`, the type of the RRset.

covers, an `int` or `str`, the covers value of the RRset. The default is `dns.rdatatype.NONE`.

deleting, an `int`, `str`, or `None`, the deleting value of the RRset. The default is `None`.

create, a `bool`. If `True`, create the RRset if it is not found. The created RRset is appended to *section*.

force_unique, a `bool`. If `True` and *create* is also `True`, create a new RRset regardless of whether a matching RRset exists already. The default is `False`. This is useful when creating DDNS Update messages, as order matters for them.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

Returns a `dns.rrset.RRset object` or `None`.

is_response(*other: Message*)→ bool [source]

Is *other*, also a `dns.message.Message`, a response to this message?

Returns a `bool`.

opcode()→ Opcode [source]

Return the opcode.

Returns a `dns.opcode.Opcode`.

property **question**: *List[RRset]*

The question section.

rcode()→ Rcode [source]

Return the rcode.

Returns a `dns.rcode.Rcode`.

section_count(*section: int | str | List[RRset]*)→ int [source]

Returns the number of records in the specified section.

section, an `int` section number, a `str` section name, or one of the section attributes of this message. This specifies the the section of the message to count. For example:

```
my_message.section_count(my_message.answer)
my_message.section_count(dns.message.ANSWER)
my_message.section_count("ANSWER")
```

section_from_number(*number: int*)→ List[RRset] [source]

Return the section list associated with the specified section number.

number is a section number *int* or the text form of a section name.

Raises `ValueError` if the section isn't known.

Returns a `list`.

`section_number`(*section: List[RRset]*) → int [source]

Return the "section number" of the specified section for use in indexing.

section is one of the section attributes of this message.

Raises `ValueError` if the section isn't known.

Returns an `int`.

`set_opcode`(*opcode: Opcode*) → None [source]

Set the opcode.

opcode, a `dns.opcode.Opcode`, is the opcode to set.

`set_rcode`(*rcode: Rcode*) → None [source]

Set the rcode.

rcode, a `dns.rcode.Rcode`, is the rcode to set.

`to_text`(*origin: Name | None = None, relativize: bool = True, **kw: Dict[str, Any]*) → str [source]

Convert the message to text.

The *origin*, *relativize*, and any other keyword arguments are passed to the RRset `to_wire()` method.

Returns a `str`.

`to_wire`(*origin: Name | None = None, max_size: int = 0, multi: bool = False, tsig_ctx: Any | None = None, prepend_length: bool = False, prefer_truncation: bool = False, **kw: Dict[str, Any]*) → bytes [source]

Return a string containing the message in DNS compressed wire format.

Additional keyword arguments are passed to the RRset `to_wire()` method.

origin, a `dns.name.Name` or `None`, the origin to be appended to any relative names. If `None`, and the message has an origin attribute that is not `None`, then it will be used.

max_size, an `int`, the maximum size of the wire format output; default is 0, which means "the message's request payload, if nonzero, or 65535".

multi, a `bool`, should be set to `True` if this message is part of a multiple message sequence.

tsig_ctx, a `dns.tsig.HMACTSig` or `dns.tsig.GSSTSig` object, the ongoing TSIG context, used when signing zone transfers.

prepend_length, a `bool`, should be set to `True` if the caller wants the message length prepended to the message itself. This is useful for messages sent over TCP, TLS (DoT), or QUIC (DoQ).

prefer_truncation, a `bool`, should be set to `True` if the caller wants the message to be truncated if it would otherwise exceed the maximum length. If the truncation occurs before the additional section, the TC bit will be set.

Raises `dns.exception.TooBig` if *max_size* was exceeded.

Returns a `bytes`.

use_edns(*edns: int | bool | None = 0, ednsflags: int = 0, payload: int = 1232, request_payload: int | None = None, options: List[Option] | None = None, pad: int = 0*)→ None [source]

Configure EDNS behavior.

edns, an `int`, is the EDNS level to use. Specifying `None`, `False`, or `-1` means "do not use EDNS", and in this case the other parameters are ignored. Specifying `True` is equivalent to specifying 0, i.e. "use EDNS0".

ednsflags, an `int`, the EDNS flag values.

payload, an `int`, is the EDNS sender's payload field, which is the maximum size of UDP datagram the sender can handle. I.e. how big a response to this message can be.

request_payload, an `int`, is the EDNS payload size to use when sending this message. If not specified, defaults to the value of *payload*.

options, a list of `dns.edns.Option` objects or `None`, the EDNS options.

pad, a non-negative `int`. If 0, the default, do not pad; otherwise add padding bytes to make the message size a multiple of *pad*. Note that if padding is non-zero, an EDNS PADDING option will always be added to the message.

use_tsig(*keyring: ~typing.Any, keyname: ~dns.name.Name | str | None = None, fudge: int = 300, original_id: int | None = None, tsig_error: int = 0, other_data: bytes = b'', algorithm: ~dns.name.Name | str = <DNS name hmac-sha256.>*)→ None [source]

When sending, a TSIG signature using the specified key should be added.

key, a `dns.tsig.Key` is the key to use. If a key is specified, the *keyring* and *algorithm* fields are not used.

keyring, a `dict`, `callable` or `dns.tsig.Key`, is either the TSIG keyring or key to use.

The format of a keyring dict is a mapping from TSIG key name, as `dns.name.Name` to `dns.tsig.Key` or a TSIG secret, a `bytes`. If a `dict` *keyring* is specified but a *keyname* is not, the key used will be the first key in the *keyring*. Note that the order of keys in a

dictionary is not defined, so applications should supply a keyname when a `dict` keyring is used, unless they know the keyring contains only one key. If a `callable` keyring is specified, the callable will be called with the message and the keyname, and is expected to return a key.

keyname, a `dns.name.Name`, `str` or `None`, the name of this TSIG key to use; defaults to `None`. If *keyring* is a `dict`, the key must be defined in it. If *keyring* is a `dns.tsig.Key`, this is ignored.

fudge, an `int`, the TSIG time fudge.

original_id, an `int`, the TSIG original id. If `None`, the message's id is used.

tsig_error, an `int`, the TSIG error code.

other_data, a `bytes`, the TSIG other data.

algorithm, a `dns.name.Name` or `str`, the TSIG algorithm to use. This is only used if *keyring* is a `dict`, and the key entry is a `bytes`.

want_dnssec(*wanted: bool = True*) → None [source]

Enable or disable 'DNSSEC desired' flag in requests.

wanted, a `bool`. If `True`, then DNSSEC data is desired in the response, EDNS is enabled if required, and then the DO bit is set. If `False`, the DO bit is cleared if EDNS is enabled.

The following constants may be used to specify sections in the `find_rrset()` and `get_rrset()` methods:

dns.message.QUESTION = *MessageSection.QUESTION*

Message sections

dns.message.ANSWER = *MessageSection.ANSWER*

Message sections

dns.message.AUTHORITY = *MessageSection.AUTHORITY*

Message sections

dns.message.ADDITIONAL = *MessageSection.ADDITIONAL*

Message sections

Making DNS Messages

dns.message.from_file(*f: Any, idna_codec: IDNACodec | None = None, one_rr_per_rrset: bool = False*)→ Message [source]

Read the next text format message from the specified file.

Message blocks are separated by a single blank line.

f, a `file` or `str`. If *f* is text, it is treated as the pathname of a file to open.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

one_rr_per_rrset, a `bool`. If `True`, then each RR is put into its own rrset. The default is `False`.

Raises `dns.message.UnknownHeaderField` if a header is unknown.

Raises `dns.exception.SyntaxError` if the text is badly formed.

Returns a `dns.message.Message object`

dns.message.from_text(*text: str, idna_codec: IDNACodec | None = None, one_rr_per_rrset: bool = False, origin: Name | None = None, relativize: bool = True, relativize_to: Name | None = None*)→ Message [source]

Convert the text format message into a message object.

The reader stops after reading the first blank line in the input to facilitate reading multiple messages from a single file with `dns.message.from_file()`.

text, a `str`, the text format message.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

one_rr_per_rrset, a `bool`. If `True`, then each RR is put into its own rrset. The default is `False`.

origin, a `dns.name.Name` (or `None`), the origin to use for relative names.

relativize, a `bool`. If true, name will be relativized.

relativize_to, a `dns.name.Name` (or `None`), the origin to use when relativizing names. If not set, the *origin* value will be used.

Raises `dns.message.UnknownHeaderField` if a header is unknown.

Raises `dns.exception.SyntaxError` if the text is badly formed.

Returns a `dns.message.Message object`

`dns.message.from_wire`(*wire: bytes, keyring: Any | None = None, request_mac: bytes | None = b'', xfr: bool = False, origin: Name | None = None, tsig_ctx: HMACTSig | GSSTSig | None = None, multi: bool = False, question_only: bool = False, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, raise_on_truncation: bool = False, continue_on_error: bool = False*)→ Message [source]

Convert a DNS wire format message into a message object.

keyring, a `dns.tsig.Key` or `dict`, the key or keyring to use if the message is signed.

request_mac, a `bytes` or `None`. If the message is a response to a TSIG-signed request, *request_mac* should be set to the MAC of that request.

xfr, a `bool`, should be set to `True` if this message is part of a zone transfer.

origin, a `dns.name.Name` or `None`. If the message is part of a zone transfer, *origin* should be the origin name of the zone. If not `None`, names will be relativized to the origin.

tsig_ctx, a `dns.tsig.HMACTSig` or `dns.tsig.GSSTSig` object, the ongoing TSIG context, used when validating zone transfers.

multi, a `bool`, should be set to `True` if this message is part of a multiple message sequence.

question_only, a `bool`. If `True`, read only up to the end of the question section.

one_rr_per_rrset, a `bool`. If `True`, put each RR into its own RRset.

ignore_trailing, a `bool`. If `True`, ignore trailing junk at end of the message.

raise_on_truncation, a `bool`. If `True`, raise an exception if the TC bit is set.

continue_on_error, a `bool`. If `True`, try to continue parsing even if errors occur. Erroneous rdata will be ignored. Errors will be accumulated as a list of MessageError objects in the message's `errors` attribute. This option is recommended only for DNS analysis tools, or for use in a server as part of an error handling path. The default is `False`.

Raises `dns.message.ShortHeader` if the message is less than 12 octets long.

Raises `dns.message.TrailingJunk` if there were octets in the message past the end of the proper DNS message, and *ignore_trailing* is `False`.

Raises `dns.message.BadEDNS` if an OPT record was in the wrong section, or occurred more than once.

Raises `dns.message.BadTSIG` if a TSIG record was not the last record of the additional data section.

Raises `dns.message.Truncated` if the TC flag is set and *raise_on_truncation* is `True`.

Returns a `dns.message.Message`.

`dns.message.make_query`(*qname: Name | str, rdtype: RdataType | str, rdclass: RdataClass | str = RdataClass.IN, use_edns: int | bool | None = None, want_dnssec: bool = False, ednsflags: int | None = None, payload: int | None = None, request_payload: int | None = None, options: List[Option] | None = None, idna_codec: IDNACodec | None = None, id: int | None = None, flags: int = Flag.RD, pad: int = 0*)→ QueryMessage [source]

Make a query message.

The query name, type, and class may all be specified either as objects of the appropriate type, or as strings.

The query will have a randomly chosen query id, and its DNS flags will be set to dns.flags.RD.

qname, a `dns.name.Name` or `str`, the query name.

rdtype, an `int` or `str`, the desired rdata type.

rdclass, an `int` or `str`, the desired rdata class; the default is class IN.

use_edns, an `int`, `bool` or `None`. The EDNS level to use; the default is `None`. If `None`, EDNS will be enabled only if other parameters (*ednsflags*, *payload*, *request_payload*, or *options*) are set. See the description of dns.message.Message.use_edns() for the possible values for use_edns and their meanings.

want_dnssec, a `bool`. If `True`, DNSSEC data is desired.

ednsflags, an `int`, the EDNS flag values.

payload, an `int`, is the EDNS sender's payload field, which is the maximum size of UDP datagram the sender can handle. I.e. how big a response to this message can be.

request_payload, an `int`, is the EDNS payload size to use when sending this message. If not specified, defaults to the value of *payload*.

options, a list of `dns.edns.Option` objects or `None`, the EDNS options.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

id, an `int` or `None`, the desired query id. The default is `None`, which generates a random query id.

flags, an `int`, the desired query flags. The default is `dns.flags.RD`.

pad, a non-negative `int`. If 0, the default, do not pad; otherwise add padding bytes to make the message size a multiple of *pad*. Note that if padding is non-zero, an EDNS PADDING option will always be added to the message.

Returns a `dns.message.QueryMessage`

dns.message.make_response(*query: Message, recursion_available: bool = False, our_payload: int = 8192, fudge: int = 300, tsig_error: int = 0, pad: int | None = None*) → Message [source]

Make a message which is a response for the specified query. The message returned is really a response skeleton; it has all of the infrastructure required of a response, but none of the content.

The response's question section is a shallow copy of the query's question section, so the query's question RRsets should not be changed.

query, a `dns.message.Message`, the query to respond to.

recursion_available, a `bool`, should RA be set in the response?

our_payload, an `int`, the payload size to advertise in EDNS responses.

fudge, an `int`, the TSIG time fudge.

tsig_error, an `int`, the TSIG error.

pad, a non-negative `int` or `None`. If 0, the default, do not pad; otherwise if not `None` add padding bytes to make the message size a multiple of *pad*. Note that if padding is non-zero, an EDNS PADDING option will always be added to the message. If `None`, add padding following RFC 8467, namely if the request is padded, pad the response to 468 otherwise do not pad.

Returns a `dns.message.Message` object whose specific class is appropriate for the query. For example, if query is a `dns.update.UpdateMessage`, response will be too.

Message Flags

DNS message flags are used for signalling of various kinds in the DNS protocol. For example, the `QR` flag indicates that a message is a response to a prior query.

Messages flags are encoded in two locations: the DNS header and the EDNS flags field.

Header Flags

dns.flags.QR= *Flag.QR*

dns.flags.AA= *Flag.AA*

dns.flags.TC= *Flag.TC*

dns.flags.RD= *Flag.RD*

dns.flags.RA= *Flag.RA*

dns.flags.AD= *Flag.AD*

dns.flags.CD= *Flag.CD*

dns.flags.from_text(*text: str*)→ int [source]

Convert a space-separated list of flag text values into a flags value.

Returns an `int`

dns.flags.to_text(*flags: int*)→ str [source]

Convert a flags value into a space-separated list of flag text values.

Returns a `str`.

EDNS Flags

dns.flags.DO= *EDNSFlag.DO*

dns.flags.edns_from_text(*text: str*) → int [source]

Convert a space-separated list of EDNS flag text values into a EDNS flags value.

Returns an `int`

dns.flags.edns_to_text(*flags: int*) → str [source]

Convert an EDNS flags value into a space-separated list of EDNS flag text values.

Returns a `str`.

Message Opcodes

DNS Opcodes describe what kind of operation a DNS message is requesting or replying to. Opcodes are embedded in the flags field in the DNS header.

dns.opcode.QUERY = *Opcode.QUERY*

dns.opcode.IQUERY = *Opcode.IQUERY*

dns.opcode.STATUS = *Opcode.STATUS*

dns.opcode.NOTIFY = *Opcode.NOTIFY*

dns.opcode.UPDATE = *Opcode.UPDATE*

dns.opcode.from_text(*text: str*) → Opcode [source]

Convert text into an opcode.

text, a `str`, the textual opcode

Raises `dns.opcode.UnknownOpcode` if the opcode is unknown.

Returns an `int`.

dns.opcode.to_text(*value: Opcode*) → str [source]

Convert an opcode to text.

value, an `int` the opcode value,

Raises `dns.opcode.UnknownOpcode` if the opcode is unknown.

Returns a `str`.

dns.opcode.from_flags(*flags: int*) → Opcode [source]

Extract an opcode from DNS message flags.

flags, an `int`, the DNS flags.

Returns an `int`.

dns.opcode.to_flags(value: Opcode) → int [source]

Convert an opcode to a value suitable for ORing into DNS message flags.

value, an `int`, the DNS opcode value.

Returns an `int`.

dns.opcode.is_update(flags: int) → bool [source]

Is the opcode in flags UPDATE?

flags, an `int`, the DNS message flags.

Returns a `bool`.

Message Rcodes

A DNS Rcode describes the result of a DNS request. If EDNS is not in use, then the rcode is encoded solely in the DNS header. If EDNS is in use, then the rcode is encoded using bits form both the header and the EDNS OPT RR.

dns.rcode.NOERROR = *Rcode.NOERROR*

dns.rcode.FORMERR = *Rcode.FORMERR*

dns.rcode.SERVFAIL = *Rcode.SERVFAIL*

dns.rcode.NXDOMAIN = *Rcode.NXDOMAIN*

dns.rcode.NOTIMP = *Rcode.NOTIMP*

dns.rcode.REFUSED = *Rcode.REFUSED*

dns.rcode.YXDOMAIN = *Rcode.YXDOMAIN*

dns.rcode.YXRRSET = *Rcode.YXRRSET*

dns.rcode.NXRRSET = *Rcode.NXRRSET*

dns.rcode.NOTAUTH = *Rcode.NOTAUTH*

dns.rcode.NOTZONE = *Rcode.NOTZONE*

dns.rcode.BADVERS = *Rcode.BADVERS*

dns.rcode.from_text(*text: str*) → Rcode [source]

Convert text into an rcode.

text, a `str`, the textual rcode or an integer in textual form.

Raises `dns.rcode.UnknownRcode` if the rcode mnemonic is unknown.

Returns a `dns.rcode.Rcode`.

`dns.rcode.to_text`(*value: Rcode, tsig: bool = False*) → str [source]

Convert rcode into text.

value, a `dns.rcode.Rcode`, the rcode.

Raises `ValueError` if rcode is < 0 or > 4095.

Returns a `str`.

`dns.rcode.from_flags`(*flags: int, ednsflags: int*) → Rcode [source]

Return the rcode value encoded by flags and ednsflags.

flags, an `int`, the DNS flags field.

ednsflags, an `int`, the EDNS flags field.

Raises `ValueError` if rcode is < 0 or > 4095

Returns a `dns.rcode.Rcode`.

`dns.rcode.to_flags`(*value: Rcode*) → Tuple[int, int] [source]

Return a (flags, ednsflags) tuple which encodes the rcode.

value, a `dns.rcode.Rcode`, the rcode.

Raises `ValueError` if rcode is < 0 or > 4095.

Returns an `(int, int)` tuple.

Message EDNS Options

EDNS allows for larger messages and also provides an extension mechanism for the protocol. EDNS *options* are typed data, and are treated much like Rdata. For example, if dnsython encounters the EDNS `ECS` option code when parsing a DNS wire format message, it will create a `dns.edns.ECSOption` object to represent it.

dns.edns.NSID= *OptionType.NSID*

dns.edns.DAU= *OptionType.DAU*

dns.edns.DHU= *OptionType.DHU*

dns.edns.N3U= *OptionType.N3U*

dns.edns.ECS= *OptionType.ECS*

dns.edns.EXPIRE= *OptionType.EXPIRE*

dns.edns.COOKIE= *OptionType.COOKIE*

dns.edns.KEEPALIVE= *OptionType.KEEPALIVE*

dns.edns.PADDING= *OptionType.PADDING*

dns.edns.CHAIN= *OptionType.CHAIN*

class **dns.edns.Option**(*otype: OptionType | str*) [source]

Base class for all EDNS option types.

Initialize an option.

otype, a `dns.edns.OptionType`, is the option type.

classmethod **from_wire_parser**(*otype: OptionType*, *parser: Parser*) → Option [source]

Build an EDNS option object from wire format.

otype, a `dns.edns.OptionType`, is the option type.

parser, a `dns.wire.Parser`, the parser, which should be restructured to the option length.

Returns a `dns.edns.Option`.

`to_wire`(*file: Any | None = None*) → bytes | None [source]

Convert an option to wire format.

Returns a `bytes` or `None`.

class **`dns.edns.GenericOption`**(*otype: OptionType | str, data: bytes | str*) [source]

Generic Option Class

This class is used for EDNS option types for which we have no better implementation.

Initialize an option.

otype, a `dns.edns.OptionType`, is the option type.

classmethod **`from_wire_parser`**(*otype: OptionType | str, parser: Parser*) → Option [source]

Build an EDNS option object from wire format.

otype, a `dns.edns.OptionType`, is the option type.

parser, a `dns.wire.Parser`, the parser, which should be restructured to the option length.

Returns a `dns.edns.Option`.

`to_wire`(*file: Any | None = None*) → bytes | None [source]

Convert an option to wire format.

Returns a `bytes` or `None`.

class **`dns.edns.ECSOption`**(*address: str, srclen: int | None = None, scopelen: int = 0*) [source]

EDNS Client Subnet (ECS, RFC7871)

address, a `str`, is the client address information.

srclen, an `int`, the source prefix length, which is the leftmost number of bits of the address to be used for the lookup. The default is 24 for IPv4 and 56 for IPv6.

scopelen, an `int`, the scope prefix length. This value must be 0 in queries, and should be set in responses.

static **`from_text`**(*text: str*) → Option [source]

Convert a string into a *dns.edns.ECSOption*

text, a *str*, the text form of the option.

Returns a *dns.edns.ECSOption*.

Examples:

```
>>> import dns.edns
>>>
>>> # basic example
>>> dns.edns.ECSOption.from_text('1.2.3.4/24')
>>>
>>> # also understands scope
>>> dns.edns.ECSOption.from_text('1.2.3.4/24/32')
>>>
>>> # IPv6
>>> dns.edns.ECSOption.from_text('2001:4b98::1/64/64')
>>>
>>> # it understands results from `dns.edns.ECSOption.to_text()`
>>> dns.edns.ECSOption.from_text('ECS 1.2.3.4/24/32')
```

classmethod **from_wire_parser**(*otype: OptionType | str, parser: Parser*)→ Option [source]

Build an EDNS option object from wire format.

otype, a `dns.edns.OptionType`, is the option type.

parser, a `dns.wire.Parser`, the parser, which should be restructed to the option length.

Returns a `dns.edns.Option`.

to_wire(*file: Any | None = None*)→ bytes | None [source]

Convert an option to wire format.

Returns a `bytes` or `None`.

dns.edns.get_option_class(*otype: OptionType*)→ Any [source]

Return the class for the specified option type.

The GenericOption class is used if a more specific class is not known.

dns.edns.option_from_wire_parser(*otype: OptionType | str, parser: Parser*)→ Option [source]

Build an EDNS option object from wire format.

otype, an `int`, is the option type.

parser, a `dns.wire.Parser`, the parser, which should be restricted to the option length.

Returns an instance of a subclass of `dns.edns.Option`.

dns.edns.option_from_wire(*otype: OptionType | str, wire: bytes, current: int, olen: int*)→ Option [source]

Build an EDNS option object from wire format.

otype, an `int`, is the option type.

wire, a `bytes`, is the wire-format message.

current, an `int`, is the offset in *wire* of the beginning of the rdata.

olen, an `int`, is the length of the wire-format option data

Returns an instance of a subclass of `dns.edns.Option`.

`dns.edns.register_type`(*implementation: Any, otype: OptionType*) → None [source]

Register the implementation of an option type.

implementation, a `class`, is a subclass of `dns.edns.Option`.

otype, an `int`, is the option type.

The dns.message.QueryMessage Class

The `dns.message.QueryMessage` class is used for ordinary DNS query messages.

class **dns.message.QueryMessage**(*id: int | None = None*) [source]

canonical_name()→ Name [source]

Return the canonical name of the first name in the question section.

Raises `dns.message.NotQueryResponse` if the message is not a response.

Raises `dns.message.ChainTooLong` if the CNAME chain is too long.

Raises `dns.message.AnswerForNXDOMAIN` if the rcode is NXDOMAIN but an answer was found.

Raises `dns.exception.FormError` if the question count is not 1.

resolve_chaining()→ ChainingResult [source]

Follow the CNAME chain in the response to determine the answer RRset.

Raises `dns.message.NotQueryResponse` if the message is not a response.

Raises `dns.message.ChainTooLong` if the CNAME chain is too long.

Raises `dns.message.AnswerForNXDOMAIN` if the rcode is NXDOMAIN but an answer was found.

Raises `dns.exception.FormError` if the question count is not 1.

Returns a ChainingResult object.

The dns.message.ChainingResult Class

Objects of the `dns.message.ChainingResult` class are returned by the `dns.message.QueryMessage.resolve_chaining()` method.

class **dns.message.ChainingResult**(*canonical_name: Name, answer: RRset | None, minimum_ttl: int, cnames: List[RRset]*) [source]

The result of a call to dns.message.QueryMessage.resolve_chaining().

The `answer` attribute is the answer RRSet, or `None` if it doesn't exist.

The `canonical_name` attribute is the canonical name after all chaining has been applied (this is the same name as `rrset.name` in cases where rrset is not `None`).

The `minimum_ttl` attribute is the minimum TTL, i.e. the TTL to use if caching the data. It is the smallest of all the CNAME TTLs and either the answer TTL if it exists or the SOA TTL and SOA minimum values for negative answers.

The `cnames` attribute is a list of all the CNAME RRSets followed to get to the canonical name.

The dns.update.UpdateMessage Class

The `dns.update.UpdateMessage` class is used for DNS Dynamic Update messages. It provides section access using the DNS Dynamic Update section names, and a variety of convenience methods for constructing dynamic updates.

class **dns.update.UpdateMessage**(*zone: ~dns.name.Name | str | None = None, rdclass: ~dns.rdataclass.RdataClass = RdataClass.IN, keyring: ~typing.Any | None = None, keyname: ~dns.name.Name | None = None, keyalgorithm: ~dns.name.Name | str = <DNS name hmac-sha256.>, id: int | None = None*) [source]

Initialize a new DNS Update object.

See the documentation of the Message class for a complete description of the keyring dictionary.

zone, a `dns.name.Name`, `str`, or `None`, the zone which is being updated. `None` should only be used by dnspython's message constructors, as a zone is required for the convenience methods like `add()`, `replace()`, etc.

rdclass, an `int` or `str`, the class of the zone.

The *keyring*, *keyname*, and *keyalgorithm* parameters are passed to `use_tsig()`; see its documentation for details.

absent(*name: Name | str, rdtype: RdataType | str | None = None*) → None [source]

Require that an owner name (and optionally an rdata type) does not exist as a prerequisite to the execution of the update.

add(*name: Name | str, *args: Any*) → None [source]

Add records.

The first argument is always a name. The other arguments can be:

- rdataset...
- ttl, rdata...
- ttl, rdtype, string...

delete(*name: Name | str, *args: Any*) → None [source]

Delete records.

The first argument is always a name. The other arguments can be:

- *empty*
- rdataset...
- rdata...
- rdtype, [string...]

property **prerequisite**: *List[RRset]*

The prerequisite section.

present(*name: Name | str, *args: Any*) → None [source]

Require that an owner name (and optionally an rdata type, or specific rdataset) exists as a prerequisite to the execution of the update.

The first argument is always a name. The other arguments can be:

- rdataset...
- rdata...
- rdtype, string...

replace(*name: Name | str, *args: Any*) → None [source]

Replace records.

The first argument is always a name. The other arguments can be:

- rdataset...
- ttl, rdata...
- ttl, rdtype, string...

Note that if you want to replace the entire node, you should do a delete of the name followed by one or more calls to add.

property **update**: *List[RRset]*

The update section.

property **zone**: *List[RRset]*

The zone section.

The following constants may be used to specify sections in the `find_rrset()` and `get_rrset()` methods:

dns.update.ZONE = *UpdateSection.ZONE*

Update sections

dns.update.PREREQ = *UpdateSection.PREREQ*

Update sections

dns.update.UPDATE = *UpdateSection.UPDATE*

Update sections

dns.update.ADDITIONAL = *UpdateSection.ADDITIONAL*

Update sections

The dns.resolver.Resolver and dns.resolver.Answer Classes

class **dns.resolver.Resolver**(*filename: str = '/etc/resolv.conf', configure: bool = True*) [source]

DNS stub resolver.

filename, a `str` or file object, specifying a file in standard /etc/resolv.conf format. This parameter is meaningful only when *configure* is true and the platform is POSIX.

configure, a `bool`. If True (the default), the resolver instance is configured in the normal fashion for the operating system the resolver is running on. (I.e. by reading a /etc/resolv.conf file on POSIX systems and from the registry on Windows systems.)

domain

A `dns.name.Name`, the domain of this host.

nameservers

A `list` of `str` or `dns.nameserver.Nameserver`. A string may be an IPv4 or IPv6 address, or an https URL.

This field is actually a property, and returns a tuple as of dnspython 2.4. Assigning this this field converts any strings into `dns.nameserver.Nameserver` instances.

search

A `list` of dns.name.Name objects. If the query name is a relative name, the resolver will construct absolute query names to try by appending values from the search list.

use_search_by_default

A `bool`, specifes whether or not `resolve()` uses the search list configured in the system's resolver configuration when the `search` parameter to `resolve()` is `None`. The default is `False`.

port

An `int`, the default DNS port to send to if not overridden by *nameserver_ports*. The default value is 53.

nameserver_ports

A `dict` mapping an IPv4 or IPv6 address `str` to an `int`. This specifies the port to use when sending to a nameserver. If a port is not defined for an address, the value of the *port* attribute will be used.

timeout

A `float`, the number of seconds to wait for a response from a server.

lifetime

A `float`, the number of seconds to spend trying to get an answer to the question. If the lifetime expires a `dns.exception.Timeout` exception will be raised.

cache

An object implementing the caching protocol, e.g. a `dns.resolver.Cache` or a `dns.resolver.LRUCache`. The default is `None`, in which case there is no local caching.

retry_servfail

A `bool`. Should we retry a nameserver if it says `SERVFAIL`? The default is `False`.

keyring

A `dict`, the TSIG keyring to use. If a *keyring* is specified but a *keyname* is not, then the key used will be the first key in the *keyring*. Note that the order of keys in a dictionary is not defined, so applications should supply a keyname when a keyring is used, unless they know the keyring contains only one key.

keyname

A `dns.name.Name` or `None`, the name of the TSIG key to use; defaults to `None`. The key must be defined in the keyring.

keyalgorithm

A `dns.name.Name` or `str`, the TSIG algorithm to use.

edns

An `int`, the EDNS level to use. Specifying `None`, `False`, or `-1` means “do not use EDNS”, and in this case the other parameters are ignored. Specifying `True` is equivalent to specifying 0, i.e. “use EDNS0”.

ednsflags

An `int`, the EDNS flag values.

payload

An `int`, is the EDNS sender's payload field, which is the maximum size of UDP datagram the sender can handle. I.e. how big a response to this message can be.

flags

An `int` or `None`, the message flags to use. If `None`, then the default flags as set by the `dns.message.Message` constructor will be used.

canonical_name(*name: Name | str*)→ Name [source]

Determine the canonical name of *name*.

The canonical name is the name the resolver uses for queries after all CNAME and DNAME renamings have been applied.

name, a `dns.name.Name` or `str`, the query name.

This method can raise any exception that `resolve()` can raise, other than `dns.resolver.NoAnswer` and `dns.resolver.NXDOMAIN`.

Returns a `dns.name.Name`.

query(*qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None*)→ Answer [source]

Query nameservers to find the answer to the question.

This method calls resolve() with `search=True`, and is provided for backwards compatibility with prior versions of dnspython. See the documentation for the resolve() method for further details.

resolve(*qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None, search: bool | None = None*)→ Answer [source]

Query nameservers to find the answer to the question.

The *qname*, *rdtype*, and *rdclass* parameters may be objects of the appropriate type, or strings that can be converted into objects of the appropriate type.

qname, a `dns.name.Name` or `str`, the query name.

rdtype, an `int` or `str`, the query type.

rdclass, an `int` or `str`, the query class.

tcp, a `bool`. If `True`, use TCP to make the query.

source, a `str` or `None`. If not `None`, bind to this IP address when making queries.

raise_on_no_answer, a `bool`. If `True`, raise `dns.resolver.NoAnswer` if there's no answer to the question.

source_port, an `int`, the port from which to send the message.

lifetime, a `float`, how many seconds a query should run before timing out.

search, a `bool` or `None`, determines whether the search list configured in the system's resolver configuration are used for relative names, and whether the resolver's domain may be added to relative names. The default is `None`, which causes the value of the resolver's `use_search_by_default` attribute to be used.

Raises `dns.resolver.LifetimeTimeout` if no answers could be found in the specified lifetime.

Raises `dns.resolver.NXDOMAIN` if the query name does not exist.

Raises `dns.resolver.YXDOMAIN` if the query name is too long after DNAME substitution.

Raises `dns.resolver.NoAnswer` if *raise_on_no_answer* is `True` and the query name exists but has no RRset of the desired type and class.

Raises `dns.resolver.NoNameservers` if no non-broken nameservers are available to answer the question.

Returns a `dns.resolver.Answer` instance.

`resolve_address`(*ipaddr: str, *args: Any, **kwargs: Any*)→ Answer [source]

Use a resolver to run a reverse query for PTR records.

This utilizes the resolve() method to perform a PTR lookup on the specified IP address.

ipaddr, a `str`, the IPv4 or IPv6 address you want to get the PTR record for.

All other arguments that can be passed to the resolve() function except for rdtype and rdclass are also supported by this function.

`resolve_name`(*name: Name | str, family: int = AddressFamily.AF_UNSPEC, **kwargs: Any*)→ HostAnswers [source]

Use a resolver to query for address records.

This utilizes the resolve() method to perform A and/or AAAA lookups on the specified name.

qname, a `dns.name.Name` or `str`, the name to resolve.

family, an `int`, the address family. If socket.AF_UNSPEC (the default), both A and AAAA records will be retrieved.

All other arguments that can be passed to the resolve() function except for rdtype and rdclass are also supported by this function.

try_ddr(*lifetime: float = 5.0*) → None [source]

Try to update the resolver's nameservers using Discovery of Designated Resolvers (DDR). If successful, the resolver will subsequently use DNS-over-HTTPS or DNS-over-TLS for future queries.

lifetime, a float, is the maximum time to spend attempting DDR. The default is 5 seconds.

If the SVCB query is successful and results in a non-empty list of nameservers, then the resolver's nameservers are set to the returned servers in priority order.

The current implementation does not use any address hints from the SVCB record, nor does it resolve addresses for the SCVB target name, rather it assumes that the bootstrap nameserver will always be one of the addresses and uses it. A future revision to the code may offer fuller support. The code verifies that the bootstrap nameserver is in the Subject Alternative Name field of the TLS certficate.

class **dns.resolver.Answer**(*qname: Name, rdtype: RdataType, rdclass: RdataClass, response: QueryMessage, nameserver: str | None = None, port: int | None = None*) [source]

DNS stub resolver answer.

Instances of this class bundle up the result of a successful DNS resolution.

For convenience, the answer object implements much of the sequence protocol, forwarding to its `rrset` attribute. E.g. `for a in answer` is equivalent to `for a in answer.rrset`. `answer[i]` is equivalent to `answer.rrset[i]`, and `answer[i:j]` is equivalent to `answer.rrset[i:j]`.

Note that CNAMEs or DNAMEs in the response may mean that answer RRset's name might not be the query name.

qname

A `dns.name.Name`, the query name.

rdclass

An `int`, the query class.

rdtype

An `int`, the query type.

response

A `dns.message.QueryMessage`, the response message.

rrset

A `dns.rrset.RRset` or `None`, the answer RRset.

expiration

A `float`, the time when the answer expires.

canonical_name

A `dns.name.Name`, the canonical name of the query name, i.e. the owner name of the answer RRset after any CNAME and DNAME chaining.

The dns.nameserver.Nameserver Classes

The `dns.nameserver.Nameserver` abstract class represents a remote recursive resolver, and is used by the stub resolver to answer queries.

class **dns.nameserver.Nameserver** [source]

The dns.nameserver.Do53Nameserver Class

The `dns.nameserver.Do53Nameserver` class is a `dns.nameserver.Nameserver` class used to make regular UDP/TCP DNS queries, typically over port 53, to a recursive server.

class **dns.nameserver.Do53Nameserver**(*address: str, port: int = 53*) [source]

The dns.nameserver.DoTNameserver Class

The `dns.nameserver.DoTNameserver` class is a `dns.nameserver.Nameserver` class used to make DNS-over-TLS (DoT) queries to a recursive server.

class **dns.nameserver.DoTNameserver**(*address: str, port: int = 853, hostname: str | None = None, verify: bool | str = True*) [source]

The dns.nameserver.DoHNameserver Class

The `dns.nameserver.DoHNameserver` class is a `dns.nameserver.Nameserver` class used to make DNS-over-HTTPS (DoH) queries to a recursive server.

class **dns.nameserver.DoHNameserver**(*url: str, bootstrap_address: str | None = None, verify: bool | str = True, want_get: bool = False*) [source]

The dns.nameserver.DoQNameserver Class

The `dns.nameserver.DoQNameserver` class is a `dns.nameserver.Nameserver` class used to make DNS-over-QUIC (DoQ) queries to a recursive server.

class **dns.nameserver.DoQNameserver**(*address: str*, *port: int = 853*, *verify: bool | str = True*, *server_hostname: str | None = None*) [source]

Resolver Functions and The Default Resolver

dns.resolver.resolve(*qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None, search: bool | None = None*)→ Answer [source]

Query nameservers to find the answer to the question.

This is a convenience function that uses the default resolver object to make the query.

See `dns.resolver.Resolver.resolve` for more information on the parameters.

dns.resolver.resolve_address(*ipaddr: str, *args: Any, **kwargs: Any*)→ Answer [source]

Use a resolver to run a reverse query for PTR records.

See `dns.resolver.Resolver.resolve_address` for more information on the parameters.

dns.resolver.resolve_name(*name: Name | str, family: int = AddressFamily.AF_UNSPEC, **kwargs: Any*)→ HostAnswers [source]

Use a resolver to query for address records.

See `dns.resolver.Resolver.resolve_name` for more information on the parameters.

dns.resolver.canonical_name(*name: Name | str*)→ Name [source]

Determine the canonical name of *name*.

See `dns.resolver.Resolver.canonical_name` for more information on the parameters and possible exceptions.

dns.resolver.try_ddr(*lifetime: float = 5.0*)→ None [source]

Try to update the default resolver's nameservers using Discovery of Designated Resolvers (DDR). If successful, the resolver will subsequently use DNS-over-HTTPS or DNS-over-TLS for future queries.

See `dns.resolver.Resolver.try_ddr()` for more information.

dns.resolver.zone_for_name(*name: Name | str, rdclass: RdataClass = RdataClass.IN, tcp: bool = False, resolver: Resolver | None = None, lifetime: float | None = None*)→ Name [source]

Find the name of the zone which contains the specified name.

name, an absolute `dns.name.Name` or `str`, the query name.

rdclass, an `int`, the query class.

tcp, a `bool`. If `True`, use TCP to make the query.

resolver, a `dns.resolver.Resolver` or `None`, the resolver to use. If `None`, the default, then the default resolver is used.

lifetime, a `float`, the total time to allow for the queries needed to determine the zone. If `None`, the default, then only the individual query limits of the resolver apply.

Raises `dns.resolver.NoRootSOA` if there is no SOA RR at the DNS root. (This is only likely to happen if you're using non-default root servers in your network and they are misconfigured.)

Raises `dns.resolver.LifetimeTimeout` if the answer could not be found in the allotted lifetime.

Returns a `dns.name.Name`.

`dns.resolver.query`(*qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None*)→ Answer [source]

Query nameservers to find the answer to the question.

This method calls resolve() with `search=True`, and is provided for backwards compatibility with prior versions of dnspython. See the documentation for the resolve() method for further details.

`dns.resolver.make_resolver_at`(*where: Name | str, port: int = 53, family: int = AddressFamily.AF_UNSPEC, resolver: Resolver | None = None*)→ Resolver [source]

Make a stub resolver using the specified destination as the full resolver.

where, a `dns.name.Name` or `str` the domain name or IP address of the full resolver.

port, an `int`, the port to use. If not specified, the default is 53.

family, an `int`, the address family to use. This parameter is used if *where* is not an address. The default is `socket.AF_UNSPEC` in which case the first address returned by `resolve_name()` will be used, otherwise the first address of the specified family will be used.

resolver, a `dns.resolver.Resolver` or `None`, the resolver to use for resolution of hostnames. If not specified, the default resolver will be used.

Returns a `dns.resolver.Resolver` or raises an exception.

`dns.resolver.resolve_at`(*where: Name | str, qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None, search: bool | None =*

None, port: int = 53, family: int = AddressFamily.AF_UNSPEC, resolver: Resolver | None = None)→ Answer [source]

Query nameservers to find the answer to the question.

This is a convenience function that calls `dns.resolver.make_resolver_at()` to make a resolver, and then uses it to resolve the query.

See `dns.resolver.Resolver.resolve` for more information on the resolution parameters, and `dns.resolver.make_resolver_at` for information about the resolver parameters *where*, *port*, *family*, and *resolver*.

If making more than one query, it is more efficient to call `dns.resolver.make_resolver_at()` and then use that resolver for the queries instead of calling `resolve_at()` multiple times.

dns.resolver.default_resolver: *Resolver | None = None*

The default resolver.

dns.resolver.get_default_resolver()→ Resolver [source]

Get the default resolver, initializing it if necessary.

dns.resolver.reset_default_resolver()→ None [source]

Re-initialize default resolver.

Note that the resolver configuration (i.e. /etc/resolv.conf on UNIX systems) will be re-read immediately.

Resolver Caching Classes

The dnspython resolver does not cache by default, but caching can be enabled by creating a cache and assigning it to the resolver's *cache* attribute. If a cache has been configured, the resolver caches both positive and negative responses. The cache respects the DNS TTL of the data, and will not return expired entries.

Two thread-safe cache implementations are provided, a simple dictionary-based Cache, and an LRUCache which provides cache size control suitable for use in web crawlers. Both are subclasses of a common base class which provides basic statistics. The LRUCache can also provide a hits count per cache entry.

class **dns.resolver.CacheBase** [source]

get_statistics_snapshot()→ CacheStatistics [source]

Return a consistent snapshot of all the statistics.

If running with multiple threads, it's better to take a snapshot than to call statistics methods such as hits() and misses() individually.

hits()→ int [source]

How many hits has the cache had?

misses()→ int [source]

How many misses has the cache had?

reset_statistics()→ None [source]

Reset all statistics to zero.

class **dns.resolver.Cache**(*cleaning_interval: float = 300.0*) [source]

Simple thread-safe DNS answer cache.

cleaning_interval, a `float` is the number of seconds between periodic cleanings.

flush(*key: Tuple[Name, RdataType, RdataClass] | None = None*)→ None [source]

Flush the cache.

If *key* is not `None`, only that item is flushed. Otherwise the entire cache is flushed.

key, a `(dns.name.Name, dns.rdatatype.RdataType, dns.rdataclass.RdataClass)` tuple whose values are the query name, rdtype, and rdclass respectively.

get(*key: Tuple[Name, RdataType, RdataClass]*) → Answer | None [source]

Get the answer associated with *key*.

Returns None if no answer is cached for the key.

key, a `(dns.name.Name, dns.rdatatype.RdataType, dns.rdataclass.RdataClass)` tuple whose values are the query name, rdtype, and rdclass respectively.

Returns a `dns.resolver.Answer` or `None`.

put(*key: Tuple[Name, RdataType, RdataClass]*, *value: Answer*) → None [source]

Associate key and value in the cache.

key, a `(dns.name.Name, dns.rdatatype.RdataType, dns.rdataclass.RdataClass)` tuple whose values are the query name, rdtype, and rdclass respectively.

value, a `dns.resolver.Answer`, the answer.

class **dns.resolver.LRUCache**(*max_size: int = 100000*) [source]

Thread-safe, bounded, least-recently-used DNS answer cache.

This cache is better than the simple cache (above) if you're running a web crawler or other process that does a lot of resolutions. The LRUCache has a maximum number of nodes, and when it is full, the least-recently used node is removed to make space for a new one.

max_size, an `int`, is the maximum number of nodes to cache; it must be greater than 0.

flush(*key: Tuple[Name, RdataType, RdataClass] | None = None*) → None [source]

Flush the cache.

If *key* is not `None`, only that item is flushed. Otherwise the entire cache is flushed.

key, a `(dns.name.Name, dns.rdatatype.RdataType, dns.rdataclass.RdataClass)` tuple whose values are the query name, rdtype, and rdclass respectively.

get(*key: Tuple[Name, RdataType, RdataClass]*) → Answer | None [source]

Get the answer associated with *key*.

Returns None if no answer is cached for the key.

key, a `(dns.name.Name, dns.rdatatype.RdataType, dns.rdataclass.RdataClass)` tuple whose values are the query name, rdtype, and rdclass respectively.

Returns a `dns.resolver.Answer` or `None`.

get_hits_for_key(*key: Tuple[Name, RdataType, RdataClass]*) → int [source]

Return the number of cache hits associated with the specified key.

put(*key: Tuple[Name, RdataType, RdataClass]*, *value: Answer*) → None [source]

Associate key and value in the cache.

key, a `(dns.name.Name, dns.rdatatype.RdataType, dns.rdataclass.RdataClass)` tuple whose values are the query name, rdtype, and rdclass respectively.

value, a `dns.resolver.Answer`, the answer.

class **dns.resolver.CacheStatistics**(*hits: int = 0*, *misses: int = 0*) [source]

Cache Statistics

Overriding the System Resolver

Sometimes it can be useful to make all of Python use dnspython's resolver rather than the default functionality in the `socket` module. Dnspython can redefine the methods in the socket module to point at its own code, and it can also restore them back to the regular Python defaults.

dns.resolver.override_system_resolver(*resolver: Resolver | None = None*) → None [source]

Override the system resolver routines in the socket module with versions which use dnspython's resolver.

This can be useful in testing situations where you want to control the resolution behavior of python code without having to change the system's resolver settings (e.g. /etc/resolv.conf).

The resolver to use may be specified; if it's not, the default resolver will be used.

resolver, a `dns.resolver.Resolver` or `None`, the resolver to use.

dns.resolver.restore_system_resolver() → None [source]

Undo the effects of prior override_system_resolver().

The dns.zone.Zone Class

The `Zone` class provides a non-thread-safe implementation of a DNS zone, as well as a lightweight translation mechanism that allows it to be atomically updated. For more complicated transactional needs, or for concurrency, please use the `dns.versioned.Zone` class (described below).

class **dns.zone.Zone**(*origin: Name | str | None, rdclass: RdataClass = RdataClass.IN, relativize: bool = True*) [source]

A DNS zone.

A `Zone` is a mapping from names to nodes. The zone object may be treated like a Python dictionary, e.g. `zone[name]` will retrieve the node associated with that name. The *name* may be a `dns.name.Name object`, or it may be a string. In either case, if the name is relative it is treated as relative to the origin of the zone.

Initialize a zone object.

origin is the origin of the zone. It may be a `dns.name.Name`, a `str`, or `None`. If `None`, then the zone's origin will be set by the first `$ORIGIN` line in a zone file.

rdclass, an `int`, the zone's rdata class; the default is class IN.

relativize, a `bool`, determine's whether domain names are relativized to the zone's origin. The default is `True`.

rdclass

The zone's rdata class, an `int`; the default is class IN.

origin

The origin of the zone, a `dns.name.Name`.

nodes

A dictionary mapping the names of nodes in the zone to the nodes themselves.

relativize

A `bool`, which is `True` if names in the zone should be relativized.

check_origin()→ None [source]

Do some simple checking of the zone's origin.
Raises `dns.zone.NoSOA` if there is no SOA RRset.

Raises `dns.zone.NoNS` if there is no NS RRset.

Raises `KeyError` if there is no origin node.

`delete_node`(*name: Name | str*)→ None [source]

Delete the specified node if it exists.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

It is not an error if the node does not exist.

`delete_rdataset`(*name: Name | str, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0*)→ None [source]

Delete the rdataset matching *rdtype* and *covers*, if it exists at the node specified by *name*.

It is not an error if the node does not exist, or if there is no matching rdataset at the node.

If the node has no rdatasets after the deletion, it will itself be deleted.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdata type desired.

covers, a `dns.rdatatype.RdataType` or `str` or `None`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

`find_node`(*name: Name | str, create: bool = False*)→ Node [source]

Find a node in the zone, possibly creating it.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

create, a `bool`. If true, the node will be created if it does not exist.

Raises `KeyError` if the name is not known and create was not specified, or if the name was not a subdomain of the origin.

Returns a `dns.node.Node`.

`find_rdataset`(*name: Name | str, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0, create: bool = False*)→ Rdataset [source]

Look for an rdataset with the specified name and type in the zone, and return an rdataset encapsulating it.

The rdataset returned is not a copy; changes to it will change the zone.

KeyError is raised if the name or type are not found.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdata type desired.

covers, a `dns.rdatatype.RdataType` or `str` the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If true, the node will be created if it does not exist.

Raises `KeyError` if the name is not known and create was not specified, or if the name was not a subdomain of the origin.

Returns a `dns.rdataset.Rdataset`.

`find_rrset`(*name: Name | str, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0*)→ RRset [source]

Look for an rdataset with the specified name and type in the zone, and return an RRset encapsulating it.

This method is less efficient than the similar `find_rdataset()` because it creates an RRset instead of returning the matching rdataset. It may be more convenient for some uses since it returns an object which binds the owner name to the rdataset.

This method may not be used to create new nodes or rdatasets; use `find_rdataset` instead.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdata type desired.

covers, a `dns.rdatatype.RdataType` or `str`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If true, the node will be created if it does not exist.

Raises `KeyError` if the name is not known and create was not specified, or if the name was not a subdomain of the origin.

Returns a `dns.rrset.RRset` or `None`.

`get_class`() [source]

The class of the transaction manager.

`get_node`(*name: Name | str, create: bool = False*)→ Node | None [source]

Get a node in the zone, possibly creating it.

This method is like `find_node()`, except it returns None instead of raising an exception if the node does not exist and creation has not been requested.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

create, a `bool`. If true, the node will be created if it does not exist.

Returns a `dns.node.Node` or `None`.

`get_rdataset`(*name: Name | str, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0, create: bool = False*)→ Rdataset | None [source]

Look for an rdataset with the specified name and type in the zone.

This method is like `find_rdataset()`, except it returns None instead of raising an exception if the rdataset does not exist and creation has not been requested.

The rdataset returned is not a copy; changes to it will change the zone.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdata type desired.

covers, a `dns.rdatatype.RdataType` or `str`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If true, the node will be created if it does not exist.

Raises `KeyError` if the name is not known and create was not specified, or if the name was not a subdomain of the origin.

Returns a `dns.rdataset.Rdataset` or `None`.

`get_rrset`(*name: Name | str, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0*)→ RRset | None [source]

Look for an rdataset with the specified name and type in the zone, and return an RRset encapsulating it.

This method is less efficient than the similar `get_rdataset()` because it creates an RRset instead of returning the matching rdataset. It may be more convenient for some uses since it returns an object which binds the owner name to the rdataset.

This method may not be used to create new nodes or rdatasets; use `get_rdataset()` instead.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

rdtype, a `dns.rdataset.Rdataset` or `str`, the rdata type desired.

covers, a `dns.rdataset.Rdataset` or `str`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If true, the node will be created if it does not exist.

Returns a `dns.rrset.RRset` or `None`.

`get_soa`(*txn: Transaction | None = None*)→ SOA [source]

Get the zone SOA rdata.

Raises `dns.zone.NoSOA` if there is no SOA RRset.

Returns a `dns.rdtypes.ANY.SOA.SOA` Rdata.

iterate_rdatas(*rdtype: RdataType | str = RdataType.ANY, covers: RdataType | str = RdataType.TYPE0*) → Iterator[Tuple[Name, int, Rdata]] [source]

Return a generator which yields (name, ttl, rdata) tuples for all rdatas in the zone which have the specified *rdtype* and *covers*. If *rdtype* is `dns.rdatatype.ANY`, the default, then all rdatas will be matched.

rdtype, a `dns.rdataset.Rdataset` or `str`, the rdata type desired.

covers, a `dns.rdataset.Rdataset` or `str`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

iterate_rdatasets(*rdtype: RdataType | str = RdataType.ANY, covers: RdataType | str = RdataType.TYPE0*) → Iterator[Tuple[Name, Rdataset]] [source]

Return a generator which yields (name, rdataset) tuples for all rdatasets in the zone which have the specified *rdtype* and *covers*. If *rdtype* is `dns.rdatatype.ANY`, the default, then all rdatasets will be matched.

rdtype, a `dns.rdataset.Rdataset` or `str`, the rdata type desired.

covers, a `dns.rdataset.Rdataset` or `str`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

map_factory

alias of `dict`

node_factory

alias of `Node`

origin_information() → Tuple[Name | None, bool, Name | None] [source]

Returns a tuple

(absolute_origin, relativize, effective_origin)

giving the absolute name of the default origin for any relative domain names, the "effective origin", and whether names should be relativized. The "effective origin" is the absolute origin if relativize is False, and the empty name if relativize is true. (The

effective origin is provided even though it can be computed from the absolute_origin and relativize setting because it avoids a lot of code duplication.)

If the returned names are *None*, then no origin information is available.

This information is used by code working with transactions to allow it to coordinate relativization. The transaction code itself takes what it gets (i.e. does not change name relativity).

`reader`()→ Transaction [source]

Begin a read-only transaction.

`replace_rdataset`(*name: Name | str, replacement: Rdataset*)→ None [source]

Replace an rdataset at name.

It is not an error if there is no rdataset matching I{replacement}.

Ownership of the *replacement* object is transferred to the zone; in other words, this method does not store a copy of *replacement* at the node, it stores *replacement* itself.

If the node does not exist, it is created.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

replacement, a `dns.rdataset.Rdataset`, the replacement rdataset.

`to_file`(*f: Any, sorted: bool = True, relativize: bool = True, nl: str | None = None, want_comments: bool = False, want_origin: bool = False*)→ None [source]

Write a zone to a file.

f, a file or *str*. If *f* is a string, it is treated as the name of a file to open.

sorted, a `bool`. If True, the default, then the file will be written with the names sorted in DNSSEC order from least to greatest. Otherwise the names will be written in whatever order they happen to have in the zone's dictionary.

relativize, a `bool`. If True, the default, then domain names in the output will be relativized to the zone's origin if possible.

nl, a `str` or None. The end of line string. If not `None`, the output will use the platform's native end-of-line marker (i.e. LF on POSIX, CRLF on Windows).

want_comments, a `bool`. If `True`, emit end-of-line comments as part of writing the file. If `False`, the default, do not emit them.

want_origin, a `bool`. If `True`, emit a $ORIGIN line at the start of the file. If `False`, the default, do not emit one.

`to_text`(*sorted: bool = True, relativize: bool = True, nl: str | None = None, want_comments: bool = False, want_origin: bool = False*)→ str [source]

Return a zone's text as though it were written to a file.

sorted, a `bool`. If True, the default, then the file will be written with the names sorted in DNSSEC order from least to greatest. Otherwise the names will be written in whatever order they happen to have in the zone's dictionary.

relativize, a `bool`. If True, the default, then domain names in the output will be relativized to the zone's origin if possible.

nl, a `str` or None. The end of line string. If not `None`, the output will use the platform's native end-of-line marker (i.e. LF on POSIX, CRLF on Windows).

want_comments, a `bool`. If `True`, emit end-of-line comments as part of writing the file. If `False`, the default, do not emit them.

want_origin, a `bool`. If `True`, emit a $ORIGIN line at the start of the output. If `False`, the default, do not emit one.

Returns a `str`.

`writer`(*replacement: bool = False*)→ Transaction [source]

Begin a writable transaction.

replacement, a `bool`. If *True*, the content of the transaction completely replaces any prior content. If False, the default, then the content of the transaction updates the existing content.

A `Zone` has a class attribute `node_factory` which is used to create new nodes and defaults to `dns.node.Node`. `Zone` may be subclassed if a different node factory is desired. The node factory is a class or callable that returns a subclass of `dns.node.Node`.

The dns.versioned.Zone Class

A versioned Zone is a subclass of `Zone` that provides a thread-safe multiversioned transactional API. There can be many concurrent readers, of possibly different versions, and at most one active writer. Others cannot see the changes being made by the writer until it commits. Versions are immutable once committed.

The read-only parts of the standard zone API continue to be available, and are equivalent to doing a single-query read-only transaction. Note that unless reading is done through a transaction, version stability is not guaranteed between successive calls. Attempts to use zone API methods that directly manipulate the zone, e.g. `replace_rdataset` will result in a `UseTransaction` exception.

Transactions are context managers, and are created with `reader()` or `writer()`. For example:

```
# Print the SOA serial number of the most recent version
with zone.reader() as txn:
    rdataset = txn.get('@', 'in', 'soa')
    print('The most recent serial number is', rdataset[0].serial)

# Write an A RR and increment the SOA serial number to the next value.
with zone.writer() as txn:
    txn.replace('node1', dns.rdataset.from_text('in', 'a', 300,
                '10.0.0.1'))
    txn.set_serial()
```

See below for more information on the `Transaction` API.

class **dns.versioned.Zone**(*origin: Name | str | None, rdclass: RdataClass = RdataClass.IN, relativize: bool = True, pruning_policy: Callable[[Zone, Version], bool | None] | None = None*) [source]

Initialize a versioned zone object.

origin is the origin of the zone. It may be a `dns.name.Name`, a `str`, or `None`. If `None`, then the zone's origin will be set by the first `$ORIGIN` line in a zone file.

rdclass, an `int`, the zone's rdata class; the default is class IN.

relativize, a `bool`, determine's whether domain names are relativized to the zone's origin. The default is `True`.

pruning policy, a function taking a `Zone` and a `Version` and returning a `bool`, or `None`. Should the version be pruned? If `None`, the default policy, which retains one version is used.

rdclass

The zone's rdata class, an `int`; the default is class IN.

origin

The origin of the zone, a `dns.name.Name`.

nodes

A dictionary mapping the names of nodes in the zone to the nodes themselves.

relativize

A `bool`, which is `True` if names in the zone should be relativized.

find_node(*name: Name | str, create: bool = False*)→ Node [source]

Find a node in the zone, possibly creating it.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

create, a `bool`. If true, the node will be created if it does not exist.

Raises `KeyError` if the name is not known and create was not specified, or if the name was not a subdomain of the origin.

Returns a `dns.node.Node`.

find_rdataset(*name: Name | str, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0, create: bool = False*)→ Rdataset [source]

Look for an rdataset with the specified name and type in the zone, and return an rdataset encapsulating it.

The rdataset returned is not a copy; changes to it will change the zone.

KeyError is raised if the name or type are not found.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdata type desired.

covers, a `dns.rdatatype.RdataType` or `str` the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If true, the node will be created if it does not exist.

Raises `KeyError` if the name is not known and create was not specified, or if the name was not a subdomain of the origin.

Returns a `dns.rdataset.Rdataset`.

get_rdataset(*name: Name | str, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0, create: bool = False*)→ Rdataset | None [source]

Look for an rdataset with the specified name and type in the zone.

This method is like `find_rdataset()`, except it returns None instead of raising an exception if the rdataset does not exist and creation has not been requested.

The rdataset returned is not a copy; changes to it will change the zone.

name: the name of the node to find. The value may be a `dns.name.Name` or a `str`. If absolute, the name must be a subdomain of the zone's origin. If `zone.relativize` is `True`, then the name will be relativized.

rdtype, a `dns.rdatatype.RdataType` or `str`, the rdata type desired.

covers, a `dns.rdatatype.RdataType` or `str`, the covered type. Usually this value is `dns.rdatatype.NONE`, but if the rdtype is `dns.rdatatype.SIG` or `dns.rdatatype.RRSIG`, then the covers value will be the rdata type the SIG/RRSIG covers. The library treats the SIG and RRSIG types as if they were a family of types, e.g. RRSIG(A), RRSIG(NS), RRSIG(SOA). This makes RRSIGs much easier to work with than if RRSIGs covering different rdata types were aggregated into a single RRSIG rdataset.

create, a `bool`. If true, the node will be created if it does not exist.

Raises `KeyError` if the name is not known and create was not specified, or if the name was not a subdomain of the origin.

Returns a `dns.rdataset.Rdataset` or `None`.

`node_factory`

alias of `VersionedNode`

`reader`(*id: int | None = None, serial: int | None = None*)→ Transaction [source]

Begin a read-only transaction.

`set_max_versions`(*max_versions: int | None*)→ None [source]

Set a pruning policy that retains up to the specified number of versions

`set_pruning_policy`(*policy: Callable[[Zone, Version], bool | None] | None*)→ None [source]

Set the pruning policy for the zone.

The *policy* function takes a *Version* and returns *True* if the version should be pruned, and *False* otherwise. *None* may also be specified for policy, in which case the default policy is used.

Pruning checking proceeds from the least version and the first time the function returns *False*, the checking stops. I.e. the retained versions are always a consecutive sequence.

`writer`(*replacement: bool = False*)→ Transaction [source]

Begin a writable transaction.

replacement, a `bool`. If *True*, the content of the transaction completely replaces any prior content. If False, the default, then the content of the transaction updates the existing content.

The TransactionManager Class

This is the abstract base class of all objects that support transactions.

class **dns.transaction.TransactionManager** [source]

from_wire_origin()→ Name | None [source]

Origin to use in from_wire() calls.

get_class()→ RdataClass [source]

The class of the transaction manager.

origin_information()→ Tuple[Name | None, bool, Name | None] [source]

Returns a tuple

(absolute_origin, relativize, effective_origin)

giving the absolute name of the default origin for any relative domain names, the "effective origin", and whether names should be relativized. The "effective origin" is the absolute origin if relativize is False, and the empty name if relativize is true. (The effective origin is provided even though it can be computed from the absolute_origin and relativize setting because it avoids a lot of code duplication.)

If the returned names are *None*, then no origin information is available.

This information is used by code working with transactions to allow it to coordinate relativization. The transaction code itself takes what it gets (i.e. does not change name relativity).

reader()→ Transaction [source]

Begin a read-only transaction.

writer(*replacement: bool = False*)→ Transaction [source]

Begin a writable transaction.

replacement, a `bool`. If *True*, the content of the transaction completely replaces any prior content. If False, the default, then the content of the transaction updates the existing content.

The Transaction Class

class **dns.transaction.Transaction**(*manager: TransactionManager*, *replacement: bool = False*, *read_only: bool = False*) [source]

add(**args: Any*) → None [source]

Add records.

The arguments may be:

- rrset
- name, rdataset...
- name, ttl, rdata...

changed() → bool [source]

Has this transaction changed anything?

For read-only transactions, the result is always *False*.

For writable transactions, the result is *True* if at some time during the life of the transaction, the content was changed.

check_delete_name(*check: Callable[[Transaction, Name], None]*) → None [source]

Call *check* before putting (storing) an rdataset.

The function is called with the transaction and the name.

The check function may safely make non-mutating transaction method calls, but behavior is undefined if mutating transaction methods are called. The check function should raise an exception if it objects to the put, and otherwise should return `None`.

check_delete_rdataset(*check: Callable[[Transaction, Name, RdataType, RdataType], None]*) → None [source]

Call *check* before deleting an rdataset.

The function is called with the transaction, the name, the rdatatype, and the covered rdatatype.

The check function may safely make non-mutating transaction method calls, but behavior is undefined if mutating transaction methods are called. The check function should raise an exception if it objects to the put, and otherwise should return `None`.

check_put_rdataset(*check: Callable[[Transaction, Name, Rdataset], None]*) → None [source]

Call *check* before putting (storing) an rdataset.

The function is called with the transaction, the name, and the rdataset.

The check function may safely make non-mutating transaction method calls, but behavior is undefined if mutating transaction methods are called. The check function should raise an exception if it objects to the put, and otherwise should return `None`.

commit()→ None [source]

Commit the transaction.

Normally transactions are used as context managers and commit or rollback automatically, but it may be done explicitly if needed. A `dns.transaction.Ended` exception will be raised if you try to use a transaction after it has been committed or rolled back.

Raises an exception if the commit fails (in which case the transaction is also rolled back.

delete(**args: Any*)→ None [source]

Delete records.

It is not an error if some of the records are not in the existing set.

The arguments may be:

- rrset
- name
- name, rdatatype, [covers]
- name, rdataset...
- name, rdata...

delete_exact(**args: Any*)→ None [source]

Delete records.

The arguments may be:

- rrset
- name
- name, rdatatype, [covers]
- name, rdataset...
- name, rdata...

Raises dns.transaction.DeleteNotExact if some of the records are not in the existing set.

get(*name: Name | str | None, rdtype: RdataType | str, covers: RdataType | str = RdataType.TYPE0*)→ Rdataset [source]

Return the rdataset associated with *name*, *rdtype*, and *covers*, or *None* if not found.

Note that the returned rdataset is immutable.

get_node(*name: Name*)→ Node | None [source]

Return the node at *name*, if any.

Returns an immutable node or `None`.

`iterate_names`()→ Iterator[Name] [source]

Iterate all the names in the transaction.

Note that as is usual with python iterators, adding or removing names while iterating will invalidate the iterator and may raise *RuntimeError* or fail to iterate over all entries.

`iterate_rdatasets`()→ Iterator[Tuple[Name, Rdataset]] [source]

Iterate all the rdatasets in the transaction, returning (*dns.name.Name*, *dns.rdataset.Rdataset*) tuples.

Note that as is usual with python iterators, adding or removing items while iterating will invalidate the iterator and may raise *RuntimeError* or fail to iterate over all entries.

`name_exists`(*name: Name | str*)→ bool [source]

Does the specified name exist?

`replace`(**args: Any*)→ None [source]

Replace the existing rdataset at the name with the specified rdataset, or add the specified rdataset if there was no existing rdataset.

The arguments may be:

- rrset
- name, rdataset...
- name, ttl, rdata...

Note that if you want to replace the entire node, you should do a delete of the name followed by one or more calls to add() or replace().

`rollback`()→ None [source]

Rollback the transaction.

Normally transactions are used as context managers and commit or rollback automatically, but it may be done explicitly if needed. A `dns.transaction.AlreadyEnded` exception will be raised if you try to use a transaction after it has been committed or rolled back.

Rollback cannot otherwise fail.

`update_serial`(*value: int = 1, relative: bool = True, name: ~dns.name.Name = <DNS name @>*)→ None [source]

Update the serial number.

value, an *int*, is an increment if *relative* is *True*, or the actual value to set if *relative* is *False*.

Raises *KeyError* if there is no SOA rdataset at *name*.

Raises *ValueError* if *value* is negative or if the increment is so large that it would cause the new serial to be less than the prior value.

Making DNS Zones

dns.zone.from_text(*text: str, origin: ~dns.name.Name | str | None = None, rdclass: ~dns.rdataclass.RdataClass = RdataClass.IN, relativize: bool = True, zone_factory: ~typing.Any = <class 'dns.zone.Zone'>, filename: str | None = None, allow_include: bool = False, check_origin: bool = True, idna_codec: ~dns.name.IDNACodec | None = None, allow_directives: bool | ~typing.Iterable[str] = True*)→ Zone [source]

Build a zone object from a zone file format string.

text, a `str`, the zone file format input.

origin, a `dns.name.Name`, a `str`, or `None`. The origin of the zone; if not specified, the first `$ORIGIN` statement in the zone file will determine the origin of the zone.

rdclass, a `dns.rdataclass.RdataClass`, the zone's rdata class; the default is class IN.

relativize, a `bool`, determine's whether domain names are relativized to the zone's origin. The default is `True`.

zone_factory, the zone factory to use or `None`. If `None`, then `dns.zone.Zone` will be used. The value may be any class or callable that returns a subclass of `dns.zone.Zone`.

filename, a `str` or `None`, the filename to emit when describing where an error occurred; the default is `'<string>'`.

allow_include, a `bool`. If `True`, the default, then `$INCLUDE` directives are permitted. If `False`, then encoutering a `$INCLUDE` will raise a `SyntaxError` exception.

check_origin, a `bool`. If `True`, the default, then sanity checks of the origin node will be made by calling the zone's `check_origin()` method.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

allow_directives, a `bool` or an iterable of *str*. If `True`, the default, then directives are permitted, and the *allow_include* parameter controls whether `$INCLUDE` is permitted. If `False` or an empty iterable, then no directive processing is done and any directive-like text will be treated as a regular owner name. If a non-empty iterable, then only the listed directives (including the `$`) are allowed.

Raises `dns.zone.NoSOA` if there is no SOA RRset.

Raises `dns.zone.NoNS` if there is no NS RRset.

Raises `KeyError` if there is no origin node.

Returns a subclass of `dns.zone.Zone`.

`dns.zone.from_file`(*f: ~typing.Any, origin: ~dns.name.Name | str | None = None, rdclass: ~dns.rdataclass.RdataClass = RdataClass.IN, relativize: bool = True, zone_factory: ~typing.Any = <class 'dns.zone.Zone'>, filename: str | None = None, allow_include: bool = True, check_origin: bool = True, idna_codec: ~dns.name.IDNACodec | None = None, allow_directives: bool | ~typing.Iterable[str] = True*)→ Zone [source]

Read a zone file and build a zone object.

f, a file or `str`. If *f* is a string, it is treated as the name of a file to open.

origin, a `dns.name.Name`, a `str`, or `None`. The origin of the zone; if not specified, the first `$ORIGIN` statement in the zone file will determine the origin of the zone.

rdclass, an `int`, the zone's rdata class; the default is class IN.

relativize, a `bool`, determine's whether domain names are relativized to the zone's origin. The default is `True`.

zone_factory, the zone factory to use or `None`. If `None`, then `dns.zone.Zone` will be used. The value may be any class or callable that returns a subclass of `dns.zone.Zone`.

filename, a `str` or `None`, the filename to emit when describing where an error occurred; the default is `'<string>'`.

allow_include, a `bool`. If `True`, the default, then `$INCLUDE` directives are permitted. If `False`, then encoutering a `$INCLUDE` will raise a `SyntaxError` exception.

check_origin, a `bool`. If `True`, the default, then sanity checks of the origin node will be made by calling the zone's `check_origin()` method.

idna_codec, a `dns.name.IDNACodec`, specifies the IDNA encoder/decoder. If `None`, the default IDNA 2003 encoder/decoder is used.

allow_directives, a `bool` or an iterable of *str*. If `True`, the default, then directives are permitted, and the *allow_include* parameter controls whether `$INCLUDE` is permitted. If `False` or an empty iterable, then no directive processing is done and any directive-like text will be treated as a regular owner name. If a non-empty iterable, then only the listed directives (including the `$`) are allowed.

Raises `dns.zone.NoSOA` if there is no SOA RRset.

Raises `dns.zone.NoNS` if there is no NS RRset.

Raises `KeyError` if there is no origin node.

Returns a subclass of `dns.zone.Zone`.

`dns.zone.from_xfr`(*xfr: ~typing.Any, zone_factory: ~typing.Any = <class 'dns.zone.Zone'>, relativize: bool = True, check_origin: bool = True*)→ Zone [source]

Convert the output of a zone transfer generator into a zone object.

xfr, a generator of `dns.message.Message` objects, typically `dns.query.xfr()`.

relativize, a `bool`, determine's whether domain names are relativized to the zone's origin. The default is `True`. It is essential that the relativize setting matches the one specified to the generator.

check_origin, a `bool`. If `True`, the default, then sanity checks of the origin node will be made by calling the zone's `check_origin()` method.

Raises `dns.zone.NoSOA` if there is no SOA RRset.

Raises `dns.zone.NoNS` if there is no NS RRset.

Raises `KeyError` if there is no origin node.

Raises `ValueError` if no messages are yielded by the generator.

Returns a subclass of `dns.zone.Zone`.

The dns.xfr.Inbound Class and make_query() function

The `Inbound` class provides support for inbound DNS zone transfers, both AXFR and IXFR. It is invoked by I/O code, i.e. `dns.query.inbound_xfr()` or `dns.asyncquery.inbound_xfr()`. When a message related to the transfer arrives, the I/O code calls the `process_message()` method which adds the content to the pending transaction.

The `make_query()` function is used to making the query message for the query methods to use in more complex situations, e.g. with TSIG or EDNS.

class **dns.xfr.Inbound**(*txn_manager: TransactionManager, rdtype: RdataType = RdataType.AXFR, serial: int | None = None, is_udp: bool = False*) [source]

State machine for zone transfers.

Initialize an inbound zone transfer.

txn_manager is a `dns.transaction.TransactionManager`.

rdtype can be *dns.rdatatype.AXFR* or *dns.rdatatype.IXFR*

serial is the base serial number for IXFRs, and is required in that case.

is_udp, a `bool` indidicates if UDP is being used for this XFR.

process_message(*message: Message*)→ bool [source]

Process one message in the transfer.

The message should have the same relativization as was specified when the *dns.xfr.Inbound* was created. The message should also have been created with *one_rr_per_rrset=True* because order matters.

Returns *True* if the transfer is complete, and *False* otherwise.

dns.xfr.make_query(*txn_manager: ~dns.transaction.TransactionManager, serial: int | None = 0, use_edns: int | bool | None = None, ednsflags: int | None = None, payload: int | None = None, request_payload: int | None = None, options: ~typing.List[~dns.edns.Option] | None = None, keyring: ~typing.Any = None, keyname: ~dns.name.Name | None = None, keyalgorithm: ~dns.name.Name | str = <DNS name hmac-sha256.>*)→ Tuple[QueryMessage, int | None] [source]

Make an AXFR or IXFR query.

txn_manager is a `dns.transaction.TransactionManager`, typically a `dns.zone.Zone`.

serial is an `int` or `None`. If 0, then IXFR will be attempted using the most recent serial number from the *txn_manager*; it is the caller's responsibility to ensure there are no write transactions active that could invalidate the retrieved serial. If a serial cannot be determined, AXFR will be forced. Other integer values are the starting serial to use. `None` forces an AXFR.

Please see the documentation for **`dns.message.make_query()`** and **`dns.message.Message.use_tsig()`** for details on the other parameters to this function.

Returns a *(query, serial)* tuple.

DNS Query Support

The `dns.asyncquery` module is for sending messages to DNS servers, and processing their responses. If you want "stub resolver" behavior, then you should use the higher level `dns.asyncresolver` module; see Stub Resolver.

For UDP and TCP, the module provides a single "do everything" query function, and also provides the send and receive halves of this function individually for situations where more sophisticated I/O handling is being used by the application.

UDP

async **dns.asyncquery.udp**(*q: Message, where: str, timeout: float | None = None, port: int = 53, source: str | None = None, source_port: int = 0, ignore_unexpected: bool = False, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, raise_on_truncation: bool = False, sock: DatagramSocket | None = None, backend: Backend | None = None, ignore_errors: bool = False*)→ Message [source]

Return the response obtained after sending a query via UDP.

sock, a `dns.asyncbackend.DatagramSocket`, or `None`, the socket to use for the query. If `None`, the default, a socket is created. Note that if a socket is provided, the *source*, *source_port*, and *backend* are ignored.

backend, a `dns.asyncbackend.Backend`, or `None`. If `None`, the default, then dnspython will use the default backend.

See `dns.query.udp()` for the documentation of the other parameters, exceptions, and return type of this method.

async **dns.asyncquery.udp_with_fallback**(*q: Message, where: str, timeout: float | None = None, port: int = 53, source: str | None = None, source_port: int = 0, ignore_unexpected: bool = False, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, udp_sock: DatagramSocket | None = None, tcp_sock: StreamSocket | None = None, backend: Backend | None = None, ignore_errors: bool = False*)→ Tuple[Message, bool] [source]

Return the response to the query, trying UDP first and falling back to TCP if UDP results in a truncated response.

udp_sock, a `dns.asyncbackend.DatagramSocket`, or `None`, the socket to use for the UDP query. If `None`, the default, a socket is created. Note that if a socket is provided the *source*, *source_port*, and *backend* are ignored for the UDP query.

tcp_sock, a `dns.asyncbackend.StreamSocket`, or `None`, the socket to use for the TCP query. If `None`, the default, a socket is created. Note that if a socket is provided *where*, *source*, *source_port*, and *backend* are ignored for the TCP query.

backend, a `dns.asyncbackend.Backend`, or `None`. If `None`, the default, then dnspython will use the default backend.

See `dns.query.udp_with_fallback()` for the documentation of the other parameters, exceptions, and return type of this method.

---

async **dns.asyncquery.send_udp**(*sock: DatagramSocket, what: Message | bytes, destination: Any, expiration: float | None = None*)→ Tuple[int, float] [source]

Send a DNS message to the specified UDP socket.

sock, a `dns.asyncbackend.DatagramSocket`.

what, a `bytes` or `dns.message.Message`, the message to send.

destination, a destination tuple appropriate for the address family of the socket, specifying where to send the query.

expiration, a `float` or `None`, the absolute time at which a timeout exception should be raised. If `None`, no timeout will occur. The expiration value is meaningless for the asyncio backend, as asyncio's transport sendto() never blocks.

Returns an `(int, float)` tuple of bytes sent and the sent time.

---

async **dns.asyncquery.receive_udp**(*sock: DatagramSocket, destination: Any | None = None, expiration: float | None = None, ignore_unexpected: bool = False, one_rr_per_rrset: bool = False, keyring: Dict[Name, Key] | None = None, request_mac: bytes | None = b'', ignore_trailing: bool = False, raise_on_truncation: bool = False, ignore_errors: bool = False, query: Message | None = None*)→ Any [source]

Read a DNS message from a UDP socket.

sock, a `dns.asyncbackend.DatagramSocket`.

See `dns.query.receive_udp()` for the documentation of the other parameters, and exceptions.

Returns a `(dns.message.Message, float, tuple)` tuple of the received message, the received time, and the address where the message arrived from.

TCP

---

async **dns.asyncquery.tcp**(*q: Message, where: str, timeout: float | None = None, port: int = 53, source: str | None = None, source_port: int = 0, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, sock: StreamSocket | None = None, backend: Backend | None = None*)→ Message [source]

Return the response obtained after sending a query via TCP.

sock, a `dns.asyncbacket.StreamSocket`, or `None`, the socket to use for the query. If `None`, the default, a socket is created. Note that if a socket is provided *where*, *port*, *source*, *source_port*, and *backend* are ignored.

backend, a `dns.asyncbackend.Backend`, or `None`. If `None`, the default, then dnspython will use the default backend.

See `dns.query.tcp()` for the documentation of the other parameters, exceptions, and return type of this method.

---

async **dns.asyncquery.send_tcp**(*sock: StreamSocket, what: Message | bytes, expiration: float | None = None*)→ Tuple[int, float] [source]

Send a DNS message to the specified TCP socket.

sock, a `dns.asyncbackend.StreamSocket`.

See `dns.query.send_tcp()` for the documentation of the other parameters, exceptions, and return type of this method.

---

async **dns.asyncquery.receive_tcp**(*sock: StreamSocket, expiration: float | None = None, one_rr_per_rrset: bool = False, keyring: Dict[Name, Key] | None = None, request_mac: bytes | None = b'', ignore_trailing: bool = False*)→ Tuple[Message, float] [source]

Read a DNS message from a TCP socket.

sock, a `dns.asyncbackend.StreamSocket`.

See `dns.query.receive_tcp()` for the documentation of the other parameters, exceptions, and return type of this method.

TLS

---

async **dns.asyncquery.tls**(*q: Message, where: str, timeout: float | None = None, port: int = 853, source: str | None = None, source_port: int = 0, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, sock: StreamSocket | None = None, backend: Backend | None = None, ssl_context: SSLContext | None = None, server_hostname: str | None = None, verify: bool | str = True*)→ Message [source]

Return the response obtained after sending a query via TLS.

sock, an `asyncbackend.StreamSocket`, or `None`, the socket to use for the query. If `None`, the default, a socket is created. Note that if a socket is provided, it must be a connected SSL stream socket, and *where*, *port*, *source*, *source_port*, *backend*, *ssl_context*, and *server_hostname* are ignored.

backend, a `dns.asyncbackend.Backend`, or `None`. If `None`, the default, then dnspython will use the default backend.

See `dns.query.tls()` for the documentation of the other parameters, exceptions, and return type of this method.

HTTPS

async **dns.asyncquery.https**(*q: Message, where: str, timeout: float | None = None, port: int = 443, source: str | None = None, source_port: int = 0, one_rr_per_rrset: bool = False, ignore_trailing: bool = False, client: httpx.AsyncClient | None = None, path: str = '/dns-query', post: bool = True, verify: bool | str = True, bootstrap_address: str | None = None, resolver: dns.asyncresolver.Resolver | None = None, family: int | None = AddressFamily.AF_UNSPEC*) → Message [source]

Return the response obtained after sending a query via DNS-over-HTTPS.

client, a `httpx.AsyncClient`. If provided, the client to use for the query.

Unlike the other dnspython async functions, a backend cannot be provided in this function because httpx always auto-detects the async backend.

See `dns.query.https()` for the documentation of the other parameters, exceptions, and return type of this method.

Zone Transfers

async **dns.asyncquery.inbound_xfr**(*where: str, txn_manager: TransactionManager, query: Message | None = None, port: int = 53, timeout: float | None = None, lifetime: float | None = None, source: str | None = None, source_port: int = 0, udp_mode: UDPMode = UDPMode.NEVER, backend: Backend | None = None*) → None [source]

Conduct an inbound transfer and apply it via a transaction from the txn_manager.

backend, a `dns.asyncbackend.Backend`, or `None`. If `None`, the default, then dnspython will use the default backend.

See `dns.query.inbound_xfr()` for the documentation of the other parameters, exceptions, and return type of this method.

Stub Resolver

Dnspython's asyncresolver module implements an asynchronous "stub resolver".

module:: dns.asyncbackend .. _async-backend:

Asynchronous Backend Functions

Dnspython has a “backend” for Trio, Curio, and asyncio which implements the library-specific functionality needed by the generic asynchronous DNS code.

Dnspython attempts to determine which backend is in use by “sniffing” for it with the `sniffio` module if it is installed. If sniffio is not available, dnspython try to detect asyncio directly.

dns.asyncbackend.get_default_backend()→ Backend [source]

Get the default backend, initializing it if necessary.

dns.asyncbackend.set_default_backend(*name: str*)→ Backend [source]

Set the default backend.

It’s not normally necessary to call this method, as `get_default_backend()` will initialize the backend appropriately in many cases. If `sniffio` is not installed, or in testing situations, this function allows the backend to be set explicitly.

dns.asyncbackend.sniff()→ str [source]

Attempt to determine the in-use asynchronous I/O library by using the `sniffio` module if it is available.

Returns the name of the library, or raises AsyncLibraryNotFoundError if the library cannot be determined.

dns.asyncbackend.get_backend(*name: str*)→ Backend [source]

Get the specified asynchronous backend.

name, a `str`, the name of the backend. Currently the “trio” and “asyncio” backends are available.

Raises NotImplementedError if an unknown backend name is specified.

Rdataclasses

dns.rdataclass.ANY= *255*

dns.rdataclass.CH= *3*

dns.rdataclass.CHAOS= *3*

dns.rdataclass.HESIOD= *4*

dns.rdataclass.HS= *4*

dns.rdataclass.IN= *1*

dns.rdataclass.INTERNET= *1*

dns.rdataclass.NONE= *254*

dns.rdataclass.RESERVED0= *0*

Rdatatypes

dns.rdatatype.A= 1

dns.rdatatype.A6= 38

dns.rdatatype.AAAA= 28

dns.rdatatype.AFSDB= 18

dns.rdatatype.AMTRELAY= 260

dns.rdatatype.ANY= 255

dns.rdatatype.APL= 42

dns.rdatatype.AVC= 258

dns.rdatatype.AXFR= 252

dns.rdatatype.CAA= 257

dns.rdatatype.CDNSKEY= 60

dns.rdatatype.CDS= 59

dns.rdatatype.CERT= 37

dns.rdatatype.CNAME= 5

dns.rdatatype.CSYNC= 62

dns.rdatatype.DHCID= 49

dns.rdatatype.DLV= 32769

dns.rdatatype.DNAME= 39

dns.rdatatype.DNSKEY= 48

dns.rdatatype.DS= 43

dns.rdatatype.EUI48= 108

dns.rdatatype.EUI64= 109

dns.rdatatype.GPOS= 27

dns.rdatatype.HINFO= 13

dns.rdatatype.HIP= 55

dns.rdatatype.HTTPS= 65

dns.rdatatype.IPSECKEY= 45

dns.rdatatype.ISDN= 20

dns.rdatatype.IXFR= 251

dns.rdatatype.KEY= 25

dns.rdatatype.KX= 36

dns.rdatatype.L32= 105

dns.rdatatype.L64= 106

dns.rdatatype.LOC= 29

dns.rdatatype.LP= 107

dns.rdatatype.MAILA= 254

dns.rdatatype.MAILB= 253

dns.rdatatype.MB= 7

dns.rdatatype.MD= 3

dns.rdatatype.MF= 4

dns.rdatatype.MG= 8

dns.rdatatype.MINFO= 14

dns.rdatatype.MR= 9

dns.rdatatype.MX= 15

dns.rdatatype.NAPTR= 35

dns.rdatatype.NID= 104

dns.rdatatype.NINFO= 56

dns.rdatatype.NS= 2

dns.rdatatype.NSAP= 22

dns.rdatatype.NSAP_PTR= 23

dns.rdatatype.NSEC= 47

dns.rdatatype.NSEC3= 50

dns.rdatatype.NSEC3PARAM= 51

dns.rdatatype.NULL= 10

dns.rdatatype.NXT= 30

dns.rdatatype.OPENPGPKEY= 61

dns.rdatatype.OPT= 41

dns.rdatatype.PTR= 12

dns.rdatatype.PX= 26

dns.rdatatype.RP= 17

dns.rdatatype.RRSIG= 46

dns.rdatatype.RT= 21

dns.rdatatype.SIG= 24

dns.rdatatype.SMIMEA= 53

dns.rdatatype.SOA= 6

dns.rdatatype.SPF= 99

dns.rdatatype.SRV= 33

dns.rdatatype.SSHFP= 44

dns.rdatatype.SVCB= 64

dns.rdatatype.TA= 32768

dns.rdatatype.TKEY= 249

dns.rdatatype.TLSA= 52

dns.rdatatype.TSIG= 250

```
dns.rdatatype.TXT= 16
```

```
dns.rdatatype.TYPE0= 0
```

```
dns.rdatatype.UNSPEC= 103
```

```
dns.rdatatype.URI= 256
```

```
dns.rdatatype.WKS= 11
```

```
dns.rdatatype.X25= 19
```

```
dns.rdatatype.ZONEMD= 63
```

The dns.asyncresolver.Resolver Class

The async resolver is a subclass of `dns.resolver.Resolver` and has the same attributes. The methods are similar, but I/O methods like `resolve()` are asynchronous.

class **dns.asyncresolver.Resolver**(*filename: str = '/etc/resolv.conf', configure: bool = True*) [source]

Asynchronous DNS stub resolver.

filename, a `str` or file object, specifying a file in standard /etc/resolv.conf format. This parameter is meaningful only when *configure* is true and the platform is POSIX.

configure, a `bool`. If True (the default), the resolver instance is configured in the normal fashion for the operating system the resolver is running on. (I.e. by reading a /etc/resolv.conf file on POSIX systems and from the registry on Windows systems.)

async **canonical_name**(*name: Name | str*) → Name [source]

Determine the canonical name of *name*.

The canonical name is the name the resolver uses for queries after all CNAME and DNAME renamings have been applied.

name, a `dns.name.Name` or `str`, the query name.

This method can raise any exception that `resolve()` can raise, other than `dns.resolver.NoAnswer` and `dns.resolver.NXDOMAIN`.

Returns a `dns.name.Name`.

async **resolve**(*qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None, search: bool | None = None, backend: Backend | None = None*) → Answer [source]

Query nameservers asynchronously to find the answer to the question.

backend, a `dns.asyncbackend.Backend`, or `None`. If `None`, the default, then dnspython will use the default backend.

See `dns.resolver.Resolver.resolve()` for the documentation of the other parameters, exceptions, and return type of this method.

async **resolve_address**(*ipaddr: str, *args: Any, **kwargs: Any*) → Answer [source]

Use an asynchronous resolver to run a reverse query for PTR records.
This utilizes the resolve() method to perform a PTR lookup on the specified IP address.

ipaddr, a `str`, the IPv4 or IPv6 address you want to get the PTR record for.

All other arguments that can be passed to the resolve() function except for rdtype and rdclass are also supported by this function.

async **resolve_name**(*name: Name | str, family: int = AddressFamily.AF_UNSPEC, **kwargs: Any*)→ HostAnswers [source]

Use an asynchronous resolver to query for address records.

This utilizes the resolve() method to perform A and/or AAAA lookups on the specified name.

qname, a `dns.name.Name` or `str`, the name to resolve.

family, an `int`, the address family. If socket.AF_UNSPEC (the default), both A and AAAA records will be retrieved.

All other arguments that can be passed to the resolve() function except for rdtype and rdclass are also supported by this function.

async **try_ddr**(*lifetime: float = 5.0*)→ None [source]

Try to update the resolver's nameservers using Discovery of Designated Resolvers (DDR). If successful, the resolver will subsequently use DNS-over-HTTPS or DNS-over-TLS for future queries.

lifetime, a float, is the maximum time to spend attempting DDR. The default is 5 seconds.

If the SVCB query is successful and results in a non-empty list of nameservers, then the resolver's nameservers are set to the returned servers in priority order.

The current implementation does not use any address hints from the SVCB record, nor does it resolve addresses for the SCVB target name, rather it assumes that the bootstrap nameserver will always be one of the addresses and uses it. A future revision to the code may offer fuller support. The code verifies that the bootstrap nameserver is in the Subject Alternative Name field of the TLS certficate.

Asynchronous Resolver Functions

async **dns.asyncresolver.resolve**(*qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None, search: bool | None = None, backend: Backend | None = None*)→ Answer [source]

Query nameservers asynchronously to find the answer to the question.

This is a convenience function that uses the default resolver object to make the query.

See `dns.asyncresolver.Resolver.resolve()` for more information on the parameters.

async **dns.asyncresolver.resolve_address**(*ipaddr: str, *args: Any, **kwargs: Any*)→ Answer [source]

Use a resolver to run a reverse query for PTR records.

See `dns.asyncresolver.Resolver.resolve_address()` for more information on the parameters.

async **dns.asyncresolver.resolve_name**(*name: Name | str, family: int = AddressFamily.AF_UNSPEC, **kwargs: Any*)→ HostAnswers [source]

Use a resolver to asynchronously query for address records.

See `dns.asyncresolver.Resolver.resolve_name()` for more information on the parameters.

async **dns.asyncresolver.canonical_name**(*name: Name | str*)→ Name [source]

Determine the canonical name of *name*.

See `dns.resolver.Resolver.canonical_name()` for more information on the parameters and possible exceptions.

async **dns.asyncresolver.try_ddr**(*timeout: float = 5.0*)→ None [source]

Try to update the default resolver's nameservers using Discovery of Designated Resolvers (DDR). If successful, the resolver will subsequently use DNS-over-HTTPS or DNS-over-TLS for future queries.

See `dns.resolver.Resolver.try_ddr()` for more information.

async **dns.asyncresolver.zone_for_name**(*name: Name | str, rdclass: RdataClass = RdataClass.IN, tcp: bool = False, resolver: Resolver | None = None, backend: Backend | None = None*)→ Name [source]

Find the name of the zone which contains the specified name.
See `dns.resolver.Resolver.zone_for_name()` for more information on the parameters and possible exceptions.

async **dns.asyncresolver.make_resolver_at**(*where: Name | str, port: int = 53, family: int = AddressFamily.AF_UNSPEC, resolver: Resolver | None = None*)→ Resolver [source]

Make a stub resolver using the specified destination as the full resolver.

where, a `dns.name.Name` or `str` the domain name or IP address of the full resolver.

port, an `int`, the port to use. If not specified, the default is 53.

family, an `int`, the address family to use. This parameter is used if *where* is not an address. The default is `socket.AF_UNSPEC` in which case the first address returned by `resolve_name()` will be used, otherwise the first address of the specified family will be used.

resolver, a `dns.asyncresolver.Resolver` or `None`, the resolver to use for resolution of hostnames. If not specified, the default resolver will be used.

Returns a `dns.resolver.Resolver` or raises an exception.

async **dns.asyncresolver.resolve_at**(*where: Name | str, qname: Name | str, rdtype: RdataType | str = RdataType.A, rdclass: RdataClass | str = RdataClass.IN, tcp: bool = False, source: str | None = None, raise_on_no_answer: bool = True, source_port: int = 0, lifetime: float | None = None, search: bool | None = None, backend: Backend | None = None, port: int = 53, family: int = AddressFamily.AF_UNSPEC, resolver: Resolver | None = None*)→ Answer [source]

Query nameservers to find the answer to the question.

This is a convenience function that calls `dns.asyncresolver.make_resolver_at()` to make a resolver, and then uses it to resolve the query.

See `dns.asyncresolver.Resolver.resolve` for more information on the resolution parameters, and `dns.asyncresolver.make_resolver_at` for information about the resolver parameters *where*, *port*, *family*, and *resolver*.

If making more than one query, it is more efficient to call `dns.asyncresolver.make_resolver_at()` and then use that resolver for the queries instead of calling `resolve_at()` multiple times.

dns.asyncresolver.default_resolver = *None*

dns.asyncresolver.get_default_resolver()→ Resolver [source]

Get the default asynchronous resolver, initializing it if necessary.

dns.asyncresolver.reset_default_resolver()→ None [source]

Re-initialize default asynchronous resolver.

Note that the resolver configuration (i.e. /etc/resolv.conf on UNIX systems) will be re-read immediately.