lars Documentation

Release 0.1

Dave Hughes

July 27, 2013

CONTENTS

lars is a small suite of tools for converting httpd logs (from a variety of common servers like Apache, nginx, and IIS) into a format more conducive for loading into databases, the default being CSV.

The project is written in Python and is open-sourced under the MIT license. The source code can be obtained from GitHub.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Install

TODO To be written...

1.2 API Reference

The framework is designed in a modular fashion with a separate module for each log input format, each data output format, a few auxilliary modules for the datatypes exposed by the framework and their functionality. Where possible, standards dictating formats are linked in the API reference.

Each module comes with documentation including examples of usage. The best way to learn the framework is to peruse the API reference and try out the examples, modifying them to suit your purposes.

1.2.1 lars - Introduction

A typical lars script opens some log source, typically a file, and uses the source and target wrappers provided by lars to convert the log entries into some other format (potentially filtering and/or modifying the entries along the way). A trivial script to convert IIS W3C style log entries into a CSV file is shown below:

```
import io
from lars import iis, csv

with io.open('webserver.log', 'r') as infile, io.open('output.csv', 'wb') as outfile:
    with iis.IISSource(infile) as source, csv.CSVTarget(outfile) as target:
        for row in source:
            target.write(row)
```

Going through this section by section we can see the following:

1. The first couple of lines import the necessary modules that we'll need; the standard Python io module for opening files, and the `iis` and `csv` modules from lars for converting the data.
2. Using `io.open` we open the input file (with mode `'r'` for reading) and the output file (with mode `'wb'` for creating a new file and writing (binary mode) to it)
3. We wrap `infile` (the input file) with `IISSource` to parse the input file, and `outfile` (the output file) with `CSVTarget` to format the output file.
4. Finally, we use a simple loop to iterate over the rows in the source file, and the `write()` method to write them to the target.

This is the basic structure of most lars scripts. Most extra lines for filtering and manipulating rows appear within the loop at the end of the file, although sometimes extra module configuration lines are required at the top.

Filtering rows

The row object declared in the loop has attributes named after the columns of the source (with characters that cannot appear in Python identifiers replaced with underscores). To see the structure of a row you can simply print one and then terminate the loop:

```
import io
from lars import iis, csv

with io.open('webserver.log', 'r') as infile, io.open('output.csv', 'wb') as outfile:
    with iis.IISSource(infile) as source, csv.CSVTarget(outfile) as target:
        for row in source:
            print(row)
            break
```

Given the following input file (long lines indented for readability):

```
#Software: Microsoft Internet Information Services 6.0
#Version: 1.0
#Date: 2002-05-24 20:18:01
#Fields: date time c-ip cs-username s-ip s-port cs-method cs-uri-stem
    cs-uri-query sc-status sc-bytes cs-bytes time-taken cs(User-Agent)
    cs(Referrer)
2002-05-24 20:18:01 172.224.24.114 - 206.73.118.24 80 GET /Default.htm -
    200 7930 248 31
    Mozilla/4.0+(compatible;+MSIE+5.01;+Windows+2000+Server)
    http://64.224.24.114/
```

This will produce this output on the command line:

```
Row(date=Date(2002, 5, 24), time=Time(20, 18, 1),
    c_ip=IPv4Address(u'172.224.24.114'), cs_username=None,
    s_ip=IPv4Address(u'206.73.118.24'), s_port=80, cs_method=u'GET',
    cs_uri_stem=Url(scheme='', netloc='', path=u'/Default.htm', params='',
    query_str='', fragment=''), cs_uri_query=None, sc_status=200,
    sc_bytes=7930, cs_bytes=248, time_taken=31.0,
    cs_User_Agent=u'Mozilla/4.0 (compatible; MSIE 5.01; Windows 2000
    Server)', cs_Referrer=Url(scheme=u'http', netloc=u'64.224.24.114',
     path=u'/', params='', query_str='', fragment=''))
```

From this one can see that field names like `c-ip` have been converted into `c_ip` (`-` is an illegal character in Python identifiers). Furthermore it is apparent that instead of simple strings being extracted, the data has been converted into a variety of appropriate datatypes (`Date` for the `date` field, `Url` for the `cs-uri-stem` field, and so on). This significantly aids in filtering rows based upon sub-attributes of the extracted data.

For example, to filter on the year of the date:

```
if row.date.year == 2002:
    target.write(row)
```

Alternatively, you could filter on whether or not the client IP belongs in a particular network:

```
if row.c_ip in datatypes.network('172.0.0.0/8'):
    target.write(row)
```

Or use Python's string methods to filter on any string:

```
if row.cs_User_Agent.startswith('Mozilla/'):
    target.write(row)
```

Or any combination of the above:

```
if row.date.year == 2002 and 'MSIE' in row.cs_User_Agent:
    target.write(row)
```

Manipulating row content

If you wish to modify the output structure,the simplest method is to declare the row structure you want at the top of the file (using the `row()` function) and then construct rows with the new structure in the loop (using the result of the function):

```
import io
from lars import datatypes, iis, csv

NewRow = datatypes.row('date', 'time', 'client', 'url')

with io.open('webserver.log', 'r') as infile, io.open('output.csv', 'wb') as outfile:
    with iis.IISSource(infile) as source, csv.CSVTarget(outfile) as target:
        for row in source:
            new_row = NewRow(row.date, row.time, row.c_ip, row.cs_uri_stem)
            target.write(new_row)
```

There is no need to convert column data back to strings for output; all datatypes produced by lars source adapters have built-in string conversions which all target adapters know to use.

1.2.2 lars.apache - Reading Apache Logs

This module provides a wrapper for Apache log files, typically in common or combined format (but technically any Apache format which is can be unambiguously parsed with regexes).

The `ApacheSource` class is the major element that this module exports; this is the class which wraps a file-like object containing a common, combined, or otherwise Apache formatted log file and yields rows from it as tuples.

Classes

class `lars.apache.`**`ApacheSource`**(*source*, *log_format=COMMON*)

Wraps a stream containing a Apache formatted log file.

This wrapper converts a stream containing an Apache log file into an iterable which yields tuples. Each tuple has fieldnames derived from the following mapping of Apache format strings (which occur in the optional *log_format* parameter):

Format String	Field Name
%a	remote_ip
%A	local_ip
%B	size
%b	size
%{Foobar}C	cookie_Foobar (1)
%D	time_taken_ms
%{FOOBAR}e	env_FOOBAR (1)

Continued on next page

Table 1.1 – continued from previous page

Format String	Field Name
%f	filename
%h	remote_host
%H	protocol
%{Foobar}i	req_Foobar (1)
%k	keepalive
%l	ident
%m	method
%{Foobar}n	note_Foobar (1)
%{Foobar}o	resp_Foobar (1)
%p	port
%{canonical}p	port
%{local}p	local_port
%{remote}p	remote_port
%P	pid
%{pid}P	pid
%{tid}P	tid
%{hextid}P	hextid
%q	url_query
%r	request
%R	handler
%s	status
%t	time
%{format}t	time
%T	time_taken
%u	remote_user
%U	url_stem
%v	server_name
%V	canonical_name
%X	connection_status
%I	bytes_received
%O	bytes_sent

Notes:

1. Any characters in the field-name which are invalid in a Python identifier are converted to underscore, e.g. `%{foo-bar}C` becomes `"cookie_foo_bar"`.

Warning: The wrapper will only operate on *log_format* specifications that can be unambiguously parsed with a regular expression. In particular, this means that if a field can contain whitespace it must be surrounded by characters that it cannot legitimately contain (or cannot contain unescaped versions of). Typically double-quotes are used as Apache (from version 2.0.46) escapes double-quotes within `%r`, `%i`, and `%o`. See Apache's Custom Log Formats documentation for full details.

Parameters

- **source** – A file-like object containing the source stream
- **format** (*str*) – Defaults to `COMMON` but can be set to any valid Apache LogFormat string

`source`

The file-like object that the source reads rows from

count
Returns the number of rows successfully read from the source

log_format
The Apache LogFormat string that the class will use to decode rows

Data

lars.apache.**COMMON**
This string contains the Apache LogFormat string for the common log format (sometimes called the CLF). This is the default format for the ApacheSource class.

lars.apache.**COMMON_VHOST**
This string contains the Apache LogFormat strnig for the common log format with an additional virtual-host specification at the beginning of the string. This is a typical configuration used by several distributions of Apache which are configured with virtualhosts by default.

lars.apache.**COMBINED**
This string contains the Apache LogFormat string for the NCSA combined/extended log format. This is a popular variant that many server administrators use as it combines the COMMON format with REFERER and USER_AGENT formats.

lars.apache.**REFERER**
This string contains the (rudimentary) referer log format which is typically used in conjunction with the COMMON format.

lars.apache.**USER_AGENT**
This string contains the (rudimentary) user-agent log format which is typically used in conjunction with the COMMON format.

Exceptions

class lars.apache.**ApacheError**(*message*, *line_number=None*, *line=None*)
Base class for ApacheSource errors.

Exceptions of this class take the optional arguments line_number and line for specifying the index and content of the line that caused the error respectively. If specified, the __str__() method is overridden to include the line number in the error message.

Parameters

- **message** (*str*) – The error message
- **line_number** (*int*) – The 1-based index of the line that caused the error
- **line** (*str*) – The content of the line that caused the error

exception lars.apache.**ApacheWarning**
Raised when an error is encountered in parsing a log row.

Examples

A typical usage of this class is as follows:

```
import io
from lars import apache, csv

with io.open('/var/log/apache2/access.log', 'rb') as infile:
```

```
with io.open('access.csv', 'wb') as outfile:
    with apache.ApacheSource(infile) as source:
        with csv.CSVTarget(outfile) as target:
            for row in source:
                target.write(row)
```

1.2.3 lars.iis - Reading IIS Logs

This module provides a wrapper for W3C extended log files, typically used by the Microsoft IIS web-server.

The `IISSource` class is the major element that this module provides; this is the class which wraps a file-like object containing a W3C formatted log file and yields rows from it as tuples.

Classes

class `lars.iis.`**`IISSource`**(*source*)

Wraps a stream containing a IIS formatted log file.

This wrapper converts a stream containing a IIS formatted log file into an iterable which yields tuples. Each tuple is a namedtuple instance with the fieldnames of the tuple being the sanitized versions of the field names in the original log file (as specified in the `#Fields` directive).

The directives contained in the file can be obtained from attributes of the wrapper itself (useful in the case that relative timestamps, e.g. with the `#Date` directive, are being used) in which case the attribute will be the lower-cased version of the directive name without the `#` prefix.

Parameters **source** – A file-like object containing the source stream

`count`

Returns the number of rows successfully read from the source

`date`

The timestamp specified by the last encountered `#Date` directive (if any), as a `DateTime` instance

`fields`

A sequence of fields names found in the `#Fields` directive in the file header

`finish`

The timestamp found in the `#End-Date` directive (if any, as a `DateTime` instance)

`remark`

The remarks recorded in the `#Remark` directive (if any)

`software`

The name of the software which produced the source file as given by the `#Software` directive (if any)

`start`

The timestamp found in the `#Start-Date` directive (if any), as a `DateTime` instance

`version`

The version of the source file, as given by the `#Version` directive in the header

Exceptions

class `lars.iis.`**`IISError`**(*message*, *line_number=None*, *line=None*)

Base class for IISSource errors.

Exceptions of this class take the optional arguments line_number and line for specifying the index and content of the line that caused the error respectively. If specified, the `__str__()` method is overridden to include the line number in the error message.

Parameters

- **message** (*str*) – The error message
- **line_number** (*int*) – The 1-based index of the line that caused the error
- **line** (*str*) – The content of the line that caused the error

exception `lars.iis.`**`IISDirectiveError`**(*message*, *line_number=None*, *line=None*)
Raised when an error is encountered in any `#Directive`.

exception `lars.iis.`**`IISFieldsError`**(*message*, *line_number=None*, *line=None*)
Raised when an error is encountered in a `#Fields` directive.

exception `lars.iis.`**`IISVersionError`**(*message*, *line_number=None*, *line=None*)
Raised for a `#Version` directive with an unknown version is found.

exception `lars.iis.`**`IISWarning`**
Raised when an error is encountered in parsing a log row.

Examples

A typical usage of this class is as follows:

```
import io
from lars import iis, csv

with io.open('logs\iis.txt', 'rb') as infile:
    with io.open('iis.csv', 'wb') as outfile:
        with iis.IISSource(infile) as source:
            with csv.CSVTarget(outfile) as target:
                for row in source:
                    target.write(row)
```

Note for maintainers

The draft standard for the W3C Extended Log File Format is not well written (see the various notes and comments in the code); actual practice deviates from the draft in several areas, and the draft is deficient in describing what is potentially permitted in other areas.

Examples of the format as produced by IIS (the major user of the draft) can be found on MSDN. When maintaining the code below, please refer to both the draft (for information on what *could* be included in W3C log files) as well as the examples (for information on what typically *is* included in W3C log files, even when it outright violates the draft), and bear in mind Postel's Law.

1.2.4 lars.csv - Writing CSV Files

This module provides a target wrapper for CSV (Comma Separated Values) formatted text files, which are typically used as a generic source format for bulk loading databases.

The `CSVTarget` class is the major element that this module provides; it is a standard target class (a context manager with a `write()` method that accepts row tuples).

Classes

class `lars.csv.`**`CSVTarget`**(*fileobj*, *header=False*, *dialect=CSV_DIALECT*, *encoding='utf-8'*, ***kwargs*)

Wraps a stream to format rows as CSV (Comma Separated Values).

This wrapper provides a simple `write()` method which can be used to format row tuples as comma separated values in a variety of common dialects. The dialect defaults to `CSV_DIALECT` which produces a typical CSV file compatible with the vast majority of products.

If you desire a different output format you can either specify a different value for the *dialect* parameter, or if you only wish to use a minimal modification of the dialect you can override its attributes with keyword arguments. For example:

```
CSVTarget(outfile, dialect=CSV_DIALECT, lineterminator='\n')
```

The *encoding* parameter controls the character set used in the output. This defaults to UTF-8 which is a sensible default for most modern systems, but is a multi-byte encoding which some legacy systems (notably mainframes) may have troubles with. In this case you can either select a single byte encoding like ISO-8859-1 or even EBCDIC. See Python standard encodings for a full list of supported encodings.

> **Warning:** The file that you wrap with `CSVTarget` *must* be opened in binary mode (`'wb'`) partly because the dialect dictates the line terminator that is used, and partly because the class handles its own character encoding.

class `lars.csv.`**`CSV_DIALECT`**

This is the default dialect used by the `CSVTarget` class which has the following attributes:

Attribute	Value
delimiter	`','` (comma)
quotechar	`'"'` (double-quote)
quoting	`QUOTE_MINIMAL`
lineterminator	`'\r\n'` (DOS line breaks)
doublequote	True
escapechar	None

This dialect is compatible with Microsoft Excel and the vast majority of of other products which accept CSV as an input format. However, please note that some UNIX based database products require UNIX style line endings (`'\n'`) in which case you may wish to override the *lineterminator* attribute (see `CSVTarget` for more information).

class `lars.csv.`**`TSV_DIALECT`**

This is a dialect which produces tab-delimited files, another common data exchange format also supported by Microsoft Excel and numerous database products. This dialect has the following properties:

Attribute	Value
delimiter	`'\t'` (tab)
quotechar	`'"'` (double-quote)
quoting	`QUOTE_MINIMAL`
lineterminator	`'\r\n'` (DOS line breaks)
doublequote	True
escapechar	None

Data

`lars.csv.`**`QUOTE_NONE`**

This value indicates that no values should ever be quoted, even if they contain the delimiter character. In this case, any delimiter characters appearing the data will be preceded by the dialect's *escapechar* which should be set to an appropriate value. If *escapechar* is not set (None) an exception will be raised if any character that require quoting are encountered.

`lars.csv.`**`QUOTE_MINIMAL`**

This is the default quoting mode. In this mode the writer will only quote those values that contain the *delimiter* or *quotechar* characters, or any of the characters in *lineterminator*.

`lars.csv.`**`QUOTE_NONNUMERIC`**

This value tells the writer to quote all numeric (int and float) values.

`lars.csv.`**`QUOTE_ALL`**

This value simply tells the writer to quote all values written.

Examples

A typical example of working with the class is shown below:

```
import io
from lars import apache, csv

with io.open('/var/log/apache2/access.log', 'rb') as infile:
    with io.open('apache.csv', 'wb') as outfile:
        with apache.ApacheSource(infile) as source:
            with csv.CSVTarget(outfile, lineterminator='\n') as target:
                for row in source:
                    target.write(row)
```

1.2.5 lars.sql - Direct Database Output

This module provides a target wrapper for SQL-based databases, which can provide a powerful means of analyzing log data.

The `SQLTarget` class accepts row objects in its `write()` method and automatically generates the required SQL `INSERT` statements to append records to the specified target table.

The implementation has been tested with SQLite3 (built into Python), and PostgreSQL, but should work with any PEP-249 (Python DB API 2.0) compatible database cursor. A list of available Python database drives is maintained on the Python wiki DatabaseInterfaces page.

Classes

class `lars.sql.`**`SQLTarget`**(*db_module*, *connection*, *table*, *commit=1000*, *create_table=False*, *drop_table=False*, *ignore_drop_errors=True*, *str_type=u'VARCHAR(1000)'*, *int_type=u'INTEGER'*, *fixed_type=u'DOUBLE'*, *bool_type=u'SMALLINT'*, *date_type=u'DATE'*, *time_type=u'TIME'*, *datetime_type=u'TIMESTAMP'*, *ip_type=u'VARCHAR(53)'*, *hostname_type=u'VARCHAR(255)'*, *path_type=u'VARCHAR(260)'*)

Wraps a database connection to insert row tuples into an SQL database table.

This wrapper provides a simple `write()` method which can be used to insert row tuples into a specified table, which can optionally by created automatically by the wrapper before insertion of the first row. The wrapper must be passed a database connection object that conforms to the Python DB-API (version 2.0) as defined by PEP-249.

The *db_module* parameter must be passed the module that defines the database interface (this odd requirement is so that the wrapper can look up the parameter style that the interface uses, and the exceptions that it declares).

The *connection* parameter must be given an active database connection object (presumably belonging to the module passed to *db_module*).

The *table* parameter is the final mandatory parameter which names the table that values are to be inserted into. If the table name requires quoting in the target SQL dialect, you should include such quoting in the *table* value (this class does not try and discern what database engine you are connecting to and thus has no idea about non-standard quoting styles like `` `MySQL` `` or `[MS-SQL]`).

The *commit* parameter controls how often a `COMMIT` statement is executed when inserting rows. By default, this is 1000 which is usually sufficient to provide decent performance but may (in certain database engines with fixed size transaction logs) cause errors, in which case you may wish to specify a lower value.

If the *create_table* parameter is set to True (it defaults to False), when the `write()` method is first called, the class will determine column names and types from the row passed in and will attempt to generate and execute a `CREATE TABLE` statement to set up the target table automatically. The database types that are used in the `CREATE TABLE` statement are controlled by other optional parameters and are documented in the table below:

Parameter	Default Value (SQL)
str_type	`VARCHAR(1000)` - typically used for URL fields.
int_type	`INTEGER` - used for fields like status and size. If your server is serving large binaries you may wish to use a 64-bit type like `BIGINT` here instead.
fixed_type	`DOUBLE` - used for fields like time_taken. Some users may wish to change this an appropriate `NUMERIC` or `DECIMAL` specification for precision.
bool_type	`SMALLINT` - used for any boolean values in the input (0 for False, 1 for True)
date_type	`DATE`
time_type	`TIME`
datetime_type	`TIMESTAMP` - MS-SQL users will likely wish to change this to `DATETIME` or `SMALLDATETIME`. MySQL users may wish to change this to `DATETIME`, although `TIMESTAMP` is technically also supported (albeit with functional differences).
ip_type	`VARCHAR(53)` - this is sufficient for storing all possible IP address and port combinations up and including an IPv6 v4-mapped address. If you are certain you will only need IPv4 support you may wish to use a length of 21 (with port) or 15 (no port). PostgreSQL users may wish to use the special `inet` type instead as this is much more efficient but cannot store port information.
hostname_type	`VARCHAR(255)`
path_type	`VARCHAR(260)`

If the *drop_table* parameter is set to True (it defaults to False), the wrapper will first attempt to use `DROP TABLE` to destroy any existing table before attempting `CREATE TABLE`. If *ignore_drop_errors* is True (which it is by default) then any errors encountered during the drop operation (e.g. if the table does not exist) will be ignored.

`commit`

The number of rows which the class will attempt to write before performing a COMMIT. It is strongly recommended to set this to a reasonably large number (e.g. 1000) to ensure decent INSERT performance

`count`

Returns the number of rows successfully written to the database so far

`create_table`

If True, the class will attempt to create the target table during the first call to the `write()` method

drop_table
If True, the class will attempt to unconditionally drop any existing target table during the first call to the `write()` method

ignore_drop_errors
If True, and `drop_table` is True, any errors encountered during the `DROP TABLE` operation will be ignored (typically useful when you are not sure the target table exists or not)

table
The name of the target table in the database, including any required escaping or quotation

Exceptions

exception `lars.sql.`**`SQLError`**
Base class for all fatal errors generated by classes in the sql module.

exception `lars.sql.`**`SQLWarning`**
Raised when an error is encountered inserting a log row.

Examples

A typical example of working with the class is shown below:

```
import io
import sqlite3
from lars import apache, sql

connection = sqlite3.connect('apache.db', detect_types=sqlite3.PARSE_DECLTYPES)

with io.open('/var/log/apache2/access.log', 'rb') as infile:
    with io.open('apache.csv', 'wb') as outfile:
        with apache.ApacheSource(infile) as source:
            with sql.SQLTarget(sqlite3, connection, 'log_entries', create_table=True) as target:
                for row in source:
                    target.write(row)
```

1.2.6 lars.geoip - GeoIP Database Access

This module provides a common interface to the GeoIP database. Most users will only need to be aware of the `init_database()` function in this module, which is used to initialize the GeoIP database(s). All other functions should be ignored; instead, users should use the `country`, `region`, `city`, and `coords` attributes of the `IPv4Address` and `IPv6Address` classes.

Functions

`lars.geoip.`**`init_database`**(*v4_filename*, *v6_filename=None*, *memcache=True*)
Initializes the global GeoIP database instance in a thread-safe manner.

This function opens GeoIP databases for use by the `IPv4Address` and `IPv6Address` classes. GeoIP databases are hierarchical: if you open a country-only database, you will only be able to use country-level lookups. However, city-level databases enable all supported lookups (country, region, city, and coordinates).

By default, the function caches the entire content of (both) the database(s) in memory (on the assumption that just about any machine has more than sufficient RAM for this), but this behaviour can be overridden with the *memcache* parameter.

The optional *v6_filename* parameter specifies the location of the IPv6 database which will be used for IPv6 addresses. The GeoIP IPv6 databases are orthogonal to the IPv4 databases (you cannot lookup IPv4 addresses using an IPv6 database) - hence why the two databases are stored and specified separately.

> **Warning:** At the time of writing, the free GeoLite IPv6 city-level database does not work (the authors seem to be using a new database format which the pygeoip API does not yet know). This does not affect the IPv4 city-level database.

Parameters

- **v4_filename** (*str*) – The filename of the IPv4 database
- **v6_filename** (*str*) – The filename of the IPv6 database (optional)
- **memcache** (*bool*) – Set to False if you don't wish to cache the db in RAM (optional)

`lars.geoip.`**`country_code_by_addr`**`(`*address*`)`

Returns the country code associated with the specified address. You should use the `country` attribute instead of this function.

Parameters **address** (*str*) – The address to lookup the country for

Returns str The country code associated with the address, or None

`lars.geoip.`**`country_code_by_addr_v6`**`(`*address*`)`

Returns the country code associated with the specified address. You should use the `country` attribute instead of this function.

Parameters **address** (*str*) – The address to lookup the country for

Returns str The country code associated with the address, or None

`lars.geoip.`**`city_by_addr`**`(`*address*`)`

Returns the city associated with the address. You should use the `city` attribute instead of this function.

Given an address, this function returns the city associated with it. Note: this function will raise an exception if the GeoIP database loaded is above city level.

Parameters **address** (*str*) – The address to lookup the city for

Returns str The city associated with the address, or None

`lars.geoip.`**`city_by_addr_v6`**`(`*address*`)`

Returns the city associated with the address. You should use the `city` attribute instead of this function.

Given an address, this function returns the city associated with it. Note: this function will raise an exception if the GeoIP database loaded is above city level.

Parameters **address** (*str*) – The address to lookup the city for

Returns str The city associated with the address, or None

`lars.geoip.`**`region_by_addr`**`(`*address*`)`

Returns the region (e.g. state) associated with the address. You should use the `region` attribute instead of this function.

Given an address, this function returns the region associated with it. In the case of the US, this is the state. In the case of other countries it may be a state, county, something GeoIP-specific or simply undefined. Note: this function will raise an exception if the GeoIP database loaded is country-level only.

Parameters **address** (*str*) – The address to lookup the region for

Returns str The region associated with the address, or None lars.geoip.**region_by_addr_v6**(*address*)

Returns the region (e.g. state) associated with the address. You should use the `region` attribute instead of this function.

Given an address, this function returns the region associated with it. In the case of the US, this is the state. In the case of other countries it may be a state, county, something GeoIP-specific or simply undefined. Note: this function will raise an exception if the GeoIP database loaded is country-level only.

Parameters **address** (*str*) – The address to lookup the region for

Returns str The region associated with the address, or None

lars.geoip.**coords_by_addr**(*address*)

Returns the coordinates (long, lat) associated with the address. You should use the `coords` attribute instead of this function.

Given an address, this function returns a tuple with the attributes longitude and latitude (in that order) representing the (very) approximate coordinates of the address on the globe. Note: this function will raise an exception if the GeoIP database loaded is above city level.

Parameters **address** (*str*) – The address to locate

Returns str The coordinates associated with the address, or None

lars.geoip.**coords_by_addr_v6**(*address*)

Returns the coordinates (long, lat) associated with the address. You should use the `coords` attribute instead of this function.

Given an address, this function returns a tuple with the attributes longitude and latitude (in that order) representing the (very) approximate coordinates of the address on the globe. Note: this function will raise an exception if the GeoIP database loaded is above city level.

Parameters **address** (*str*) – The address to locate

Returns str The coordinates associated with the address, or None

Examples

1.2.7 lars.datatypes - Web Log Datatypes

This module wraps various Python data-types which are commonly found in log files to provide them with default string coercions and enhanced attributes. Each datatype is given a simple constructor function which accepts a string in a common format (for example, the `date()` function which accepts dates in `YYYY-MM-DD` format), and returns the converted data.

Most of the time you will not need the functions in this module directly, but the attributes of the classes are extremely useful for filtering and transforming log data for output.

Classes

class lars.datatypes.**DateTime**

Represents a timestamp.

This type is returned by the `datetime()` function and represents a timestamp (with optional timezone). A `DateTime` object is a single object containing all the information from a `Date` object and a `Time` object. Like a `Date` object, `DateTime` assumes the current Gregorian calendar extended in both directions; like a time object, `DateTime` assumes there are exactly 3600*24 seconds in every day.

Other constructors, all class methods:

classmethod `today` ()

Return the current local datetime, with `tzinfo` `None`. This is equivalent to `DateTime.fromtimestamp(time.time())`. See also `now()`, `fromtimestamp()`.

classmethod `now` ([*tz*])

Return the current local date and time. If optional argument *tz* is `None` or not specified, this is like `today()`, but, if possible, supplies more precision than can be gotten from going through a `time.time()` timestamp (for example, this may be possible on platforms supplying the C `gettimeofday()` function).

Else *tz* must be an instance of a class `tzinfo` subclass, and the current date and time are converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(DateTime.utcnow().replace(tzinfo=tz))`. See also `today()`, `utcnow()`.

classmethod `utcnow` ()

Return the current UTC date and time, with `tzinfo` `None`. This is like `now()`, but returns the current UTC date and time, as a naive `DateTime` object. See also `now()`.

classmethod `fromtimestamp` (*timestamp*[, *tz*])

Return the local date and time corresponding to the POSIX timestamp, such as is returned by `time.time()`. If optional argument *tz* is `None` or not specified, the timestamp is converted to the platform's local date and time, and the returned `DateTime` object is naive.

Else *tz* must be an instance of a class `tzinfo` subclass, and the timestamp is converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(DateTime.utcfromtimestamp(timestamp).replace(tzinfo=tz))`.

`fromtimestamp()` may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by `fromtimestamp()`, and then it's possible to have two timestamps differing by a second that yield identical `DateTime` objects. See also `utcfromtimestamp()`.

classmethod `utcfromtimestamp` (*timestamp*)

Return the UTC `DateTime` corresponding to the POSIX timestamp, with `tzinfo` `None`. This may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `gmtime()` function. It's common for this to be restricted to years in 1970 through 2038. See also `fromtimestamp()`.

classmethod `combine` (*date*, *time*)

Return a new `DateTime` object whose date components are equal to the given `date` object's, and whose time components and `tzinfo` attributes are equal to the given `Time` object's. For any `DateTime` object *d*, `d == DateTime.combine(d.date(), d.timetz())`. If date is a `DateTime` object, its time components and `tzinfo` attributes are ignored.

classmethod `strptime` (*date_string*, *format*)

Return a `DateTime` corresponding to *date_string*, parsed according to *format*. This is equivalent to `DateTime(*(time.strptime(date_string, format)[0:6]))`. `ValueError` is raised if the date_string and format can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple.

Class attributes:

`min`

The earliest representable `DateTime`.

`max`

The latest representable `DateTime`.

resolution
The smallest possible difference between non-equal `DateTime` objects, `timedelta(microseconds=1)`.

Instance attributes (read-only):

year
Between `MINYEAR` and `MAXYEAR` inclusive.

month
Between 1 and 12 inclusive.

day
Between 1 and the number of days in the given month of the given year.

hour
In `range(24)`.

minute
In `range(60)`.

second
In `range(60)`.

microsecond
In `range(1000000)`.

tzinfo
The object passed as the *tzinfo* argument to the `DateTime` constructor, or `None` if none was passed.

Supported operations:

Operation	Result
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 < datetime2`	Compares `DateTime` to `DateTime`. (4)

1. datetime2 is a duration of timedelta removed from datetime1, moving forward in time if `timedelta.days` > 0, or backward if `timedelta.days` < 0. The result has the same `tzinfo` attribute as the input datetime, and datetime2 - datetime1 == timedelta after. `OverflowError` is raised if datetime2.year would be smaller than `MINYEAR` or larger than `MAXYEAR`. Note that no time zone adjustments are done even if the input is an aware object.

2. Computes the datetime2 such that datetime2 + timedelta == datetime1. As for addition, the result has the same `tzinfo` attribute as the input datetime, and no time zone adjustments are done even if the input is aware. This isn't quite equivalent to datetime1 + (-timedelta), because -timedelta in isolation can overflow in cases where datetime1 - timedelta does not.

3. Subtraction of a `DateTime` from a `DateTime` is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, `TypeError` is raised.

   If both are naive, or both are aware and have the same `tzinfo` attribute, the `tzinfo` attributes are ignored, and the result is a `timedelta` object *t* such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

   If both are aware and have different `tzinfo` attributes, `a-b` acts as if *a* and *b* were first converted to naive UTC datetimes first. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

4.*datetime1* is considered less than *datetime2* when *datetime1* precedes *datetime2* in time.

If one comparand is naive and the other is aware, `TypeError` is raised. If both comparands are aware, and have the same `tzinfo` attribute, the common `tzinfo` attribute is ignored and the base datetimes are compared. If both comparands are aware and have different `tzinfo` attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

Note: In order to stop comparison from falling back to the default scheme of comparing object addresses, datetime comparison normally raises `TypeError` if the other comparand isn't also a `DateTime` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `DateTime` object is compared to an object of a different type, `TypeError` is raised unless the comparison is == or `!=`. The latter cases return `False` or `True`, respectively.

`DateTime` objects can be used as dictionary keys. In Boolean contexts, all `DateTime` objects are considered to be true.

Instance methods:

`date`()

Return `date` object with same year, month and day.

`time`()

Return `Time` object with same hour, minute, second and microsecond. `tzinfo` is `None`. See also method `timetz()`.

`timetz`()

Return `Time` object with same hour, minute, second, microsecond, and tzinfo attributes. See also method `time()`.

`replace`([*year*[, *month*[, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]]]]])

Return a DateTime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive DateTime from an aware DateTime with no conversion of date and time data.

`astimezone`(*tz*)

Return a `DateTime` object with new `tzinfo` attribute *tz*, adjusting the date and time data so the result is the same UTC time as *self*, but in *tz*'s local time.

tz must be an instance of a `tzinfo` subclass, and its `utcoffset()` and `dst()` methods must not return `None`. *self* must be aware (`self.tzinfo` must not be `None`, and `self.utcoffset()` must not return `None`).

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in time zone *tz*, representing the same UTC time as *self*: after `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` will usually have the same date and time data as `dt - dt.utcoffset()`. The discussion of class `tzinfo` explains the cases at Daylight Saving Time transition boundaries where this cannot be achieved (an issue only if *tz* models both standard and daylight time).

If you merely want to attach a time zone object *tz* to a DateTime *dt* without adjustment of date and time data, use `dt.replace(tzinfo=tz)`. If you merely want to remove the time zone object from an aware DateTime *dt* without conversion of date and time data, use `dt.replace(tzinfo=None)`.

Note that the default `tzinfo.fromutc()` method can be overridden in a `tzinfo` subclass to affect the result returned by `astimezone()`. Ignoring error cases, `astimezone()` acts like:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

utcoffset()

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.utcoffset(self)`, and raises an exception if the latter doesn't return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

dst()

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.dst(self)`, and raises an exception if the latter doesn't return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

tzname()

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.tzname(self)`, raises an exception if the latter doesn't return `None` or a string object,

weekday()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also `isoweekday()`.

isoweekday()

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also `weekday()`, `isocalendar()`.

isocalendar()

Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as `self.date().isocalendar()`.

isoformat([*sep*])

Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if `microsecond` is 0, YYYY-MM-DDTHH:MM:SS

If `utcoffset()` does not return `None`, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM or, if `microsecond` is 0 YYYY-MM-DDTHH:MM:SS+HH:MM

The optional argument *sep* (default `'T'`) is a one-character separator, placed between the date and time portions of the result. For example,

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

class `lars.datatypes.`**`Date`**

Represents a date.

This type is returned by the `date()` function and represents a date. A `Date` object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book Calendrical Calculations, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

Other constructors, all class methods:

classmethod `today``()`
Return the current local date. This is equivalent to `date.fromtimestamp(time.time())`.

classmethod `fromtimestamp``(`*timestamp*`)`
Return the local date corresponding to the POSIX timestamp, such as is returned by `time.time()`. This may raise `ValueError`, if the timestamp is out of the range of values supported by the platform C `localtime()` function. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by `fromtimestamp()`.

Class attributes:

`min`
The earliest representable date, `date(MINYEAR, 1, 1)`.

`max`
The latest representable date, `date(MAXYEAR, 12, 31)`.

`resolution`
The smallest possible difference between non-equal date objects, `timedelta(days=1)`.

Instance attributes (read-only):

`year`
Between `MINYEAR` and `MAXYEAR` inclusive.

`month`
Between 1 and 12 inclusive.

`day`
Between 1 and the number of days in the given month of the given year.

Supported operations:

Operation	Result
`date2 = date1 + timedelta`	*date2* is `timedelta.days` days removed from *date1*. (1)
`date2 = date1 - timedelta`	Computes *date2* such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 < date2`	*date1* is considered less than *date2* when *date1* precedes *date2* in time. (4)

Notes:

1. *date2* is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. `OverflowError` is raised if `date2.year` would be smaller than `MINYEAR` or larger than `MAXYEAR`.

2. This isn't quite equivalent to date1 + (-timedelta), because -timedeltan i isolation can overflow in cases where date1 - timedelta does not . `timedelta.seconds` and `timedelta.microseconds` are ignored .

3. This is exact, and cannot overflow. timedelta.seconds and timedelta.microseconds are 0, and date2 + timedelta == date1 after.

4.In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`. In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises `TypeError` if the other comparand isn't also a `date` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `date` object is compared to an object of a different type, `TypeError` is raised unless the comparison is == or `!=`. The latter cases return `False` or `True`, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all `date` objects are considered to be true.

Instance methods:

`replace`(*year*, *month*, *day*)

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if `d == Date(2002, 12, 31)`, then `d.replace(day=26) == Date(2002, 12, 26)`.

`weekday`()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, `Date(2002, 12, 4).weekday() == 2`, a Wednesday. See also `isoweekday()`.

`isoweekday`()

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `Date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also `weekday()`, `isocalendar()`.

`isocalendar`()

Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian calendar. See http://www.phys.uu.nl/~vgent/calendar/isocalendar.htm for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that `Date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `Date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

`isoformat`()

Return a string representing the date in ISO 8601 format, 'YYYY-MM-DD'. For example, `Date(2002, 12, 4).isoformat() == '2002-12-04'`.

`strftime`(*format*)

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values.

class `lars.datatypes.`**`Hostname`**(*s*)

Represents an Internet hostname and provides attributes for DNS resolution.

This type is returned by the `hostname()` function and represents a DNS hostname. The `address` property allows resolution of the hostname to an IP address.

Parameters **hostname** (*str*) – The hostname to parse

`address`

Attempts to resolve the hostname into an IPv4 or IPv6 address (returning an `IPv4Address` or `IPv6Address` object repsectively). The result of the DNS query (including negative lookups is cached, so repeated queries for the same hostname should be extremely fast.

class `lars.datatypes.`**`IPv4Address`**(*address*)

Represents an IPv4 address.

This type is returned by the `address()` function and represents an IPv4 address and provides various attributes and comparison operators relevant to such addresses.

For example, to test whether an address belongs to particular network you can use the `in` operator with the result of the `network()` function:

```
address('192.168.0.64') in network('192.168.0.0/16')
```

The `hostname` attribute will perform reverse DNS resolution to determine a hostname associated with the address (if any). The result of the query (including negative lookups) is cached so subsequent queries of the same address should be extermely rapid.

If the `geoip` module has been initialized with a database, the GeoIP-related attributes `country`, `region`, `city`, and `coords` will return the country, region, city and a (longitude, latitude) tuple respectively.

compressed
Returns the shorthand version of the IP address as a string (this is the default string conversion).

exploded
Returns the longhand version of the IP address as a string.

is_link_local
Returns True if the address is reserved for link-local. See RFC 3927 for details.

is_loopback
Returns True if the address is a loopback address. See RFC 3330 for details.

is_multicast
Returns True if the address is reserved for multicast use. See RFC 3171 for details.

is_private
Returns True if this address is allocated for private networks. See RFC 1918 for details.

is_reserved
Returns True if the address is otherwise IETF reserved.

is_unspecified
Returns True if the address is unspecified. See RFC 5735 3 for details.

packed
Returns the binary representation of this address.

city
If `init_database()` has been called with a city-level GeoIP database, returns the city of the address.

coords
If `init_database()` has been called with a city-level GeoIP database, returns a (longitude, latitude) tuple describing the approximate location of the address.

country
If `init_database()` has been called to initialize a GeoIP database, returns the country of the address.

hostname
Performs a reverse DNS lookup to attempt to determine a hostname for the address. Lookups (including negative lookups) are cached so that repeated lookups are extremely quick. Returns a `Hostname` object if the lookup is successful, or None.

region
If `init_database()` has been called with a region-level (or lower) GeoIP database, returns the region of the address.

class `lars.datatypes.`**`IPv4Network`**(*address*, *strict=True*)
This type is returned by the `network()` function. This class represents and manipulates 32-bit IPv4 networks.

Attributes: [examples for IPv4Network('192.0.2.0/27')]

- network_address: IPv4Address('192.0.2.0')
- hostmask: IPv4Address('0.0.0.31')
- broadcast_address: IPv4Address('192.0.2.32')
- netmask: IPv4Address('255.255.255.224')
- prefixlen: 27

address_exclude(*other*)

Remove an address from a larger block.

For example:

```
addr1 = network('192.0.2.0/28')
addr2 = network('192.0.2.1/32')
addr1.address_exclude(addr2) = [
    IPv4Network('192.0.2.0/32'), IPv4Network('192.0.2.2/31'),
    IPv4Network('192.0.2.4/30'), IPv4Network('192.0.2.8/29'),
    ]
```

Parameters **other** – An IPv4Network object of the same type.

Returns An iterator of the IPv4Network objects which is self minus other.

compare_networks(*other*)

Compare two IP objects.

This is only concerned about the comparison of the integer representation of the network addresses. This means that the host bits aren't considered at all in this method. If you want to compare host bits, you can easily enough do a `HostA._ip < HostB._ip`.

Parameters **other** – An IP object.

Returns -1, 0, or 1 for less than, equal to or greater than respectively.

hosts()

Generate iterator over usable hosts in a network.

This is like `__iter__()` except it doesn't return the network or broadcast addresses.

overlaps(*other*)

Tells if self is partly contained in *other*.

subnets(*prefixlen_diff=1*, *new_prefix=None*)

The subnets which join to make the current subnet.

In the case that self contains only one IP (self._prefixlen == 32 for IPv4 or self._prefixlen == 128 for IPv6), yield an iterator with just ourself.

Parameters

- **prefixlen_diff** (*int*) – An integer, the amount the prefix length should be increased by. This should not be set if *new_prefix* is also set.
- **new_prefix** (*int*) – The desired new prefix length. This must be a larger number (smaller prefix) than the existing prefix. This should not be set if *prefixlen_diff* is also set.

Returns An iterator of IPv(4|6) objects.

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing the current network.

Parameters

- **prefixlen_diff** (*int*) – An integer, the amount the prefix length of the network should be decreased by. For example, given a `/24` network and a prefixlen_diff of `3`, a supernet with a `/21` netmask is returned.
- **new_prefix** (*int*) – The desired new prefix length. This must be a smaller number (larger prefix) than the existing prefix. This should not be set if *prefixlen_diff* is also set.

Returns An IPv4Network object.

`is_link_local`
Returns True if the address is reserved for link-local. See RFC 4291 for details.

`is_loopback`
Returns True if the address is a loopback address. See RFC 2373 2.5.3 for details.

`is_multicast`
Returns True if the address is reserved for multicast use. See RFC 2373 2.7 for details.

`is_private`
Returns True if this address is allocated for private networks. See RFC 4193 for details.

`is_reserved`
Returns True if the address is otherwise IETF reserved.

`is_unspecified`
Returns True if the address is unspecified. See RFC 2373 2.5.2 for details.

class `lars.datatypes.`**`IPv4Port`**`(`*address*`)`
Represents an IPv4 address and port number.

This type is returned by the `address()` function and represents an IPv4 address and port number. Other than this, all properties of the base `IPv4Address` class are equivalent.

`port`
An integer representing the network port for a connection

class `lars.datatypes.`**`IPv6Address`**`(`*address*`)`
Represents an IPv6 address.

This type is returned by the `address()` function and represents an IPv6 address and provides various attributes and comparison operators relevant to such addresses.

For example, to test whether an address belongs to particular network you can use the `in` operator with the result of the `network()` function:

```
address('::1') in network('::/16')
```

The `hostname` attribute will perform reverse DNS resolution to determine a hostname associated with the address (if any). The result of the query (including negative lookups) is cached so subsequent queries of the same address should be extermely rapid.

If the `geoip` module has been initialized with a database, the GeoIP-related attributes `country`, `region`, `city`, and `coords` will return the country, region, city and a (longitude, latitude) tuple respectively.

`compressed`
Returns the shorthand version of the IP address as a string (this is the default string conversion).

`exploded`
Returns the longhand version of the IP address as a string.

`ipv4_mapped`
Returns the IPv4 mapped address if the IPv6 address is a v4 mapped address, or `None` otherwise.

is_link_local
Returns True if the address is reserved for link-local. See RFC 4291 for details.

is_loopback
Returns True if the address is a loopback address. See RFC 2373 2.5.3 for details.

is_multicast
Returns True if the address is reserved for multicast use. See RFC 2373 2.7 for details.

is_private
Returns True if this address is allocated for private networks. See RFC 4193 for details.

is_reserved
Returns True if the address is otherwise IETF reserved.

is_site_local
Returns True if the address is reserved for site-local.

Note that the site-local address space has been deprecated by RFC 3879. Use `is_private` to test if this address is in the space of unique local addresses as defined by RFC 4193. See RFC 3513 2.5.6 for details.

is_unspecified
Returns True if the address is unspecified. See RFC 2373 2.5.2 for details.

packed
Returns the binary representation of this address.

sixtofour
Returns the IPv4 6to4 embedded address if present, or `None` if the address doesn't appear to contain a 6to4 embedded address.

teredo
Returns a `(server, client)` tuple of embedded Teredo IPs, or `None` if the address doesn't appear to be a Teredo address (doesn't start with `2001::/32`).

city
If `init_database()` has been called with a city-level GeoIP IPv6 database, returns the city of the address.

coords
If `init_database()` has been called with a city-level GeoIP IPv6 database, returns a (longitude, latitude) tuple describing the approximate location of the address.

country
If `init_database()` has been called to initialize a GeoIP IPv6 database, returns the country of the address.

hostname
Performs a reverse DNS lookup to attempt to determine a hostname for the address. Lookups (including negative lookups) are cached so that repeated lookups are extremely quick. Returns a `Hostname` object if the lookup is successful, or None.

region
If `init_database()` has been called with a region-level (or lower) GeoIP IPv6 database, returns the region of the address.

class `lars.datatypes.`**`IPv6Network`**(*address*, *strict=True*)
This type is returned by the `network()` function. This class represents and manipulates 128-bit IPv6 networks.

address_exclude(*other*)
Remove an address from a larger block.

For example:

```
addr1 = network('192.0.2.0/28')
addr2 = network('192.0.2.1/32')
addr1.address_exclude(addr2) = [
    IPv4Network('192.0.2.0/32'), IPv4Network('192.0.2.2/31'),
    IPv4Network('192.0.2.4/30'), IPv4Network('192.0.2.8/29'),
    ]
```

Parameters **other** – An IPv4Network object of the same type.

Returns An iterator of the IPv4Network objects which is self minus other.

compare_networks(*other*)

Compare two IP objects.

This is only concerned about the comparison of the integer representation of the network addresses. This means that the host bits aren't considered at all in this method. If you want to compare host bits, you can easily enough do a `HostA._ip < HostB._ip`.

Parameters **other** – An IP object.

Returns -1, 0, or 1 for less than, equal to or greater than respectively.

hosts()

Generate iterator over usable hosts in a network.

This is like `__iter__()` except it doesn't return the network or broadcast addresses.

overlaps(*other*)

Tells if self is partly contained in *other*.

subnets(*prefixlen_diff=1*, *new_prefix=None*)

The subnets which join to make the current subnet.

In the case that self contains only one IP (self._prefixlen == 32 for IPv4 or self._prefixlen == 128 for IPv6), yield an iterator with just ourself.

Parameters

- **prefixlen_diff** (*int*) – An integer, the amount the prefix length should be increased by. This should not be set if *new_prefix* is also set.
- **new_prefix** (*int*) – The desired new prefix length. This must be a larger number (smaller prefix) than the existing prefix. This should not be set if *prefixlen_diff* is also set.

Returns An iterator of IPv(4|6) objects.

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing the current network.

Parameters

- **prefixlen_diff** (*int*) – An integer, the amount the prefix length of the network should be decreased by. For example, given a `/24` network and a prefixlen_diff of `3`, a supernet with a `/21` netmask is returned.
- **new_prefix** (*int*) – The desired new prefix length. This must be a smaller number (larger prefix) than the existing prefix. This should not be set if *prefixlen_diff* is also set.

Returns An IPv4Network object.

is_link_local

Returns True if the address is reserved for link-local. See RFC 4291 for details.

is_loopback
Returns True if the address is a loopback address. See RFC 2373 2.5.3 for details.

is_multicast
Returns True if the address is reserved for multicast use. See RFC 2373 2.7 for details.

is_private
Returns True if this address is allocated for private networks. See RFC 4193 for details.

is_reserved
Returns True if the address is otherwise IETF reserved.

is_unspecified
Returns True if the address is unspecified. See RFC 2373 2.5.2 for details.

class lars.datatypes.**IPv6Port**(*address*)
Represents an IPv6 address and port number.

This type is returned by the address() function an represents an IPv6 address and port number. The string representation of an IPv6 address with port necessarily wraps the address portion in square brakcets as otherwise the port number will make the address ambiguous. Other than this, all properties of the base IPv6Address class are equivalent.

port
An integer representing the network port for a connection

class lars.datatypes.**Path**
Represents a path.

This type is returned by the path() function and represents a path in POSIX format (forward slash separators and no drive portion). It is used to represent the path portion of URLs and provides attributes for extracting parts of the path there-in.

The original path can be obtained as a string by asking for the string conversion of this class, like so:

```
p = datatypes.path('/foo/bar/baz.ext')
assert p.dirname == '/foo/bar'
assert p.basename == 'baz.ext'
assert str(p) == '/foo/bar/baz.ext'
```

dirname
A string containing all of the path except the basename at the end

basename
A string containing the basename (filename and extension) at the end of the path

ext
A string containing the filename's extension (including the leading dot)

basename_no_ext
Returns a string containing basename with the extension removed (including the final dot separator).

dirs
Returns a sequence of the directories making up dirname

isabs
Returns True if the path is absolute (dirname begins with one or more forward slashes).

join(**paths*)
Joins this path with the specified parts, returning a new Path object.

Parameters ***paths** – The parts to append to this path

Returns A new Path object representing the extended path

class `lars.datatypes.`**`Time`**

Represents a time.

This type is returned by the `time()` function and represents a time. A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a `tzinfo` object.

Class attributes:

`min`

The earliest representable `Time`, `time(0, 0, 0, 0)`.

`max`

The latest representable `Time`, `time(23, 59, 59, 999999)`.

`resolution`

The smallest possible difference between non-equal `Time` objects, `timedelta(microseconds=1)`, although note that arithmetic on `Time` objects is not supported.

Instance attributes (read-only):

`hour`

In `range(24)`.

`minute`

In `range(60)`.

`second`

In `range(60)`.

`microsecond`

In `range(1000000)`.

`tzinfo`

The object passed as the tzinfo argument to the `Time` constructor, or `None` if none was passed.

Supported operations:

- comparison of `Time` to `Time`, where *a* is considered less than *b* when *a* precedes *b* in time. If one comparand is naive and the other is aware, `TypeError` is raised. If both comparands are aware, and have the same `tzinfo` attribute, the common `tzinfo` attribute is ignored and the base times are compared. If both comparands are aware and have different `tzinfo` attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a `Time` object is compared to an object of a different type, `TypeError` is raised unless the comparison is == or `!=`. The latter cases return `False` or `True`, respectively.
- hash, use as dict key
- efficient pickling
- in Boolean contexts, a `Time` object is considered to be true if and only if, after converting it to minutes and subtracting `utcoffset()` (or `0` if that's `None`), the result is non-zero.

Instance methods:

`replace``(`[*hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]]`)`

Return a `Time` with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive `Time` from an aware `Time`, without conversion of the time data.

`isoformat``()`

Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if self.microsecond is 0, HH:MM:SS If `utcoffset()` does not return `None`, a 6-character string is appended, giving the

UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM

strftime (*format*)
Return a string representing the time, controlled by an explicit format string.

utcoffset ()
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return `None` or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

dst ()
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn't return `None`, or a `timedelta` object representing a whole number of minutes with magnitude less than one day.

tzname ()
If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.tzname(None)`, or raises an exception if the latter doesn't return `None` or a string object.

class `lars.datatypes.`**`Url`**
Represents a URL.

This type is returned by the `url()` function and represents the parts of the URL. You can obtain the original URL as a string by requesting the string conversion of this class, for example:

```
>>> u = datatypes.url('http://foo/bar/baz')
>>> print u.scheme
http
>>> print u.hostname
foo
>>> print str(u)
http://foo/bar/baz
```

scheme
The scheme of the URL, before the first `:`

netloc
The "network location" of the URL, comprising the hostname and port (separated by a colon), and historically the username and password (prefixed to the hostname and separated with an ampersand)

path_str
The path of the URL from the first slash after the network location

path
The path of the URL, parsed into a tuple which splits out the directory, filename, and extension:

```
>>> u = datatypes.url('foo/bar/baz.html')
>>> u.path
Path(dirname='foo/bar', basename='baz.html', ext='.html')
>>> u.path.isabs
False
```

params
The parameters of the URL

query_str
The query string of the URL from the first question-mark in the path

query
The query string, parsed into a mapping of keys to lists of values. For example:

```
>>> u = datatypes.url('foo/bar?a=1&a=2&b=3&c=')
>>> print u.query
{'a': ['1', '2'], 'c': [''], 'b': ['3']}
>>> print 'a' in u.query
True
```

fragment
The fragment of the URL from the last hash-mark to the end of the URL

Additionally, the following attributes can be used to separate out the various parts of the `netloc` attribute:

username
The username (historical, rare to see this used on the modern web)

password
The password (historical, almost unheard of on the modern web as it's extremely insecure to include credentials in the URL)

hostname
The hostname from the network location. This attribute returns a `Hostname` object which can be used to resolve the hostname into an IP address if required.

port
The optional network port

Functions

`lars.datatypes.`**address**(*s*)
Returns an `IPv4Address`, `IPv6Address`, `IPv4Port`, or `IPv6Port` instance for the given string.

Parameters **s** (*str*) – The string containing the IP address to parse

Returns An `IPv4Address`, `IPv4Port`, `IPv6Address`, or `IPv6Port` instance

`lars.datatypes.`**date**(*s, format=u'%Y-%m-%d'*)
Returns a `Date` object for the given string.

Parameters

- **s** (*str*) – The string containing the date to parse
- **format** (*str*) – Optional string containing the date format to parse

Returns A `Date` object representing the date

`lars.datatypes.`**datetime**(*s, format=u'%Y-%m-%d %H:%M:%S'*)
Returns a `DateTime` object for the given string.

Parameters

- **s** (*str*) – The string containing the timestamp to parse
- **format** (*str*) – Optional string containing the datetime format to parse

Returns A `DateTime` object representing the timestamp

`lars.datatypes.`**hostname**(*s*)
Returns a `Hostname`, `IPv4Address`, or `IPv6Address` object for the given string depending on whether it represents an IP address or a hostname.

Parameters **s** (*str*) – The string containing the hostname to parse

Returns A `Hostname`, `IPv4Address`, or `IPv6Address` instance lars.datatypes.**network**(*s*)

Returns an IPv4Network or IPv6Network instance for the given string.

Parameters **s** (*str*) – The string containing the IP network to parse

Returns An IPv4Network or IPv6Network instance

lars.datatypes.**path**(*s*)

Returns a Path object for the given string.

Parameters **s** (*str*) – The string containing the path to parse

Returns A Path object representing the path

lars.datatypes.**row**(**args*)

Returns a new tuple sub-class type containing the specified fields. For example:

```
NewRow = row('foo', 'bar', 'baz')
a_row = NewRow(1, 2, 3)
print(a_row.foo)
```

Parameters ***args** – The set of fields to include in the row definition.

Returns A tuple sub-class with the specified fields.

lars.datatypes.**time**(*s*, *format=u'%H:%M:%S'*)

Returns a Time object for the given string.

Parameters

- **s** (*str*) – The string containing the time to parse
- **format** (*str*) – Optional string containing the time format to parse

Returns A Time object representing the time

lars.datatypes.**url**(*s*)

Returns a Url object for the given string.

Parameters **s** (*str*) – The string containing the URL to parse

Returns A Url tuple representing the URL

1.2.8 lars.progress - Rendering Progress

This module provides a wrapper that outputs simple progress meters to the command line based on source file positions, or an arbitrary counter. The ProgressMeter class is the major element that this module provides.

Classes

class lars.progress.**ProgressMeter**(*fileobj=None*, *value=0*, *total=None*, *max_wait=0.1*, *stream=sys.stderr*, *mode='w'*, *style=BarStyle*, *hide_on_finish=True*)

This class provides a simple means of rendering a progress meter at the command line. It can be driven either with a file object (in which case the current position of the file is used) or with an arbitrary value (which your code must provide). In the case of a file-object, the file must be seekable (so that the class can determine the overall length of the file). If *fileobj* is not specified, then *total* must be specified.

The class is intended to be used as a context manager. Upon entry it will render an initial progress meter, and will update it at reasonable intervals (dictated by the max_wait parameter) in response to calls to the update()

method. When you leave the context, the progress meter will be automatically erased if *hide_on_finish* is True (which it is by default).

Within the context, the `hide()` and `show()` methods can be used to temporarily hide and show the progress meter (in order to display some status text, for example).

Parameters

- **fileobj** (*file*) – A file-like object from which to determine progress
- **value** (*int*) – An arbitrary value from which to determine progress
- **total** (*int*) – In the case that *value* is set, this must be set to the maximum value that *value* will take
- **max_wait** (*float*) – The minimum length of time that must elapse before a screen update is permitted
- **stream** (*file*) – The stream object that output should be written to, defaults to stderr
- **style** – A reference to a class which will be used to render the progress meter, defaults to `BarStyle`
- **hide_on_finish** (*bool*) – If True (the default), the progress meter will be erased when the context exits

class `lars.progress.`**`SpinnerStyle`**(*meter*)
A `ProgressMeter` style that renders a simple spinning line.

class `lars.progress.`**`PercentageStyle`**(*meter*)
A `ProgressMeter` style that renders a simple percentage counter.

class `lars.progress.`**`EllipsisStyle`**(*meter*)
A `ProgressMeter` style that renders an looping series of dots.

class `lars.progress.`**`BarStyle`**(*meter*)
A `ProgressMeter` style that renders a full progress bar and percentage.

class `lars.progress.`**`HashStyle`**(*meter*)
A `ProgressMeter` style for those that remember FTP's `hash` command!

Examples

The most basic usage of this class is as follows:

```
import io
from lars import iis, csv, progress

with io.open('logs\iis.txt', 'rb') as infile, \
        io.open('iis.csv', 'wb') as outfile, \
        progress.ProgressMeter(infile) as meter, \
        iis.IISSource(infile) as source, \
        csv.CSVTarget(outfile) as target:
    for row in source:
        target.write(row)
        meter.update()
```

Note that you do not need to worry about the detrimental performance effects of calling `update()` too often; the class ensures that repeated calls are ignored until `max_wait` seconds have elapsed since the last update.

Alternatively, if you wish to update according to, say, the number of files to process you could use something like the following example (which also demonstrates temporarily hiding the progress meter in order to show the current filename):

```
import os
import io
from lars import iis, csv, progress

files = os.listdir('.')
with progress.ProgressMeter(total=len(files), style=progress.BarStyle) as meter:
    for file_num, file_name in enumerate(files):
        meter.hide()
        print "Processing %s" % file_name
        meter.show()
        with io.open(file_name, 'rb') as infile, \
                io.open(os.path.splitext(file_name)[0] + '.csv', 'wb') as outfile, \
                iis.IISSource(infile) as source, \
                csv.CSVTarget(outfile) as target:
            for row in source:
                target.write(row)
        meter.update(file_num)
```

1.2.9 lars.dns - DNS Resolution

This module provides a couple of trivial DNS resolution functions, enhanced with LRU caches. Most users should never need to access these functions directly. Instead, use the `address` and `hostname` properties of relevant objects.

Functions

`lars.dns.`**`from_address`**`(`**args*, ***kwds*`)`

Reverse resolve an address to a hostname.

Given a string containing an IPv4 or IPv6 address, this functions returns a hostname associated with the address, using an LRU cache to speed up repeat queries. If the address does not reverse, the function returns the original address.

Parameters **address** (*str*) – The address to resolve to a hostname

Returns The resolved hostname

`lars.dns.`**`to_address`**`(`**args*, ***kwds*`)`

Resolve a hostname to an address, preferring IPv4 addresses.

Given a string containing a DNS hostname, this function resolves the hostname to an address, using an LRU cache to speed up repeat queries. The function prefers IPv4 addresses, but will return IPv6 addresses if no IPv4 addresses are present in the result from getaddrinfo. If the hostname does not resolve, the function returns None rather than raise an exception (this is preferable as it provides a negative lookup cache).

Parameters **hostname** (*str*) – The hostname to resolve to an address

Returns The resolved address

1.2.10 lars.cache - Cache Decorators

This module provides a backport of the Python 3.3 LRU caching decorator. Users should never need to access this module directly; its contents are solely present to ensure DNS lookups can be cached under a Python 2.7 environment.

Source adapted from Raymond Hettinger's recipe licensed under the MIT license.

Functions

`lars.cache.lru_cache(maxsize=100, typed=False)`

Least-recently-used cache decorator.

If *maxsize* is set to None, the LRU features are disabled and the cache can grow without bound.

If *typed* is True, arguments of different types will be cached separately. For example, f(3.0) and f(3) will be treated as distinct calls with distinct results.

Arguments to the cached function must be hashable.

View the cache statistics named tuple (hits, misses, maxsize, currsize) with f.cache_info(). Clear the cache and statistics with f.cache_clear(). Access the underlying function with f.__wrapped__.

1.3 License

1.3.1 DateTime, Date, and Time documentation license

The documentation for the `DateTime`, `Date`, and `Time` classes in this module are derived from the documentation sources for the datetime, date, and time classes in Python 2.7.4 and thus are subject to the following copyright and license:

PSF LICENSE AGREEMENT FOR PYTHON 2.7.4

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.7.4 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.7.4 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., “Copyright © 2001-2013 Python Software

Foundation; All Rights Reserved" are retained in Python 2.7.4 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.7.4 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.7.4.

4. PSF is making Python 2.7.4 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.7.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.4 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.4, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.7.4, Licensee agrees to be bound by the terms and conditions of this License Agreement.

1.3.2 _strptime license

The `strptime` and `timezone` modules are derived from the _strptime and datetime modules in Python 3.2 respectively, and therefore are subject to the following license:

Copyright (c) 1990-2013, Python Software Foundation

PSF LICENSE AGREEMENT FOR PYTHON 3.2.3

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.2.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.2.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2012 Python Software Foundation; All Rights Reserved" are retained in Python 3.2.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.2.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.2.3.

4. PSF is making Python 3.2.3 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.2.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.2.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.2.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.2.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

1.3.3 IPNetwork & IPAddress documentation license

The documentation for the `IPv4Address`, `IPv4Network`, `IPv6Address`, and `IPv6Network` classes in lars are derived from the ipaddress documentation sources which are subject to the following copyright and are licensed to the PSF under the contributor agreement which makes them subject to the PSF 3.2.3 license from the section above:

Copyright (c) 2007 Google Inc.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX