map_resources Documentation

Release

Author

Jun 13, 2016

Contents

1 MapResources - A Whois Resource Discovery Tool. **3**

2 MapResources Installation **5**
- 2.1 Dependencies . . . 5
- 2.2 Installation . . . 5
- 2.3 MongoDB Setup . . . 5
- 2.4 Route Views database setup . . . 6

3 MapResources Examples **7**
- 3.1 1. Initiate search from an Organizational handle . . . 7
- 3.2 2. Initiate search from an Organizational handle, with a larger threshold . . . 9
- 3.3 3. Initiate search from multiple handles . . . 11
- 3.4 4. Comparison against Route Views . . . 12
- 3.5 5. Generation of a report . . . 12

4 map_resources package **13**
- 4.1 Module contents . . . 13
- 4.2 map_resources.map_whois . . . 14
- 4.3 map_resources.query_resources . . . 15
- 4.4 map_resources.analyze module . . . 16
- 4.5 map_resources.fetch_whois module . . . 21
- 4.6 map_resources.whois_rv_cmp module . . . 27

5 Indices and tables **31**

Python Module Index **33**

Index **35**

Contents:

CHAPTER 1

MapResources - A Whois Resource Discovery Tool.

There a number of reasons why an organization may wish to build an inventory of all routing resources (ASNs, network blocks) that it holds. While much of this information can be found in Whois registries, the process of building such a list and the maintenance of such a list over time is often non-trivial. This is because organizations, through mergers and splits may change in form and composition over time. The Whois database itself may become stale from not receiving timely updates, or could simply become fractured enough over time such that no one person has full knowledge of the organizational routing resources.

Clearly, an automated interface to Whois is required.

ARIN, the RIR for the North America region, offers access to its Whois database through a RESTful API. While ARIN does not itself provide a tool to automate Whois access, the RESTful API provides the necessary building block to implement such capability. The MapResources package is an implementation of this capability.

A starting point in the form of a known POC handle, organization handle, net handle or ASN number is assumed. Using information contained within the whois object, the MapResources tool identifies other resource dependencies and makes further queries through ARIN's RESTful API in order to find other resources that the organization may hold. The end result is a report that constitutes a rough resource inventory and a network graph that depicts how these resources are related to one another.

The main driver utility program for this package is map_resources.map_whois.py. The -h option to this script provides more information on the different options that are available to the user.

Note that even though most interfaces in the map_resources module are marked as public, they are still in flux and subject to change.

CHAPTER 2

MapResources Installation

2.1 Dependencies

The MapResources package has the following external dependencies. These packages must be installed prior to installing the MapResources package.

- pymongo
- pprint
- urllib
- requests
- xmltodict
- pygraphviz
- networkx
- matplotlib
- json2html
- sqlite3
- ipaddress

Some of these python packages may have additional library and system pacakge dependencies. For example, pygraphviz has a dependency on the Graphviz package.

2.2 Installation

The package can be installed by running the setup.py script as follows:

```
$ python setup.py install
```

2.3 MongoDB Setup

In order to use MongoDB as the persistent data store, follow directions given at http://docs.mongodb.org/manual/installation/ in order to install MongoDB.

2.4 Route Views database setup

When trying to compare against route views data the scripts in this package expect the route views data to be stored in a database that has the following schema.

- routeadv table

Column	Type
advId	integer primary key autoincrement
at	integer
prefix	varchar(255)
prefixStart	int
prefixEnd	int
sourceAddr	varchar(255)
sourceAS	integer
originated	int
nexthop	varchar(255)
lastAS	integer
asciiPath	varchar(4096)

- path table

Column	Type
pathId	integer primary key autoincrement
advId	integer REFERENCES routeadv(advId)
fromAS	integer
toAS	integer
count	integer

CHAPTER 3

MapResources Examples

Note that the following examples are for illustration purposes only. The resources listed below were found at a particular instance in time. For a more accurate representation of organizational resources the ARIN whois database should be queried directly.

3.1 1. Initiate search from an Organizational handle

Suppose we wish to obtain the graph of resources assigned ARIN. We know that one of the organizational handles for ARIN is 'ARIN', thus we can use this handle as one of our starting points:

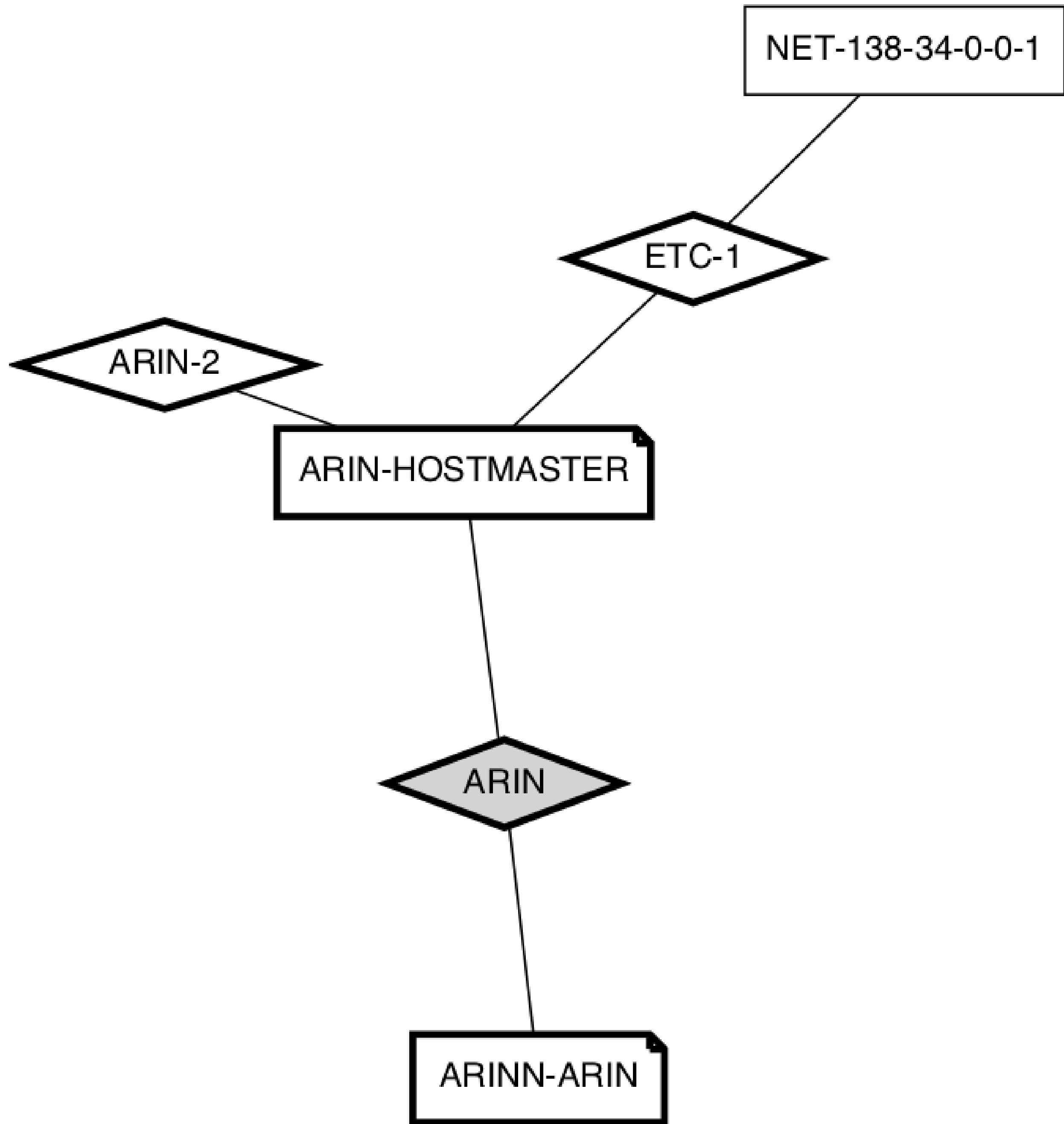
```
$ python map_whois.py -X -o ARIN -g ARIN.png
```

The -o option specifies the organizational handle of 'ARIN', while the -g option specifies the name of the file for the resource graph.

The -X option in the above command specifies that no caching is to be performed during query lookup.

If the -H option were used instead of the -X option, the script would use a hash store as the caching store instead. The hash data store allows scripts that wrap around the map_resources module to make use of cached values in subsequent calls to the graph generation routine. However, the hash data is still non-persistent. In order to use a persistent data store the above script should be called with the '-D host:port' option. The -D option is the default.

The graph produced by the above command is shown below:



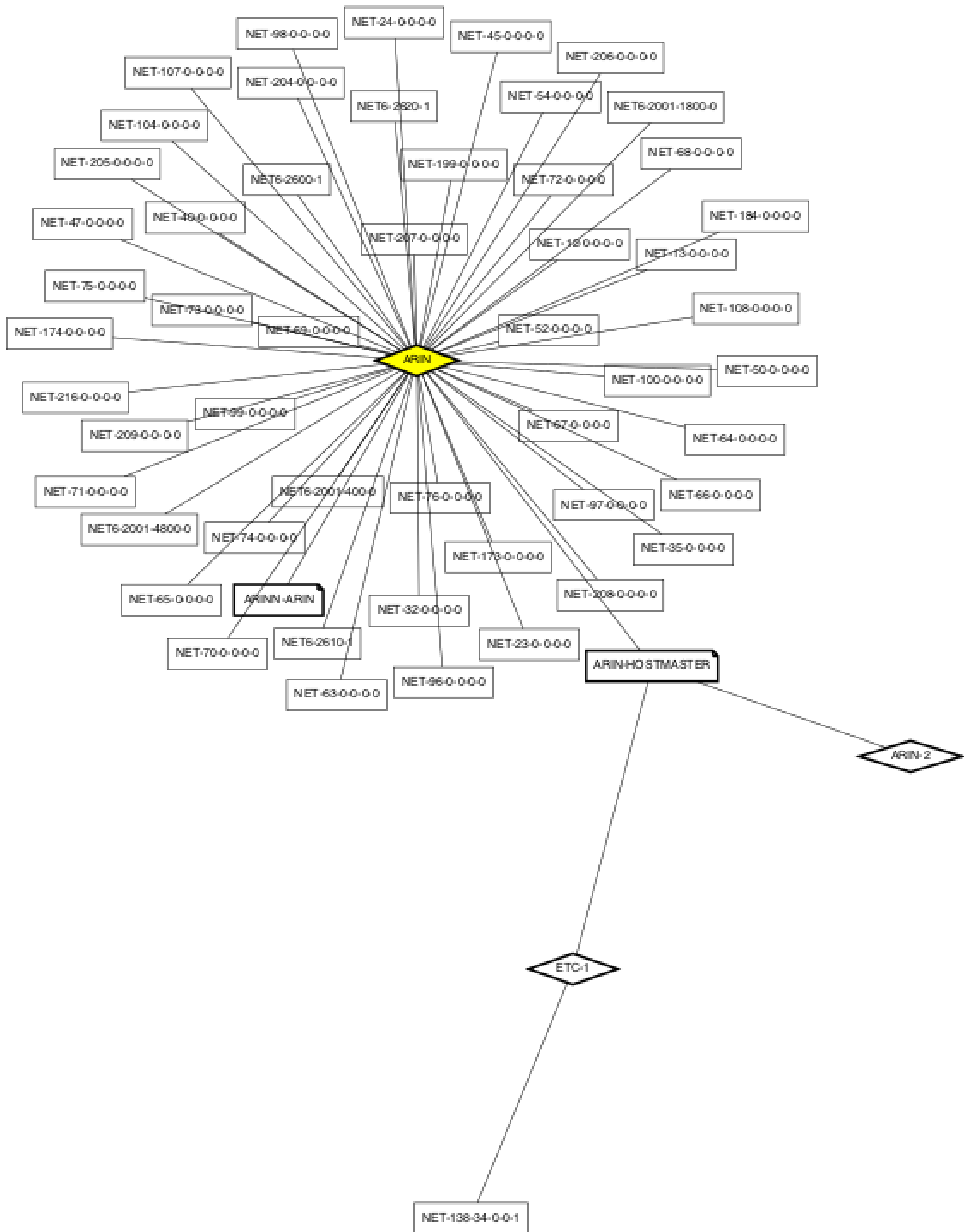
We note the following:

- Only a single network block is shown. For ARIN, this is likely an incomplete set.
- The ARIN block is grayed out. This indicates that the number of resource dependencies from the ARIN block exceeded some threshold. The hover text that appears over the grayed block indicates that the number of resources for http://whois.arin.net/rest/org/ARIN/nets is 51, which is clearly above our default threshold of 25.

3.2 2. Initiate search from an Organizational handle, with a larger threshold

If we re-run the above command (this time, with a local DB store) with the -t 51 option, we get a different output:

```
$ python map_whois.py -o ARIN -t 51 -g ARIN.png
```



As can be seen from the figure, a much larger set of resources are returned now.

3.3 3. Initiate search from multiple handles

A search for 'ARIN' on http://whois.arin.net indicates that two ASNs, AS10745 and AS394018, not depicted in the figure above are also assigned to ARIN. In order to include these resources in our resource graph, we can renew the search using multiple starting handles. Note that since we make use of a local persistant data store, resources that were fetched previously are not re-queried.

The command that we use and the image that is produced as a result are as follows:

```
$ python map_whois.py -o ARIN -a AS10745 -a AS394018 \
        -t 51 -g ARIN.png
```

The figure shows two different clusters of resources. This explains why the combined set of resources could not be found with the 'ARIN' handle alone.

3.4 4. Comparison against Route Views

The -R option enables map_whois.py to find new resource handles after a comparison with Route Views data.

NOTE: The Route Views data has to be stored into a database first.

New resources are identified in two ways:

- ASNs that were previously unknown, but which originated known prefixes.
- Prefixes that were previously unknown, but which are originated by known ASNs.

In addition, this option also checks whether any known prefixes were originated by an ASN other than the one listed in ARIN's Whois database.

For example, given an SQLite file 'rib.sqlite' containing the Route Views RIB data, the following command lists a number of 'unknown' ASNs:

```
$ python map_whois.py -o ARIN -a AS10745 -a AS394018 \
        -t 51 -g ARIN.png -R rib.sqlite
```

However most (if not all) such ASNs correspond to different clusters of resources that belong to other organizations that are recipients of resource assignments from ARIN. If any of these clusters are not re-assignments, resources within such clusters can be used as additional starting points in the resource graph construction process.

3.5 5. Generation of a report

Determining a complete list of starting handles is an iterative process. Once a list of handles has been obtained, a report listing out the different resources found as part of the mapping operation can be generated using the -r option to map_whois:

```
$ python map_whois.py -o ARIN -a AS10745 -a AS394018 \
    -t 51 -g ARIN.png -r report.html
```

Note that the report formating is highly primitive and is a work in progress.

CHAPTER 4

map_resources package

4.1 Module contents

This package enables one to query and inspect ARIN whois objects.

This package exports the following classes:

Data Stores:

- **GenericStore: No caching data store (base class)**
  - HashStore: Data store with hash backend
  - DBStore: Data store with MongoDB as the backend

Whois Collection Objects:

- **WhoisCollection: Base class**
  - **POCCollection: Point of Contact Collection**
    - URLCollection: URL Collection (Ephemeral Class)
  - OrgCollection: Organization Collection
  - **NetCollection: Network resource collection**
    - IPCollection: IP address collection (Ephemeral Class)
    - CIDRCollection: CIDR block Collection (Ephemeral Class)
  - ASNCollection: Autonomous System Number Collection

Analysis and reporting:

- WhoisAnalyzer: Cluster analyzer
- ResourceReporter: formats cluster information for reporting

CLI Argument Parser:

- WhoisOptParser: Parse base command line options
- AnalyzeOptExtension: Parse analyzer specific command line options

RouteViews Interface:

- RVFetcher: Fetch route view data from local DB
- RVComparator: Compare whois and route views data

- RVComparatorRes: Container for RVComparator results

4.2 map_resources.map_whois

map_whois.py - Map whois resources

This script enables one to discover network resources in ARIN's whois database that are could belong to an organization. A starting point in the form of a known POC handle, organization handle, net handle or asn number is assumed; the tool thereafter queries ARIN's RESTful API in order to detect other resources that the organization may hold.

usage: map_whois [-h] [-v] [-a ASN] [-p POC] [-o ORG] [-n NET] [-c CIDR] [-i IP] [-u URL] [-s ORGSTR] [-L RESOURCELIST] [-j JSONFILE] [-J CJSONFILE] [-e] [-X | -H | -D DBSTORE] [-t THRESHOLD] [-w WHITELIST] [-b BLACKLIST] [-S] [-g GRAPHFILE] [-r REPORTFILE] [-G] [-P] [-R RVDB]

optional arguments:

-h, --help show this help message and exit

-v, --verbose increase output verbosity

-a ASN, --asn ASN Start from the given ASN handle

-p POC, --poc POC Start from the given POC handle

-o ORG, --org ORG Start from the given Org handle

-n NET, --net NET Start from the given Net handle

-c CIDR, --cidr CIDR Start from the given CIDR block

-i IP, --ip IP Start from the given IP address

-u URL, --url URL Start from the given domain

-s ORGSTR, --orgstr ORGSTR Start from the given org string

-L RESOURCELIST, --resourcelist RESOURCELIST Extract resource handles from the given file. Each line of the file should be formatted as <type>:<value>, where the different supported types are 'asn', 'poc', 'org', 'net', 'cidr', 'ip' and 'url'. Args: rsrcfile(file handle): Return Values: A dict object containing a mapping between the resource handle and the resource type.

-j JSONFILE, --jsonfile JSONFILE Output raw resource information in json format

-J CJSONFILE, --cjsonfile CJSONFILE Output cluster information in json format

-e, --extended Display detailed information

-X, --nostore Do not use any data store

-H, --hashstore Use a hash store

-D DBSTORE, --dbstore DBSTORE Use a DB store

-t THRESHOLD, --threshold THRESHOLD Maximum number of dependencies to follow

-w WHITELIST, --whitelist WHITELIST Whitelisted handles

-b BLACKLIST, --blacklist BLACKLIST Blacklisted handles

-S, --showgraph Dsplay the graph

-g GRAPHFILE, --graphfile GRAPHFILE Output graph image

-r REPORTFILE, --reportfile REPORTFILE Output report

-G, --clustergraph Include graph image in report

-P, --clusterplot Include resource plot in report

-R RVDB, --rvdb RVDB Check against given Route Views Database file

`map_resources.map_whois.main(argv)`

4.3 map_resources.query_resources

query_resource.py - Query whois resources

This script enables one to query the ARIN whois database for data corresponding to given object handles.

usage: query_resources [-h] [-v] [-a ASN] [-p POC] [-o ORG] [-n NET] [-c CIDR] [-i IP] [-u URL] [-s ORGSTR] [-L RESOURCELIST] [-j JSONFILE] [-J CJSONFILE] [-e] [-X | -H | -D DBSTORE]

optional arguments:

-h, --help show this help message and exit

-v, --verbose increase output verbosity

-a ASN, --asn ASN Start from the given ASN handle

-p POC, --poc POC Start from the given POC handle

-o ORG, --org ORG Start from the given Org handle

-n NET, --net NET Start from the given Net handle

-c CIDR, --cidr CIDR Start from the given CIDR block

-i IP, --ip IP Start from the given IP address

-u URL, --url URL Start from the given domain

-s ORGSTR, --orgstr ORGSTR Start from the given org string

-L RESOURCELIST, --resourcelist RESOURCELIST Extract resource handles from the given file. Each line of the file should be formatted as <type>:<value>, where the different supported types are 'asn', 'poc', 'org', 'net', 'cidr', 'ip' and 'url'. Args: rsrcfile(file handle): Return Values: A dict object containing a mapping between the resource handle and the resource type.

-j JSONFILE, --jsonfile JSONFILE Output raw resource information in json format

-J CJSONFILE, --cjsonfile CJSONFILE Output cluster information in json format

-e, --extended Display detailed information

-X, --nostore Do not use any data store

-H, --hashstore Use a hash store

-D DBSTORE, --dbstore DBSTORE Use a DB store

`map_resources.query_resources.main(argv)`

4.4 map_resources.analyze module

Analyze and report on resource clusters.

This module analyzes the data that was fetched through the ARIN RESTful API and generates reportable output. It also implements the helper class that is used for processing Command line input to any driver scripts.

Attributes: verbose (boolean): Turns on verbosity of log messages.

class `map_resources.analyze.`**`AnalyzeOptExtension`**(*base*)
Class to parse options related to any analysis driver script.

`get_help`()
Return the formatted help text.

Returns: Str value containing formatted help text.

`parse`(*argv*)
Parse the list of options.

Args: A list of arguments provided in argv.

Returns: A dict structure that contains different analyzer options.

class `map_resources.analyze.`**`ResourceReporter`**(*analyzer*, *resources=None*, *links=None*, *filtered=None*)
This class formats cluster information for reporting.

`display_graph`()
Display the resource graph using matplotlib.

`get_agraph`(*G*, *o*)
Generate a Graphviz Agraph from graph information.

Produce the Agraph from the networkx object. Also set node attributes from the values stored in the different WhoisCollection objects.

Args: G(Graph): A networkx graph object. o(dict): A dict of object collections indexed by the origin handle.

Returns: A graphviz Agraph.

`get_clusterinfo`(*extended=False*, *rvf=None*)
Return cluster info.

Build a list of cluster information along with relevant network information for reporting.

Args: extended(boolean): If True produce additional details.

rvf(RouteViewsFetcher): If not None, fetch information from a RouteViews database to augment reported cluster information.

Returns:

A dict that contains for each cluster, a list of resources against each resource type.

`get_raw_clusterinfo`()
Get the raw cluster information

Returns: A tuple consisting of a dict of resources, a dict of links in the graph and the list of handles whose that were filtered.

`plot_graph`(*outputfile=None*)
Build a graph of whois resources.

Use graphviz to build the graph. Default format is neato The graph format is either png or svg. An svg image can show tooltips.

Args: outputfile(file handle): If not None, write graph to this file.

Returns: A base64 encoded image of the whois resource graph.

plot_resources (*plotfile=None*)

Scatter Plot number of URLs and IPs in each cluster.

Produce a Scatter Plot of the number of resources of each type in each cluster. Annotate the top 10 clusters in terms of resource density.

Args: plotfile (str, default=None): Graph file to write to

Returns: A base64 encoded image of the scatter plot.

write_cluster_json (*jh*, *extended=True*)

Write the cluster information in json format.

Args: jh(file handle): The target file for the JSON data.

Returns: None.

write_cluster_summary (*messages*)

Write the cluster summary information.

Args:

messages(array of strings): A list of messages associated with the clusters.

Returns: None.

write_raw_json (*jh*)

Write the raw resource information in json format.

Args: jh(file handle): The target file for the JSON data.

Returns: None.

write_report (*rh*, *clusterplot=False*, *clustergraph=False*, *extended=False*, *rvf=None*)

Write cluster info

Args: rh(file handle): The target file for the HTML data.

clusterplot(boolean): If True, generate and write cluster plot information to report.

clustergraph(boolean): If True, generate and write network resource information to report.

extended(boolean): If True produce additional details.

rvf(RouteViewsFetcher): If not None, use this object to augment report with RouteViews derived information.

Returns: None.

class map_resources.analyze.**WhoisAnalyzer** (*store=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)

Define a class for analyzing a list of collection objects.

analyze (*objlist*)

Analyze a list of handles.

Each handle in the list is used as a starting point for the collection of resources through the process_new_collection() method.

Args: objlist (dict): A dict of handles->type mappings.

Returns: None.

append_message(*msg*)
Append a new message to the analyzer object.

fetch(*ctype*, *loc*)
Fetch the data corresponding to the given type and loc str.

If we have a collection object try fetching through that first in order to get the benefit of caching. Otherwise look at the data store

Args: ctype (str): Collection type. loc (str): The ID string.

Returns: The result object structure.

generate_clusters()
Group resources according to their subgraphs.

Returns: A tuple consisting of a dict of resources, a dict of links in the graph and the list of handles that were filtered.

generate_results(*options*, *rvf=None*)
Generate various result components.

The different result components can include a report file, a JSON structure with cluster information, a Graphviz graph and a plot of the graph using matplotlib.

Args: options(dict): A dict containing the following parameters:

reportfile(file handle): If not None, the target file to write the HTML report into.

jsonfile(file handle) [The target file for raw jason] resource information.

cjsonfile(file handle) [The target file for jason] cluster information.

showgraph(boolean): If True, display network graph plot using matplotlib.

graphfile(filehandle): If True, write graphviz image to this file.

extended(boolean): If True produce extended details.

rvf(RouteViewsFetcher): If not None, use this object to augment report with RouteViews derived information.

Returns: None.

get_messages()
Return messages.

get_resobj()
Get the collection container object.

Return Values: The WhoisCollection object that serves as the handle to the collection container

pack(*resources*, *links*, *filtered*)
Pack resource and links into a dict.

Args: resources (dict): A dict of resources indexed by type links (dict): A dict of links filtered (list): A list of handles that are to be filtered

Returns: A dict that has 'resources' and 'links' as two of its keys.

process_new_collection(*t*, *h*, *comment=None*)
Process a new collection of given type and handle.

Create a new collection object with the given type. Reuse the cache and the data store if available, but dont link to any other collection object. Finally, subsume the resulting object into the main collection container object within self.

Args: t (str): Collection type. h (str): The collection handle.

Returns: None.

unpack(*clustobj*)

Unpack a dict containing resources and links.

Args:

clustobj(dict): A dict that has 'resources' and 'links' as two of its keys.

Returns:

A tuple of three elements: A dict of resources indexed by type A dict of links A list of handles that are to be filtered

class map_resources.analyze.**WhoisObjectFormatter**(*getter*)

A class that allows us to format data received from whois.

get_asninfo(*loc*)

Generate asn info.

Build a dict structure that contains ASN related information.

Args: loc(str): The object identifier.

Returns:

A dict structure with the following keys: 'handle': ASN handle

get_netinfo(*loc*, *rvf=None*)

Generate net info.

Build a dict structure that contains information such as the registration date, the different netblocks, and the network handle. Also fetch information from the routing table if we have access to a Route Views database.

Args: loc(str): The object identifier.

rvf(RouteViewsFetcher): If not None, fetch information such as routed netbocks and origination AS in order to determine if any network resources are originated by resources in a different cluster.

Returns:

A dict structure with the following keys: 'handle': network handle 'regstration': registration date 'netblocks': a JSONized string of a list of network blocks 'routed': a JSONized string of a list of routed network blocks

get_orginfo(*loc*)

Generate org info.

Build a dict structure that contains organization related information.

Args: loc(str): The object identifier.

Returns:

A dict structure with the following keys: 'handle': organizational handle

get_pocinfo(*loc*)

Generate poc info.

Build a dict structure that contains Point of Contact related information.

Args: loc(str): The object identifier.

Returns:

A dict structure with the following keys: 'handle': POC handle

class map_resources.analyze.**WhoisOptParser**(*prog*)
Class to parse options related to any whois interfacing script.

add_stores()
Register the options associated with different data store types.

get_help()
Return the formatted help text.

Returns: Str value containing formatted help text.

get_parser()
Return the parser associated with this object.

host_port(*s*)
A simple type for a host:port identifier

Throws argparse.ArgumentTypeError if the provided string is not formated as hostname:port.

Args: s (str): a string of the form hostname:port

Returns: The host(str) and port(int) tuple.

parse(*argv*)
Parse the list of options.

Args: A list of arguments provided in argv.

Returns: A dict structure that contains different analyzer options.

parse_objs(*p*)
Extract object handles and types from provided options.

The object handle could come from a file or from the relevant option associated with each resource type.

Args: p(Namespace): Contains various command line parameters.

Returns: A dict objects whose keys correspond to the different handle identifiers and whose values correspond to the resource type.

parse_objs_from_file(*rsrcfile*)
Extract resource handles from the given file.

Each line of the file should be formatted as <type>:<value>, where the different supported types are 'asn', 'poc', 'org', 'net', 'cidr', 'ip' and 'url'.

Args: rsrcfile(file handle):

Return Values: A dict object containing a mapping between the resource handle and the resource type.

parse_opts(*p*)
Parse the list of options given in the parser namespace.

Args: A namespace containing the different parsed options.

Returns: A dict structure that contains different analyzer options.

parse_store(*p*)
Extract the store type from provided options.

Args: p(Namespace): Contains various command line parameters.

Returns: A GenericStore object corresponding to the selected data store type.

4.5 map_resources.fetch_whois module

Query ARIN whois objects

This module provides the functionality to query the ARIN RESTful API for all resource dependencies starting from the given ASN, POC, Org or Net handle.

Attributes: verbose (boolean): Turns on verbosity of log messages.

class `map_resources.fetch_whois.`**`ASNCollection`**(*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)

Bases: *`map_resources.fetch_whois.WhoisCollection`*

ASN Resource Class

`slurp`(*handle*)

Look for all objects that can be reached from this ASN container.

ASN collections may be comprised of POC and Org collections.

Args: base (str): The base URL for lookups.

`slurp_set`(*base*)

Find the ASN handle and slurp data.

ASN resources may be found in XML elements named asnPocLinkRef or asnRef.

Args: base (str): The base URL for lookups.

class `map_resources.fetch_whois.`**`CIDRCollection`**(*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)

Bases: *`map_resources.fetch_whois.NetCollection`*

IP container class (ephemeral).

`slurp`(*handle*)

Look for all objects that can be reached from this CIDR block.

Look for the associated NetCollection object and perform the slurp operation over that object.

Args: handle (str): The origin handle for the slurp operation

class `map_resources.fetch_whois.`**`DBStore`**(*dbhost*, *dbport*, *local=False*)

Bases: *`map_resources.fetch_whois.GenericStore`*

Wrapper around a MongoDB data store.

`fetch`(*ctype*, *idstr*)

Fetch data for the given ID string and collection type.

First look at the MongoDB store for any matching data. If found return that data; if not, fetch new data but only if we are not limiting lookups to already-cached values.

`find_collection`(*ctype*)

Find the DB collection associated with the given object type.

Args: ctype (str): The collection object type.

Returns: The mongoDB collection object.

class `map_resources.fetch_whois.`**`GenericStore`**(*base='http://whois.arin.net/rest'*)
Base class for all data stores with no caching support.

`fetch`(*ctype*, *idstr*)
Fetch data for the given ID string and collection type.

No caching is done in this routine. Just query and return.

Args: ctype (str): The collection type. idstr (str): The location reference for the whois object.

Returns: A dict object representing the result.

`fetchAssociated`(*obj*, *idstr*)
Fetch an object's associated data, given an ID string.

Args: obj (str): The object for which we are seeking associated data. idstr (str): The ID string.

Returns: A dict object representing the result.

`get_idstr`(*typepfx*, *handle*)
Determine the set of IDs for given type and handle.

This is the default way of constructing an ID.

Args: typepfx (str): The collection type. handle (str): The resource handle.

Returns: A tuple comprising the handle and a list of ID strings and collection type values.

`query`(*idstr*)
Query the data store for the given ID string.

This is the common query method for all types of data stores.

Args: idstr (str): The ID string.

Returns: A dict object representing the result.

class `map_resources.fetch_whois.`**`HashStore`**(*base='http://whois.arin.net/rest'*)
Bases: *`map_resources.fetch_whois.GenericStore`*

Implementation of a simple hash data store (non-persistent).

`fetch`(*ctype*, *idstr*)
Fetch data for the given ID string and collection type.

First look at the hash for any matching data. If found return that data; if not fetch new data.

class `map_resources.fetch_whois.`**`IPCollection`**(*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)
Bases: *`map_resources.fetch_whois.NetCollection`*

IP container class (ephemeral).

`slurp`(*handle*)
Look for all objects that can be reached from this IP address.

Look for the associated NetCollection object and perform the slurp operation over that object.

Args: handle (str): The origin handle for the slurp operation.

class `map_resources.fetch_whois.`**`NetCollection`**(*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)
Bases: *`map_resources.fetch_whois.WhoisCollection`*

Net Resource Class.

slurp (*handle*)
Look for all objects that can be reached from this Net container.

Net collections may be comprised of POC or Org collections.

Args: base (str): The base URL for lookups.

slurp_set (*base*)
Find the Net handle and slurp data.

Net resources may be found in XML elements named netPocLinkRef or netRef.

Args: base (str): The base URL for lookups.

class map_resources.fetch_whois.**OrgCollection** (*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)
Bases: *map_resources.fetch_whois.WhoisCollection*

Organization Resouce Class

slurp (*handle*)
Look for all objects that can be reached from this Org container.

Org collections may be comprised of POC, ASN and Net collections.

Args: base (str): The base URL for lookups.

slurp_set (*base*)
Find the Org handle and slurp data.

Orgs may be found in XML elements named orgPocLinkRef or orgRef.

Args: base (str): The base URL for lookups.

class map_resources.fetch_whois.**OrgstrCollection** (*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)
Bases: *map_resources.fetch_whois.OrgCollection*

slurp (*handle*)
Look for all objects that can be reached from this OrgName

Look for the associated OrgCollection object and perform the slurp operation over that object.

Args: handle (str): The origin handle for the slurp operation.

class map_resources.fetch_whois.**POCCollection** (*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)
Bases: *map_resources.fetch_whois.WhoisCollection*

Point of Contact Resource Class.

slurp (*handle*)
Look for all objects that can be reached from this POC container.

POC collections may be comprised of Org, ASN and Net collections.

Args: handle (str): The origin handle for the slurp operation.

slurp_set (*base*)
Find the POC handle and slurp data.

POCs may be found in XML elements named pocLinkRef or pocRef.

Args: base (str): The base URL for lookups.

class map_resources.fetch_whois.**URLCollection**(*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)

Bases: *map_resources.fetch_whois.POCCollection*

slurp(*handle*)
Look for all objects that can be reached from this URL.

Look for the associated POCCollection object and perform the slurp operation over that object.

Args: handle (str): The origin handle for the slurp operation.

class map_resources.fetch_whois.**WhoisCollection**(*origin_handle*, *origin=None*, *store=None*, *cache=None*, *tt=None*, *threshold=None*, *whitelist=None*, *blacklist=None*)

Base class for all Whois collection objects.

add_collection(*col*)
Add a new associated ollection to the current object.

Collections are indexed by the origin handle. There can be multiple collections per origin handle. As part of adding the collection also update the cache from the to-be-added collection to the main (parent) collection holder.

Args: col (WhoisCollection): the collection object to be added.

Returns: None.

add_link(*handle*)
Create a link between the current object and the given handle.

Args: handle (str): the handle of the collection to link with.

Returns: None.

add_tooltip(*handle*, *msg*)
Add a new tooltip to the object.

Append a new tooltip to the list indexed by the given handle.

Args: handle (str): the handle to associate the tool-tip with. msg (str): the tool-tip message.

Returns: None.

do_slurp()
Entry point for looking up resource objects.

fetchAssociatedObj(*idstr*)
Get associated data for given idstr.

This method asks the data store for any associated objects.

Args: idstr (str): object ID str.

Returns: A tuple with two values: the first is whether the data was cached or not; and the second is the actual result object.

fetchObj(*typepfx*, *handle*, *cache=True*)
Get data corresponding to given handle and collection type.

This method first constructs the ID string(s) corresponding to the handle and then calls get_data() to get the actual result object.

Args: typepfx (string): Collection type. handle (string): Resource handle. cache (boolean): If true, any data returned by get_data() is cached.

Returns: A list of object tuples. Each object tuple contains three values: the first is whether the data was fresh (not cached) or not; the second, is the ID string, and the third is the result object.

fill_draw_attribs(*node_h*, *a*)
Get different attributes of this collection.

Returns: A dict structure that contains the following attributes for the collection object:

shape: the shape to use in any graphical representation. style: the figure style, such as its filled status. fillcolor: the fillcolor to use. color: the color to use. penwidth: the penwidth to use.

get_blacklist()
Return the blacklisted handles.

Returns: List of object handle strings that are blacklisted.

get_cache()
Return the cache structure for the collection object.

Returns: A dict value that holds the cache information for the WhoisCollection object.

get_collections(*recurse=True*)
Get list of collections associated with the given object.

Return all objects below and including the current one grouped by their handle.

Args:

recurse (boolean): return collection objects by traversing all collections that are linked through previous calls to add_collection().

Returns: A dict of lists of WhoisCollection objects. The keys of the dict are the origin handles that were used in the construction of the associated WhoisCollection objects.

get_data(*ctype*, *idstr*)
Get data corresponding to given ID string and collection type.

First check if the data exists in the cache. If it doesn't then look for data in the data store.

Args: ctype (string): Collection type. idstr (string): ID string.

Returns: A tuple of two values, where the first is a boolean value that indicates whether the data was cached or not, and the second is the result dict object.

get_filtered(*recurse=True*)
Get the list of handles that were filtered

Args:

recurse (boolean): traverse all collections that are linked through previous calls to add_collection().

Returns: A lists of handles that were filtered.

get_links(*recurse=True*)
Get the list of links associated with the given object.

For each object below and including the current node return the list of connected nodes.

Args:

recurse (boolean): return links by traversing all collections that are linked through previous calls to add_collection().

Returns: A dict of lists of handles. The keys of the dict are the origin handles that were used in the construction of the associated WhoisCollection objects.

get_parent()
Return the parent object for the object.

Returns: The WhoisContainer object corresponding to the parent.

get_parent_handle()
Return the origin (parent) handle for the object.

Returns: The string value for the origin handle.

get_resources(*recurse=True*)
Get the list of resources associated with the given object.

Args:

recurse (boolean): return resources by traversing all collections that are linked through previous calls to add_collection().

Returns: A dict of lists of resources. The keys are the resource types.

get_store()
Get the store associated with this object.

Returns: The GenericStore object that is associated with the given collection object.

get_threshold()
Return the threshold for the collection object.

Returns: An integer value that holds the threshold value.

get_tooltip(*recurse=True*)
Get the list of tooltips associated with the given object.

Args:

recurse (boolean): return tooltips by traversing all collections that are linked through previous calls to add_collection().

Returns: A dict of lists of tooltips. The keys are the origin handles.

get_type()
Get the collection type for the given collection object.

Returns: A str value corresponding to the type of the whois collection object.

get_whitelist()
Return the whitelisted handles.

Returns: List of object handle strings that are whitelisted from fitering.

set_limit_exceeded(*handle*, *idstr*, *lim*)
Set the object state to indicate too many dependencies.

Args:

handle (str): the handle against which the limit exceeded state is to be associated.

idstr (str): the ID string that triggered the limit exceeded event.

lim (int): the number of dependencies that were detected.

Returns: None.

slurp_common (*p*, *idstr*)
Common convenience function for data slurping.

If we're presented with a dict, look for the handle in all list elements; if not, check in the provided list. Do not process those objects whose child collections have number of elements greater than a particular threshold.

Args: p (str): the link reference to process.

idstr (str): object ID str.

Returns: None.

subsume (*col*)
Subsume the new collection object.

Merge various pertinent data associated with the new collection object into the current collection object.

Args: col (WhoisCollection): the collection object that we want to subsume.

Returns: None

4.6 map_resources.whois_rv_cmp module

Get data from a local database instance of RouteViews.

This module serves as the interface between the WhoisAnalyzer object and the database view of RouteViews data. The Route Views data must first be pre-populated within a database.

This module provides two classes:

RVFetcher: provides a simple interface to fetch netblocks from the database

RVComparator: Does some comparisons between a set of ASNs and network block data and the RouteViews DB.

Attributes: verbose (boolean): Turns on verbosity of log messages.

class map_resources.whois_rv_cmp.**RVComparator** (*dbfile*)
comparison between Whois objects and Route Views data.

add_asn (*asn*)
Add the ASN to the SQL query.

Args: asn(str): The ASN handle

Returns: True if the ASN was added.

add_net (*startAddr*, *endAddr*, *oaslist*)
Add the network block to the SQL query.

Note that IPv6 address blocks are ignored.

Args: startAddr(str): The start address in the block. endAddr(str): The end address in the block. oaslist(list): a list of origin ASNs associated with this netblock.

Returns: True if the network block was added.

compare ()
Compare known resources against Route Views data.

Issue the sql query, check which prefixes and ASNs are unknown.

Returns: An RVComparatorRes object that encapsulates the results of the comparison.

compare_resources(*resob*)
Register known resources and compare against Route Views.

Register all ASNs and Net objects from the WhoisAnalyzer object and then call compare() in order to check for differences against Route Views data.

Args:

resob(WhoisAnalyzer): object that holds the list of ASN and Net resources we are interested in analyzing.

Returns: An RVComparatorRes object that encapsulates the results of the comparison.

is_pfx_known(*start*, *end*)
Check if the given network block is one that we know about.

Args: start(str): network block start. end(str): network block end.

Returns: True if the network block exists in our list; False if not.

class map_resources.whois_rv_cmp.**RVComparatorRes**
Class for encapsulating the comparator result.

add_unknown_asn(*asn*, *pfxlist*)
Register an unknown AS.

Register the previously unknown ASN that was found to originate a known prefix.

Args: asn: The (unknown) ASN originating the prefix. pfx: The prefix from a known block that was originated.

Returns: None.

add_unknown_oasn(*asn*, *pfx*)
Register an unknown originating AS.

Register the ASN originating a known prefix but was not configured as the originating AS in whois.

Args: asn: The (known) ASN originating the prefix. pfx: The prefix from a known block that was originated.

Returns: None.

add_unknown_pfx(*asn*, *pfx*)
Register an unknown prefix.

Register the previously unknown prefix that was originated from a known ASN.

Args: asn: The known ASN originating the prefix. pfx: The unknown prefix that was originated.

Returns: None.

get_unknown_asn()
Return the list of unknown ASNs.

Returns: A dict structure that maps ASNs to the prefix values.

get_unknown_oasn()
Return the list of unknown originating ASNs.

Returns: A dict structure that maps ASNs to the prefix values.

get_unknown_pfx()
Return the list of unknown prefixes.

Returns: A dict structure that maps prefixes to the ASNs they were originated from.

process_unknown_resources(*analyzer*)

Process all unknown dependencies.

Feed all unknown dependencies that were detected in Routeviews through the WhoisAnalyzer object.

Args: analyzer(WhoisAnalyzer): the analyzer object

Returns: None.

class map_resources.whois_rv_cmp.**RVFetcher**(*dbfile*)

Class to fetch objects from route views.

find_netblocks(*start*, *end*)

Find netblocks that fall between the given prefix bounds.

Look up all netblocks that fall between the start and end bounds. IPv6 address blocks are ignored.

Args: start(str): start prefix end(str): end prefix

Returns: A cursor iterator that holds the results of the lookup.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

m

map_resources, 13
map_resources.analyze, 16
map_resources.fetch_whois, 21
map_resources.map_whois, 14
map_resources.query_resources, 15
map_resources.whois_rv_cmp, 27

A

add_asn() (map_resources.whois_rv_cmp.RVComparator method), 27
add_collection() (map_resources.fetch_whois.WhoisCollection method), 24
add_link() (map_resources.fetch_whois.WhoisCollection method), 24
add_net() (map_resources.whois_rv_cmp.RVComparator method), 27
add_stores() (map_resources.analyze.WhoisOptParser method), 20
add_tooltip() (map_resources.fetch_whois.WhoisCollection method), 24
add_unknown_asn() (map_resources.whois_rv_cmp.RVComparatorRes method), 28
add_unknown_oasn() (map_resources.whois_rv_cmp.RVComparatorRes method), 28
add_unknown_pfx() (map_resources.whois_rv_cmp.RVComparatorRes method), 28
analyze() (map_resources.analyze.WhoisAnalyzer method), 17
AnalyzeOptExtension (class in map_resources.analyze), 16
append_message() (map_resources.analyze.WhoisAnalyzer method), 18
ASNCollection (class in map_resources.fetch_whois), 21

C

CIDRCollection (class in map_resources.fetch_whois), 21
compare() (map_resources.whois_rv_cmp.RVComparator method), 27
compare_resources() (map_resources.whois_rv_cmp.RVComparator method), 27

D

DBStore (class in map_resources.fetch_whois), 21
display_graph() (map_resources.analyze.ResourceReporter method), 16
do_slurp() (map_resources.fetch_whois.WhoisCollection method), 24

F

fetch() (map_resources.analyze.WhoisAnalyzer method), 18
fetch() (map_resources.fetch_whois.DBStore method), 21
fetch() (map_resources.fetch_whois.GenericStore method), 22
fetch() (map_resources.fetch_whois.HashStore method), 22
fetchAssociated() (map_resources.fetch_whois.GenericStore method), 22
fetchAssociatedObj() (map_resources.fetch_whois.WhoisCollection method), 24
fetchObj() (map_resources.fetch_whois.WhoisCollection method), 24
fill_draw_attribs() (map_resources.fetch_whois.WhoisCollection method), 25
find_collection() (map_resources.fetch_whois.DBStore method), 21
find_netblocks() (map_resources.whois_rv_cmp.RVFetcher method), 29

G

generate_clusters() (map_resources.analyze.WhoisAnalyzer method), 18
generate_results() (map_resources.analyze.WhoisAnalyzer method), 18
GenericStore (class in map_resources.fetch_whois), 21
get_agraph() (map_resources.analyze.ResourceReporter method), 16
get_asninfo() (map_resources.analyze.WhoisObjectFormatter method), 19
get_blacklist() (map_resources.fetch_whois.WhoisCollection method), 25
get_cache() (map_resources.fetch_whois.WhoisCollection method), 25
get_clusterinfo() (map_resources.analyze.ResourceReporter method), 16

get_collections() (map_resources.fetch_whois.WhoisCollection method), 25
get_data() (map_resources.fetch_whois.WhoisCollection method), 25
get_filtered() (map_resources.fetch_whois.WhoisCollection method), 25
get_help() (map_resources.analyze.AnalyzeOptExtension method), 16
get_help() (map_resources.analyze.WhoisOptParser method), 20
get_idstr() (map_resources.fetch_whois.GenericStore method), 22
get_links() (map_resources.fetch_whois.WhoisCollection method), 25
get_messages() (map_resources.analyze.WhoisAnalyzer method), 18
get_netinfo() (map_resources.analyze.WhoisObjectFormatter method), 19
get_orginfo() (map_resources.analyze.WhoisObjectFormatter method), 19
get_parent() (map_resources.fetch_whois.WhoisCollection method), 26
get_parent_handle() (map_resources.fetch_whois.WhoisCollection method), 26
get_parser() (map_resources.analyze.WhoisOptParser method), 20
get_pocinfo() (map_resources.analyze.WhoisObjectFormatter method), 19
get_raw_clusterinfo() (map_resources.analyze.ResourceReporter method), 16
get_resobj() (map_resources.analyze.WhoisAnalyzer method), 18
get_resources() (map_resources.fetch_whois.WhoisCollection method), 26
get_store() (map_resources.fetch_whois.WhoisCollection method), 26
get_threshold() (map_resources.fetch_whois.WhoisCollection method), 26
get_tooltip() (map_resources.fetch_whois.WhoisCollection method), 26
get_type() (map_resources.fetch_whois.WhoisCollection method), 26
get_unknown_asn() (map_resources.whois_rv_cmp.RVComparatorRes method), 28
get_unknown_oasn() (map_resources.whois_rv_cmp.RVComparatorRes method), 28
get_unknown_pfx() (map_resources.whois_rv_cmp.RVComparatorRes method), 28
get_whitelist() (map_resources.fetch_whois.WhoisCollection method), 26

H

HashStore (class in map_resources.fetch_whois), 22
host_port() (map_resources.analyze.WhoisOptParser method), 20

I

IPCollection (class in map_resources.fetch_whois), 22
is_pfx_known() (map_resources.whois_rv_cmp.RVComparator method), 28

M

main() (in module map_resources.map_whois), 15
main() (in module map_resources.query_resources), 15
map_resources (module), 13
map_resources.analyze (module), 16
map_resources.fetch_whois (module), 21
map_resources.map_whois (module), 14
map_resources.query_resources (module), 15
map_resources.whois_rv_cmp (module), 27

N

NetCollection (class in map_resources.fetch_whois), 22

O

OrgCollection (class in map_resources.fetch_whois), 23
OrgstrCollection (class in map_resources.fetch_whois), 23

P

pack() (map_resources.analyze.WhoisAnalyzer method), 18
parse() (map_resources.analyze.AnalyzeOptExtension method), 16
parse() (map_resources.analyze.WhoisOptParser method), 20
parse_objs() (map_resources.analyze.WhoisOptParser method), 20
parse_objs_from_file() (map_resources.analyze.WhoisOptParser method), 20
parse_opts() (map_resources.analyze.WhoisOptParser method), 20
parse_store() (map_resources.analyze.WhoisOptParser method), 20
plot_graph() (map_resources.analyze.ResourceReporter method), 16
plot_resources() (map_resources.analyze.ResourceReporter method), 17
POCCollection (class in map_resources.fetch_whois), 23
process_new_collection() (map_resources.analyze.WhoisAnalyzer method), 18
process_unknown_resources() (map_resources.whois_rv_cmp.RVComparatorRes method), 28

Q

query() (map_resources.fetch_whois.GenericStore method), 22

R

ResourceReporter (class in map_resources.analyze), 16
RVComparator (class in map_resources.whois_rv_cmp), 27
RVComparatorRes (class in map_resources.whois_rv_cmp), 28
RVFetcher (class in map_resources.whois_rv_cmp), 29

S

set_limit_exceeded() (map_resources.fetch_whois.WhoisCollection method), 26
slurp() (map_resources.fetch_whois.ASNCollection method), 21
slurp() (map_resources.fetch_whois.CIDRCollection method), 21
slurp() (map_resources.fetch_whois.IPCollection method), 22
slurp() (map_resources.fetch_whois.NetCollection method), 22
slurp() (map_resources.fetch_whois.OrgCollection method), 23
slurp() (map_resources.fetch_whois.OrgstrCollection method), 23
slurp() (map_resources.fetch_whois.POCCollection method), 23
slurp() (map_resources.fetch_whois.URLCollection method), 24
slurp_common() (map_resources.fetch_whois.WhoisCollection method), 26
slurp_set() (map_resources.fetch_whois.ASNCollection method), 21
slurp_set() (map_resources.fetch_whois.NetCollection method), 23
slurp_set() (map_resources.fetch_whois.OrgCollection method), 23
slurp_set() (map_resources.fetch_whois.POCCollection method), 23
subsume() (map_resources.fetch_whois.WhoisCollection method), 27

U

unpack() (map_resources.analyze.WhoisAnalyzer method), 19
URLCollection (class in map_resources.fetch_whois), 24

W

WhoisAnalyzer (class in map_resources.analyze), 17
WhoisCollection (class in map_resources.fetch_whois), 24
WhoisObjectFormatter (class in map_resources.analyze), 19
WhoisOptParser (class in map_resources.analyze), 20
write_cluster_json() (map_resources.analyze.ResourceReporter method), 17
write_cluster_summary() (map_resources.analyze.ResourceReporter method), 17
write_raw_json() (map_resources.analyze.ResourceReporter method), 17
write_report() (map_resources.analyze.ResourceReporter method), 17