BigBang

Release v0.3.0

BigBang Team

Apr 25, 2022

GETTING STARTED

BigBang is a toolkit for studying processes of open collaboration and deliberation, especially with respect to the production of digital infrastructures, to make them more transparent and accountable. This is achieved by utilising public communication channels and documents to reveal which actors are leading, following, or left out. It enables the analysis and visualisation of relationships, discourses, time series and knowledge networks.

BigBang is a community of researchers sharing code and best practices. BigBang comes with a large directory of examples showcasing what kind of questions can be answered through the communications data. These examples are designed as tutorial and support data science and research pedagogy.

CHAPTER

ONE

MOTIVATION

BigBang was originally designed to study communities developing open sources scientific software. It was quickly thereafter adopted by researchers studying Internet and network standards-setting. Organizations such as the IETF conduct much of their technical work via open mailing list and document records. There is tremendous opportunity to learn from these data sources. BigBang codifies the processes of collecting and cleaning that data, making it available to researchers.

More broadly, BigBang is designed to give researchers and students more transparent insight into the sociotechnical governance and infrastructure processes that shape their world. BigBang is a telescope designed to study the originating singularities that gave rise to the Internet, scientific computation, and other pivotal technological developments.

CHAPTER

TWO

LICENSE

BigBang is open source software. It is released under the MIT license.

CHAPTER

THREE

INSTALLATION

3.1 conda

You can use Anaconda. This will also install the *conda* package management system, which you can use to complete installation.

Install Anaconda, with Python version 3.*.

If you choose not to use Anaconda, you may run into issues with versioning in Python. Add the Conda installation directory to your path during installation.

You also need need to have Git and Pip (for Python3) installed.

Run the following commands:

```
git clone https://github.com/datactive/bigbang.git
cd bigbang
bash conda-setup.sh
python3 setup.py develop --user
```

3.2 pip

```
git clone https://github.com/datactive/bigbang.git
# optionally create a new virtualenv here
pip3 install -r requirements.txt
python3 setup.py develop --user
```

3.3 Video Tutorial

If you have problems installing, you might want to have a look at the video tutorial below (clicking on the image will take you to YouTube).

BigBang Video Tutorial

CHAPTER

FOUR

DATASETS

This section outlines, how various public mailing lists can be scraped from the web and stored to disk for further processing. Currently, the `BigBang` repository does not contain personally identifiable information of any kind. The datasets included in `BigBang` pertain to organizational entities and provide *ancillary data* useful in preprocessing and analysis of those entities. As the mailing-list archives are large and time consuming to scrape from the web, we are working on GDPR compliant method to share the datasets with other researchers.

4.1 Mailinglists

Below we describe, how the public mailing lists of each of the Internet standard developing organisations can be scrabed from the web. Some mailng lists reach back to 1998 and is multiple GBs in size. Therefore, it can take a considerable amount of time to scrape an entire mailing list. This process can't be speed up, since one would commit a DDoS attack otherwise. So be prepared to leave your machine running over (multiple) night(s).

4.1.1 IETF

To scrabed public mailing lists of the Internet Engineering Task Force (IETF), there are two options outlined below.

Public Mailman Web Archive

BigBang comes with a script for collecting files from public Mailman web archives. An example of this is the scipy-dev mailing list page. To collect the archives of the scipy-dev mailing list, run the following command from the root directory of this repository:

```
python3 bin/collect_mail.py -u http://mail.python.org/pipermail/scipy-dev/
```

You can also give this command a file with several urls, one per line. One of these is provided in the *examples/* directory.

```
python3 bin/collect_mail.py -f examples/urls.txt
```

Once the data has been collected, BigBang has functions to support analysis.

Datatracker

BigBang can also be used to analyze data of IETF RFC drafts.

It does this using the Glasgow IPL group's `ietfdata` tool.

The script takes an argument, the working group acronym

```
python3 bin/collect_draft_metadata.py -w httpbis
```

4.1.2 W3C

The World Wide Web Consortium (W3C) mailing archive is managed using the Hypermail software and is hosted at:

`https://lists.w3.org/Archives/Public/`

There are two ways you can scrape the public mailing-list from that domain. First, one can write their own python script containing a variation of:

```
from bigbang.ingress import ListservMailList

mlist = W3CMailList.from_url(
    name="public-testtwf",
    url="https://lists.w3.org/Archives/Public/public-testtwf/",
    select={"years": 2014, "fields": "header"},
)
mlist.to_mbox(path_to_file)
```

Or one can use the command line script and a file containg all mailing-list URLs one wants to scrape:

```
python bin/collect_mail.py -f examples/url_collections/W3C.txt
```

4.1.3 3GPP

The 3rd Generation Partnership Project (3GPP) mailing archive is managed using the LISTSERV software and is hosted at:

`https://list.etsi.org/scripts/wa.exe?HOME`

In order to successfully scrape all public mailing lists, one needs to create an account here: https://list.etsi.org/scripts/wa.exe?GETPW1=&X=&Y=

There are two ways you can scrape the public mailing-list from that domain. First, one can write their own python script containing a variation of:

```
from bigbang.ingress import ListservMailList

mlist = ListservMailList.from_url(
    name="3GPP_TSG_SA_WG2_EMEET",
    url="https://list.etsi.org/scripts/wa.exe?A0=3GPP_TSG_SA_WG2_EMEET",
    select={"fields": "header",},
    url_login="https://list.etsi.org/scripts/wa.exe?LOGON=INDEX",
    url_pref="https://list.etsi.org/scripts/wa.exe?PREF",
    login=auth_key,
)
mlist.to_mbox(path_to_file)
```

Or one can use the command line script and a file containg all mailing-list URLs one wants to scrape:

```
python bin/collect_mail.py -f examples/url_collections/listserv.3GPP.txt
```

4.1.4 IEEE

The Institute of Electrical and Electronics Engineers (W3C) mailing archive is managed using the LISTSERV software and is hosted at:

```
https://listserv.ieee.org/cgi-bin/wa?INDEX
```

There are two ways you can scrape the public mailing-list from that domain. First, one can write their own python script containing a variation of:

```
from bigbang.ingress import ListservMailList

mlist = ListservMailList.from_url(
    name="IEEE-TEST",
    url="https://listserv.ieee.org/cgi-bin/wa?A0=IEEE-TEST",
    select={"fields": "header",},
    url_login="https://listserv.ieee.org/cgi-bin/wa?LOGON",
    url_pref="https://listserv.ieee.org/cgi-bin/wa?PREF",
    login=auth_key,
)
mlist.to_mbox(path_to_file)
```

Or one can use the command line script and a file containg all mailing-list URLs one wants to scrape:

```
python bin/collect_mail.py -f examples/url_collections/listserv.IEEE.txt
```

4.2 Ancillary Datasets

In addition to providing tools for gathering data from public sources, `BigBang` also includes some datasets that have been curated by contributors and researchers.

4.2.1 General

Email domain categories

BigBang comes with a partial list of email domains, categorized as:

- **Generic**. A domain associated with a generic email provider. E.g. `gmail.com`
- **Personal**. A domain associated with a single individual. E.g `csperkins.org`
- **Company**. A domain associated with a particular company. E.g. `apple.com`
- **Academic**. A domain associated with a university or academic professional organization. E.g. `mit.edu`
- **SDO**. A domain associated with a Standards Development Organization. E.g. `ietf.org`

This data can be loaded as a Pandas DataFrame with indices as email domains and categories in the `category` column with the following code:

```
import bigbang.datasets.domains as domains
domain_data = domains.load_data()
```

The sources of this data are a hand-curated list of domains provided by BigBang contributors and a list of generic email domain providers provided by this public gist.

Organization Metadata

BigBang comes with a curated list of metadata about organizations. This data is provided as a DataFrame with the following columns:

- **name**. Organization name. E.g. `gmail.com`
- **Category**. Kind of organization. E.g `Infrastructure Company`
- **subsidiary**. This column describes when a company is the subsidiary of another company in the list. If the cell in this column is empty, this company can be understood as the parent company.. E.g. `apple.com`
- **stakeholdergroup**. Stakeholdergroups are used as they have been defined in the WSIS process and the Tunis-agenda.
- **nationality**. The country name in which the stakeholder or subsidiary is registered.
- **email domain names**. Email domains associated with the organization. May include multiple, comma separated, domain names.
- **Membership Organization**. Membership of regional SDOs, derived from 3GPP data.

This data can be loaded as a Pandas DataFrame with indices as email domains and categories in the `category` column with the following code:

```
import bigbang.datasets.organizations as organizations
organization_data = organizations.load_data()
```

The sources of this data are a hand-curated list of domains provided by BigBang contributors and a list of generic email domain providers provided by this public gist.

4.2.2 IETF

Publication date of protocols.

4.2.3 3GPP

Release dates of standards.

4.3 Data Source - Git

After the git repositories have been cloned locally, you will be able to start analyzing them. To do this, you will need a GitRepo object, which is a convenient wrapper which does the work of extracting and generating git information and storing it internally in a pandas dataframe. You can then use this GitRepo object's methods to gain access to the large pandas dataframe.

There are many ways to generate a GitRepo object for a repository, using RepoLoader:

- Bash scripts (in the bigbang directory):
  - single url `python bin/collect_git.py -u https://github.com/scipy/scipy.git`
  - file of urls `python bin/collect_git.py -f examples/git_urls.txt`

  - Github organization name `python bin/collect_git.py -g glass-bead-labs`

- Single Repo:
  - remote `get_repo("https://github.com/sbenthall/bigbang.git", in_type = "remote" )`
  - local `get_repo("~/urap/bigbang/archives/sample_git_repos/bigbang", in_type = "local" )`
  - name `get_repo("bigbang", in_type = "name")`
- Multiple Repos:
  - With repo names: `get_multi_repo(repo_names=["bigbang","django"])`
  - With repo objects: `get_multi_repo(repos=[{list of existing GitRepo objects}]`
  - With Github Organization names `get_org_multirepo("glass-bead-labs")`

4.3.1 Repo Locations

As of now, repos are clones into `archives/sample_git_repos/{repo_name}`. Their caches are stored at `archives/sample_git_repos/{repo_name}_backup.csv`.

4.3.2 Caches

Caches are stored at `archives/sample_git_repos/{repo_name}_backup.csv`. They are the dumped `.csv` files of a GitRepo object's `commit_data` attribute, which is a pandas dataframe of all commit information. We can initialize a GitRepo object by feeding the cache's Pandas dataframe into the GitRepo init function. However, the init function will need to do some processing before it can use the cache as its commit data. It needs to convert the `"Touched File"` attribute of the cache dataframe from unicode `"[file1, file2, file3]"` to an actual list `["file1", "file2", "file3"]`. It will also need to convert the time index of the cache from string to datetime.

4.3.3 Bash Scripts

Run the following commands while in the bigbang directory. The repo information will go into the default repo location.

```
python bin/collect_git.py -u https://github.com/scipy/scipy.git
```

You can also give this command a file with several urls, one per line. One of these is provided in the `examples/` directory.

```
python bin/collect_git.py -f examples/git_urls.txt
```

This command will load all of the repos of a github organization. Make sure that the name is exactly as it appears on Github.

```
python bin/collect_git.py -g glass-bead-labs
```

4.3.4 Single Repos

Here, we can load in three ways. We can use a github url, a local path to a repo, or the name of a repo. All of these return a `GitRepo` object. Here is an example, with explanations below.

```
from bigbang import repo_loader # The file that handles most loading

repo = repo_loader.get_repo("https://github.com/sbenthall/bigbang.git", in_type =
↪"remote" )
# repo = repo_loader.get_repo("../",  in_type = "local" ) # I commented this out␣
↪because it may take too long
repo = repo_loader.get_repo("bigbang", in_type = "name")

repo.commit_data # The pandas df of commit data
```

Remote

A remote call to `get_repo` will extract the repo's name from its git url. Thus, `https://github.com/sbenthall/bigbang.git` will yield `bigbang` as its name. It will check if the repo already exists. If it doesn't it will send a shell command to clone the remote repository to `archives/sample_git_repos/{repo_name}`. It will then return `get_repo({name}, in_type="name")`. Before returning, however, it will cache the GitRepo object at `archives/sample_git_repos/{repo_name}_backup.csv` to make loading faster the next time.

Local

A local call is the simplest. It will first extract the repo name from the filepath. Thus, `~/urap/bigbang/archives/sample_git_repos/bigbang` will yield `bigbang`. It will check to see if a git repo exists at the given address. If it does, it will initialize a GitPython object, which only needs a name and a filepath to a Git repo. Note that this option does not check or create a cache.

4.3.5 Name

This is the preferred and easiest way to load a git repository. It works under the assumptions above about where a git repo and its cache should be stored. It will check to see if a cache exists. If it does, then it will load a GitPython object using that cache.

If a cache is not found, then the function constructs a filepath from the name, using the above rule about where repo locations. It will pass off the function to `get_repo(filepath, in_type="local")`. Before returning the answer, it will cache the result.

4.3.6 MultiRepos

These are the ways we can get MultiGitRepo objects. MultiGitRepo objects are GitRepos that were created with a list of GitRepos. Basically, a MultiGitRepo's `commit_data` contains the commit_data from all of its GitRepos. The only difference is that each entry has an extra attribute, `Repo Name` that tells us which Repo that commit is initially from. Here are some examples, with explanations below. Note that the examples below will not work if you don't have an internet connection, and may take some time to process. The first call may also fail if you do not have all of the repositories

```
from bigbang import repo_loader # The file that handles most loading

## Using GitHub API
multirepo = repo_loader.get_org_multirepo("glass-bead-labs")

## List of repo names
multirepo = repo_loader.get_multi_repo(repo_names = ["bigbang","bead.glass"])

## List of actual repos
repo1 = repo_loader.get_repo("bigbang", in_type="name")
repo2 = repo_loader.get_repo("bead.glass", in_type="name")
multirepo = repo_loader.get_multi_repo(repos = [repo1, repo2])

multirepo.commit_data # The pandas df of commit data
```

List of Repos / List of Repo Names (`get_multi_repo`)

This is rather simple. We can call the `get_multi_repo` method with either a list of repo names `["bigbang", "django", "scipy"]` or a list of actual GitRepo objects. This returns us the merged MultiGitRepo. Please note that this will not work if a local clone / cache of the repos does not exist for every repo name (e.g. if you ask for `["bigbang", "django", "scipy"]`, you must already have a local copy of those in your sample_git_repos directory.

Github Organization's Repos (`get_org_multirepo`)

This is more useful to us. We can use this method to get a MultiGitRepo that contains the information from every repo in a Github Organization. This requires that we input the organization's name *exactly* as it appears on Github (edX, glass-bead-labs, codeforamerica, etc.)

It will look for `examples/{org_name}_urls.txt`, which should be a file that contains all of the git urls of the projects that belong to that organization. If this file doesn't yet exist, it will make a call to the Github API. This requires a stable internet connection, and it may randomly stall on requests that do not time out.

The function will then use the list of git urls and the `get_repo` method to get each repo. It will use this list of repos to create a MultiGitRepo object, using `get_multi_repo`.

CHAPTER

FIVE

ANALYSIS

5.1 3GPP

This page introduces a collection of simple functions with which a comprehensive overview of 3GPP mailinglists, ingressed using *bigbang/ingress/listserv.py*, is gained. Without extensive editing, these functions should also be applicable to IETF, ICANN, W3C, and IEEE mailinglists, however it hasn't been tested yet.

To start, a `ListservList` class instance needs to be created using either `.from_mbox()` or `.from_pandas_dataframe()`. Using the former as an example:

```
from bigbang.analysis.listserv import ListservList

mlist_name = "3GPP_TSG_CT_WG1_122E_5G"
mlist = ListservList.from_mbox(
    name=mlist_name,
    filepath=f"/path/to/{mlist_name}.mbox",
    include_body=True,
)
```

The function argument `include_body` is by default `True`, but if one has to work with a large quantity of Emails, it might be necessary to set it to *False* to avoid out-of-memory errors.

5.1.1 Cropping of mailinglist

If one is interested in specific subgroups contained in a mailinglist, then the *ListservList* class instance can be cropped using the following functions:

```
# select Emails send in a specific year
mlist.crop_by_year(yrs=[2011])

# select Emails send within a period
mlist.crop_by_year(yrs=[2011, 2021])

# select Emails send or received from specified addresses
mlist.crop_by_address(
    header_field='from',
    per_address_field={'domain': ['t-mobile.at', 'nokia.com']}
)

# select Emails containing string in subject
mlist.crop_by_subject(match='OpenPGP')
```

In the second example, the function has an `per_address_field` argument. This argument is a dictionary in which the top-level keys can be `localpart` and `domain`, where the former is the part of an Email address that stands in front of the @ and the latter after. Thus for *Heinrich.vonKleist@selbst.org*, localpart is *Heinrich.vonKleist* and the domain is *selbst.org*.

5.1.2 Who is sending/receiving?

To get an insight in which actors are involved in a mailinglist, a `ListservList` class instance can be return the unique email domains and the unique email localparts per domain for multiple header fields:

```
mlist.get_domains(header_fields=['from', 'reply-to'])

mlist.get_localparts(header_fields=['from', 'reply-to'])
```

This will return a dictionary, in which each key (both 'from' and 'reply-to') contains a list of all domains. If one wants see not just who contributes, but also how much, change the default argument of `return_msg_counts=False` to `True`:

```
mlist.get_domains(header_fields=['from', 'reply-to'], return_msg_counts=True)
```

Alternatively, one can also get the number of Emails send or received by a certain address via,

```
mlist.get_messagescount(
    header_fields=['from', 'reply-to'],
    per_address_field={
        'domain': ['t-mobile.at', 'nokia.com'],
        'localpart': ['ian.hacking', 'victor.klemperer'],
    }
)
```

5.1.3 Communication Network

For a more in-depth view into who is sending (receiving) to (from) whom in a mailing list, one can use the `return_msg_counts=False` as follows:

```
mlist.create_sender_receiver_digraph()
```

This will create a new `networkx.DiGraph()` instance attribute for `mlist`, which can be used to perform a number of standard calculations using the `networkx` python package:

```
import networkx as nx

nx.betweenness_centrality(mlist.dg, weight="weight")
nx.closeness_centrality(mlist.dg)
nx.degree_centrality(mlist.dg)
```

5.1.4 Time-series

To study, e.g., the continuity of an actors contribution to a mailinglist, many function have an optional `per_year` boolean argument.

To simply find out during which period Emails were in a mailinglist, one can call `mlist.period_of_activity()`.

5.2 Networks

Documentation for the analysis and preprocessing scripts of BigBang.

5.3 Timeseries

Documentation for the analysis and preprocessing scripts of BigBang.

CHAPTER

SIX

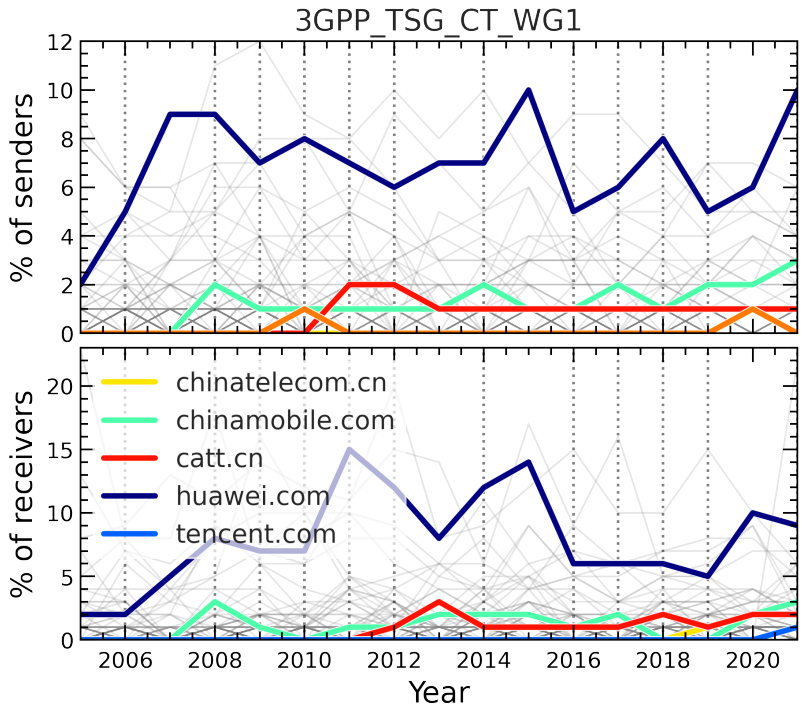
VISUALISATION

6.1 Lines

To help visualise, e.g., time-series data obtained through *3GPP*, we provide a number of support functions. If, for example, one has executed the `mlist.get_localpartscount()` command with `per_year=True`, one can use `lines.evolution_of_participation_1D()` to visualise how the number of get_localparts changed over time for each domain, which is related to the number of participants belonging to each organisation:

```
from bigbang.analysis.listserv import ListservMailList
from bigbang.visualisation import graphs

mlist_name = "3GPP_TSG_SA_WG3_LI"
filepath = f"/home/christovis/InternetGov/bigbang-archives/3GPP/{mlist_name}.mbox"
mlist = ListservMailList.from_mbox(
    name=mlist_name,
    filepath=filepath,
)

dic = mlist.get_localpartscount(
    header_fields=['from'],
    per_domain=True,
    per_year=True,
)

entities_in_focus = [
    'catt.cn',
    'chinaunicom.cn',
    'huawei.com',
    'chinatelecom.cn',
    'chinamobile.com',
]

fig, axis = plt.subplots()
lines.evolution_of_participation_1D(
    dic['from'],
    ax=axis,
    entity_in_focus=entities_in_focus,
    percentage=False,
)
axis.set_xlabel('Year')
axis.set_ylabel('Nr of senders')
```

The above code produces the following figure:



Alternatively it can also be visualised as a heat map using `lines.evolution_of_participation_2D()`. Similarly, one can plot the evolution of, e.g., different types of centrality of domain names in the communication network:

```
from bigbang.analysis.listserv import ListservMailList
from bigbang.visualisation import graphs

dic = mlist.get_graph_prop_per_domain_per_year(func=nx.degree_centrality)

fig, axis = plt.subplots()
lines.evolution_of_graph_property_by_domain(
    dic,
    "year",
    "degree_centrality",
    entity_in_focus=entities_in_focus,
    ax=axis,
)
axis.set_xlabel('Year')
axis.set_ylabel(r'$C_{\rm D}$')
```

6.2 Histograms

Documentation for the visualization scripts of BigBang.

6.3 Graphs

To help visualise the results obtained in *Communication Network*, we provide support functions, such that the thickness of graph edges and nodes can be adjusted. Assuming that one as already executed the `mlist.create_sender_receiver_digraph()` command, we can use `graphs.edge_thickness()` to highlight the relation between specific actors or `graphs.node_size()` to let the node size increase with their betweenness centrality.

```
import networkx as nx
from bigbang.visualisation import graphs

edges, edge_width = graphs.edge_thickness(
    mlist.dg,
    entity_in_focus=['t-mobile.at', 'nokia.com'],
)
node_size = graphs.node_size(mlist.dg)

nx.draw_networkx_nodes(
    mlist.dg, pos,
    node_size=node_size,
)

nx.draw_networkx_edges(
    mlist.dg, pos,
    width=edge_width,
    edgelist=edges,
    edge_color=edge_width,
    edge_cmap=plt.cm.rainbow,
)
```

CHAPTER

SEVEN

REFERENCE

7.1 archive

This module supports the Archive class, a generic structure representing a collection of archived emails, typically from a single mailing list.

class bigbang.archive.**Archive**(*data*, *archive_dir='/home/docs/checkouts/readthedocs.org/user_builds/bigbang-py/checkouts/stable/archives/'*, *mbox=False*)

Bases: object

A representation of a mailing list archive.

Initialize an Archive object.

The behavior of the constructor depends on the type of its first argument, data.

If data is a Pandas DataFrame, it is treated as a representation of email messages with columns for Message-ID, From, Date, In-Reply-To, References, and Body. The created Archive becomes a wrapper around a copy of the input DataFrame.

If data is a string, then it is interpreted as a path to either a single .mbox file (if the optional argument single_file is True) or else to a directory of .mbox files (also in .mbox format). Note that the file extensions need not be .mbox; frequently they will be .txt.

Upon initialization, the Archive object drops duplicate entries and sorts its member variable *data* by Date.

Parameters

- **data** (*pandas.DataFrame, or str*) –
- **archive_dir** (*str, optional*) – Defaults to CONFIG.mail_path
- **mbox** (*bool*) –

activity = None

add_affiliation(*rel_email_affil*)

Uses a DataFrame of email affiliation information and adds it to the archive's data table.

The email affilation data is expected to have a regular format, with columns:

- email - strings, complete email addresses
- affilation - strings, names of organizatiosn of affilation
- min_date - datetime, the starting date of the affiliation
- max_date - datetime,the end date of the affilation.

Note that this mutates the dataframe in `self.data` to add the affiliation data.

rel_email_affil : pandas.DataFrame

compute_activity(*clean=True*)
Return the computed activity.

data = None

entities = None

get_activity(*resolved=False*)
Get the activity matrix of an Archive.

Columns of the returned DataFrame are the Senders of emails. Rows are indexed by ordinal date. Cells are the number of emails sent by each sender on each data.

If *resolved* is true, then default entity resolution is run on the activity matrix before it is returned.

get_personal_headers(*header='From'*)
Returns a dataframe with a row for every message of the archive, containing column entries for:

- The personal header specified. Defaults to "From". Could be "Repy-To".
- The email address extracted from the From field
- The domain of the From field

This dataframe is computed the first time this method is called and then cached.

Parameters **header** (*string, default "From"*) –

Returns **data**

Return type pandas.DataFrame

get_threads(*verbose=False*)
Get threads.

preprocessed = None

resolve_entities(*inplace=True*)
Return data with resolved entities.

Parameters **inplace** (*bool, default True*) –

Returns Returns None if inplace == True

Return type pandas.DataFrame or None

save(*path*, *encoding='utf-8'*)
Save data to csv file.

threads = None

exception bigbang.archive.**ArchiveWarning**
Bases: BaseException

Base class for Archive class specific exceptions

exception bigbang.archive.**MissingDataException**(*value*)
Bases: Exception

bigbang.archive.**archive_directory**(*base_dir*, *list_name*)
Creates a new archive directory for the given list_name unless one already exists. Returns the path of the archive directory.

bigbang.archive.**find_footer**(*messages*, *number=1*)
Returns the footer of a DataFrame of emails.

A footer is a string occurring at the tail of most messages. Messages can be a DataFrame or a Series

bigbang.archive.**load**(*path*)

bigbang.archive.**load_data**(*name: str*, *archive_dir: str = '/home/docs/checkouts/readthedocs.org/user_builds/bigbang-py/checkouts/stable/archives/'*, *mbox: bool = False*)
Load the data associated with an archive name, given as a string.

Attempt to open {archives-directory}/NAME.csv as data.

Failing that, if the the name is a URL, it will try to derive the list name from that URL and load the .csv again.

Parameters

- **name** (*str*) –
- **archive_dir** (*str, default CONFIG.mail_path*) –
- **mbox** (*bool, default False*) – If true, expects and opens an mbox file at this path

Returns data

Return type pandas.DataFrame

bigbang.archive.**messages_to_dataframe**(*messages*)
Turn a list of parsed messages into a dataframe of message data, indexed by message-id, with column-names from headers.

bigbang.archive.**open_list_archives**(*archive_name: str*, *archive_dir: str = '/home/docs/checkouts/readthedocs.org/user_builds/bigbang-py/checkouts/stable/archives/'*, *mbox: bool = False*) → pandas.core.frame.DataFrame
Return a list of all email messages contained in the specified directory.

Parameters

- **archive_name** (*str*) – the name of a subdirectory of the directory specified in argument *archive_dir*. This directory is expected to contain files with extensions .txt, .mail, or .mbox. These files are all expected to be in mbox format– i.e. a series of blocks of text starting with headers (colon-separated key-value pairs) followed by an email body.
- **archive_dir** (*str:*) – directory containing all messages.
- **mbox** (*bool, default False*) – True if there's an mbox file already available for this archive.

Returns data

Return type pandas.DataFrame

7.2 bigbang_io

bigbang.bigbang_io.**email_to_dict**(*msg: mailbox.mboxMessage*) → Dict[str, str]
Handles data type transformation from *mailbox.mboxMessage* to *Dictionary*.

bigbang.bigbang_io.**email_to_mbox**(*msg: mailbox.mboxMessage*, *filepath: str*, *mode: str = 'w'*) → None
Saves *mailbox.mboxMessage* as .mbox file.

bigbang.bigbang_io.**email_to_pandas_dataframe**(*msg: mailbox.mboxMessage*) → pandas.core.frame.DataFrame

Handles data type transformation from *mailbox.mboxMessage* to *pandas.DataFrame*.

bigbang.bigbang_io.**get_paths_to_dirs_in_directory**(*directory: str*, *folder_dsc: str = '*'*) → List[str]

Get paths of all directories matching file_dsc in directory

bigbang.bigbang_io.**get_paths_to_files_in_directory**(*directory: str*, *file_dsc: str = '*'*) → List[str]

Get paths of all files matching file_dsc in directory

bigbang.bigbang_io.**mlist_from_mbox**(*filepath: str*) → list

Reads *mailbox.mboxMessage* objects from .mbox file. For a clearer definition on what a mailing list is, see: bigbang.ingress.abstract.AbstractList

bigbang.bigbang_io.**mlist_from_mbox_to_pandas_dataframe**(*filepath: str*) → pandas.core.frame.DataFrame

Reads *mailbox.mboxMessage* objects from .mbox file and transforms it to a *pandas.DataFrame*. For a clearer definition on what a mailing list is, see: bigbang.ingress.abstract.AbstractList

bigbang.bigbang_io.**mlist_to_dict**(*msgs: List[mailbox.mboxMessage]*, *include_body: bool = True*) → Dict[str, List[str]]

Handles data type transformation from a List[mailbox.mboxMessage] to a Dictionary. For a clearer definition on what a mailing list is, see: bigbang.ingress.abstract.AbstractList

bigbang.bigbang_io.**mlist_to_mbox**(*msgs: List[mailbox.mboxMessage]*, *dir_out: str*, *filename: str*) → None

Saves a List[mailbox.mboxMessage] as .mbox file. For a clearer definition on what a mailing list is, see: bigbang.ingress.abstract.AbstractList

bigbang.bigbang_io.**mlist_to_pandas_dataframe**(*msgs: List[mailbox.mboxMessage]*, *include_body: bool = True*) → pandas.core.frame.DataFrame

Handles data type transformation from a List[mailbox.mboxMessage] to a pandas.DataFrame. For a clearer definition on what a mailing list is, see: bigbang.ingress.abstract.AbstractList

bigbang.bigbang_io.**mlistdom_to_dict**(*mlists: List[List[mailbox.mboxMessage]]*, *include_body: bool = True*) → Dict[str, List[str]]

Handles data type transformation from a List[AbstractList] to a *Dictionary*. For a clearer definition on what a mailing archive is, see: bigbang.ingress.abstract.AbstractArchive

bigbang.bigbang_io.**mlistdom_to_mbox**(*mlists: List[List[mailbox.mboxMessage]]*, *dir_out: str*)

Saves a List[AbstractList] as .mbox file. For a clearer definition on what a mailing archive is, see: bigbang.ingress.abstract.AbstractArchive

bigbang.bigbang_io.**mlistdom_to_pandas_dataframe**(*mlists: List[List[mailbox.mboxMessage]]*, *include_body: bool = True*) → pandas.core.frame.DataFrame

Handles data type transformation from a List[AbstractList] to a pandas.DataFrame. For a clearer definition on what a mailing archive is, see: bigbang.ingress.abstract.AbstractArchive

7.3 parse

bigbang.parse.**clean_from**(*m_from*)
Return a person's name extracted from 'From' field of email, based on heuristics.

bigbang.parse.**clean_mid**(*mid*)

bigbang.parse.**clean_name**(*name*)
Clean just the name portion from email.utils.parseaddr.

Returns None if the name portion is missing anything name-like. Otherwise, returns the cleaned name.

bigbang.parse.**get_date**(*message*)

bigbang.parse.**get_refs**(*refs*)

bigbang.parse.**get_text**(*msg*)
Get text from a message.

bigbang.parse.**guess_first_name**(*cleaned_from*)
Attempt to extract a person's first name from the cleaned version of their name (from a 'From' field). This may or may not be the given name. Returns None if heuristic doesn't recognize a separable first name.

bigbang.parse.**normalize_email_address**(*address*)
Takes a valid email address and returns a normalized one, for matching purposes.

bigbang.parse.**split_references**(*refs*)

bigbang.parse.**tokenize_name**(*clean_name*)
Create a tokenized version of a name, good for comparison and sorting for entity resolution.

Takes a Unicode name already cleaned of most punctuation and spurious characters, hopefully.

7.4 utils

Miscellaneous utility functions used in other modules.

bigbang.utils.**add_freq**(*idx*, *freq=None*)
Add a frequency attribute to idx, through inference or directly.

Returns a copy. If *freq* is None, it is inferred.

bigbang.utils.**clean_message**(*mess*)

bigbang.utils.**get_common_foot**(*str1*, *str2*, *delimiter=None*)

bigbang.utils.**get_common_head**(*str1*, *str2*, *delimiter=None*)

bigbang.utils.**get_paths_to_dirs_in_directory**(*directory: str*, *folder_dsc: str = '*'*) → List[str]
Get paths of all directories matching file_dsc in directory

bigbang.utils.**get_paths_to_files_in_directory**(*directory: str*, *file_dsc: str = '*'*) → List[str]
Get paths of all files matching file_dsc in directory

bigbang.utils.**labeled_blockmodel**(*g*, *partition*)
Perform blockmodel transformation on graph *g* and partition represented by dictionary *partition*. Values of *partition* are used to partition the graph. Keys of *partition* are used to label the nodes of the new graph.

bigbang.utils.**remove_quoted**(*mess*)

bigbang.utils.**repartition_dataframe**(*df*, *partition*)
Create a new dataframe with the same index as argument dataframe *df*, where columns are the keys of dictionary *partition*. The data of the returned dataframe are the combinations of columns listed in the keys of *partition*

7.5 analysis.repo_loader

exception bigbang.analysis.repo_loader.**RepoLoaderWarning**
Bases: BaseException

Base class for Archive class specific exceptions

bigbang.analysis.repo_loader.**cache_path**(*name*)
Takes in a name (bigbang) Returns where its cached file should be (../sample_git_repos/bigbang_backup.csv)

bigbang.analysis.repo_loader.**create_graph**(*dic*)
Converts a dictionary of dependencies into a NetworkX DiGraph.

bigbang.analysis.repo_loader.**fetch_repo**(*url*)
Takes in a git url and uses shell commands to clone the git repo into sample_git_repos/

TODO: We shouldn't use this with shell=True because of security concerns.

bigbang.analysis.repo_loader.**filepath_to_name**(*filepath*)
Converts a filepath (../archives/sample_git_repos/{name}) to a name. Note that this will fail if the filepath ends in a "/". It must end in the name of the folder. Thus, it should be ../archives/sample_git_repos/{name} not ../archives/sample_git_repos/{name}/

bigbang.analysis.repo_loader.**get_cache**(*name*)
Takes in a name (bigbang) Returns a GitRepo object containing the cache data if the cache exists Returns None otherwise.

bigbang.analysis.repo_loader.**get_dependency_network**(*filepath*)
Given a directory, collects all Python and IPython files and uses the Python AST to create a dictionary of dependencies from them. Returns the dependencies converted into a NetworkX graph.

bigbang.analysis.repo_loader.**get_files**(*filepath*)
Returns a list of the Python files in a directory, and converts IPython notebooks into Python source code and includes them with the Python files.

bigbang.analysis.repo_loader.**get_multi_repo**(*repo_names=None*, *repos=None*)
As of now, this only accepts names/repos, not local urls TODO: This could be optimized

bigbang.analysis.repo_loader.**get_org_multirepo**(*org_name*)

bigbang.analysis.repo_loader.**get_org_repos**(*org_name*)
Checks to see if we have the urls for a given org If we don't, it fetches them. Once we do, it returns a list of GitRepo objects from the urls.

bigbang.analysis.repo_loader.**get_repo**(*repo_in*, *in_type='name'*, *update=False*)

Takes three different options for type:

- remote: basically a git url
- name (default): a name like 'scipy' which the method can expand to a url
- **local: a filepath to a file on the local system** (basically an existing git directory on this computer)

This returns an initialized GitRepo object with its data and name already loaded.

bigbang.analysis.repo_loader.**load_org_repos**(*org_name*)
fetches a list of all repos in an organization from github and gathers their URL's (of the form *.git) It dumps these into ../examples/{org_name}_urls.txt

bigbang.analysis.repo_loader.**name_to_filepath**(*name*)
Converts a name of a repo to its filepath. Currently, these go to ../archives/sample_git_repos/{name}/

bigbang.analysis.repo_loader.**repo_already_exists**(*filepath*)

bigbang.analysis.repo_loader.**url_to_name**(*url*)
Converts a github url (e.g. https://github.com/sbenthall/bigbang.git) to a human-readable name (bigbang) by looking at the word between the last "/" and ".git".

7.6 get_dependencies

7.7 datasets.domains

This submodule is responsible for making data about the classification of email domains available in Python memory.

The data is stored in a CSV file that is provided with the BigBang repository.

This file was generated using a script that is provided for the library for reproducibility. The script can be found in `Create Domain-Category Data.ipynb`

bigbang.datasets.domains.domains.**load_data**()
Returns a datafarme with email domains labeled by category.

Categories include: generic, personal, company, academic, sdo

Returns **data**

Return type pandas.DataFrame

7.8 ingress.abstract

class bigbang.ingress.abstract.**AbstractMailList**(*name: str*, *source: Union[List[str], str]*, *msgs: List[mailbox.mboxMessage]*)
Bases: abc.ABC

This class handles the scraping of a all public Emails contained in a single mailing list. To be more precise, each contributor to a mailing list sends their message to an Email address that has the following structure: <mailing_list_name>@<mail_list_domain_name>. Thus, this class contains all Emails send to a specific <mailing_list_name> (the Email localpart).

Parameters

- **name** (*The of whom the list (e.g. 3GPP_COMMON_IMS_XFER, IEEESCO-DIFUSION, ..)*) –
- **source** (*Contains the information of the location of the mailing list.*) – It can be either an URL where the list or a path to the file(s).
- **msgs** (*List of mboxMessage objects*) –

from_url()

from_messages()

from_mbox()

get_message_urls()

get_messages_from_url()

get_index_of_elements_in_selection()

get_name_from_url()

to_dict()

to_pandas_dataframe()

to_mbox()

__getitem__(*index*) → mailbox.mboxMessage
Get specific message at position *index* within the mailing list.

__iter__()
Iterate over each message within the mailing list.

__len__() → int
Get number of messsages within the mailing list.

abstract classmethod from_mbox(*name: str*, *filepath: str*) → *bigbang.ingress.abstract.AbstractMailList*

Parameters

- **name** (*Name of the list of messages, e.g. '3GPP_TSG_SA_WG2_UPCON'.*) –
- **filepath** (*Path to file in which mailing list is stored.*) –

abstract classmethod from_messages(*name: str*, *url: str*, *messages: Union[List[str], List[mailbox.mboxMessage]]*, *fields: str = 'total'*, *url_login: str = None*, *url_pref: str = None*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[str] = None*) → *bigbang.ingress.abstract.AbstractMailList*

Parameters

- **name** (*Name of the list of messages, e.g. 'public-bigdata'*) –
- **url** (*URL to the Email list.*) –
- **messages** (*Can either be a list of URLs to specific messages*) – or a list of *mboxMessage* objects.
- **url_login** (*URL to the 'Log In' page.*) –
- **url_pref** (*URL to the 'Preferences'/settings page.*) –
- **login** (*Login credentials (username and password) that were used to set*) – up AuthSession.
- **session** (*requests.Session() object for the Email list domain website.*) –

abstract classmethod from_url(*name: str*, *url: str*, *select: Optional[dict] = {'fields': 'total'}*, *url_login: Optional[str] = None*, *url_pref: Optional[str] = None*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[requests.sessions.Session] = None*) → *bigbang.ingress.abstract.AbstractMailList*

Parameters

- **name** (*Name of the mailing list.*) –
- **url** (*URL to the mailing list.*) –
- **select** (*Selection criteria that can filter messages by:*) –
  - content, i.e. header and/or body
  - period, i.e. written in a certain year, month, week-of-month
- **url_login** (*URL to the 'Log In' page*) –
- **url_pref** (*URL to the 'Preferences'/settings page*) –
- **login** (*Login credentials (username and password) that were used to set*) – up AuthSession.
- **session** (*requests.Session() object for the Email list domain website.*) –

static get_index_of_elements_in_selection(*times: List[Union[int, str]]*, *urls: List[str]*, *filtr: Union[tuple, list, int, str]*) → List[int]

Filter out messages that where in a specific period. Period here is a set containing units of year, month, and week-of-month which can have the following example elements:

- years: (1992, 2010), [2000, 2008], 2021
- months: ["January", "July"], "November"
- weeks: (1, 4), [1, 5], 2

Parameters

- **times** (*A list containing information of the period for each*) – group of mboxMessage.
- **urls** (*Corresponding URLs of each group of mboxMessage of which the*) – period info is contained in *times*.
- **filtr** (*Containing info on what should be filtered.*) –

Returns

Return type Indices of to the elements in *times*/*ursl*.

abstract classmethod get_message_urls(*name: str*, *url: str*, *select: Optional[dict] = None*) → List[str]

Parameters

- **name** (*Name of the list of messages, e.g. 'public-bigdata'*) –
- **url** (*URL to the mailing list.*) –
- **select** (*Selection criteria that can filter messages by:*) –

- content, i.e. header and/or body
- period, i.e. written in a certain year and month

Returns

Return type List of all selected URLs of the messages in the mailing list.

static get_messages_from_urls(*name: str*, *msg_urls: list*, *msg_parser*, *fields: Optional[str] = 'total'*) → List[mailbox.mboxMessage]

Generator that returns all messages within a certain period (e.g. January 2021, Week 5).

Parameters

- **name** (*Name of the list of messages, e.g. '3GPP_TSG_SA_WG2_UPCON'*) –
- **url** (*URL to the mailing list.*) –
- **fields** (*Content, i.e. header and/or body*) –

abstract get_name_from_url() → str

to_dict(*include_body: bool = True*) → Dict[str, List[str]]

Parameters **include_body** (*A boolean that indicates whether the message body should*) – be included or not.

Returns

- *A Dictionary with the first key layer being the header field names and*
- *the "body" key. Each value field is a list containing the respective*
- *header field contents arranged by the order as they were scraped from*
- *the web. This format makes the conversion to a pandas.DataFrame easier.*

to_mbox(*dir_out: str*, *filename: Optional[str] = None*)

Safe mailing list to .mbox files.

to_pandas_dataframe(*include_body: bool = True*) → pandas.core.frame.DataFrame

Parameters **include_body** (*A boolean that indicates whether the message body should*) – be included or not.

Returns

- *Converts the mailing list into a pandas.DataFrame object in which each*
- *row represents an Email.*

class bigbang.ingress.abstract.**AbstractMailListDomain**(*name: str*, *url: str*, *lists: List[Union[*bigbang.ingress.abstract.AbstractMailList, *str]]*)

Bases: abc.ABC

This class handles the scraping of a all public Emails contained in a mail list domain. To be more precise, each contributor to a mailing archive sends their message to an Email address that has the following structure: <mailing_list_name>@<mail_list_domain_name>. Thus, this class contains all Emails send to <mail_list_domain_name> (the Email domain name). These Emails are contained in a list of *AbstractMailList* types, such that it is known to which <mailing_list_name> (the Email localpart) was send.

Parameters

- **name** (*The mail list domain name (e.g. 3GPP, IEEE, W3C)*) –
- **url** (*The URL where the archive lives*) –

- **lists** (A list containing the mailing lists as *AbstractMailList* types) –

from_url()

from_mailing_lists()

from_mbox()

get_lists_from_url()

to_dict()

to_pandas_dataframe()

to_mbox()

__getitem__(*index*)
Get specific mailing list at position *index* from the mail list domain.

__iter__()
Iterate over each mailing list within the mail list domain.

__len__()
Get number of mailing lists within the mail list domain.

abstract classmethod from_mailing_lists(*name: str*, *url_root: str*, *url_mailing_lists: Union[List[str], List[bigbang.ingress.abstract.AbstractMailList]]*, *select: Optional[dict] = None*, *url_login: Optional[str] = None*, *url_pref: Optional[str] = None*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[str] = None*, *only_mlist_urls: bool = True*, *instant_save: Optional[bool] = True*) → *bigbang.ingress.abstract.AbstractMailListDomain*
Create mailing mail list domain from a given list of 'AbstractMailList` instances or URLs pointing to mailing lists.

Parameters

- **name** (*mail list domain name, such that multiple instances of*) – *AbstractMailListDomain* can easily be distinguished.
- **url_root** (*The invariant root URL that does not change no matter what*) – part of the mail list domain we access.
- **url_mailing_lists** (This argument can either be a list of *AbstractMailList*) – objects or a list of string containing the URLs to the mailing list of interest.
- **url_login** (*URL to the 'Log In' page.*) –
- **url_pref** (*URL to the 'Preferences'/settings page.*) –
- **login** (*Login credentials (username and password) that were used to set*) – up AuthSession.
- **session** (*requests.Session() object for the mail list domain website.*) –
- **only_list_urls** (*Boolean giving the choice to collect only mailing list*) – URLs or also their contents.

- **instant_save** (Boolean giving the choice to save a *AbstractMailList* as) – soon as it is completely scraped or collect entire mail list domain. The prior is recommended if a large number of mailing lists are scraped which can require a lot of memory and time.

abstract classmethod from_mbox(*name: str*, *directorypath: str*, *filedsc: str = '*.mbox'*) → *bigbang.ingress.abstract.AbstractMailListDomain*

Parameters

- **name** (*mail list domain name, such that multiple instances of*) – *AbstractMailListDomain* can easily be distinguished.
- **directorypath** (Path to the folder in which *AbstractMailListDomain* is stored.) –
- **filedsc** (*Optional filter that only reads files matching the description.*) – By default all files with an mbox extension are read.

abstract classmethod from_url(*name: str*, *url_root: str*, *url_home: Optional[str] = None*, *select: Optional[dict] = None*, *url_login: Optional[str] = None*, *url_pref: Optional[str] = None*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[str] = None*, *instant_save: bool = True*, *only_mlist_urls: bool = True*) → *bigbang.ingress.abstract.AbstractMailListDomain*

Create a mail list domain from a given URL. :param name: *AbstractMailListDomain* can easily be distinguished. :type name: Email list domain name, such that multiple instances of :param url_root: part of the mail list domain we access. :type url_root: The invariant root URL that does not change no matter what :param url_home: it contains the different sections which we obtain using *get_sections()*. :type url_home: The 'home' space of the mail list domain. This is required as :param select:

- content, i.e. header and/or body
- period, i.e. written in a certain year, month, week-of-month

Parameters

- **url_login** (*URL to the 'Log In' page.*) –
- **url_pref** (*URL to the 'Preferences'/settings page.*) –
- **login** (*Login credentials (username and password) that were used to set*) – up AuthSession.
- **session** (*requests.Session() object for the mail list domain website.*) –
- **instant_save** (Boolean giving the choice to save a *AbstractMailList* as) – soon as it is completely scraped or collect entire mail list domain. The prior is recommended if a large number of mailing lists are scraped which can require a lot of memory and time.
- **only_list_urls** (Boolean giving the choice to collect only *AbstractMailList*) – URLs or also their contents.

abstract classmethod get_lists_from_url(*url_home: str*, *select: dict*, *session: Optional[str] = None*, *instant_save: bool = True*, *only_mlist_urls: bool = True*) → List[Union[*bigbang.ingress.abstract.AbstractMailList*, str]]

Created dictionary of all lists in the mail list domain.

Parameters

- **url_root** (*The invariant root URL that does not change no matter what*) – part of the mail list domain we access.
- **url_home** (*The 'home' space of the mail list domain. This is required as*) – it contains the different sections which we obtain using *get_sections()*.
- **select** (*Selection criteria that can filter messages by:*) –
  - content, i.e. header and/or body
  - period, i.e. written in a certain year, month, week-of-month
- **session** (*requests.Session() object for the mail list domain website.*) –
- **instant_save** (Boolean giving the choice to save a *AbstractMailList* as) – soon as it is completely scraped or collect entire mail list domain. The prior is recommended if a large number of mailing lists are scraped which can require a lot of memory and time.
- **only_list_urls** (Boolean giving the choice to collect only *AbstractMailList*) – URLs or also their contents.

Returns archive_dict

Return type the keys are the names of the lists and the value their url

to_dict(*include_body: bool = True*) → Dict[str, List[str]]
Concatenates mailing list dictionaries created using *AbstractMailList.to_dict()*.

to_mbox(*dir_out: str*)
Save mail list domain content to .mbox files

to_pandas_dataframe(*include_body: bool = True*) → pandas.core.frame.DataFrame
Concatenates mailing list pandas.DataFrames created using *AbstractMailList.to_pandas_dataframe()*.

exception bigbang.ingress.abstract.**AbstractMailListDomainWarning**
Bases: BaseException

Base class for AbstractMailListDomain class specific exceptions

exception bigbang.ingress.abstract.**AbstractMailListWarning**
Bases: BaseException

Base class for AbstractMailList class specific exceptions

class bigbang.ingress.abstract.**AbstractMessageParser**(*website=False*, *url_login: Optional[str] = None*, *url_pref: Optional[str] = None*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[requests.sessions.Session] = None*)
Bases: abc.ABC

This class handles the creation of an mailbox.mboxMessage object (using the from_*() methods) and its storage in various other file formats (using the to_*() methods) that can be saved on the local memory.

create_email_message(*archived_at: str*, *body: str*, ***header*) → mailbox.mboxMessage

Parameters

- **archived_at** (*URL to the Email message.*) –
- **body** (*String that contains the body of the message.*) –

- **header** (*Dictionary that contains all available header fields of the*) – message.

from_url(*list_name: str*, *url: str*, *fields: str = 'total'*) → mailbox.mboxMessage

Parameters

- **list_name** (*The name of the mailing list.*) –
- **url** (*URL of this Email*) –
- **fields** (*Indicates whether to return 'header', 'body' or 'total'/both or*) – the Email. The latter is the default.

static to_dict(*msg: mailbox.mboxMessage*) → Dict[str, List[str]]
Convert mboxMessage to a Dictionary

static to_mbox(*msg: mailbox.mboxMessage*, *filepath: str*)

Parameters

- **msg** (*The Email.*) –
- **filepath** (*Path to file in which the Email will be stored.*) –

static to_pandas_dataframe(*msg: mailbox.mboxMessage*) → pandas.core.frame.DataFrame
Convert mboxMessage to a pandas.DataFrame

exception bigbang.ingress.abstract.**AbstractMessageParserWarning**
Bases: BaseException

Base class for AbstractMessageParser class specific exceptions

7.9 ingress.listserv

class bigbang.ingress.listserv.**ListservMailList**(*name: str*, *source: Union[List[str], str]*, *msgs: List[mailbox.mboxMessage]*)
Bases: *bigbang.ingress.abstract.AbstractMailList*

This class handles the scraping of a all public Emails contained in a single mailing list in the LISTSERV 16.5 and 17 format. To be more precise, each contributor to a mailing list sends their message to an Email address that has the following structure: <mailing_list_name>@LIST.ETSI.ORG. Thus, this class contains all Emails send to a specific <mailing_list_name> (the Email localpart, such as "3GPP_TSG_CT_WG1" or "3GPP_TSG_CT_WG3_108E_MAIN").

Parameters

- **name** (*The of whom the list (e.g. 3GPP_COMMON_IMS_XFER, IEEESCO-DIFUSION, ..)*) –
- **source** (*Contains the information of the location of the mailing list.*) – It can be either an URL where the list or a path to the file(s).
- **msgs** (*List of mboxMessage objects*) –

Example

To scrape a Listserv mailing list from an URL and store it in run-time memory, we do the following >>> mlist = ListservMailList.from_url(>>> name="IEEE-TEST", >>> url="https://listserv.ieee.org/cgi-bin/wa?A0=IEEE-TEST", >>> select={ >>> "years": 2015, >>> "months": "November", >>> "weeks": 4, >>> "fields": "header", >>> }, >>> login={"username": <your_username>, "password": <your_password>}, >>>)

To save it as *.mbox file we do the following >>> mlist.to_mbox(path_to_file)

classmethod from_listserv_directories(*name: str*, *directorypaths: List[str]*, *filedsc: str = '*.LOG?????'*, *select: Optional[dict] = None*) → *bigbang.ingress.listserv.ListservMailList*

This method is required if the files that contain the list messages were directly exported from LISTSERV 16.5 (e.g. by a member of 3GPP). Each mailing list has its own directory and is split over multiple files with an extension starting with LOG and ending with five digits.

Parameters

- **name** (*Name of the list of messages, e.g. '3GPP_TSG_SA_WG2_UPCON'.*) –
- **directorypaths** (*List of directory paths where LISTSERV formatted*) – messages are.
- **filedsc** (A description of the relevant files, e.g. *.LOG?????) –
- **select** (*Selection criteria that can filter messages by:*) –
  - content, i.e. header and/or body
  - period, i.e. written in a certain year, month, week-of-month

classmethod from_listserv_files(*name: str*, *filepaths: List[str]*, *select: Optional[dict] = None*) → *bigbang.ingress.listserv.ListservMailList*

This method is required if the files that contain the list messages were directly exported from LISTSERV 16.5 (e.g. by a member of 3GPP). Each mailing list has its own directory and is split over multiple files with an extension starting with LOG and ending with five digits. Compared to *ListservMailList.from_listserv_directories()*, this method reads messages from single files, instead of all the files contained in a directory.

Parameters

- **name** (*Name of the list of messages, e.g. '3GPP_TSG_SA_WG2_UPCON'*) –
- **filepaths** (*List of file paths where LISTSERV formatted messages are.*) – Such files can have a file extension of the form: *.LOG1405D
- **select** (*Selection criteria that can filter messages by:*) –
  - content, i.e. header and/or body
  - period, i.e. written in a certain year, month, week-of-month

classmethod from_mbox(*name: str*, *filepath: str*) → *bigbang.ingress.listserv.ListservMailList*

Docstring in *AbstractMailList*.

classmethod from_messages(*name: str, url: str, messages: Union[List[str], List[mailbox.mboxMessage]], fields: str = 'total', url_login: str = 'https://list.etsi.org/scripts/wa.exe?LOGON', url_pref: str = 'https://list.etsi.org/scripts/wa.exe?PREF', login: Optional[Dict[str, str]] = {'password': None, 'username': None}, session: Optional[str] = None*) → *bigbang.ingress.listserv.ListservMailList*

Docstring in *AbstractMailList*.

classmethod from_url(*name: str, url: str, select: Optional[dict] = {'fields': 'total'}, url_login: str = 'https://list.etsi.org/scripts/wa.exe?LOGON', url_pref: str = 'https://list.etsi.org/scripts/wa.exe?PREF', login: Optional[Dict[str, str]] = {'password': None, 'username': None}, session: Optional[requests.sessions.Session] = None*) → *bigbang.ingress.listserv.ListservMailList*

Docstring in *AbstractMailList*.

static get_all_periods_and_their_urls(*url: str*) → Tuple[List[str], List[str]]

LISTSERV groups messages into weekly time bundles. This method obtains all the URLs that lead to the messages of each time bundle.

Returns

- *Returns a tuple of two lists that look like*
- *(['April 2017, 2', 'January 2001', . . .], ['ulr1', 'url2', . . .])*

classmethod get_line_numbers_of_header_starts(*content: List[str]*) → List[int]

By definition LISTSERV logs seperate new messages by a row of 73 equal signs.

Parameters **content** (*The content of one LISTSERV file.*) –

Returns

Return type List of line numbers where header starts

classmethod get_message_urls(*name: str, url: str, select: Optional[dict] = None*) → List[str]

Docstring in *AbstractMailList*.

This routine is needed for Listserv 16.5

static get_name_from_url(*url: str*) → str

Get name of mailing list.

classmethod get_period_urls(*url: str, select: Optional[dict] = None*) → List[str]

All messages within a certain period (e.g. January 2021, Week 5).

Parameters

- **url** (*URL to the LISTSERV list.*) –
- **select** (*Selection criteria that can filter messages by:*) –
  - content, i.e. header and/or body
  - period, i.e. written in a certain year, month, week-of-month

class bigbang.ingress.listserv.**ListservMailListDomain**(*name: str, url: str, lists: List[Union[*bigbang.ingress.abstract.AbstractMailList, *str]]*)

Bases: *bigbang.ingress.abstract.AbstractMailListDomain*

This class handles the scraping of a all public Emails contained in a mail list domain that has the LISTSERV 16.5 and 17 format, such as 3GPP. To be more precise, each contributor to a mail list domain sends their message

to an Email address that has the following structure: <mailing_list_name>@w3.org. Thus, this class contains all Emails send to <mail_list_domain_name> (the Email domain name). These Emails are contained in a list of *ListservMailList* types, such that it is known to which <mailing_list_name> (the Email localpart) was send.

Parameters

- **name** (*The mailing list domain name (e.g. 3GPP, IEEE, ..)*) –
- **url** (*The URL where the mailing list domain lives*) –
- **lists** (A list containing the mailing lists as *ListservMailList* types) –

All methods in the `AbstractMailListDomain` class in addition to:

from_listserv_directory()

get_sections()

Example

To scrape a Listserv mailing list domain from an URL and store it in run-time memory, we do the following >>> mlistdom = ListservMailListDomain.from_url(>>> name="IEEE", >>> url_root="https://listserv.ieee.org/cgi-bin/wa?", >>> url_home="https://listserv.ieee.org/cgi-bin/wa?HOME", >>> select={ >>> "years": 2015, >>> "months": "November", >>> "weeks": 4, >>> "fields": "header", >>> }, >>> login={"username": <your_username>, "password": <your_password>}, >>> instant_save=False, >>> only_mlist_urls=False, >>>)

To save it as *.mbox file we do the following >>> mlistdom.to_mbox(path_to_directory)

classmethod from_listserv_directory(*name: str*, *directorypath: str*, *folderdsc: str = '*'*, *filedsc: str = '*.LOG?????'*, *select: Optional[dict] = None*) → *bigbang.ingress.listserv.ListservMailListDomain*

This method is required if the files that contain the mail list domain messages were directly exported from LISTSERV 16.5 (e.g. by a member of 3GPP). Each mailing list has its own subdirectory and is split over multiple files with an extension starting with LOG and ending with five digits.

Parameters

- **name** (*mail list domain name, such that multiple instances of*) – *ListservMailListDomain* can easily be distinguished.
- **directorypath** (Where the *ListservMailListDomain* can be initialised.) –
- **folderdsc** (*A description of the relevant folders*) –
- **filedsc** (A description of the relevant files, e.g. *.LOG?????) –
- **select** (*Selection criteria that can filter messages by:*) –
  - content, i.e. header and/or body
  - period, i.e. written in a certain year, month, week-of-month

classmethod from_mailing_lists(*name: str*, *url_root: str*, *url_mailing_lists: Union[List[str], List[*bigbang.ingress.listserv.ListservMailList*]]*, *select: Optional[dict] = {'fields': 'total'}*, *url_login: str = 'https://list.etsi.org/scripts/wa.exe?LOGON'*, *url_pref: str = 'https://list.etsi.org/scripts/wa.exe?PREF'*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[str] = None*, *only_mlist_urls: bool = True*, *instant_save: Optional[bool] = True*) → *bigbang.ingress.listserv.ListservMailListDomain*

Docstring in *AbstractMailList*.

classmethod from_mbox(*name: str*, *directorypath: str*, *filedsc: str = '*.mbox'*) → *bigbang.ingress.listserv.ListservMailList*
Docstring in *AbstractMailList*.

classmethod from_url(*name: str*, *url_root: str*, *url_home: str*, *select: Optional[dict] = {'fields': 'total'}*, *url_login: str = 'https://list.etsi.org/scripts/wa.exe?LOGON'*, *url_pref: str = 'https://list.etsi.org/scripts/wa.exe?PREF'*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[str] = None*, *instant_save: bool = True*, *only_mlist_urls: bool = True*) → *bigbang.ingress.listserv.ListservMailListDomain*
Docstring in *AbstractMailList*.

static get_lists_from_url(*url_root: str*, *url_home: str*, *select: dict*, *session: Optional[str] = None*, *instant_save: bool = True*, *only_mlist_urls: bool = True*) → List[Union[*bigbang.ingress.listserv.ListservMailList*, str]]
Docstring in *AbstractMailList*.

get_sections(*url_home: str*) → int
Get different sections of mail list domain. On the Listserv 16.5 website they look like: [3GPP] [3GPP–AT1] [AT2–CONS] [CONS–EHEA] [EHEA–ERM_] ... On the Listserv 17 website they look like: [<<][<]1-50(798)[>][>>]

Returns

- *If sections exist, it returns their urls and names. Otherwise it returns*
- *the url_home.*

exception bigbang.ingress.listserv.**ListservMailListDomainWarning**
Bases: BaseException

Base class for ListservMailListDomain class specific exceptions

exception bigbang.ingress.listserv.**ListservMailListWarning**
Bases: BaseException

Base class for ListservMailList class specific exceptions

class bigbang.ingress.listserv.**ListservMessageParser**(*website=False*, *url_login: Optional[str] = None*, *url_pref: Optional[str] = None*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[requests.sessions.Session] = None*)
Bases: *bigbang.ingress.abstract.AbstractMessageParser*, email.parser.Parser

This class handles the creation of an mailbox.mboxMessage object (using the from_*() methods) and its storage in various other file formats (using the to_*() methods) that can be saved on the local memory.

Parameters

- **website** (*Set 'True' if messages are going to be scraped from websites,*) – otherwise 'False' if read from local memory.
- **url_login**(*URL to the 'Log In' page.*)–
- **url_pref**(*URL to the 'Preferences'/settings page.*)–

- **login** (*Login credentials (username and password) that were used to set*) – up AuthSession. You can create your own for the 3GPP mail list domain.
- **session** (*requests.Session() object for the mail list domain website.*) –

from_url()

from_listserv_file()

_get_header_from_html()

_get_body_from_html()

_get_header_from_listserv_file()

_get_body_from_listserv_file()

Example

To create a Email message parser object, use the following syntax: >>> msg_parser = ListservMessageParser(>>> website=True, >>> login={"username": <your_username>, "password": <your_password>}, >>>)

To obtain the Email message content and return it as *mboxMessage* object, you need to do the following: >>> msg = msg_parser.from_url(>>> list_name="3GPP_TSG_RAN_DRAFTS", >>> url="https://list.etsi.org/scripts/wa.exe?A2=ind2010B&L=3GPP_TSG_RAN_DRAFTS&O=D&P=29883", >>> fields="total", >>>)

empty_header = {}

from_listserv_file(*list_name: str*, *file_path: str*, *header_start_line_nr: int*, *fields: str = 'total'*) → mailbox.mboxMessage

This method is required if the message is inside a file that was directly exported from LISTSERV 16.5 (e.g. by a member of 3GPP). Such files have an extension starting with LOG and ending with five digits.

Parameters

- **list_name** (*The name of the LISTSERV Email list.*) –
- **file_path** (*Path to file that contains the Email list.*) –
- **header_start_line_nr** (*Line number in the file on which a new message starts.*) –
- **fields** (*Indicates whether to return 'header', 'body' or 'total'/both or*) – the Email.

exception bigbang.ingress.listserv.**ListservMessageParserWarning**

Bases: BaseException

Base class for ListservMessageParser class specific exceptions

7.10 ingress.w3c

class bigbang.ingress.w3c.**W3CMailList**(*name: str, source: Union[List[str], str], msgs: List[mailbox.mboxMessage]*)

Bases: *bigbang.ingress.abstract.AbstractMailList*

This class handles the scraping of a all public Emails contained in a single mailing list in the hypermail format. To be more precise, each contributor to a mailing list sends their message to an Email address that has the following structure: <mailing_list_name>@w3.org. Thus, this class contains all Emails send to a specific <mailing_list_name> (the Email localpart, such as "public-abcg" or "public-accesslearn-contrib").

Parameters

- **name** (*The name of the list (e.g. public-2018-permissions-ws, ..)*) –
- **source** (*Contains the information of the location of the mailing list.*) – It can be either an URL where the list or a path to the file(s).
- **msgs** (*List of mboxMessage objects*) –

Example

To scrape a W3C mailing list from an URL and store it in run-time memory, we do the following

```
>>> mlist = W3CMailList.from_url(
>>>     name="public-bigdata",
>>>     url="https://lists.w3.org/Archives/Public/public-bigdata/",
>>>     select={
>>>         "years": 2015,
>>>         "months": "August",
>>>         "fields": "header",
>>>     },
>>> )
```

To save it as `*.mbox` file we do the following >>> mlist.to_mbox(path_to_file)

classmethod from_mbox(*name: str, filepath: str*) → *bigbang.ingress.w3c.W3CMailList*
Docstring in *AbstractMailList*.

classmethod from_messages(*name: str, url: str, messages: Union[List[str], List[mailbox.mboxMessage]], fields: str = 'total'*) → *bigbang.ingress.w3c.W3CMailList*
Docstring in *AbstractMailList*.

classmethod from_url(*name: str, url: str, select: Optional[dict] = {'fields': 'total'}*) → *bigbang.ingress.w3c.W3CMailList*
Docstring in *AbstractMailList*.

static get_all_periods_and_their_urls(*url: str*) → Tuple[List[str], List[str]]
W3C groups messages into monthly time bundles. This method obtains all the URLs that lead to the messages of each time bundle.

Returns

- *Returns a tuple of two lists that look like*
- *(['April 2017', 'January 2001', ...], ['ulr1', 'url2', ...])*

classmethod get_message_urls(*name: str, url: str, select: Optional[dict] = None*) → List[str]
Docstring in *AbstractMailList*.

classmethod get_messages_urls(*name: str*, *url: str*) → List[str]

Parameters

- **name** (*Name of the W3C mailing list.*) –
- **url** (*URL to group of messages that are within the same period.*) –

Returns

Return type List of URLs from which *mboxMessage* can be initialized.

static get_name_from_url(*url: str*) → str
Get name of mailing list.

classmethod get_period_urls(*url: str*, *select: Optional[dict] = None*) → List[str]
All messages within a certain period (e.g. January 2021).

Parameters

- **url** (*URL to the W3C list.*) –
- **select** (*Selection criteria that can filter messages by:*) –
  - content, i.e. header and/or body
  - period, i.e. written in a certain year and month

class bigbang.ingress.w3c.**W3CMailListDomain**(*name: str*, *url: str*, *lists: List[Union[*bigbang.ingress.abstract.AbstractMailList, *str]]*)
Bases: *bigbang.ingress.abstract.AbstractMailListDomain*

This class handles the scraping of a all public Emails contained in a mail list domain that has the hypermail format, such as W3C. To be more precise, each contributor to a mail list domain sends their message to an Email address that has the following structure: <mailing_list_name>@w3.org. Thus, this class contains all Emails send to <mail_list_domain_name> (the Email domain name). These Emails are contained in a list of *W3CMailList* types, such that it is known to which <mailing_list_name> (the Email localpart) was send.

Parameters

- **name** (*The name of the mailing list domain.*) –
- **url** (*The URL where the mailing list domain lives*) –
- **lists** (A list containing the mailing lists as *W3CMailList* types) –

All methods in the `AbstractMailListDomain` class.

Example

To scrape a W3C mailing list mailing list domain from an URL and store it in run-time memory, we do the following >>> mlistdom = W3CMailListDomain.from_url(>>> name="W3C", >>> url_root="https://lists.w3.org/Archives/Public/", >>> select={ >>> "years": 2015, >>> "months": "November", >>> "weeks": 4, >>> "fields": "header", >>> }, >>> instant_save=False, >>> only_mlist_urls=False, >>>)

To save it as *.mbox file we do the following >>> mlistdom.to_mbox(path_to_directory)

classmethod from_mailing_lists(*name: str*, *url_root: str*, *url_mailing_lists: Union[List[str], List[*bigbang.ingress.w3c.W3CMailList*]]*, *select: Optional[dict] = {'fields': 'total'}*, *only_mlist_urls: bool = True*, *instant_save: Optional[bool] = True*) → *bigbang.ingress.w3c.W3CMailListDomain*

Docstring in *AbstractMailListDomain*.

classmethod from_mbox(*name: str*, *directorypath: str*, *filedsc: str = '*.mbox'*) → *bigbang.ingress.w3c.W3CMailListDomain*

Docstring in *AbstractMailListDomain*.

classmethod from_url(*name: str*, *url_root: str*, *url_home: Optional[str] = None*, *select: Optional[dict] = {'fields': 'total'}*, *instant_save: bool = True*, *only_mlist_urls: bool = True*) → *bigbang.ingress.w3c.W3CMailListDomain*

Docstring in *AbstractMailListDomain*.

static get_lists_from_url(*name: str*, *select: dict*, *url_root: str*, *url_home: Optional[str] = None*, *instant_save: bool = True*, *only_mlist_urls: bool = True*) → List[Union[*bigbang.ingress.w3c.W3CMailList*, str]]

Docstring in *AbstractMailListDomain*.

exception bigbang.ingress.w3c.**W3CMailListDomainWarning**

Bases: BaseException

Base class for W3CMailListDomain class specific exceptions

exception bigbang.ingress.w3c.**W3CMailListWarning**

Bases: BaseException

Base class for W3CMailList class specific exceptions

class bigbang.ingress.w3c.**W3CMessageParser**(*website=False*, *url_login: Optional[str] = None*, *url_pref: Optional[str] = None*, *login: Optional[Dict[str, str]] = {'password': None, 'username': None}*, *session: Optional[requests.sessions.Session] = None*)

Bases: *bigbang.ingress.abstract.AbstractMessageParser*, email.parser.Parser

This class handles the creation of an *mailbox.mboxMessage* object (using the from_*() methods) and its storage in various other file formats (using the to_*() methods) that can be saved on the local memory.

Parameters

- **website** (*Set 'True' if messages are going to be scraped from websites,*) – otherwise 'False' if read from local memory. This distinction needs to be made if missing messages should be added.
- **url_pref** (*URL to the 'Preferences'/settings page.*) –

Example

To create a Email message parser object, use the following syntax: >>> msg_parser = W3CMessageParser(website=True)

To obtain the Email message content and return it as *mboxMessage* object, you need to do the following: >>> msg = msg_parser.from_url(>>> list_name="public-2018-permissions-ws", >>> url="https://lists.w3.org/Archives/Public/public-2018-permissions-ws/2019May/0000.html", >>> fields="total", >>>)

empty_header = {}

exception bigbang.ingress.w3c.**W3CMessageParserWarning**

Bases: BaseException

Base class for W3CMessageParser class specific exceptions

bigbang.ingress.w3c.**parse_dfn_header**(*header_text*)

bigbang.ingress.w3c.**text_for_selector**(*soup: bs4.BeautifulSoup*, *selector: str*)
Filter out header or body field from website and return them as utf-8 string.

7.11 ingress.git_repo

class bigbang.ingress.git_repo.**GitRepo**(*name*, *url=None*, *attribs=['HEXSHA', 'Committer Name', 'Committer Email', 'Commit Message', 'Time', 'Parent Commit', 'Touched File']*, *cache=None*)

Bases: object

Store a git repository given the address to that repo relative to this file.

It returns the data in many forms.

Index a Pandas DataFrame object by time.

That stores the raw form of the repo's commit data as a table.

Each row in this table is a commit.

And each column represents an attribute of that commit: (eg.: time, message, commiter name, committer email, commit hexsha).

by_committer()
Return commit data grouped by commiter.

property commit_data
Return commit data.

commits_for_committer(*committer_name*)
Return commits for committer given the commiter name.

commits_per_day()
Return commits grouped by day.

commits_per_day_full()
Return commits grouped by day and by commiter.

commits_per_week()
Return commits grouped by week.

gen_data(*repo*, *raw*)
Generate data to repo.

merge_with_repo(*other*)
Append commit to a repo.

populate_data(*attribs=['HEXSHA', 'Committer Name', 'Committer Email', 'Commit Message', 'Time', 'Parent Commit', 'Touched File']*)
Populate data.

class bigbang.ingress.git_repo.**MultiGitRepo**(*repos*, *attribs=['HEXSHA', 'Committer Name', 'Committer Email', 'Commit Message', 'Time', 'Parent Commit', 'Touched File']*)

Bases: *bigbang.ingress.git_repo.GitRepo*

Repos must have a "Repo Name" column.

Index a Pandas DataFrame object by time.

That stores the raw form of the repo's commit data as a table.

Each row in this table is a commit.

And each column represents an attribute of that commit: (eg.: time, message, commiter name, committer email, commit hexsha).

bigbang.ingress.git_repo.**cache_fixer**(*r*)
Adds info from row to graph.

7.12 ingress.mailman

exception bigbang.ingress.mailman.**InvalidURLException**(*value*)
Bases: Exception

bigbang.ingress.mailman.**access_provenance**(*directory*)
Return an object with provenance information located in the given directory, or None if no provenance was found.

bigbang.ingress.mailman.**collect_archive_from_url**(*url: Union[list, str]*, *archive_dir='/home/docs/checkouts/readthedocs.org/user_bui py/checkouts/stable/archives/'*, *notes=None*)
Collect archives (generally tar.gz) files from mailmain archive page.

Return True if archives were downloaded, False otherwise (for example if the page lists no accessible archive files).

bigbang.ingress.mailman.**collect_from_file**(*urls_file: str*, *archive_dir: str = '/home/docs/checkouts/readthedocs.org/user_builds/bigbang-py/checkouts/stable/archives/'*, *notes=None*)
Collect urls from a file.

bigbang.ingress.mailman.**collect_from_url**(*url: Union[list, str]*, *archive_dir: str = '/home/docs/checkouts/readthedocs.org/user_builds/bigbang-py/checkouts/stable/archives/'*, *notes=None*)
Collect data from a given url.

bigbang.ingress.mailman.**get_list_name**(*url*)
Return the 'list name' from a canonical mailman archive url.

Otherwise return the same URL.

bigbang.ingress.mailman.**normalize_archives_url**(*url*)
Normalize url.

will try to infer, find or guess the most useful archives URL, given a URL.

Return normalized URL, or the original URL if no improvement is found.

bigbang.ingress.mailman.**open_activity_summary**(*url*, *archive_dir='/home/docs/checkouts/readthedocs.org/user_buil py/checkouts/stable/archives/'*)
Open the message activity summary for a particular mailing list (as specified by url).

Return the dataframe, or return None if no activity summary export file is found.

bigbang.ingress.mailman.**populate_provenance**(*directory*, *list_name*, *list_url*, *notes=None*)
Create a provenance metadata file for current mailing list collection.

bigbang.ingress.mailman.**recursive_get_payload**(*x*)
Get payloads recursively.

bigbang.ingress.mailman.**unzip_archive**(*url*, *archive_dir='/home/docs/checkouts/readthedocs.org/user_builds/bigbang-py/checkouts/stable/archives/'*)
Unzip archive files.

bigbang.ingress.mailman.**update_provenance**(*directory*, *provenance*)
Update provenance file with given object.

bigbang.ingress.mailman.**urls_to_collect**(*urls_file: str*)
Collect urls given urls in a file.

7.13 ingress.utils

Miscellaneous utility functions used in other modules.

bigbang.ingress.utils.**ask_for_input**(*request: str*) → Optional[str]

bigbang.ingress.utils.**get_auth_session**(*url_login: str*, *username: str*, *password: str*) → requests.sessions.Session
Create AuthSession.

There are three ways to create an AuthSession:

- parse username & password directly into method
- create a /bigbang/config/authentication.yaml file that contains keys
- **type then into terminal when the method 'get_login_from_terminal'** is raised

bigbang.ingress.utils.**get_login_from_terminal**(*username: Optional[str]*, *password: Optional[str]*, *file_auth: str = '/home/docs/checkouts/readthedocs.org/user_builds/bigbang-py/checkouts/stable/bigbang/config/authentication.yaml'*) → Tuple[Optional[str]]
Get login key from user during run time if 'username' and/or 'password' is 'None'. Return 'None' if no reply within 15 sec.

bigbang.ingress.utils.**get_website_content**(*url: str*, *session: Optional[requests.sessions.Session] = None*) → Union[str, bs4.BeautifulSoup]
Get HTML code from website

Note: Servers don't like it when one is sending too many requests from same ip address in short period of time. Therefore we need to:

a) **catch 'requests.exceptions.RequestException' errors** (includes all possible errors to be on the safe side),

b) safe intermediate results,

c) continue where we left off at a later stage.

---

bigbang.ingress.utils.**loginkey_to_file**(*username: str*, *password: str*, *file_auth: str*) → None
Safe login key to yaml

bigbang.ingress.utils.**set_website_preference_for_header**(*url_pref: str*, *session: requests.sessions.Session*) → requests.sessions.Session

Set the 'Email Headers' of the 'Archive Preferences' for the auth session to 'Show All Headers'. Otherwise only a restricted list of header fields is shown.

7.14 analysis.attendance

bigbang.analysis.attendance.**name_email_affil_relations_from_IETF_attendance**(*meeting_range=[106, 107, 108]*, *threshold=None*)

Extract and infer from IETF attendance records relations between full names, email address, and affiliations.

In the returned dataframes, each row represents a relation between two of these forms of entity, along with the maximum and minimum date associated with it in the data.

Two forms of inference are used when generating these relational tables:

- Missing values in time are filled forward, then filled backward
- TODO: Affiliations are ran through the entity resolution script to reduce them to a 'cannonical form'

Parameters

- **meeting_range** (*list of ints*) – The numbers of the IETF meetings to use for source data
- **threshold** (*float*) – Defaults to None. If not None, activate entity resolution on the affiliations. Threshold value is used for the entity resolution.

Returns

- **rel_name_affil** (*pandas.DataFrame*)
- **rel_email_affil** (*pandas.DataFrame*)
- **rel_name_email** (*pandas.DataFrame*)

7.15 analysis.listserv

class bigbang.analysis.listserv.**ListservMailList**(*name: str*, *filepath: str*, *msgs: pandas.core.frame.DataFrame*)

Bases: object

Note: Issues loading 3GPP_TSG_RAN_WG1 which is 3.3Gb large

__iter__()
Iterate over each message within the mailing list.

__len__() → int
Get number of messsages within the mailing list.

add_thread_info()
Edit pd.DataFrame to include extra column to identify which thread a message belongs to.

add_weight_to_edge(*dic: dict*, *key1: str*, *key2: str*) → dict

Parameters

- **dic** –
- **key1** –
- **key2** –

static contract(*count: numpy.array*, *label: list*, *contract: float*) → Dict[str, int]
This function contracts all domain names that contributed to a mailinglists below the *contract* threshold into one entity called *Others*. Meaning, if *contract=3* and *nokia.com*, *nokia.com*, *t-mobile.at* all wrote less then three Emails to the mailinglist in question, their contributions are going to be summed into one entity denoted as *Others*.

Parameters

- **count** (*Number of Emails send to mailinglist.*) –
- **label** (*Names of contributers to mailinglist.*) –
- **contract** (*Threshold below which all contributions will be summed.*) –

create_sender_receiver_digraph(*nw: Optional[dict] = None*, *entity_in_focus: Optional[list] = None*, *node_attributes: Optional[Dict[str, list]] = None*)
Create directed graph from messaging network created with ListservMailList.get_sender_receiver_dict().

Parameters

- **nw** (*dictionary created with self.get_sender_receiver_dict()*) –
- **entity_in_focus** (*This can be a list of domain names or localparts. If*) – such a list is provided, the creaed di-graph will only focus on their relations.

crop_by_address(*header_field: str*, *per_address_field: Dict[str, List[str]]*) → *bigbang.analysis.listserv.ListservMailList*

Parameters

- **header_field** (*For a Listserv mailing list the most representative*) – header fields for senders and receivers are 'from' and 'comments-to' respectively.
- **per_address_field** (*Filter by 'local-part' or 'domain' part of an address.*) –

  The data structure of the argument should be, e.g.: {'localpart': [string-1, string-2, ...]}

Returns

Return type *ListservMailList* object cropped to specification.

crop_by_subject(*match=<class 'str'>*, *place: int = 2*) → *bigbang.analysis.listserv.ListservMailList*

Parameters

- **match** (Only keep messages with subject lines containing *match* string.) –
- **place** (Define how to filter for *match*. Use on of the following methods:) – 0 = Using Regex expression 1 = String ends with match 2 =

Returns

Return type *ListservMailList* object cropped to message subject.

crop_by_year (*yrs: Union[int, list]*) → *bigbang.analysis.listserv.ListservMailList*
Filter *self.df* DataFrame by year in message date.

Parameters **yrs** (*Specify a specific year, such as 2021, or a range of years, such*) – as [2011, 2021].

Returns

Return type *ListservMailList* object cropped to specification.

crop_dic_to_entity_in_focus (*dic: dict*, *entity_in_focus: list*) → dict

Parameters **entity_in_focus** (*This can a list of domain names or localparts.*) –

classmethod from_mbox (*name: str*, *filepath: str*, *include_body: bool = True*) → *bigbang.analysis.listserv.ListservMailList*

classmethod from_pandas_dataframe (*df: pandas.core.frame.DataFrame*, *name: Optional[str] = None*, *filepath: Optional[str] = None*) → *bigbang.analysis.listserv.ListservMailList*

get_domains (*header_fields: List[str]*, *return_msg_counts: bool = False*, *df: Optional[pandas.core.frame.DataFrame] = None*) → dict
Get contribution of members per affiliation.

Note: For a Listserv mailing list the most representative header fields of senders and receivers are 'from' and 'comments-to' respectively.

Parameters

- **header_fields** (*Indicate which Email header field to process*) – (e.g. 'from', 'reply-to', 'comments-to'). For a listserv mailing list the most representative header fields of senders and receivers are 'from' and 'comments-to' respectively.
- **return_msg_counts** (*If 'True', return # of messages per domain.*) –

get_domainscount (*header_fields: List[str]*, *per_year: bool = False*) → dict

Parameters

- **header_fields** (*Indicate which Email header field to process*) – (e.g. 'from', 'reply-to', 'comments-to'). For a listserv mailing list the most representative header fields of senders and receivers are 'from' and 'comments-to' respectively.
- **per_year** (*Aggregate results for each year.*) –

get_graph_prop_per_domain_per_year (*years: Optional[tuple] = None*, *func=<function betweenness_centrality>*, ***args*) → dict

Parameters

- **years** –
- **func** –

get_localparts(*header_fields: List[str]*, *per_domain: bool = False*, *return_msg_counts: bool = False*, *df: Optional[pandas.core.frame.DataFrame] = None*) → dict
Get contribution of members per affiliation.

Parameters

- **header_fields** (*Indicate which Email header field to process*) – (e.g. 'from', 'reply-to', 'comments-to'). For a listserv mailing list the most representative header fields of senders and receivers are 'from' and 'comments-to' respectively.
- **per_domain** –
- **return_msg_counts** (*If 'True', return # of messages per localpart.*) –

get_localpartscount(*header_fields: List[str]*, *per_domain: bool = False*, *per_year: bool = False*) → dict

Parameters

- **header_fields** (*Indicate which Email header field to process*) – (e.g. 'from', 'reply-to', 'comments-to'). For a listserv mailing list the most representative header fields of senders and receivers are 'from' and 'comments-to' respectively.
- **per_domain** (*Aggregate results for each domain.*) –
- **per_year** (*Aggregate results for each year.*) –

get_messagescount(*header_fields: Optional[List[str]] = None*, *per_address_field: Optional[str] = None*, *per_year: bool = False*) → dict

Parameters

- **header_fields** (*Indicate which Email header field to process*) – (e.g. 'from', 'reply-to', 'comments-to'). For a listserv mailing list the most representative header fields of senders and receivers are 'from' and 'comments-to' respectively.
- **per_year** (*Aggregate results for each year.*) –

get_messagescount_per_timezone(*percentage: bool = False*) → Dict[str, int]
Get contribution of messages per time zone.

Parameters **percentage** (*Whether to return count of messages percentage w.r.t. total.*) –

static get_name_localpart_domain(*string: str*) → tuple
Split an address field which has (ideally) a format as 'Heinrich von Kleist <Heinrich.vonKleist@SELBST.org>' into name, local-part, and domain. All strings are returned in lower case only to avoid duplicates.

Note: Test whether the incorporation of email.utils.parseaddr() can improve this function.

---

get_sender_receiver_dict(*address_field: str = 'domain'*, *entity_in_focus: Optional[list] = None*, *df: Optional[pandas.core.frame.DataFrame] = None*) → Dict

Parameters

- **address_field** –
- **entity_in_focus** (*This can a list of domain names or localparts. If such*) – a list is provided, the creaed dictionary will only contain their information.

Returns

- *Nested dictionary with first layer the 'from' domain keys and*
- *the second layer the 'comments-to' domain keys with the*
- *integer indicating the number of messages between them.*

get_threads (*return_length: bool = False*) → dict
Collect all messages that belong to the same thread.

Note: Computationally very intensive.

Parameters **return_length** – If 'True', the returned dictionary will be of the form {'subject1': # of messages, 'subject2': # of messages, ... }. If 'False', the returned dictionary will be of the form {'subject1': list of indices, 'subject2': list of indices, ... }.

get_threadsroot (*per_address_field: Optional[str] = None, df: Optional[pandas.core.frame.DataFrame] = None*) → dict
Find all unique message subjects. Replies not treated as a new subject.

Note: The most reliable ways to find the beginning of threads is to check whether the subject line of a message contains an element of reply_labels at the beginning. Checking whether the header field 'comments-to' is empty is not reliable, as 'reply-all' is often chosen by mistake as seen here: 2020-04-01 10:08:58+00:00 joern.krause@etsi.org, juergen.hofmann@nokia.com 2020-03-26 21:41:27+00:00 joern.krause@etsi.org NaN 2020-03-26 21:00:08+00:00 joern.krause@etsi.org juergen.hofmann@nokia.com

i) **Some Emails start with 'AW:', which comes from German and has** the same meaning as 'Re:'.

ii) Some Emails start with '=?utf-8?b?J+WbnuWkjTo=?=' or '=?utf-8?b?J+etlOWkjTo=?=', which are UTF-8 encodings of the Chinese characters '' and '' both of which have the same meaning as 'Re:'.

iii) Leading strings such as 'FW:' are treated as new subjects.

Parameters **per_address_field** –

Returns

- **A dictionary of the form {'subject1'** (*index of message, 'subject2': ... }*)
- *is returned. If per_address_field is specified, the subjects are sorted*
- *into the domain or localpart from which they originate.*

get_threadsrootcount (*per_address_field: Optional[str] = None, per_year: bool = False*) → Union[int, dict]
Identify number conversation threads in mailing list.

Parameters

- **per_address_field** (*Aggregate results for each address field, which can*) – be, e.g., *from*, *send-to*, *received-by*.
- **per_year** (*Aggregate results for each year.*) –

static iterator_name_localpart_domain (*li: list*) → tuple
Generator for the self.get_name_localpart_domain() function.

period_of_activity(*format: str = '%a, %d %b %Y %H:%M:%S %z'*) → list
Return a list containing the datetime of the first and last message written in the mailing list.

static to_percentage(*arr: numpy.array*) → numpy.array

class bigbang.analysis.listserv.**ListservMailListDomain**(*name: str*, *filedsc: str*, *lists: pandas.core.frame.DataFrame*)
Bases: object

Parameters

- **name** – The of whom the mail list domain is (e.g. 3GPP, IEEE, . . .)
- **filedsc** – The file description of the mail list domain
- **lists** – A list containing the mailing lists as *ListservMailList* types

get_mlistscount_per_institution()

classmethod from_mbox(*name: str*, *directorypath: str*, *filedsc: str = '*.mbox'*) → *bigbang.analysis.listserv.ListservMailListDomain*

get_mlistscount_per_institution() → Dict[str, int]
Get a dictionary that lists the mailing lists/working groups in which a institute/company is active.

exception bigbang.analysis.listserv.**ListservMailListDomainWarning**
Bases: BaseException

Base class for ListservMailListDomain class specific exceptions

exception bigbang.analysis.listserv.**ListservMailListWarning**
Bases: BaseException

Base class for ListservMailList class specific exceptions

7.16 analysis.thread

class bigbang.analysis.thread.**Node**(*ID*, *data=None*, *parent=None*)
Bases: object

Form a Node object. ID: Message ID, data: Information about that message, parent: the message's reply-to

add_successor(*successor: list*)
Add a node which has a message that is a reply to this node

get_data()
Return the Information about this message

get_id()
Return message ID

get_parent()
Return Information in the data set about this message

get_successors()
Return a list of nodes of messages which are replies to this node

properties()
Return various properties about the tree with this node as root.

class bigbang.analysis.thread.**Thread**(*root*, *known_root=True*)
Bases: object

Form a thread object. root: the node of the message that start the thread known_root: indicator whether the root node is in our data set

get_content()

get_duration()
Return the time duration of the thread

get_leaves()

get_not_leaves()

get_num_messages()
Return the number of messages in the thread

get_num_people()
Return the number of people in the thread

get_root()
Return the root node.

7.17 analysis.entity_resolution

Tools for resolving entities in a data set, in particular individual persons based on their name and email address.

bigbang.analysis.entity_resolution.**entity_resolve**(*row*, *emailCol*, *nameCol*)
Return a row with name and email by ID.

bigbang.analysis.entity_resolution.**getID**(*name*, *email*)
Get ID from a name and email.

bigbang.analysis.entity_resolution.**name_for_id**(*id*)
Return name by ID.

bigbang.analysis.entity_resolution.**store**(*id*, *name*, *email*)
Store name and email by ID.

7.18 analysis.graph

Tools for studying the graph of interactions between message senders. An interaction, for the purposes of this module, is a direct reply

bigbang.analysis.graph.**ascendancy**(*am*)
Ulanowicz ecosystem health measures Input is weighted adjacency matrix.

bigbang.analysis.graph.**capacity**(*am*)
Return the capacity given a adjacency matrix.

bigbang.analysis.graph.**compute_ascendancy**(*messages*, *duration=50*)
Compute ascendancy given messages.

bigbang.analysis.graph.**interaction_graph_to_matrix**(*dg*)
Turn an interaction graph into a weighted edge matrix.

bigbang.analysis.graph.**messages_to_interaction_graph**(*messages*, *verbose=False*, *clean=True*)
Return a interactable graph given messages.

bigbang.analysis.graph.**messages_to_reply_graph**(*messages*)
Return a graph given messages.

bigbang.analysis.graph.**overhead**(*am*)
Return overhead given a adjacency matrix.

7.19 analysis.process

bigbang.analysis.process.**ai**(*m*, *parts*, *i*)

bigbang.analysis.process.**bi**(*m*, *parts*, *i*)

bigbang.analysis.process.**consolidate_senders_activity**(*activity_df*, *to_consolidate*)
takes a DataFrame in the format returned by activity takes a list of tuples of format ('from 1', 'from 2') to consolidate returns the consolidated DataFrame (a copy, not in place)

bigbang.analysis.process.**containment_distance**(*a*, *b*)
A case-insensitive distance measure on strings.

Returns

- 0 if strings are identical
- positive infinity if neither string contains the other
- 1 / (minimum string length) if one string contains the other.

Good for Organizations. I.e. "cisco" "Cisco" "Cisco Systems" are all 'close' (< .2)

bigbang.analysis.process.**domain_name_from_email**(*name*)

bigbang.analysis.process.**eij**(*m*, *parts*, *i*, *j*)

bigbang.analysis.process.**from_header_distance**(*a*, *b*, *verbose=False*)
A distance measure specifically for the 'From' header of emails. Normalizes based on common differences in client handling of email, then computes Levenshtein distance between components of the field.

bigbang.analysis.process.**matricize**(*series*, *func*)
create a matrix by applying func to pairwise combos of elements in a Series returns a square matrix as a DataFrame should return a symmetric matrix if func(a,b) == func(b,a) should return the identity matrix if func == '=='

bigbang.analysis.process.**minimum_but_not_self**(*column*, *dataframe*)

bigbang.analysis.process.**modularity**(*m*, *parts*)
Compute modularity of an adjacency matrix. Use metric from:

> Zanetti, M. and Schweitzer, F. 2012. "A Network Perspective on Software Modularity" ARCS Workshops 2012, pp. 175-186.

bigbang.analysis.process.**resolve_entities**(*significance*, *distance_function*, *threshold=0*)
Takes a Series mapping entities (index) to significance (values, numerical).

Resolves the entities based on a lexical distance function.

Returns a dictionary of labeled (keys) entity lists (values). Key is the most significant member of the entity list.

bigbang.analysis.process.**resolve_sender_entities**(*act*, *lexical_distance=0*)
Given an Archive's activity matrix, return a dict of lists, each containing message senders ('From' fields) that have been groups to be probably the same entity.

bigbang.analysis.process.**sorted_matrix**(*from_dataframe*, *limit=None*, *sort_key=None*)

Takes a dataframe with 'from' fields for column headers

. Returns a sorted distance matrix for the column headers, using from_header_distance (see method).

7.20 analysis.twopeople

twopeople.py

Written by Raj Agrawal and Ki Deuk Kim

Contains functions used to analyze communication between two people in mailing list Examples can be found in ipython notebook "Collaboration Robustness" in examples folder Each function needs a pandas DataFrame called "exchanges" that contains every two-pair communication between participants in a mailing list.

bigbang.analysis.twopeople.**duration**(*exchanges*, *A*, *B*)
Gets the target two people A, B to analyze and returns the amount of time they communicated in the mailing list in TimeDelta type

bigbang.analysis.twopeople.**num_replies**(*exchanges*, *A*, *B*)
Returns the number of replies that two people A and B sent to each other in a tuple (# of replies from A to B, # of replies from B to A)

bigbang.analysis.twopeople.**panda_allpairs**(*exchanges*, *pairs*)
With given pairs of communication, returns a Pandas DataFrame that contains communication information between two people A and B in every pair

bigbang.analysis.twopeople.**panda_pair**(*exchanges*, *A*, *B*)
Forms a new Pandas DataFrame that contains information about communication between a pair A and B using functions provided above and returns the result

bigbang.analysis.twopeople.**reciprocity**(*exchanges*, *A*, *B*)
Returns the reciprocity of communication between two people A and B in float type. This expresses how interactively they communicated to each other

bigbang.analysis.twopeople.**unique_pairs**(*exchanges*)
Finds every unique pair (A, B) from the pandas DataFrame "exchanges" and returns them in set data type

7.21 analysis.utils

bigbang.analysis.utils.**clean_addresses**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

bigbang.analysis.utils.**clean_datetime**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

bigbang.analysis.utils.**clean_subject**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

bigbang.analysis.utils.**domain_entropy**(*domain*, *froms*)
Compute the entropy of the distribution of counts of email prefixes within the given archive.

Parameters

- **domain** (*string*) – An email domain
- **froms** (*pandas.DataFrame*) – A pandas.DataFrame with From fields, email address, and domains. See the Archive method `get_froms()`

Returns entropy

Return type float

`bigbang.analysis.utils.`**`extract_domain`**(*from_field*)
Returns the domain of an email address from a string.

`bigbang.analysis.utils.`**`extract_email`**(*from_field*)
Returns an email address from a string.

`bigbang.analysis.utils.`**`get_index_of_msgs_with_datetime`**(*df: pandas.core.frame.DataFrame*, *return_boolmask: bool = False*) → numpy.array

`bigbang.analysis.utils.`**`get_index_of_msgs_with_subject`**(*df: pandas.core.frame.DataFrame*, *return_boolmask: bool = False*) → numpy.array

7.22 visualisation.lines

7.23 visualisation.plot

Complex plotting functions.

`bigbang.visualisation.plot.`**`draw_adjacency_matrix`**(*G*, *node_order=None*, *partitions=[]*, *colors=[]*, *cmap='Greys'*, *figsize=(6, 6)*)

- G is a networkx graph
- **node_order (optional) is a list of nodes, where each node in G** appears exactly once
- **partitions is a list of node lists, where each node in G appears** in exactly one node list
- **colors is a list of strings indicating what color each** partition should be

If partitions is specified, the same number of colors needs to be specified.

`bigbang.visualisation.plot.`**`stack`**(*df*, *partition=None*, *smooth=1*, *figsize=(12.5, 7.5)*, *time=True*, *cm=<matplotlib.colors.ListedColormap object>*)
Plots a stackplot based on a dataframe. Includes support for partitioning and convolution.

df - a dataframe partition - a (dictionary or list) of lists of columns of df

- if dictionary, keys are used as labels

smooth - an integer amount of convolution

7.24 visualisation.graphs

7.25 visualisation.utils

7.26 visualisation.stackedareachart

CHAPTER

EIGHT

CONTRIBUTING

The BigBang community welcomes contributions.

8.1 Release Procedure

When the community decides that it is time to cut a new release, the Core Developers select somebody to act as release manager. That release manager then performs the following steps.

1. Determine the next release number via the standards of semantic versioning.
2. Solicit a worthy name for the release.
3. Address any remaining tickets in the GitHub milestone corresponding to the release, perhaps moving them to other milestones.
4. Consult the GitHub records of merged PRs and issues to write release notes documenting the changes made in this release.
5. If the dependencies in main are not already frozen, use `pip freeze` to create a new frozen dependency list. Consider testing the code against unfrozen dependencies first to update version numbers.
6. Use the GitHub Releases interface. to cut a new release from the main branch, with the selected name and number. This should create a new tag corresponding to the release commit.
7. Write a message to the BigBang development list announcing the new release, including the release notes.

8.2 README for BigBang Docs

To build the docs, go to the *docs/* directory and run

```
` make html `
```

The built docs will be deposited in *docs/_build*

CHAPTER

NINE

RELEASE NOTES

9.1 v0.3.0 Syzygy

Work on this release was supported by the Prototype Fund.

- Robust ListServ data ingress #460 #459 #457
- New ReadTheDocs/Sphinx based documentation: domain name and organization metadata #414 #499 #548
- New code submodule organization
- 3GPP analysis #465
- *datasets* submodule for ancillary data #509
- Integration of IETF datatracker source and analysis of IETF attendance data #368 #560 #434
- IETF draft analysis #370
- Tools to identify the institution of email senders #25
- Improved test coverage #343
- Change from nose to unittest for testing framework #366
- Updates and corrections to example notebooks #364
- Bug fixes #538 #553 #555 #390
- Preliminary work towards entity resolution #405

9.2 v0.3.0 Joie de vivre

This release converted the codebase to Python 3 and introduced the DataTracker and LISTSERV data sources, among with several new scientific notebooks and maintenance improvements.

- Converted to Python 3 (#347, #373, #382, #388)
- Installation improvements (#345, #410, #423)
- Tenure calculation (#355)
- Improved documentation (#351, #389, #450)
- Improved testing (#372, #443)
- W3C data source improvements (#344, #381)
- Organization entity resolution (#385)

- Integration of IETF DataTracker data (#386, #394, #444)
- Organization and affiliation analysis notebooks (#396)
- Code style pre-commit hooks (#403)
- LISTSERV data source (#409, #442, #454, #456)

9.3 0.2.0 Tulip Revolution

We have released BigBang v0.2.0 Tulip Revolution.

This release marks a new milestone in BigBang development.

- Gender participation estimation
- Improved support for IETF mailing list ingest
- Extensive gardening of the example notebooks
- Upgraded all notebooks to Jupyter 4
- Improved installation process based on user testing

En route to this milestone, the BigBang community made a number of changes to its procedures. These include:

- The adoption of a Governance document for guiding decision-making.
- The adoption of a Code of Conduct establishing norms of respectful behavior within the community.
- The creation of an ombudsteam for handling personal disputes.

We have also for this milestone adopted by community decision the GNU Affero General Public License v3.0.

9.4 0.1.0 Anteplanck I

An initial public release of BigBand. Proof of concept.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

I

L

S

T

U

W