pdm_utils

Release 0.9.17

Travis Mavrich

Jan 19, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

`pdm_utils` is a Python package designed in combination with a pre-defined MySQL database schema in order to facilitate the creation, management, and manipulation of phage genomics databases in the SEA-PHAGES program. The package is directly connected to the structure of the MySQL database, and it provides several types of functionality:

1. *Python library* including:
    a. Classes to store/parse phage genomes, interact with a local MySQL genomics database, and manage the process of making database changes.
    b. Functions and methods to manipulate those classes as well as interact with several databases and servers, including PhagesDB, GenBank, PECAAN, and MySQL.
2. A command line *toolkit* to process data and maintain a phage genomics database.

`pdm_utils` is useful for:

1. The Hatfull lab to maintain MySQL phage genomics databases in the SEA-PHAGES program (current pipeline).
2. Researchers to evaluate new genome annotations (*flat file QC*).
3. Researchers to directly access and retrieve phage genomics data from any compatible MySQL database (*tutorial*).
4. Researchers to create *custom* MySQL phage genomics databases.
5. Developers to build downstream data analysis tools (*tutorial*).

1.1 `pdm_utils` source code

This project is maintained using git and is available on GitHub.

CHAPTER

TWO

INSTALLATION

The `pdm_utils` package is written for Python >=3.7 and can be installed on MacOS and Linux platforms. There are several third-party general dependencies that need to be installed in advance, and there are several dependencies that are only needed for a subset of `pdm_utils` tools. Below is a general step- by-step installation guide.

2.1 Required

2.1.1 MySQL

MySQL is a database management system, and it is required for practically all `pdm_utils` tools. `pdm_utils` has been developed and tested using MySQL Community Server 5.7. The package may work with newer versions of MySQL, but it has not been tested. The MySQL user guide provides very thorough, detailed installation instructions, and there are many online tutorials to help overcome common installation challenges. Below is a brief summary of installation for MacOS and Ubuntu.

MacOS installation

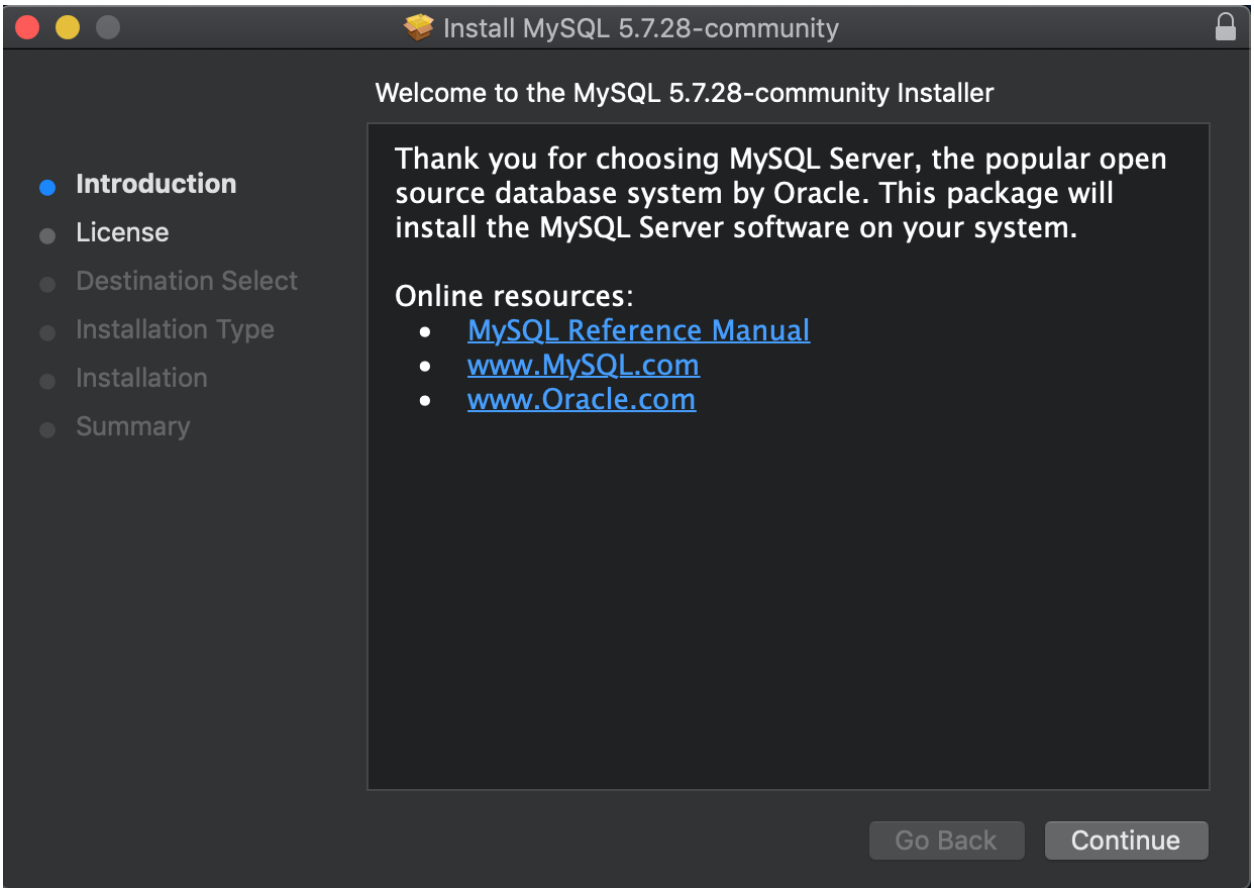
`pdm_utils` is tested on MySQL 5.7.27 on MacOS 10.14, although it is expected to be compatible with MySQL >=5.7, <8.0 and MacOS >=10.13.

Installation

1. Navigate to the MySQL Community Server download site.
2. Verify the 'General Availability (GA) Releases' tab is selected.
3. Verify 'macOS' is selected from the 'Select Operating System' menu.
4. By default, the most recent version of MySQL Community Server (version 8.0) is displayed for download. Select the 'Looking for previous GA versions?' option. Verify that version 5.7 is now displayed.
5. Download the 'macOS DMG Archive'.

> **Note:** Only the most recent version of 5.7 is available on this page. However, if you are using an older version of MacOS, there may be compatibility issues. If this is the case: select the 'Archives' tab, verify 'macOS' is selected from 'Operating System', and find the MySQL version from 'Product Version' that is compatible with your MacOS version as indicated further below on the page. Download the DMG archive.

6. Follow the graphical installation instructions:



7. Agree with the license agreement:

8. Install at the default location:

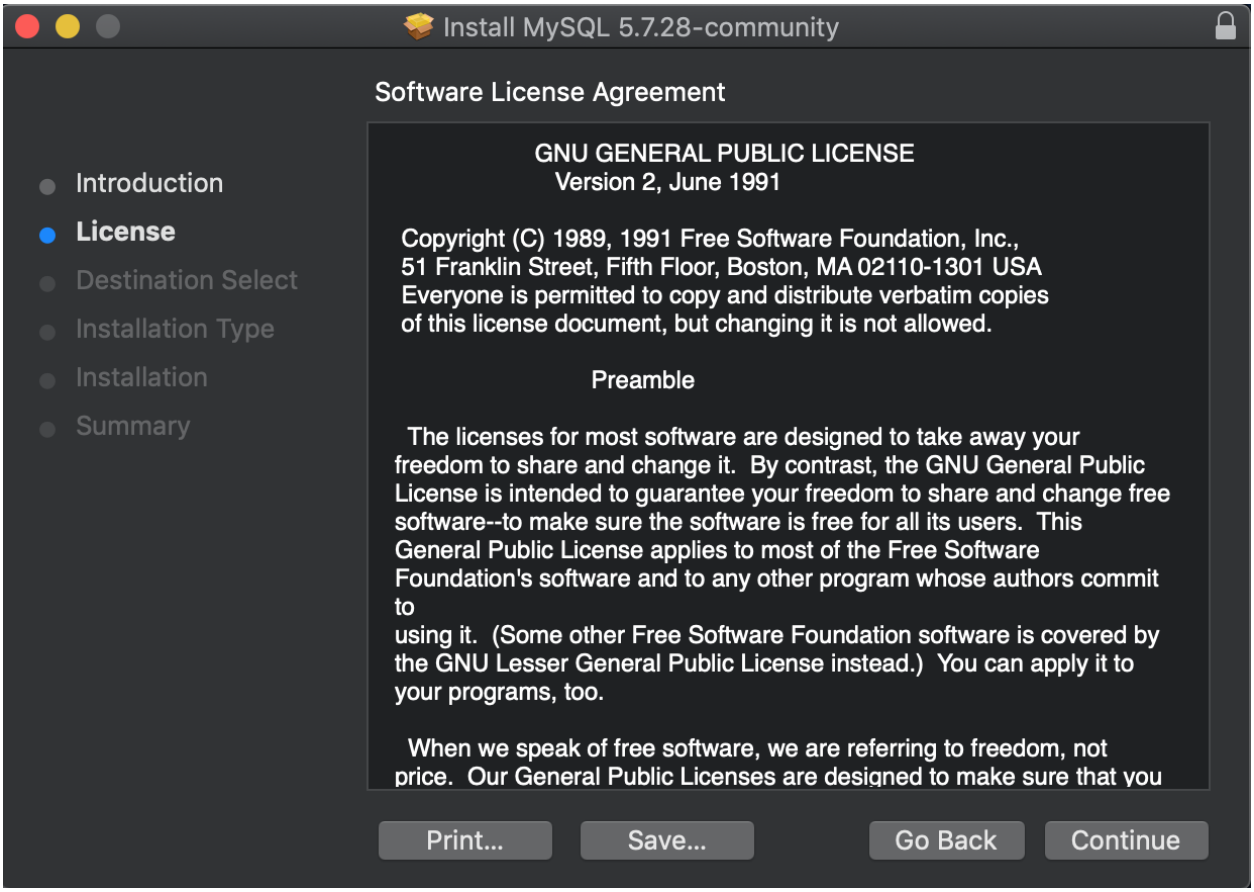
9. Allow access to 'System Events.app':

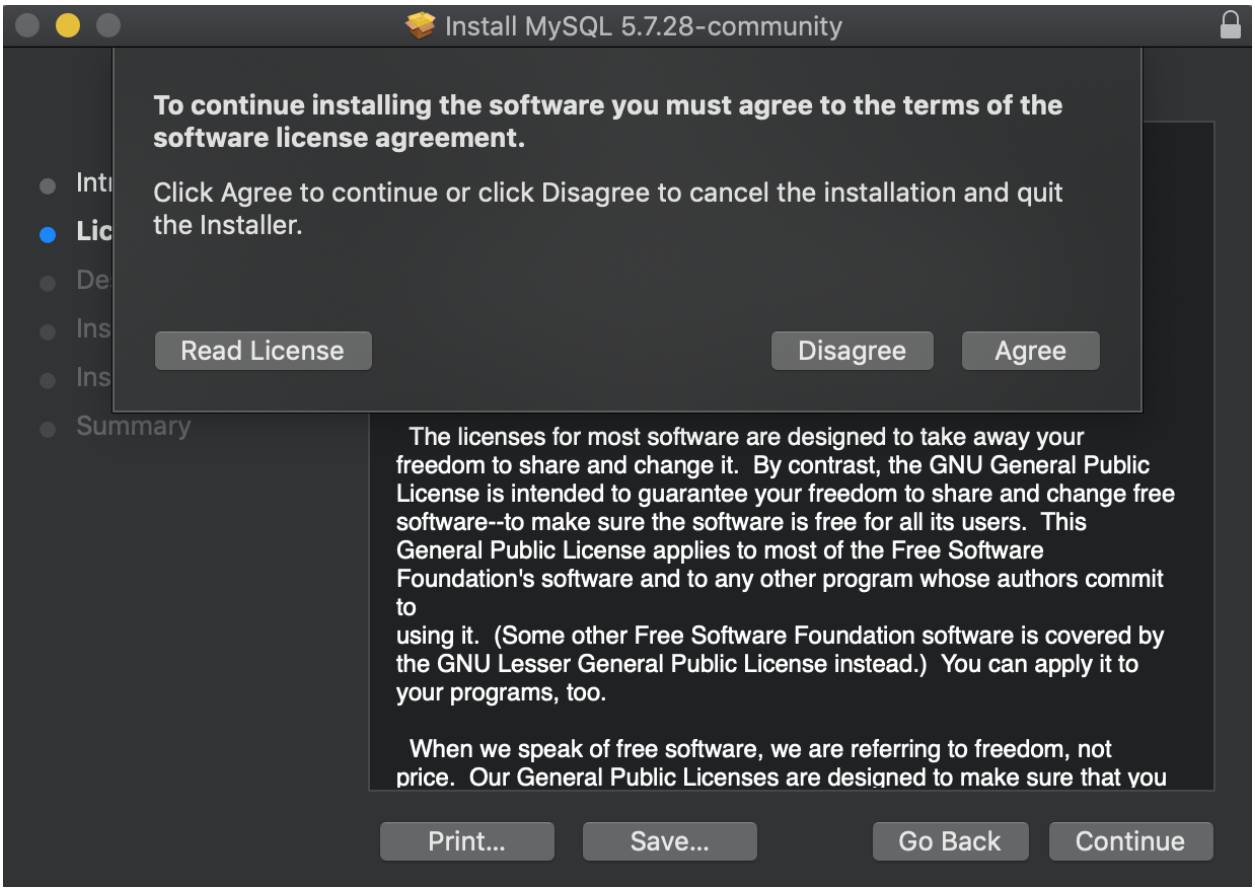
10. Make note of the temporary password provided during the installation process (highlighted in red):

> **Warning:** Be sure to record the temporary password that is generated!!! Once MySQL is installed, this password can only be used ONCE to login.

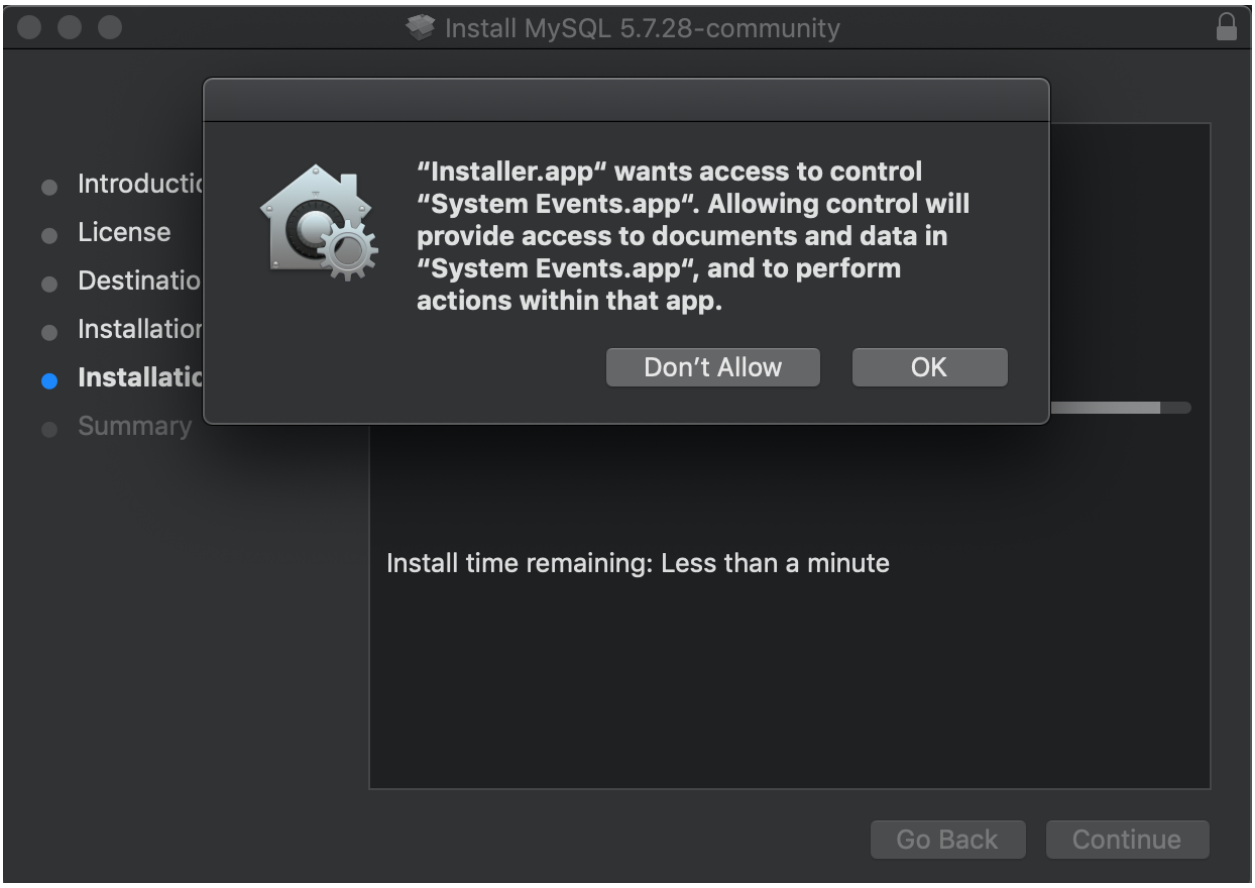
11. Installation should now be complete:

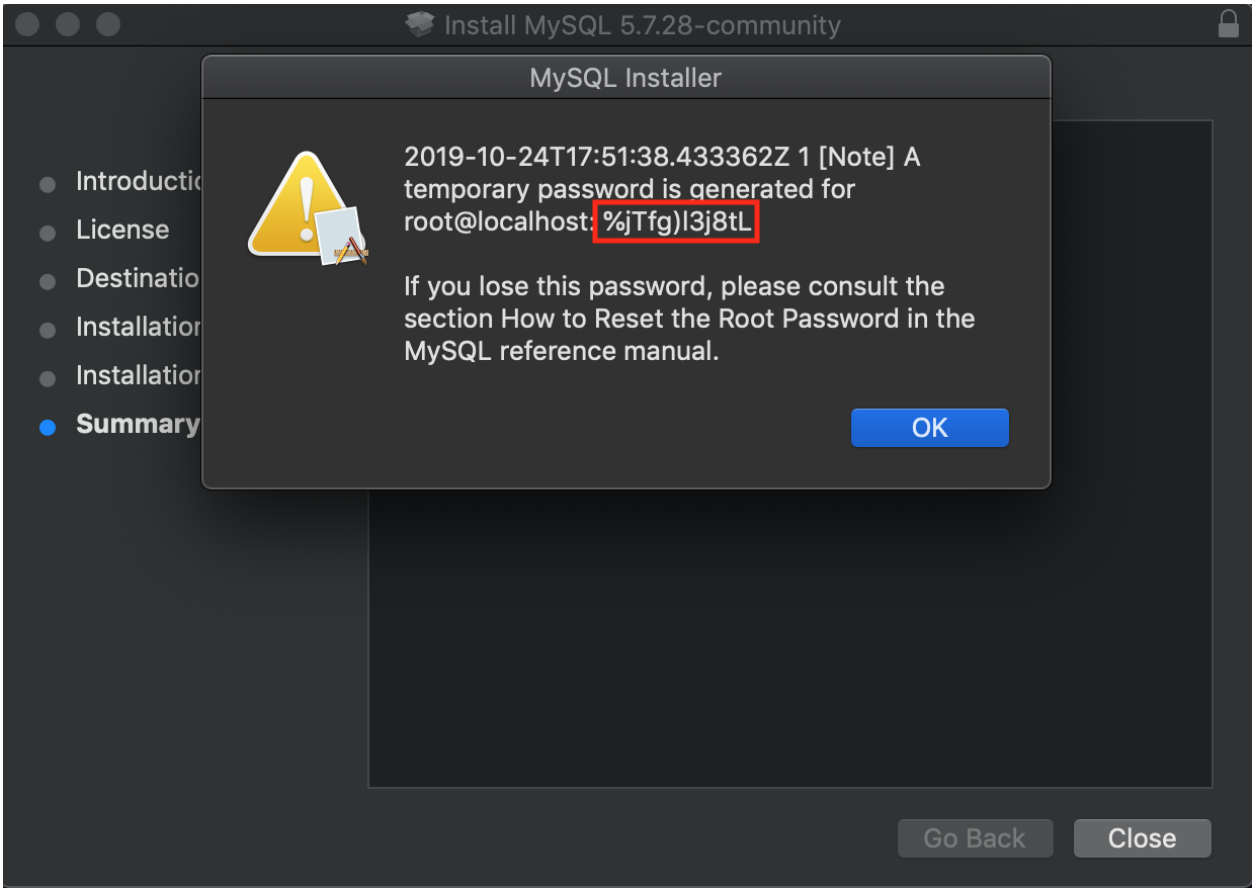
12. Open Terminal.
13. Log in to MySQL as the root user:

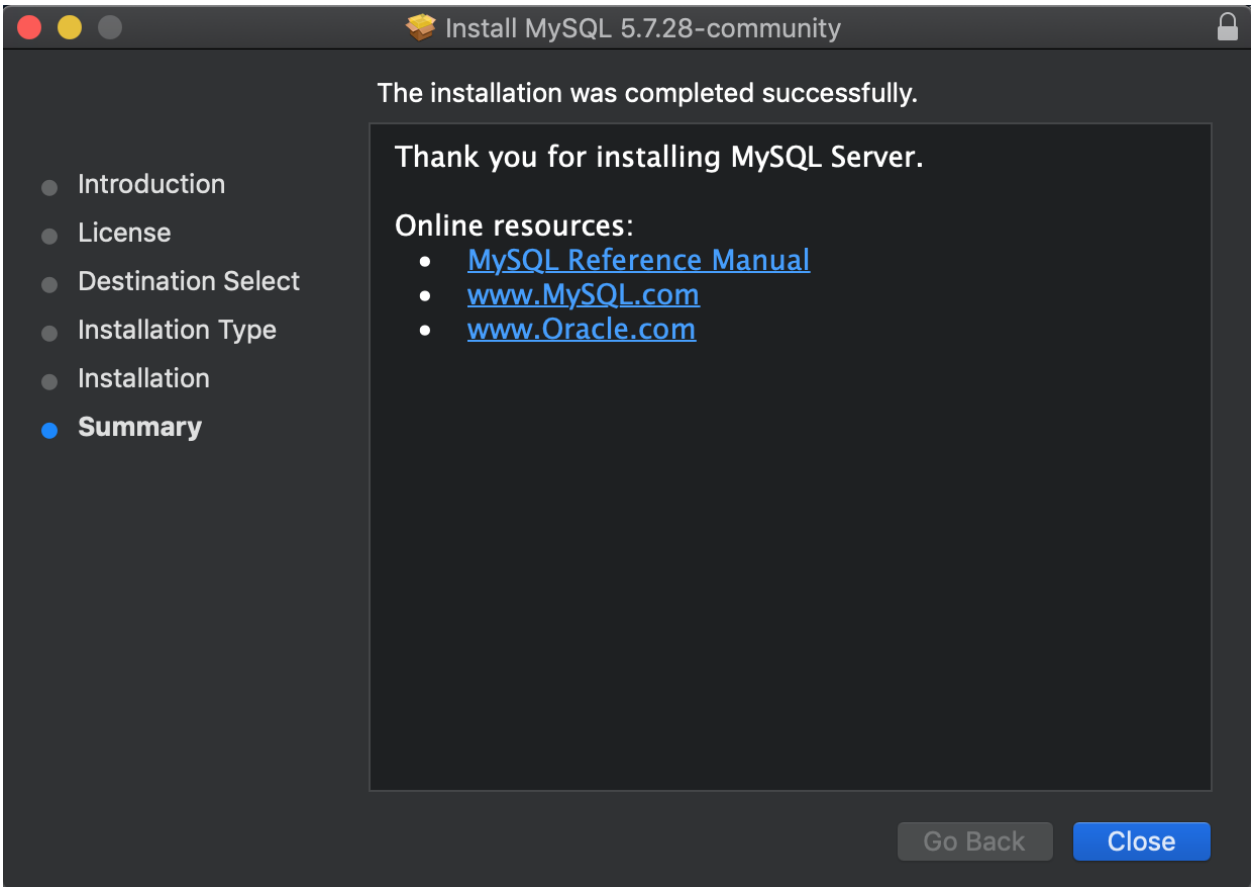

Install MySQL 5.7.28-community
Software License Agreement
Introduction
License
Destination Select
Installation Type
Installation
Summary
GNU GENERAL PUBLIC LICENSE
Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
Preamble
The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users. This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit
to
using it. (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.) You can apply it to
your programs, too.
When we speak of free software, we are referring to freedom, not
price. Our General Public Licenses are designed to make sure that you
Print...
Save...
Go Back
Continue


Install MySQL 5.7.28-community
To continue installing the software you must agree to the terms of the software license agreement.
Click Agree to continue or click Disagree to cancel the installation and quit the Installer.
Read License
Disagree
Agree
Summary
The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to
using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.
When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you
Print...
Save...
Go Back
Continue


Install MySQL 5.7.28-community
Standard Install on "Macintosh HD"
Introduction
License
Destination Select
Installation Type
Installation
Summary
This will take 1.32 GB of space on your computer.
Click Install to perform a standard installation of this software for all users of this computer. All users of this computer will be able to use this software.
Change Install Location...
Customize
Go Back
Install


Install MySQL 5.7.28-community
Introducti
License
Destinatio
Installatior
Installatic
Summary
"Installer.app" wants access to control "System Events.app". Allowing control will provide access to documents and data in "System Events.app", and to perform actions within that app.
Don't Allow
OK
Install time remaining: Less than a minute
Go Back
Continue


Install MySQL 5.7.28-community
MySQL Installer
2019-10-24T17:51:38.433362Z 1 [Note] A temporary password is generated for root@localhost: %jTfg)l3j8tL
If you lose this password, please consult the section How to Reset the Root Password in the MySQL reference manual.
OK
Introductio
License
Destinatio
Installatior
Installatior
Summary
Go Back
Close


Install MySQL 5.7.28-community
The installation was completed successfully.
Introduction
License
Destination Select
Installation Type
Installation
Summary
Thank you for installing MySQL Server.
Online resources:
MySQL Reference Manual
www.MySQL.com
www.Oracle.com
Go Back
Close

```
> mysql -u root -p
```

14. Enter the temporary password when prompted:

```
Enter password: <temporary password>
```

15. At the MySQL prompt, change the password for the root user:

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY '<new password>';
mysql> exit
```

Password reset

If the MySQL password is lost, it can be reset.

1. In Mac Preferences Pane, turn off MySQL server.
2. Open a Terminal window.
3. Enter the following command:

```
> sudo /usr/local/mysql/bin/mysqld_safe --skip-grant-tables
```

4. Enter the password for the computer user (not MySQL), at the prompt:

```
Enter password: <user password>
```

5. Open a SECOND Terminal window.
6. Enter the following command:

```
> sudo /usr/local/mysql/bin/mysql -u root
```

7. Enter the password for the computer user (not MySQL), at the prompt:

```
Enter password: <user password>
```

8. You should now be logged in to mysql. Execute the following commands:

```
mysql> UPDATE mysql.user SET authentication_string=PASSWORD('<new password>') WHERE␣
↪User='root';
mysql> FLUSH PRIVILEGES;
mysql> exit
```

9. You should now be returned to the bash command line. Enter the following command:

```
> sudo /usr/local/mysql/support-files/mysql.server restart
```

10. Close the second Terminal window.
11. Close the first Terminal window.

Server control

Access to MySQL, even on your local computer, is controlled through a server-client model. The server needs to be turned ON in order to use MySQL. This can be accomplished manually or it can be set to start automatically every time your Mac is restarted.

1. Click on the Apple icon in the top left corner of your desktop.
2. Select 'System Preferences'.
3. Click on the MySQL icon.
4. If 'MySQL Server Instance is stopped' is displayed, then click on 'Start MySQL Server'.
5. To perform this step every time automatically, select 'Automatically Start MySQL Server on Startup'.

If the automatic option is not selected, anytime your Mac is restarted the server is turned OFF, and you will be unable to use any `pdm_utils` tools that require access to MySQL until you manually turn the server ON.

Ubuntu installation

`pdm_utils` is tested on MySQL 5.7.29 on Ubuntu 18.04.3, although it is expected to be compatible with MySQL >=5.7, <8.0 and Ubuntu >=16. MySQL 5.7 can be downloaded through either the Ubuntu repositories or the official MySQL repositories. Installing MySQL using the Ubuntu repositories is outlined below:

1. Open a Terminal window.
2. Update all available repositories (provide the computer login password when prompted):

```
> sudo apt update
```

3. Enter the following command to install the MySQL version 5.7 (answer 'yes' to proceed with installing the new packages, when prompted):

```
> sudo apt install mysql-server=5.7.*
```

4. MySQL Community Server should now be installed, but the server may not be running.
   - Check the server status:
     A. Enter the following command:

```
> systemctl status mysql.service
```

     B. If the server is running, it should display:

```
Active: active (running))
```

     C. If the server is not running, it should display:

```
Active: inactive (dead)
```

   - If the server is not running, it needs to be started:

```
> sudo systemctl start mysql
```

   - Check status again to confirm it is running:

```
> systemctl status mysql.service
```

5. Although MySQL is installed, no password has yet been set for the 'root' user. Login to MySQL without a username (provide the computer login password if prompted):

```
> sudo mysql
mysql>
```

6. Now set a password for the 'root' user:

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY '<new␣
↪password>';
mysql> FLUSH PRIVILEGES;
mysql> exit;
```

Create additional users (optional)

After MySQL is installed (on MacOS or Ubuntu), additional user accounts with different types of access privileges can be created, if needed.

1. Login to mysql as 'root' (provide the password when prompted):

```
> mysql -u root -p
mysql>
```

2. Create a new user 'new_user', and specify the password:

```
mysql> CREATE USER 'new_user'@'localhost' IDENTIFIED BY '<new_password>';
```

3. Grant different levels of access using one of the following commands:
   - Grant unrestricted access to all databases:

```
mysql> GRANT ALL ON *.* TO 'new_user'@'localhost' WITH GRANT OPTION;
```

   - Grant access with all privileges to a specific database (such as Actino_Draft):

```
mysql> GRANT ALL ON Actino_Draft.* TO 'new_user'@'localhost';
```

   - Grant access to all databases, but only with the privilege to retrieve data:

```
mysql> GRANT SELECT ON *.* TO 'new_user'@'localhost';
```

4. Implement the changes:

```
mysql> FLUSH PRIVILEGES;
mysql> exit;
```

2.1.2 Anaconda

`pdm_utils` requires Python >=3.6 and several third-party Python packages:

- Biopython
- NetworkX
- Paramiko
- PyMySQL
- PyYAML
- SQLAlchemy
- Tabulate

Note: The most recent version of Biopython introduces some problems for the current pdm_utils. We are working on fixing it, but in the meantime, we have edited the conda environment creation command to install the latest non-code-breaking version of Biopython.

Some of them also have Python or non-Python dependencies. Manual installation of these dependencies can be tricky, but the Conda environment manager is a simple, automated alternative. First install Conda, then use Conda to install Python and other dependencies.

Conda is available as part of Anaconda or Miniconda, and complete installation instructions are available in the Conda user guide. `pdm_utils` has been developed and tested using Conda >=4.5. The directions below highlight installation of Anaconda, but either of these tools is fine since they both install Conda:

1. Install Conda locally through the Anaconda package.
2. Navigate to the 'Anaconda Distribution' option.
3. Begin the installation:
    - For MacOS: download the Python 3.8 graphical installer and follow the graphical prompts.
    - For Linux:
        A. Download the Python 3.8 x86 Linux installer (e.g. Anaconda3-2019.10-Linux-x86_64.sh) to the Downloads folder.
        B. Open a Terminal window.
        C. Execute the following command:

```
> bash ~/Downloads/Anaconda3-2019.10-Linux-x86_64.sh
```

4. Follow the manufacturer's installation instructions.
    - Accept the license agreement.
    - Install at the default directory.
    - Enter 'yes' when prompted for the installer to run conda init.
5. Optional: execute the following command to prevent Conda from starting up automatically every time a Terminal window is opened:

```
> conda config --set auto_activate_base false
```

6. Close the Terminal window and open a new window.

7. After installing Conda, create an environment to be able to install and use `pdm_utils` (the example below creates a Conda environment named 'pdm_utils', but it can be named anything). Enter 'y' when prompted to install all dependencies:

```
> conda create --name pdm_utils curl python pip biopython==1.77 networkx paramiko␣
↪pymysql sqlalchemy tabulate urllib3
```

8. After the Conda environment is created, it needs to be activated using the following command. The command line prompt will now include '(pdm_utils)', indicating it is operating within this environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

9. Optional: enter the following command to exit the Conda environment. The default command line prompt will be displayed, and the name of the Conda environment will no longer be displayed:

```
(pdm_utils)> conda deactivate
>
```

Note: If Conda is used to manage dependencies, the Conda environment must be activated every time you want to use `pdm_utils`. Otherwise, an error will be encountered.

The 'pdm_utils' Conda environment now contains several required dependencies, and the actual `pdm_utils` Python package can be (*installed*).

2.1.3 The `pdm_utils` package

Once a Conda environment is created, `pdm_utils` can be easily installed:

1. Open a Terminal window.
2. Activate the (*Conda environment*).
3. Install the `pdm_utils` package using pip:

```
(pdm_utils)> pip install pdm_utils
```

4. The package is routinely updated, and the most recent version can be retrieved:

```
(pdm_utils)> pip install --upgrade pdm_utils
```

2.1.4 MySQL database instance

Many `pdm_utils` modules and pipelines require access to a specifically structured MySQL database.

The primary database instance that reflects the most up-to-date actinobacteriophage genomics data in the SEA-PHAGES program is the 'Actino_Draft' database. Typically, different versions, or instances, of the database are created ('frozen') for specific studies/publications. The unique name of the database is normally published in the Materials and Methods.

The `pdm_utils get_db` installation management tool can be used to retrieve, install, and update these databases, or any custom MySQL database that is compliant with the database schema, from a local file or from the Hatfull lab server (*get_db*).

Alternatively, databases can be manually downloaded and installed, as described below (using Actino_Draft as an example):

pdm_utils installation

1. Open a Terminal window.
2. Start the Conda environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

3. Use the pdm_utils package to retrieve the Actino_Draft database (enter your username and password when prompted):

```
(pdm_utils)> python3 -m pdm_utils get_db server -u http://databases.hatfull.org/ -
↪db Actino_Draft -v
```

The following software should be installed in the order indicated:

1. *MySQL*
2. *Anaconda*
3. *pdm_utils Python package*
4. *MySQL phage genomics database instance*

2.2 Optional

2.2.1 ARAGORN

ARAGORN (*Laslett & Canback, 2004*) is used to evaluate tRNA features when importing data into the database.

MacOS and Ubuntu installation

1. Open a Terminal window and start the Conda environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

2. The most straightforward option is to use Conda to install Aragorn:

```
(pdm_utils)> conda install -c bioconda aragorn -y
```

3. Test whether ARAGORN has been successfully installed:

```
(pdm_utils)> aragorn -h
```

If successful, a detailed description of the software's options should be displayed.

If unsuccessful, an error message should be displayed, such as:

```
-bash: aragorn: command not found
```

2.2.2 Infernal

Infernal (*Nawrocki & Eddy, 2013*) is required to properly run tRNAscan-SE.

MacOS and Ubuntu installation

1. Open a Terminal window and start the Conda environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

2. The most straightforward option is to use Conda to install Infernal:

```
(pdm_utils)> conda install -c bioconda infernal -y
```

3. Test whether Infernal has been successfully installed:

```
(pdm_utils)> cmsearch -h
```

If successful, a detailed description of the software's options should be displayed.

If unsuccessful, an error message should be displayed, such as:

```
-bash: cmsearch: command not found
```

2.2.3 tRNAScan-SE v2.0

tRNAScan-SE v2.0 (*Chan & Lowe, 2019*) is used to evaluate tRNA features when importing data into the database.

MacOS and Ubuntu installation

1. Open a Terminal window and start the Conda environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

2. The most straightforward option is to use Conda to install tRNAscan-SE:

```
(pdm_utils)> conda install -c bioconda trnascan-se -y
```

3. Test whether tRNAscan-SE has been successfully installed:

```
(pdm_utils)> tRNAscan-SE -h
```

If successful, a detailed description of the software's options should be displayed.

If unsuccessful, an error message should be displayed, such as:

```
-bash: tRNAscan-SE: command not found
```

2.2.4 MMseqs2

MMseqs2 is a suite of tools for rapid searching and clustering of protein and nucleotide sequences (*Steinegger and Söding, 2017*). This software is required only if gene phamilies need to be identified using MMseqs2 in the `phamerate` pipeline. Complete, detailed installation instructions are provided by the developers on the project's GitHub page. The following instructions provide an example of how it can be installed on your local MacOS or Ubuntu machine.

MacOS and Ubuntu installation

1. Open a Terminal window and start the Conda environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

2. The most straightforward option is to use Conda to install MMseqs2:

```
(pdm_utils)> conda install -c bioconda -c conda-forge mmseqs2=13.45111 -y
```

3. Test whether MMseqs2 has been successfully installed:

```
(pdm_utils)> mmseqs cluster -h
```

If successful, a detailed description of the software's options should be displayed.

If unsuccessful, an error message should be displayed, such as:

```
-bash: mmseqs: command not found
```

2.2.5 NCBI Tools

NCBI BLAST+

The BLAST+ software package is a stand-alone version of BLAST applications and is maintained by the NCBI (*Camacho et al., 2009*). This software is required in the `find_domains` pipeline to identify conserved domains within genes. Follow the installation instructions specific to your operating system provided in the BLAST+ user guide.

MacOS and Ubuntu installation

1. Open a Terminal window and start the Conda environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

2. The most straightforward option is to use Conda to install blast:

```
(pdm_utils)> conda install -c bioconda blast -y
```

3. Test whether blast has been successfully installed:

```
(pdm_utils)> blastp -help
```

If successful, a detailed description of the software's options should be displayed.

If unsuccessful, an error message should be displayed, such as:

```
-bash: blastp: command not found
```

NCBI Conserved Domain Database

The NCBI Conserved Domain Database is a curated set of protein domain families (*Lu et al., 2020*). This database is required in the `find_domains` pipeline to identify conserved domains within genes.

1. Download the compressed NCBI CDD.
2. Expand the archived file into a directory of CDD files. The entire directory represents the database and no files should be added or removed.

2.2.6 Markov Clustering

Markov Clustering (MCL) is a graph clustering program (*van Dongen & Abreu-Goodger, 2012*) that can be used in combination a pairwise blastp output to infer phamilies of homologous proteins.

MacOS and Ubuntu installation

1. Open a Terminal window and start the Conda environment:

```
> conda activate pdm_utils
(pdm_utils)>
```

2. The most straightforward option is to use Conda to install MCL:

```
(pdm_utils)> conda install -c bioconda mcl -y
```

3. Test whether MCL has been successfully installed:

```
(pdm_utils)> mcl --help
```

If successful, a detailed description of the software's options should be displayed.

If unsuccessful, an error message should be displayed, such as:

```
-bash: mcl: command not found
```

2.2.7 pdm_utils source code repository

Some `pdm_utils` tools may require non-Python data files that are not directly installed with the Python package. Instead, these files are available on the `pdm_utils` git repository, which can be accessed through GitHub and downloaded as follows:

1. Open a Terminal window.
2. Navigate to a directory where the source code subdirectory should be installed (for example, your home directory):

```
> cd
```

3. Enter the following command:

```
> git clone https://github.com/SEA-PHAGES/pdm_utils.git
```

There should now be a `pdm_utils` directory installed locally.

Several `pdm_utils` tools have specific dependencies. Install the following software as needed:

- *ARAGORN (importing tRNA and tmRNA data)*
- *Infernal (importing tRNA and tmRNA data)*
- *tRNAScan-SE (importing tRNA and tmRNA data)*
- *MMseqs2 (creating gene phamilies)*
- *NCBI Blast+ (finding conserved domains)*
- *NCBI Conserved Domain Database (finding conserved domains)*
- *Markov Clustering (creating gene phamilies)*
- *pdm_utils repository (source code, MySQL scripts, tests)*

CHAPTER

THREE

THE PDM_UTILS MYSQL DATABASE

Since the `pdm_utils` Python package is designed to manage a MySQL database structured specifically for phage genomics, the Python package and the database structure are developed in a coordinated fashion. Below is a description of the current database structure (schema), a log of schema changes, and Entity Relationship diagrams for each prior schema version.

3.1 Database structure

The database (schema version 10) contains the following tables:

3.1.1 domain

This table stores information about NCBI-defined conserved domains relevant to CDS features within the database.

Field	Data origin
ID	MySQL
HitID	find_domains
Description	find_domains
DomainID	find_domains
Name	find_domains

ID Auto-incrementing values. This is the primary key.

HitID Identifier to match conserved domain data in this table to location of conserved domain in the gene, stored in the *gene_domain* table.

Description Description of the conserved domain.

DomainID Conserved domain identifier in CDD.

Name Conserved domain name in CDD.

3.1.2 gene

This table contains information that pertains to individual genes, including coordinates, orientation, the gene product, etc.

Field	Data origin (GenBank field or pipeline)
GeneID	import
PhageID	import
Start	LOCATION
Stop	LOCATION
Length	import
Name	MULTIPLE FIELDS
Translation	TRANSLATION
Orientation	STRAND
Notes	PRODUCT, FUNCTION, or NOTE
DomainStatus	find_domains
LocusTag	LOCUS_TAG
Parts	PARTS
PhamID	phamerate

GeneID Unique identifier for the gene feature in the entire database. It is distinct from other common gene identifiers in a flat file such as LOCUS_TAG or GENE.

Name This field is an identifier for the annotation but does not need to be unique, analogous to the distinction between the PhageID and Name fields in the *phage* table. Most of the time (but not always), it is an integer, but is ocassionally a float, an alphanumeric string, or a strictly alphabetic. This field is displayed on Phamerator genome maps. This can be derived from several locations in the flat file: the LOCUS_TAG, GENE, PRODUCT, NOTE, and FUNCTION qualifiers. If no gene identifier is present in any of these qualifiers, this field remains empty.

PhageID The name of the phage genome from which the gene is derived, matching one of the phage names in the PhageID of the *phage* table.

Start, Stop These fields store the genomic coordinates marking the coordinate boundaries of the gene. The coordinates are stored in '0-based half-open' format (as opposed to the '1-based closed' format used in other representations, such as a GenBank-formatted flat file). For practical purposes, the start coordinate has been decreased by 1 nucleotide. Start and Stop reflect the left and right (respectively) boundaries of the feature based on the genome orientation stored in the database. They do not directly reflect the translational start and stop coordinates of the feature, which are dependent on orientation. Since only two coordinates are stored for each feature, compound features spanning more than one contiguous region of the genome (such as features that wrap-around genome termini or features with a translational frameshift) are not completely represented in the database.

Orientation This field indicates the strand in which the feature is encoded.

Parts This field indicates the number of regions in the genome that define the feature. Only two coordinates are stored for each feature, which is an accurate representation of the majority of features. However, the definition of some features, such as those that extend across the genome termini or those that contain a frameshift, are not completely represented with this strategy. This field is used to discriminate between these types of features.

Length This field indicates the length of the transcribed and translated nucleotide sequence and is directly proportional to Translation. If the feature is comprised of a single ORF, this is equal to Stop - Start. If the feature is comprised of two ORFs (such as due to ribosome frameshifting), Length will be larger than Stop - Start.

Translation This field contains the translated amino acid sequence and is derived directly from the GenBank record. Note: currently, the maximum length of the translation product is 5,000 amino acids.

LocusTag This field facilitates automatic updating of GenBank records. Once a genome has been submitted to GenBank, genes are assigned unique locus tags in the LOCUS_TAG field. These identifiers cannot be changed, and anno-

tators are required to use them when requesting to update details about individual genes. This field provides a direct link to the corresponding GenBank feature. Note: this field is only populated for records retrieved from GenBank.

Notes This field contains data on the gene function, and is derived from one of several fields of the GenBank feature.

DomainStatus Indicates whether the `find_domains` pipeline has searched for conserved domain data for this feature (0 or 1). When new phage genomes are added to the *gene* table, the DomainStatus field for each new gene is set to 0. The `find_domains` pipeline updates this to 1 after searching for conserved domains (regardless of whether the feature contains any conserved domains).

PhamID Pham designation for the translation, matching one of the PhamIDs in the *pham* table.

3.1.3 gene_domain

This table stores the positions of NCBI-defined conserved domains within each CDS feature in the *gene* table.

Field	Data origin
ID	MySQL
GeneID	find_domains
HitID	find_domains
QueryStart	find_domains
QueryEnd	find_domains
Expect	find_domains

ID Auto-incrementing values. This is the primary key.

GeneID Unique gene identifier matching GeneID in the *gene* table.

HitID Identifier to match location of conserved domain in this table to conserved domain data, stored in the *domain* table.

QueryStart First amino acid position within the conserved domain.

QueryEnd Last amino acid position within the conserved domain.

Expect E-value reflecting significance of the domain hit.

3.1.4 phage

This table contains information that pertains to the entire phage genome, such as the genome sequence, the host strain, the designated cluster, etc.

Column	Data origin (GenBank field or pipeline)
PhageID	ticket
Accession	ticket
Name	ticket
HostGenus	ticket
Sequence	SEQUENCE
Length	import
DateLastModified	import
Notes	update
GC	import
Status	ticket
RetrieveRecord	ticket
AnnotationAuthor	ticket
Cluster	ticket
Subcluster	ticket

PhageID This field is the primary key of the *phage* table and is the unique identifier for all phages in the database. There is a direct correspondence between phage names in PhagesDB or phage names in GenBank records to PhageIDs in the Actino_Draft database (although there are a few exceptions, due to naming restrictions in different external databases).

Name This field also reflects the phage name, but it is not as constrained as the PhageID, and does not have to be unique. For all 'draft' genomes, the Name contains the PhageID with a '_Draft' suffix appended, indicating the annotations have been automatically annotated. For all other genomes, the Name corresponds to the PhageID. In some downstream applications, such as Phamerator, this serves as the phage's display name.

Accession This field is populated and updated directly from import tickets and is used for auto-updating genomes from GenBank records. It is important to note that the NCBI generates RefSeq records that are derived from GenBank records. After data is submitted to GenBank, authors retain control of the GenBank record but not the RefSeq record. As a result, this field should always store the GenBank ACCESSION number (and not the RefSeq ACCESSION number) for SEA-PHAGES genomes. For non-SEA-PHAGES genomes, either accession number may be stored. In either case, the Accession should not contain the sequence version (represented by the integer to the right of the decimal).

HostGenus This field indicates the host genus (e.g. *Mycobacterium*, *Streptomyces*, etc.) from which the phage was isolated.

Sequence This genome nucleotide sequence of the phage.

Length The length of the phage's genome sequence.

GC The GC% of the genome sequence.

Cluster This field indicates the phage's cluster designation if it has been clustered. If the phage is a singleton, it remains empty (NULL).

Subcluster This field indicates the phage's subcluster designation if it has been subclustered, otherwise it remains empty (NULL).

DateLastModified This field records the date in which a genome and its annotations have been imported. This keeps track of which annotation version has been imported, and it facilitates automated updating of the database. It is important to note that the date stored in this field reflects the date the annotation data were imported, and not the date that the annotation data were created. Although the field is a DATETIME data type, only date data is stored, and no time data is retained.

AnnotationAuthor This field indicates if the genome sequence and annotations are (1) or are not (0) maintained by the SEA-PHAGES program, and it facilitates automatic updates from GenBank. If a genome has been sequenced and annotated through the SEA-PHAGES program, its GenBank record is actively updated/maintained. In this case, "Graham Hatfull" is expected to be a listed author in the GenBank record. (All genomes through the SEA-PHAGES program should have "Graham Hatfull" as a listed author, but not all GenBank records listing "Graham Hatfull" as an author are derived from the SEA-PHAGES program.)

RetrieveRecord This field will be 0 or 1, and it facilitates automatic updates from GenBank records . Most SEA-PHAGES genomes are expected to be automatically updated from GenBank once they are assigned a unique GenBank accession. However, some genomes, such as those generated from non-SEA-PHAGES researchers, may not need to be automatically updated. This field is set to 1 for genomes that are to be automatically updated and set to 0 for those genomes that are not to be automatically updated. Initially, this field is indirectly determined by the AnnotationAuthor field. For newly added genomes, if AnnotationAuthor = 1 in the import ticket, this field is set to 1, otherwise it is set to 0. For genomes being replaced (by automatic updates from GenBank or by the creation of manual tickets), the value in this field is retained.

Status This field indicates whether the gene annotations have automatically (draft) or manually (final) annotated, or whether the annotation strategy is unknown (unknown).

3.1.5 pham

This table contains a list of color codes for each unique pham.

Field	Data origin
PhamID	phamerate
Color	phamerate

PhamID The primary key of the table. Unique identifier for each pham.

Color The hexrgb color code reflecting unique phams, which is used by downstream applications such as Phamerator. The script attempts to maintain consistency of pham designations and colors between rounds of clustering.

3.1.6 trna

This table contains information that pertains to individual tRNA features similar to the gene table.

Field	Data origin (GenBank field or pipeline)
GeneID	import
PhageID	import
Start	LOCATION
Stop	LOCATION
Length	import
Name	MULTIPLE FIELDS
Orientation	STRAND
Note	NOTE
LocusTag	LOCUS_TAG
AminoAcid	PRODUCT
Anticodon	PRODUCT
Structure	import
Source	import

GeneID Same usage as in the *gene* table.

Name Same usage as in the *gene* table.

PhageID Same usage as in the *gene* table.

Start, Stop Same usage as in the *gene* table.

Orientation Same usage as in the *gene* table.

Length This field indicates the length of the nucleotide sequence.

LocusTag Same usage as in the *gene* table.

Note This field contains data from the NOTE qualifier.

AminoAcid The feature's annotated amino acid.

Anticodon The feature's annotated anti-codon.

Structure The predicted secondary structure.

Source Indicates whether ARAGORN and/or tRNAScan-SE identified this feature.

3.1.7 tmrna

This table contains information that pertains to individual tmRNA features similar to the gene table.

Field	Data origin (GenBank field or pipeline)
GeneID	import
PhageID	import
Start	LOCATION
Stop	LOCATION
Length	import
Name	MULTIPLE FIELDS
Orientation	STRAND
Note	NOTE
LocusTag	LOCUS_TAG
PeptideTag	NOTE

GeneID Same usage as in the *gene* table.

Name Same usage as in the *gene* table.

PhageID Same usage as in the *gene* table.

Start, Stop Same usage as in the *gene* table.

Orientation Same usage as in the *gene* table.

Length This field indicates the length of the nucleotide sequence.

LocusTag Same usage as in the *gene* table.

Note This field contains data from the NOTE qualifier.

PeptideTag The feature's annotated peptide tag.

3.1.8 version

This table keeps track of the database version and is updated every time the database is changed.

Field	Data origin
Version	update
SchemaVersion	update

Version This field reflects the current version of the database. Every time changes are made to the database, this integer is incremented by 1.

SchemaVersion This field indicates the current version of the database structure (schema) and enhances version control of downstream tools that utilize the database. As the structure of the database changes, such as by the addition or removal of tables or fields, the database schema number can be incremented to reflect that changes have been made. This does not occur often, and needs to be manually changed.

3.2 MySQL database schema changelog

Below is a history of schema changes. For maps of each schema version: *schema maps*.

3.2.1 Schema version 10

Single column datatype changed.

Column modified

- gene.Translation datatype = blob

3.2.2 Schema version 9

Substantial restructuring of trna and tmrna tables.

Column moved

- trna_structures.Structure to trna.Structure

Column removed

- trna.Sequence
- trna.Product
- trna.InfernalScore

Column renamed

- trna.TrnaID TO trna.GeneID
- tmrna.TmrnaID TO tmrna.GeneID

Column modified

- trna.Structure datatype
- trna.AminoAcid datatype
- trna.PhageID position in table and foreign key constraint

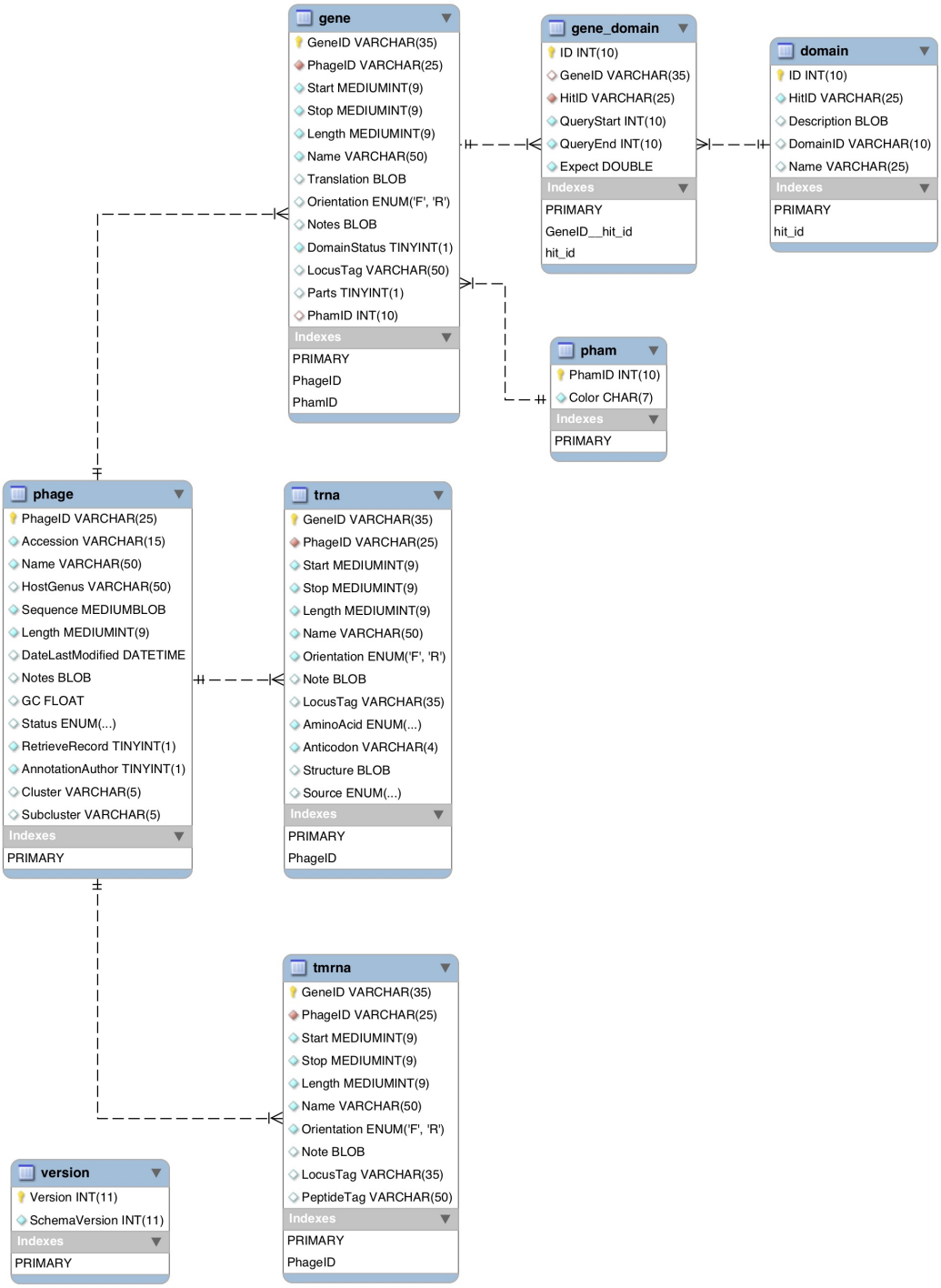


Fig. 1: Schema Entity Relationship Diagram.

- trna.LocusTag position in table
- tmrna.PhageID position in table and foreign key constraint
- tmrna.LocusTag position in table

Column added

- trna.Source
- trna.Name
- tmrna.Length
- tmrna.Name

Table removed

- trna_structures

3.2.3 Schema version 8

Modified foreign key constraint.

Column modified

- gene.PhamID foreign key constraint

3.2.4 Schema version 7

Renamed several columns. Dropped the pham table and renamed the pham_color table.

Column removed

- phage.Cluster
- pham_color.ID (primary key)
- gene.ID

Column renamed

- phage.HostStrain TO phage.HostGenus
- phage.Cluster2 TO phage.Cluster
- phage.Subcluster2 TO phage.Subcluster
- phage.SequenceLength TO phage.Length
- pham_color.Name TO pham_color.PhamID

Column modified

- pham_color.PhamID now primary key

Column added

- gene.Parts
- gene.PhamID

Table removed

- pham

Table renamed

- pham_color TO pham

3.2.5 Schema version 6

Standardized column nomenclature.

Column renamed

- domain.id TO domain.ID
- domain.description TO domain.Description
- gene_domain.id TO gene_domain.ID
- gene_domain.expect TO gene_domain.Expect
- phage.status TO phage.Status
- pham.name TO pham.Name
- pham_color.id TO pham_color.ID
- pham_color.name TO pham_color.Name
- pham_color.color TO pham_color.Color
- version.version TO version.Version
- gene.translation TO gene.Translation
- gene.id TO gene.ID
- gene.cdd_status TO gene.DomainStatus
- version.schema_version TO version.SchemaVersion
- domain.hit_id TO domain.HitID
- gene_domain.hit_id TO gene_domain.HitID
- gene_domain.query_start TO gene_domain.QueryStart
- gene_domain.query_end TO gene_domain.QueryEnd

3.2.6 Schema version 5

Removed several tables and columns.

Table removed

- node
- host_range
- host
- pham_history
- pham_old
- scores_summary

Column removed

- phage.Prophage
- phage.Isolated

- phage.ProphageOffset
- phage.DateLastSearched
- phage.AnnotationQC
- gene.StartCodon
- gene.StopCodon
- gene.GC1
- gene.GC2
- gene.GC3
- gene.GC
- gene.LeftNeighbor
- gene.RightNeighbor
- gene.clustalw_status
- gene.blast_status
- gene.TypeID
- pham.orderAdded

Column modified

- phage.status datatype = enum('unknown','draft','final')

3.2.7 Schema version 4

Added several tables.

Table created

- tmrna
- trna
- trna_structures

Column modified

- gene.translation datatype = VARCHAR(5000)

3.2.8 Schema version 3

Added/removed several columns.

Column created

- gene.LocusTag
- version.schema_version
- phage.Subcluster2
- phage.Cluster2

Column removed

- phage.Program

3.2.9 Schema version 2

Added several columns.

Column created

- phage.AnnotationAuthor
- phage.Program
- phage.AnnotationQC
- phage.RetrieveRecord

3.2.10 Schema version 1

Misc changes to maintain referential integrity.

Table created

- version

Column created

- gene.cdd_status

CASCADE setting updated

- gene.PhageID
- gene_domain.GeneID
- pham.GeneID
- scores_summary.query
- scores_summary.subject

3.2.11 Schema version 0

The base schema.

3.3 Database schema Entity Relationship Diagrams

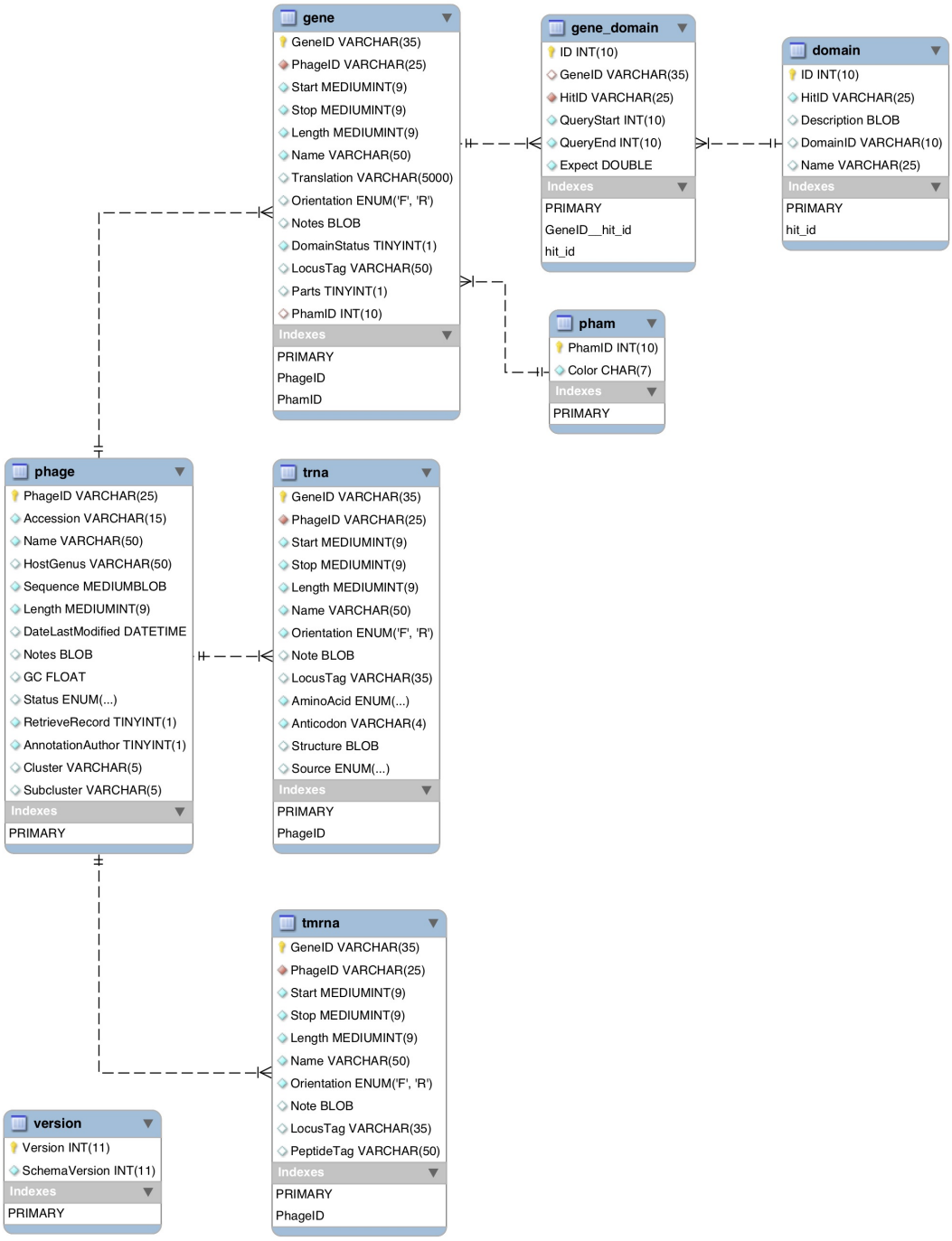


Fig. 2: Schema version 9.

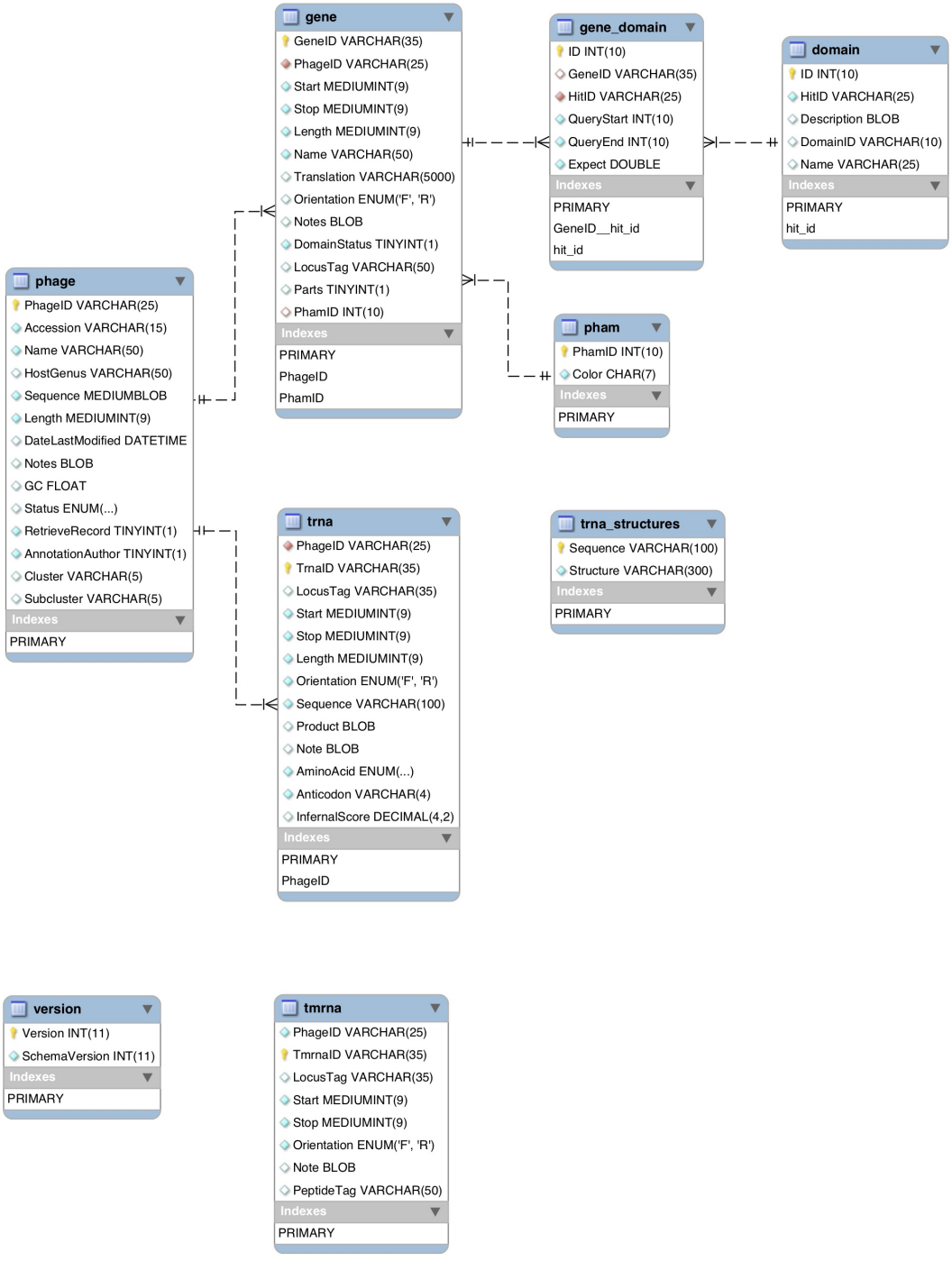


Fig. 3: Schema version 8.

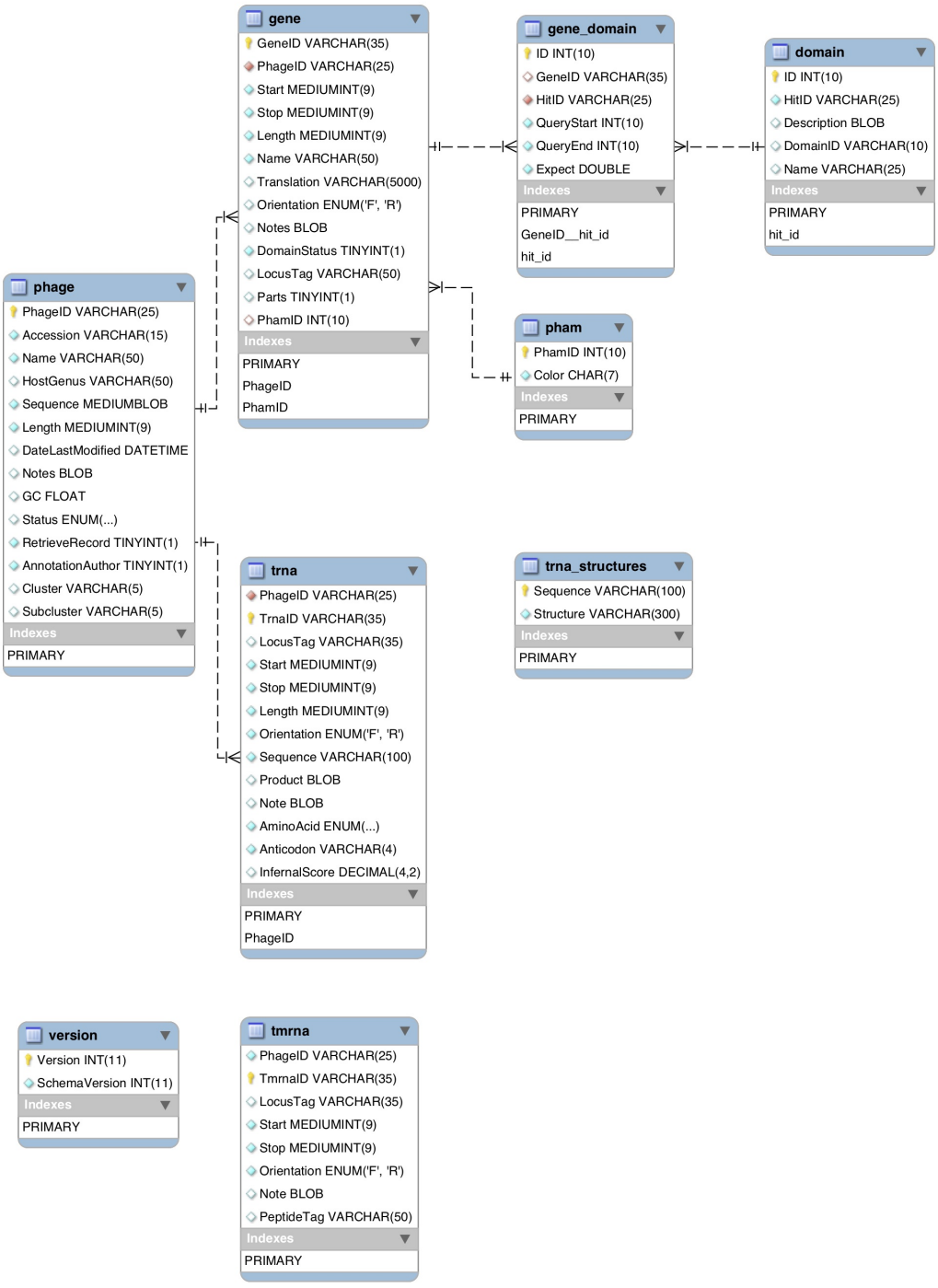


Fig. 4: Schema version 7.

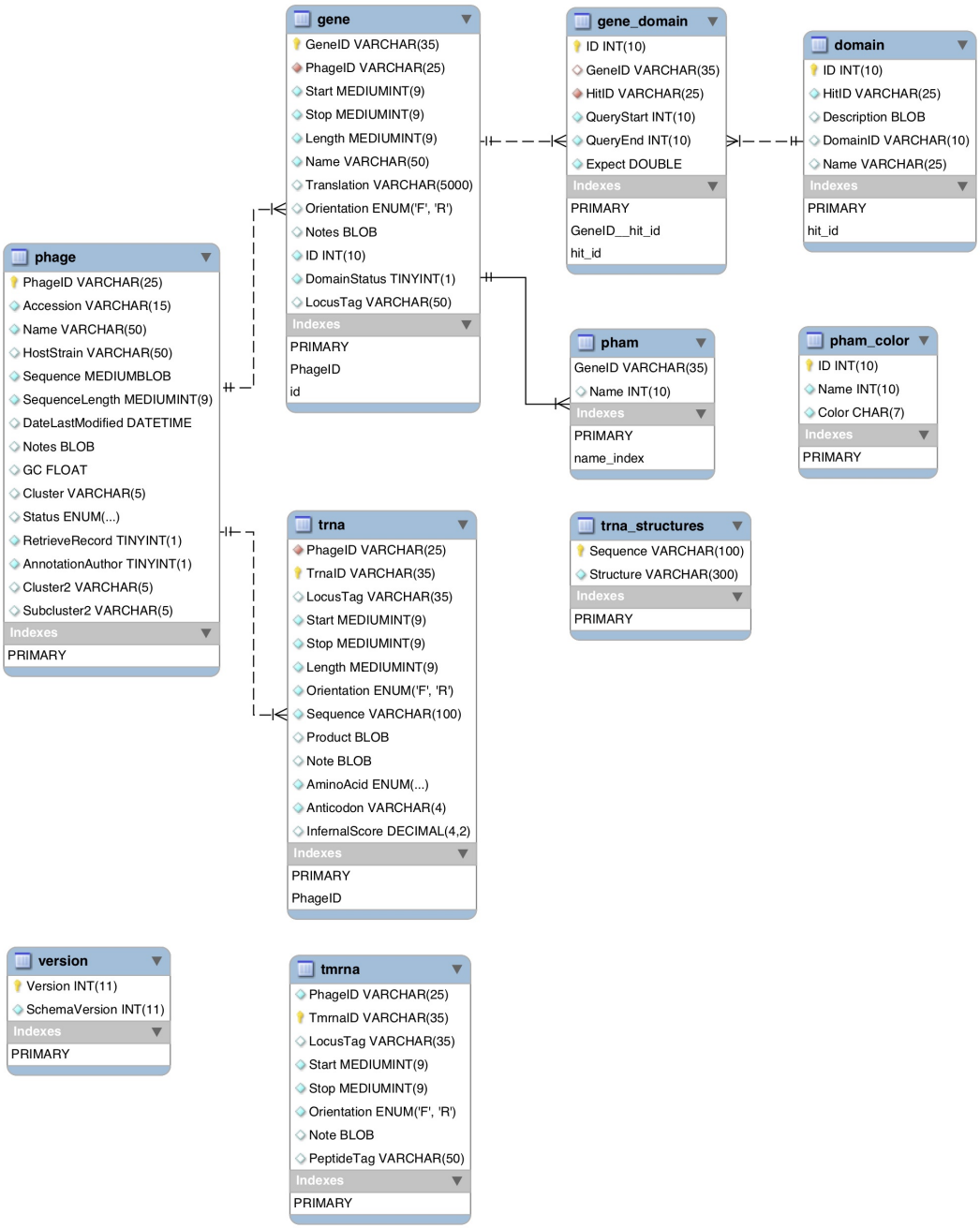


Fig. 5: Schema version 6.

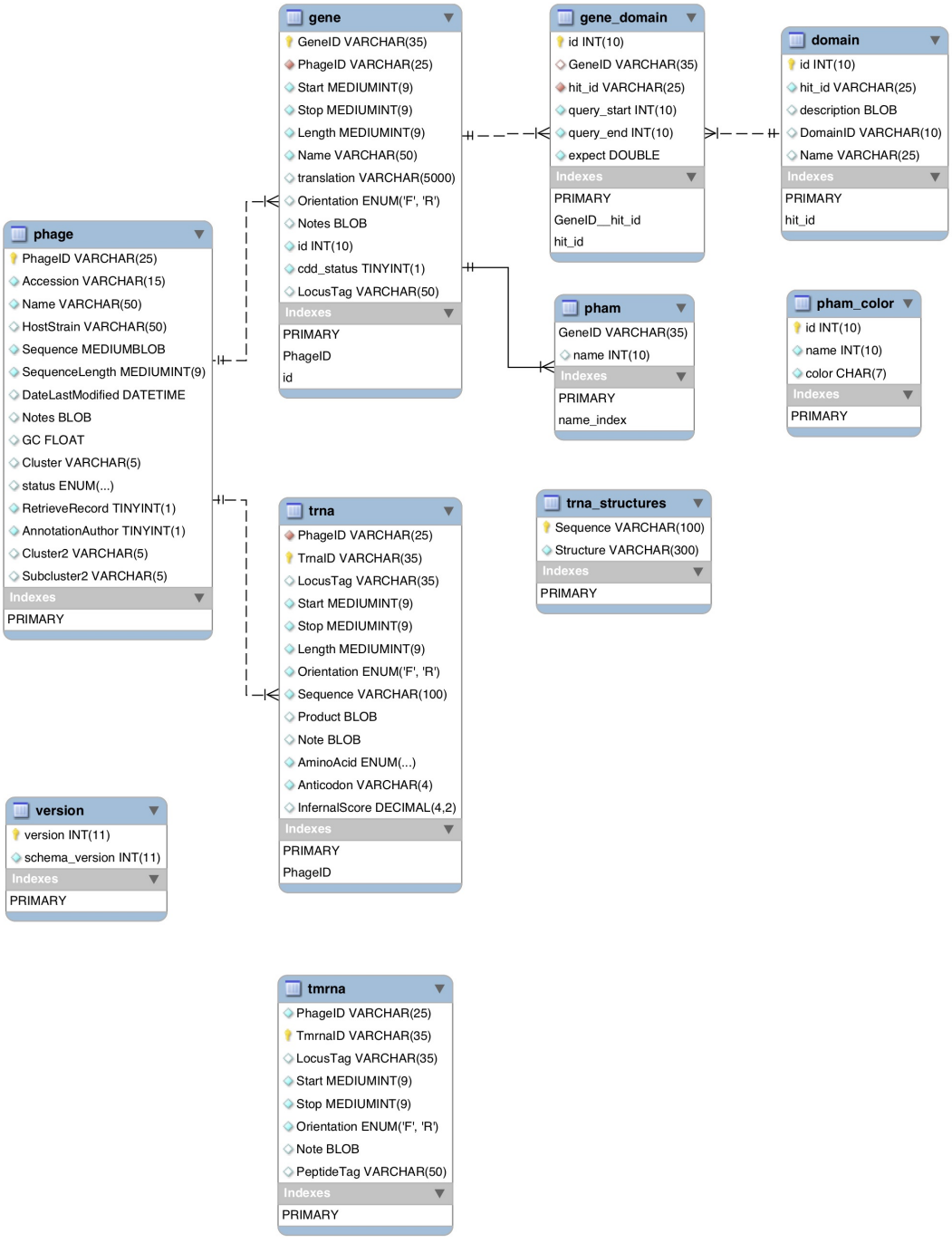


Fig. 6: Schema version 5.

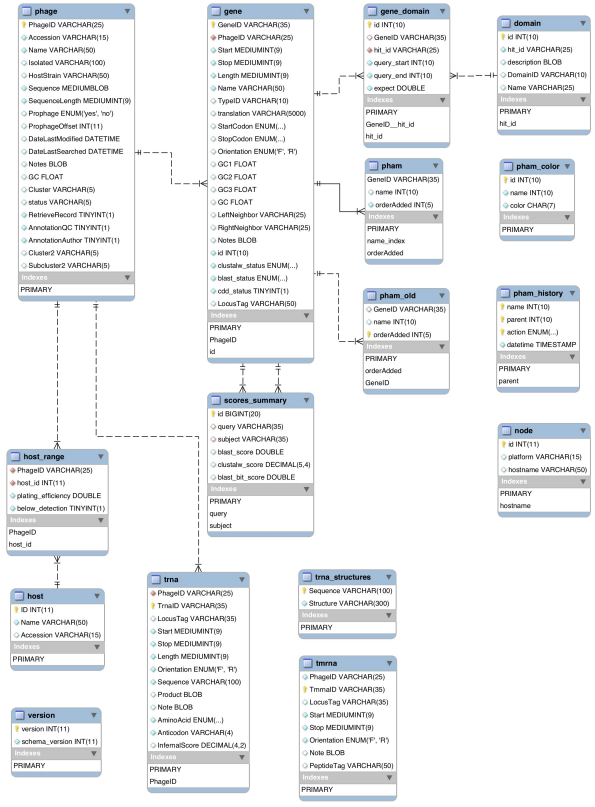


Fig. 7: Schema version 4.

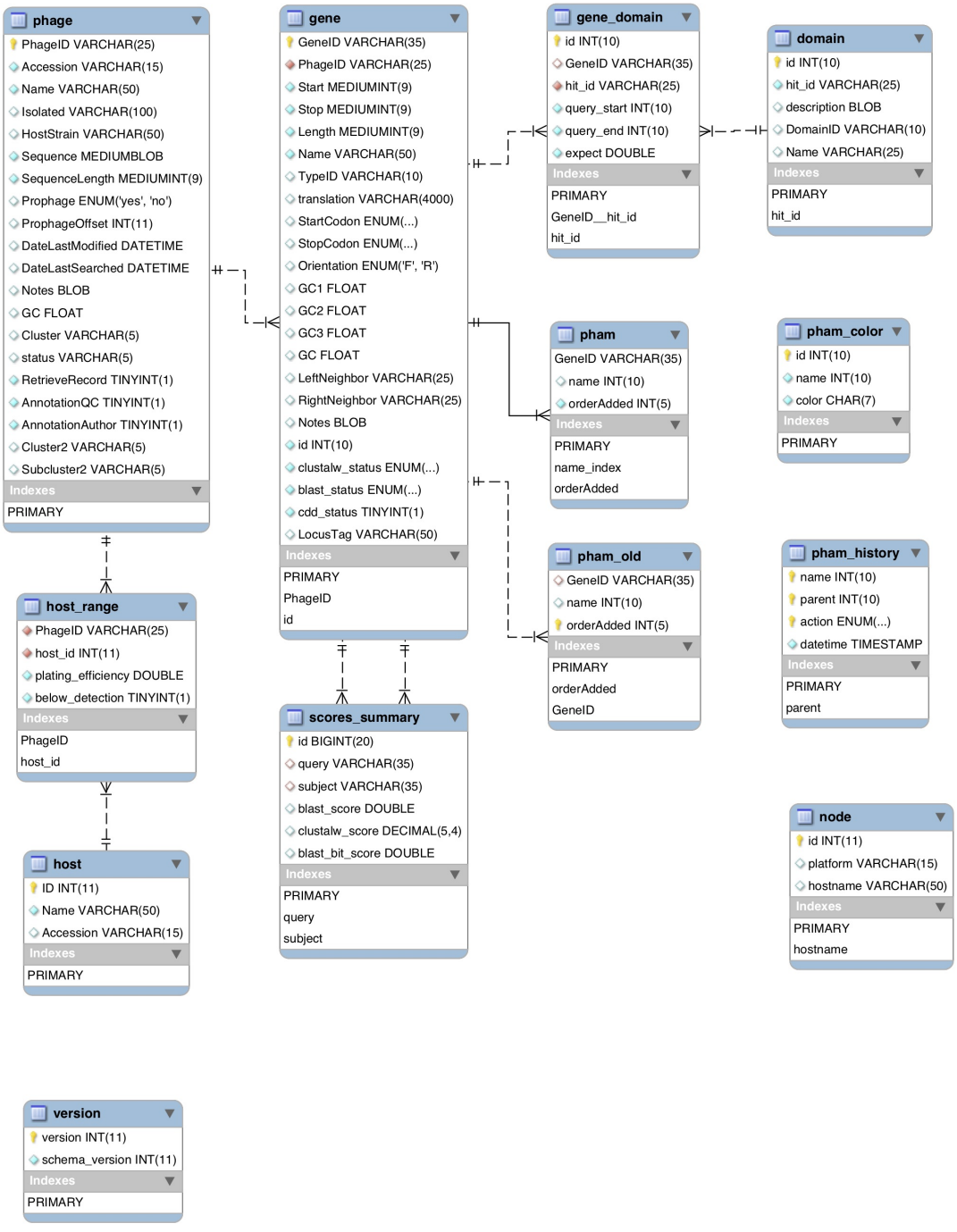


Fig. 8: Schema version 3.

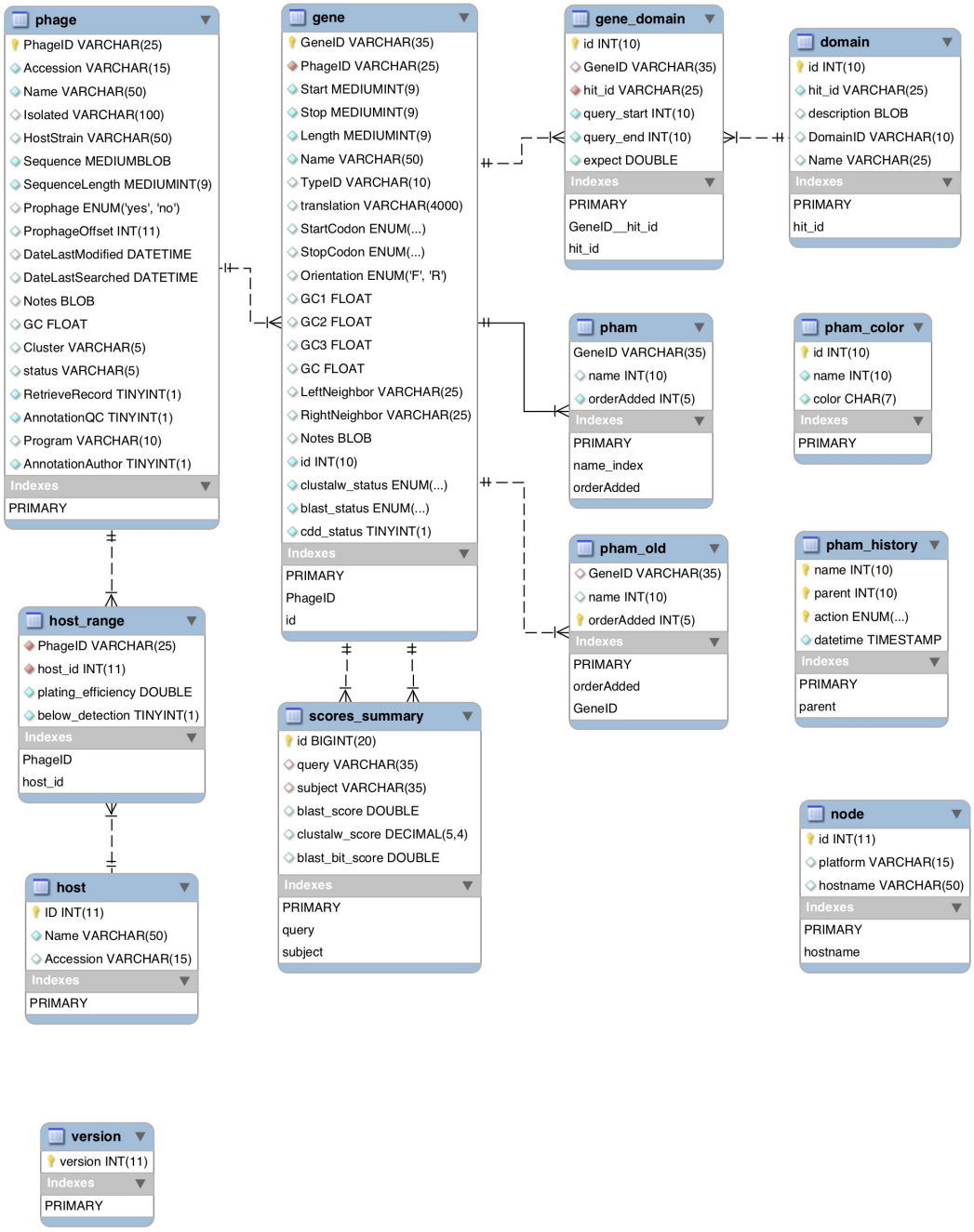


Fig. 9: Schema version 2.

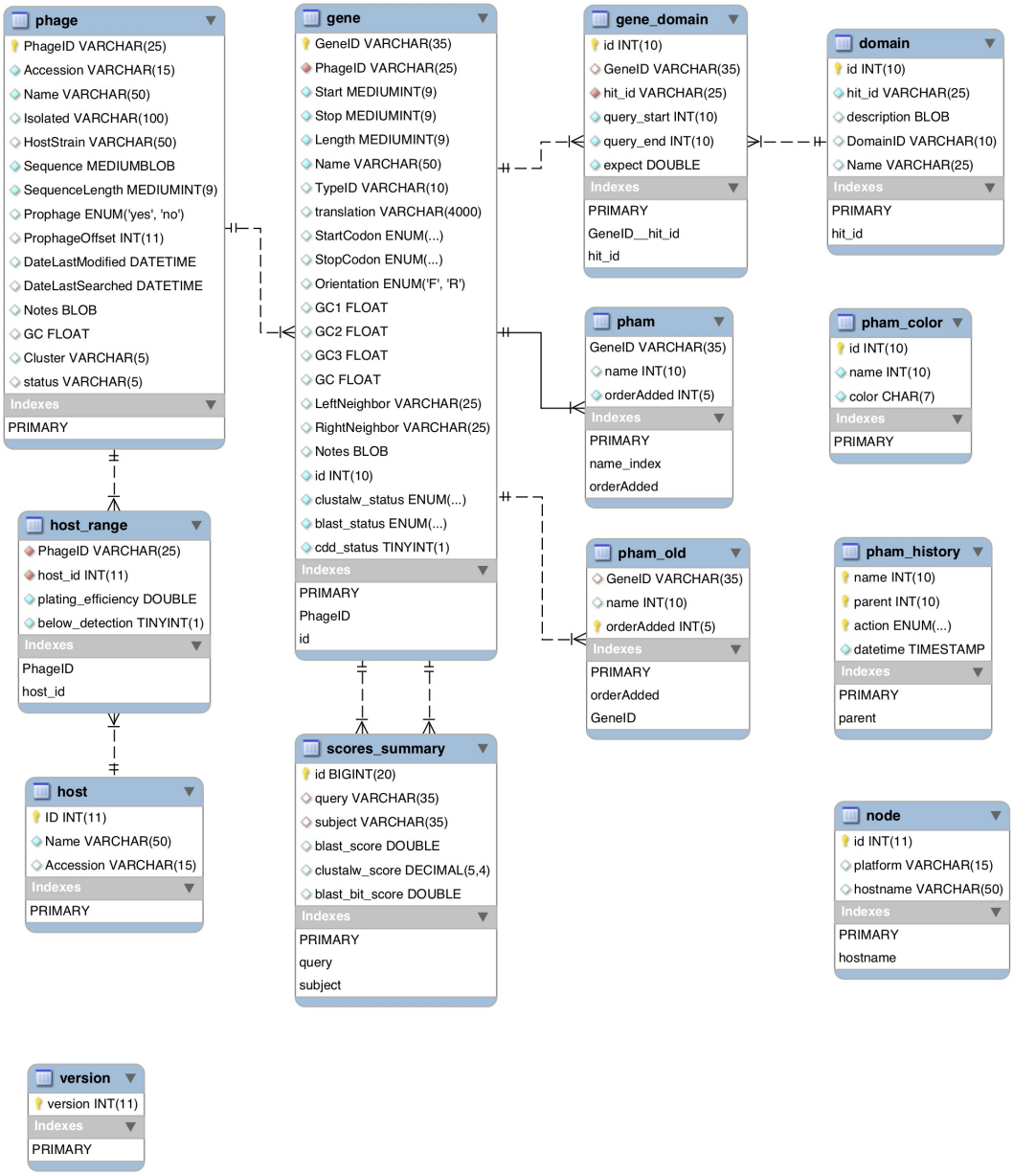


Fig. 10: Schema version 1.

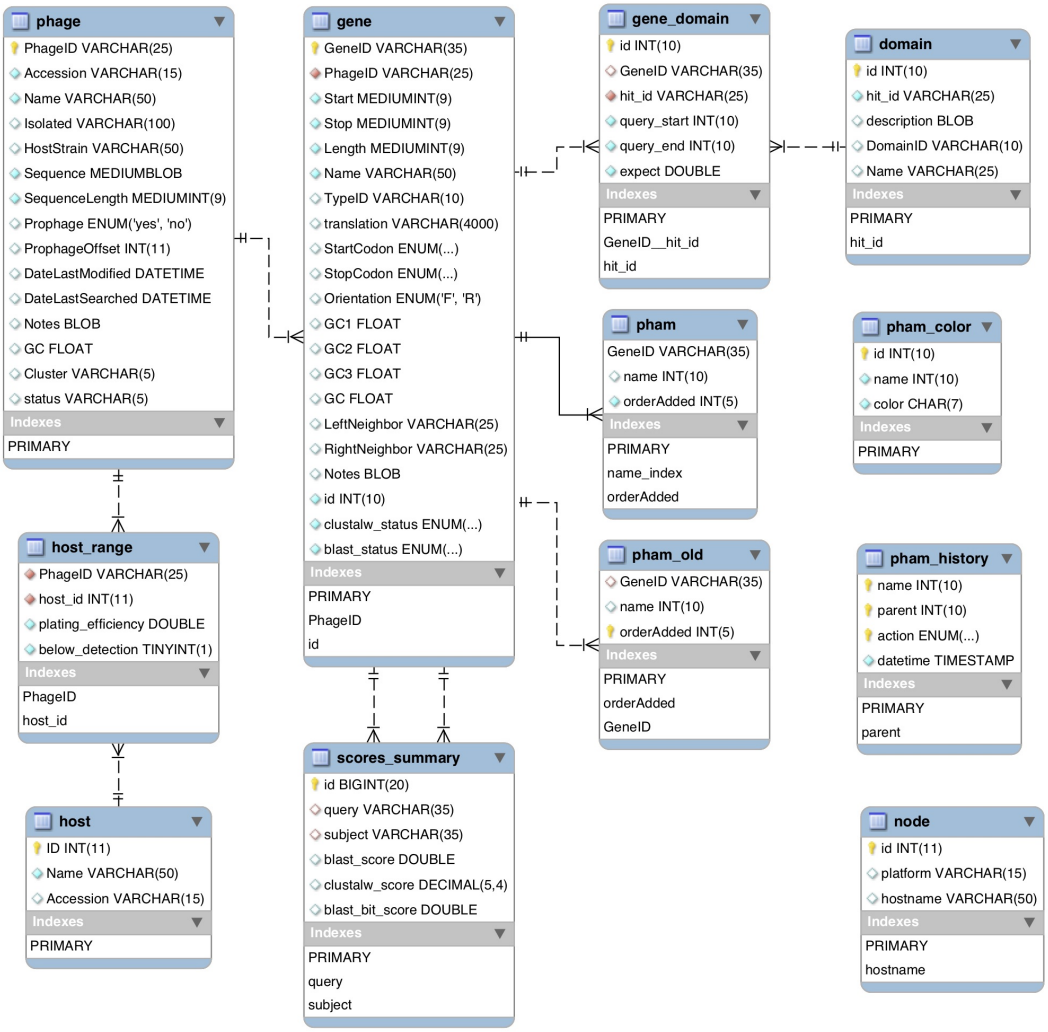


Fig. 11: Schema version 0.

CHAPTER

FOUR

DATABASE MANAGEMENT PIPELINES

Below is a description of command line tools that the `pdm_utils` package contains to analyze and manipulate data within a MySQL database:

Some tools are bound to the most current schema version ("yes"), as they need to know where to find types of data in the database. Other tools ("no"), or sub-tools ("yes"/"no"), are schema-agnostic, and can be used for any type of MySQL database.

Most tools provide functionality without respect to SEA-PHAGES-specific assumptions or goals. Some tools are more oriented to the SEA-PHAGES program by providing (optional) interactions with PhagesDB or by encoding SEA-PHAGES-specific assumptions about the data. For instance, a series of evaluations are implemented in the import and compare pipelines to ensure data quality, and while some of these evaluations are SEA-PHAGES-specific, others are not.

4.1 compare: compare data between databases

In the SEA-PHAGES program, genomics data may be stored in three separate databases (a local MySQL database, PhagesDB, and GenBank). As a result, data inconsistencies may arise without a mechanism to compare and synchronize different data sources. Although the import tool implements many QC checks for this purpose, it only ensures consistency for the specific genomes being imported and it only cross-checks databases as directed by the import ticket. In order to ensure comprehensive consistency, the `pdm_utils compare` tool can perform an all-against-all data assessment:

```
> python3 -m pdm_utils compare Actino_Draft -p -g -c config_file.txt
```

The argument 'Actino_Draft' indicates the name of the local MySQL database that serves as the central data source for the comparison. The '-p' and '-g' arguments indicate that data from PhagesDB and GenBank should be compared to the MySQL database as well as to each other (an all-against-all-against-all comparison).

> **..note:** It does not directly compare PhagesDB and GenBank data unless it also compares MySQL database data.

The '-c config_file.txt' argument indicates a local file containing login information for accessing MySQL and NCBI.

Selected subsets of data can be compared using the filtering argument. For instance, to only compare phages that infect Mycobacterium and that are 'final' status:

```
> python3 -m pdm_utils compare Actino_Draft -p -g -c config_file.txt -f "phage.
↪HostGenus=Mycobacterium AND phage.Status=final"
```

The `compare` tool retrieves data stored in the *phage* and *gene* tables. PhagesDB data for all sequenced phages are retrieved from: http://phagesdb.org/api/sequenced_phages/. All GenBank records are retrieved using accession numbers stored in the Accession field of the *phage* table. All data is matched between the local MySQL database and

PhagesDB using the PhageID field in the *phage* and the phage name in PhagesDB. All data is matched between the local MySQL database and GenBank using the Accession field in the *phage* table. After retrieving and matching data from all databases, the script compares the genome data (e.g. phage name, host strain, genome sequence, etc.) and gene data (e.g. locus tags, coordinates, descriptions, etc.), and generates several results files. Additionally, the script can output genomes retrieved from all three databases for future reference and analysis if selected by the user.

4.2 convert: upgrade and downgrade a database schema

Occasionally, the structure (schema) of the MySQL database is modified, which can result in certain types of data being stored differently. Even though the actual data may not have changed, these schema changes may prevent other tools from interacting with and retrieving data from the database. The `pdm_utils convert` tool is built to upgrade and downgrade the schema of a the database on a local computer to ensure your tools can still access the data.

To upgrade or downgrade a database (e.g. Actino_Draft) to the current version maintained in the Hatfull lab:

```
> python3 -m pdm_utils convert Actino_Draft
```

To convert a database to a specific schema version that your tool requires, the schema version can be selected with the '-s' flag:

```
> python3 -m pdm_utils convert Actino_Draft -s 5
```

Upgrading or downgrading to some schemas result in some tables or columns to be removed and added (and therefore auto-populated), resulting in some data to be lost or possibly inaccurate. After schema conversion, any tables or columns that may now have missing or inaccurate data are listed.

To avoid overwriting your primary database during conversion, a new database name can be indicated using the '-n' flag (although, if a database already exists with the same name, it will be overwritten):

```
> python3 -m pdm_utils convert Actino_Draft -s 5 -n Actino_Draft_s5
```

Additionally, if your local database schema doesn't exactly match one of the pre-defined schemas, MySQL may encounter a problem during conversion that results in the `convert` tool to terminate. This will result in a database that may only be partially converted to the intended schema. In these cases, the schema can be manually converted by executing the individual SQL statements in each conversion step. The list of MySQL conversion statements associated with each version change are stored in the pdm_utils GitHub repository. Clone the repository onto your local computer. The folder of the conversion scripts is located at 'pdm_utils/schema/conversion_scripts/'.

For instance, to manually convert a schema from version 6 to 5, execute each SQL statement in 'pdm_utils/schema/conversion_scripts/downgrade_6_to_5.sql' in mysql:

1. Log in to MySQL (enter your password when prompted):

   ```
   > mysql -u root -p
   ```

2. Execute each query from the conversion script, such as below:

   ```
   mysql> ALTER TABLE `gene_domain` CHANGE `QueryEnd` `query_end` int(10) unsigned NOT␣
   ↪NULL;
   ```

As with other pipelines, use of the *Database management configuration file* option can automate accessing MySQL.

4.3 export: export data from a database

This tool is used to export data from a MySQL database in a variety of formats, including:

1. A SQL file that represents the entire database.
2. CSV-formatted tables for selected tables.
3. Biopython SeqIOFormatted files for selected genomes, such as:
    a. GenBank-formatted flat files (.gb)
    b. Fasta-formatted files (.fasta)

4.3.1 Main Pipelines

SQL File Export

Export the entire database as a SQL file:

```
> python3 -m pdm_utils export Actino_Draft sql
```

Database information stored in a SQL database may need to be uploaded to a server for end-users. This option allows a file to be exported from MySQL into a single file that can be easily uploaded to a server (e.g. Actino_Draft.sql). The database version is tracked as an integer in the Version field of the *version* table, and a version file is also generated (e.g. Actino_Draft.version), which is a text file that contains a single integer corresponding to the database version.

CSV File Export

Export specific tables from the database into CSV-formatted files:

```
> python3 -m pdm_utils export Actino_Draft csv ...
```

Database information stored in a SQL database may need to be viewed in a more universal file. This option exports database information from a table in a comma-separated-values file that can be viewed in common spreadsheet or text-editing softwares (e.g. phage.sql).

BioPython SeqIO Files

Export genomes into biologically-relevant formatted files:

```
> python3 -m pdm_utils export Actinobacteriophage gb ...
```

Database information stored in a SQL database may be representative of biological constructs, and common bio-file currencies are often used to exchange and update information in the SQL database. This option exports database information to a formatted file that are common for bio-softwares (e.g. Trixie.gb, Trixie.fasta etc.)

4.3.2 Basic Export Options

Changing the export folder path

Change the path where a new directory containing exported files will be created:

```
> python3 -m pdm_utils export Actinobacteriophage gb -o /new/folder/path

> python3 -m pdm_utils export Actinobacteriophage csv --folder_path /new/folder/path
```

The command-line flag **-o** or **–folder_path** followed by the path to a desired directory will set the path where a the new directory will be created.

Changing the export folder name

Change the path where a new directory containing exported files will be created:

```
> python3 -m pdm_utils export Actinobacteriophage fasta -m new_folder_name

> python3 -m pdm_utils export Actinobacteriophage sql --folder_name new_folder_name
```

The command-line flag **-m** or **–folder_name** followed by a name will set the name of the new directory to be created.

Toggling the verbosity of export

Toggle on export progress statements:

```
> python3 -m pdm_utils export Actinobacteriophage csv -v

> python3 -m pdm_utils export Actinobacteriophage sql --verbose
```

The command-line flag **-v** or **–verbose** will toggle on progress report and status statements (verbosity).

Dumping created files into an existing directory

Dump created files into the current working directory or the specified folder path (see *Changing the export folder name*) avoiding the creation of a new directory:

```
> python3 -m pdm_utils export Actinobacteriophage sql -d

> python3 -m pdm_utils export Actinobacteriophage csv --dump
```

The command-line flag **-d** or **–dump** will toggle on file export dumping.

Utilizing more power

Use more of your computational resources via multithreading/parallel processing:

```
> python3 -m pdm_utils export Actinobacteriophage sql -np 8

> python3 -m pdm_utils export Actinobacteriophage gb --number_processes 8
```

The command-line flag **-np** or **–number_processes** followed by an integer specifies the number of virtual cores to utilize during the export pipeline

Forcing aggressive exports

Toggle on aggressive file and directory overwritting, avoiding cowardly responses to encountering pre-existing directories during export:

```
> python3 -m pdm_utils export Actinobacteriophage sql -f

> python3 -m pdm_utils export Actinobacteriophage gb --force
```

The command-line flag **-f** or **–force** will toggle on forced file export.

4.3.3 Import and Selection Export Options

Changing the table

Csv or SeqIO option to change the database table centered on for data export.:

```
> python3 -m pdm_utils export Actinobacteriophage gb -t phage

> python3 -m pdm_utils export Actinobacteriophage csv --table gene
```

The command-line flag **-t** or **–table** followed by a valid table from the selected MySQL database from which data is selected to be exported. Changing the table for csv export will change which columns are selected for export while changing the table for BioPython SeqIO file types will determine the data the formatted file will present.

Importing values with the command line

Csv or SeqIO option to pre-select data for export.:

```
> python3 pdm_utils export Actinobacteriophage gb -in Trixie

> python3 pdm_utils export Actinobacteriophage csv --import_names D29 L5
```

The command-line flag **-in** or **–import_names** followed by primary-key values from the database table selected for export (see *Changing the table*) begins export conditioned on the given set of values.

Importing values from a file

Csv or SeqIO option to pre-select data for export.:

```
> python3 pdm_utils export Actinobacteriophage gb -if /path/to/file

> python3 pdm_utils export Actinobacteriophage csv --import_file /path/to/file
```

The command-line flag **-if** or **–import_file** followed by a comma-separated-values file to be read for values. The first row of this file will be used as primary-key values from the database table selected for export (see *Changing the table*) to condition export on (similar to *Importing values with the command line*).

Dumping pham fasta-formatted multiple sequence and sequence alignment files

SQL option to include compressed files containing fasta-formatted multiple sequence and sequence alignment files:

```
> python3 pdm_utils export Actinobacteriophage sql -pho

> python3 pdm_utils export Actinobacteriophage sql --phams_out
```

The command line flag **-pho** or **–phams_out** toggles the export of all phams as fasta-formatted multiple sequence files, subsequent generation of sequence alignment files with clustal omega, and compression into zip files placed at the specified directory.

Including additional csv export columns

Csv option to add additional columns from the database for data export.:

```
> python3 pdm_utils export Actinobacteriophage csv -ic gene.GeneID

> python3 pdm_utils export Actinobacteriophage csv --include_columns gene.PhamID gene.
↪Notes
```

The command-line flag **-ic** or **–include_columns** followed by a MySQL-formatted column from the MySQL database selected for export to additionally be exported. Included columns must follow the format *table.column* and can be columns from different tables than the one selected for export (see *Changing the table*).

Excluding csv export columns

Csv option to exclude columns from the database for data export.:

```
> python3 pdm_utils export Actinobacteriophage csv -ec phage.Subcluster

> python3 pdm_utils export Actinobacteriophage csv --exclude_columns phage.Length
```

The command-line flag **-ec** or **–exclude_columns** followed by a MySQL-formatted column from the MySQL database selected for export tagged to not be exported. Excluded columns must follow the format *table.column* and can be columns from different tables than the one selected for export (see *Changing the table*).

4.3.4 Filtering and Organization Export Options

Filtering export

Csv or SeqIO option to filter data retrieved from the database.:

```
> python3 pdm_utils export Actinobacteriophage gb -w "phage.Cluster = A AND phage.
↪Subcluster IS NOT NULL"

> python3 pdm_utils export Actinobacteriophage csv --where "domain.Description LIKE
↪%helix-turn-helix% OR gene.Notes = 'helix-turn-helix DNA binding protein'"
```

The command-line flag **-f** or **–where** followed by a MySQL-formatted WHERE expression clauses separated by ANDs and ORs. Clauses can be expressed with the following format *table.column [operator] value* and can be using columns from different tables than the one selected for export (see *Changing the table*)

Grouping export

Csv option to exclude columns from the database for data export.:

```
> python3 pdm_utils export Actinobacteriophage csv -g phage.Status

> python3 pdm_utils export Actinobacteriophage csv --group_by phage.Cluster
```

The command-line flag **-g** or **–group_by** followed by a MySQL-formatted column from the MySQL database to group the data by for export. Grouping creates multiple subdirectories during export, and additional groups layer the subdirectories and group within already formed groups. Group by columns must follow the format *table.column* and can be columns from different tables than the one selected for export (see *Changing the table*).

Sorting export

Csv option to exclude columns from the database for data export.:

```
> python3 pdm_utils export Actinobacteriophage csv -s phage.Length

> python3 pdm_utils export Actinobacteriophage csv --order_by phage.PhageID phage.
↪Subcluster
```

The command-line flag **-s** or **–order_by** followed by a MySQL-formatted column from the MySQL database to sort the data by for export. Ordering sorts the data exported and additional orderings subsort the data. Order by columns must follow the format *table.column* and can be columns from different tables than the one selected for export (see *Changing the table*).

4.3.5 Additional Export Options

Renaming exported sql files

Sql option to rename the exported database file:

```
> python3 pdm_utils export Actinobacteriophage sql -n Actino_Draft

> python3 pdm_utils export Actinobacteriophage sql --name Actino_Draft
```

The command line flag **-n** or **–name** followed by a desired file name exports a sql file and version file named accordingly.

Concatenating SeqIO files

SeqIO option to add all export data into one contiguous formatted file.:

```
> python3 pdm_utils export Actinobacteriophage gb -cc

> python3 pdm_utils export Actinobacteriophage gb --concatenate
```

The command line flag **-cc** or **–concatenate** toggles the concatenation of exported SeqIO formatted flat files.

Including sequence data

Csv option to include all sequence and translation data.:

```
> python pdm_utils export Actinobacteriophage csv -sc

> python pdm_utils export Actinobacteriophage csv -sequence_columns
```

The command line flag **-sc** or **–sequence_columns** toggles the inclusion of sequence or translation type data into the csv for export.

Conserving raw byte data

Csv option to conserve and export raw byte data.:

```
> python pdm_utils export Actinobacteriophage csv -rb

> python pdm_utils export Actinobacteriophage csv --raw_bytes
```

The command line flag **-rb** or **–raw_bytes** toggles off the conversion of blob and byte-type data flagged for export, exporting the raw-byte format of the data.

4.4 find_domains: find NCBI conserved domains

The NCBI maintains a conserved domain database (CDD) in which the functions of protein sequences are systematically categorized and organized [ref Marchler-Bauer 2011]. Every gene product in the database can be evaluated for conserved domains using a local copy of the CDD, and this conserved domain data is stored in the database using the `pdm_utils find_domains` tool.

To identify conserved domains:

```
> python3 -m pdm_utils find_domains Actino_Draft -d /path/to/CDD/
```

The argument 'Actino_Draft' indicates the name of the database that will be evaluated. The '/path/to/CDD/' indicates where the NCBI conserved domain database is stored on your local computer.

In the *gene* table, there is a field called DomainStatus. When new phage genomes are added, the DomainStatus field for each new gene is set to '0'. The `find_domains` tool retrieves gene products (stored in the Translation field of the *gene* table) for all genes with DomainStatus < '1'. As part of the BLAST+ package, the rpsblast+ tool is used to identity conserved domains using BLAST with an e-value threshold = 0.001. For each gene, retrieved CDD data is inserted into the *domain* and *gene_domain* tables, and the DomainStatus field in the *gene* table is set to 1 so that this gene is not re-processed during subsequent rounds of updates.

`find_domains` tries to insert all hits from rpsblast+ that pass the threshold. However, rpsblast+ may report multiple hits to a domain within the same translation that pass the threshold. Duplicate hits are not permitted in the database, so when there is an attempt to insert a duplicated hit into the database, the user is notified:

```
Warning: (1062, "Duplicate entry 'gnl|CDD|334100' for key 'hit_id'")
```

This message is simply a warning, and no action needs to be taken.

As with other pipelines, use of the *Database management configuration file* option can automate accessing MySQL.

4.5 freeze: create a static database for long-term reference

The primary database instance, Actino_Draft, contains all available actinobacteriophage data, and is routinely updated and modified. However, specific projects may require a version of the database that:

1. contains a subset of genome data (such as all genomes that have NOT been auto-annotated),
2. is no longer routinely modified/updated, and/or
3. has an identifier distinct from other databases.

The `pdm_utils freeze` tool can create these "frozen" databases:

```
> python3 -m pdm_utils freeze Actino_Draft ./
```

The argument 'Actino_Draft' indicates the name of the local database from which a frozen database will be generated. The './' argument indicates the working directory for manipulating files and storing output files.

The tool copies the database indicated in the first script argument, deletes all data pertaining to draft genomes (and thus retains all final and unknown genomes), and saves the new database with a unique identifier that indicates both the type of phage database and the number of genomes (e.g. Actino_Draft_<0123>, where <0123> is an integer). The tool creates a folder for this new database in the directory indicated by the second script argument, and creates three subfolders (current, backup, and updates_history) analogous to the folders regularly used to maintain the Actino_Draft instance. Since the new frozen database no longer contains draft genomes, gene phams need to be recomputed. As a result, the script does not export the new database, and it resets the database version to '0'. After re-computing gene

phams, the 'update' tool can increment the version number, the 'export' tool can generate a new SQL file, and the 'push' tool can upload it to the server.

Different types of databases may need to be frozen. For instance, sometimes all actinobacteriophages (other than draft genomes) need to be retained. Other times, only the *Mycobacterium* phages need to be retained. As a result, the script prompts the user to choose an appropriate database name (e.g. Actinobacteriophage, Mycobacteriophage, etc.) before it appends the genome count to the database name. A customized database name can be provided if needed. However, this script does not yet provide the functionality of enabling the database administrator to indicate which types of phages should be retained (other than the annotation status), and this step currently needs to be completed with the mysql command line utility. It is also important to note that although this database is regarded as "frozen", it is still able to be modified. Therefore, if minor errors are encountered in the database that need to be modified, the database can be adjusted in MySQL, the version number can be incremented, the database can be re-exported, and the updated database file can overwrite the older file on the server.

As with other pipelines, use of the *Database management configuration file* option can automate accessing MySQL.

4.6 get_data: get new data to import into the database

New genomics data routinely becomes available for adding to the Actino_Draft database, including:

1. Metadata pertaining to individual phages (such as host, cluster, subcluster, and accession)
2. Newly-sequenced and auto-annotated 'draft' genomes
3. New manually-annotated 'final' genomes
4. Updated annotations from GenBank

These data can be automatically retrieved using the `pdm_utils get_data` tool:

```
> python3 -m pdm_utils get_data Actino_Draft -o ./ -c config_file.txt
```

The argument 'Actino_Draft' indicates the name of the database from which updates are determined. The '-o ./' indicates the directory where the data should be downloaded (if omitted, the default is the working directory from where the pipeline is called).

As with other pipelines, use of the *Database management configuration file* option can automate accessing MySQL and NCBI. If only certain types of updates are required, there are separate command line flags for each type of update.

Each type of data is retrieved and staged for import. A new folder is created that contains:

1. a CSV-formatted import table listing each 'ticket' pertaining to the type of update
2. a 'genomes' subdirectory containing flat files for import (if applicable)

4.6.1 Metadata updates

PhagesDB is the primary source for Cluster, Subcluster, Host, and Accession data for phages in the Actino_Draft database. `get_data` compares these data for each phage in the selected database to the corresponding data in PhagesDB, and creates a new update ticket for all discrepant data that need to be corrected in the Actino_Draft database. New metadata is retrieved from PhagesDB: sequenced phages. For each phage, PhagesDB stores both GenBank and RefSeq accession data, but only GenBank accession data (stored in the *genbank_accession* field) are stored in the Actino_Draft database.

4.6.2 New 'draft' auto-annotations

PhagesDB is the primary source for new genome sequences generated through the SEA-PHAGES program. Automatically-generated 'draft' annotations can be imported into a MySQL database for immediate reference. `get_data` identifies genomes present in PhagesDB that are not present in the selected MySQL database. For each new genome, a request is sent to the phage genomics tool PECAAN to generate auto-annotations with the URL: 'https://discoverdev.kbrinsgd.org/phameratoroutput/phage/<PhageID>' (where <PhageID> indicates the specific phage name of interest). PECAAN retrieves the new sequence from PhagesDB, automatically annotates genes, and returns a GenBank-formatted flat file. `get_data` stages the new file and a corresponding import ticket in an import table that are ready to be processed with the 'import' tool.

4.6.3 New 'final' annotations

The 'draft' annotations are eventually replaced with manually-generated 'final' annotations. The refined annotations are submitted by senior annotators to PhagesDB in GenBank-formatted flat files, which are are stored in the *qced_genbank_file* field with a timestamp stored in the *qced_genbank_file_date* field. Similar to metadata updates, `get_data` matches phage data in the selected MySQL database to the corresponding data in PhagesDB, and determines whether there is a new version of a 'final' annotation available for import. It reviews the date that the genome data was uploaded to PhagesDB, and if it is more recent than the date of the annotations stored in the selected MySQL database (indicated in the DateLastModified field of the *phage* table), it stages the new flat file from PhagesDB and a corresponding import ticket in an import table that are ready to be processed with the 'import' tool.

4.6.4 Updated GenBank annotations

The 'final' annotations are eventually submitted to GenBank with a unique accession number. The GenBank records are routinely updated with improved annotation data, so these records are subsequently re-imported into the database to replace the prior annotations.

`get_data` matches phage data in the selected database to the corresponding data in GenBank (indicated in the Accession field of the *phage* table) and assesses whether the date of the record is more recent than the date of the annotations stored in the database (indicated in the DateLastModified field of the *phage* table). If the GenBank record is more recent, `get_data` stages the new flat file from GenBank and a corresponding import ticket in an import table that are ready to be processed with the 'import' tool.

Additionally, a CSV-formatted summary table of all PhageIDs, their accession, and the results of data retrieval from GenBank is generated.

4.6.5 New non-SEA-PHAGES annotations

Actino_Draft genomes that have been sequenced and annotated outside of the SEA-PHAGES program occasionally become available in GenBank. If the genomes should be imported into a MySQL database, the GenBank-formatted flat files need to be manually retrieved from GenBank and staged in a local directory with a manually-generated import ticket.

4.7 get_db: download and install a database

The Hatfull lab server hosts the primary actinobacteriophage database instance, Actino_Draft, which is routinely updated with new genomics data, as well as databases that have been frozen for publication. These databases can be downloaded and installed using the `pdm_utils get_db` tool.

To view and interactively select the list of available databases on the Hatfull lab server:

```
> python3 -m pdm_utils get_db server
```

The `pdm_utils get_db` interactive tool functions as a faux command line interface that allows you to view the database packages at the specified url and navigate through subdirectories. Navigation through subdirectories is modelled after common CLI commands:

```
root@http://databases.hatfull.org/::$ ls
        Abscessus_prophages             PhageDatabase
        Actino_Draft_v6                 Actino_Draft
        Actino_prophage                 Published/

root@http://databases.hatfull.org/::$ cd Published
root@http://databases.hatfull.org/::/Published$ ls
        Actinobacteriophage_1060        Actinobacteriophage_1321
        Actinobacteriophage_2422        Actinobacteriophage_554
        Actinobacteriophage_685         Actinobacteriophage_692
        ../
```

With the `pdm_utils get_db` interactive tool you can see a description of a database package that is at available at the specified url:

```
root@http://databases.hatfull.org/::$ desc PhageDatabase
Name:
        PhageDatabase
Date:
        202x-xx-xx
Description:
        This database contains sequenced phages from the John Doe lab.
```

A database of choice can be selected using the `pdm_utils get_db` interactive tool with the following:

```
root@http://databases.hatfull.org/::$ select PhageDatabase
```

To download and install the current version of a database, like the Actino_Draft database, without the interactive tool:

```
> python3 -m pdm_utils get_db server -db Actino_Draft

> python3 -m pdm_utils get_db server --database Actino_Draft
```

The -db argument 'Actino_Draft' indicates the name of the database to download from the server, and it will install it under the same name in your local MySQL. The database will be downloaded, installed, and then the file will be removed.

Note: This will overwrite an existing Actino_Draft database, if it exists.

To download and install a database from a non-standard server, specify the URL:

```
> python3 -m pdm_utils get_db server -db PhageDatabaseName -u http://custom/server/
↪website

> python3 -m pdm_utils get_db server -db PhageDatabaseName --url http://custom/server/
↪website
```

The `pdm_utils get_db` tool checks your local database version against the specified server database package and will not download if the local database version is equal to or higher than the database package to prevent redundancies and/or loss of data. To ignore this check:

```
> python3 -m pdm_utils get_db server -db Actino_Draft -fp

> python3 -m pdm_utils get_db server -db Actino_Draft --force_pull
```

Databases can be downloaded and installed in two steps, which can be used to install a database under a new name:

1. First download the database sql and version files from the Hatfull server using the '-d' and '-v' flags. This will save the database to your local computer as a SQL file (e.g. Actino_Draft.sql) without installing it in MySQL. Also specify where the file should be downloaded using the '-o' flag (if omitted, the default is the /tmp/ directory):

   ```
   > python3 -m pdm_utils get_db server -db Actino_Draft -o ./ -d -v
   ```

2. Next, indicate the new name of the database to be created (e.g. NewDB), indicate that a local file will be used with 'file' , and indicate the path to the downloaded SQL file:

   ```
   > python3 -m pdm_utils get_db NewDB file ./downloaded_db/Actino_Draft.sql
   ```

Use of a *Database management configuration file* can automate the pipeline so that user input such as MySQL credentials or server URL is not needed:

```
> python3 -m pdm_utils get_db server -db Actino_Draft -o ./ -d -v -c config_file.txt
```

4.8 get_gb_records: retrieve records from GenBank

Genome data in the MySQL database may be directly associated with genome data stored in GenBank. The 'Accession' field in the 'phage' table provides a link to these records, and all GenBank records associated with any particular database can be retrieved using the `pdm_utils get_gb_records` tool.

To retrieve all GenBank records relative to your database, indicate the name of the database (e.g. 'Actino_Draft'). The '-o ./' indicates the directory where the files should be downloaded (if omitted, the default is the working directory from where the pipeline is called):

```
> python3 -m pdm_utils get_gb_records Actino_Draft -o ./
```

Note: The current version of Actino_Draft contains thousands of accession numbers, so retrieving GenBank records for the entire database can be slow.

This tool relies upon the NCBI E-utilities (using a Biopython wrapper), and NCBI requests that you provide information about yourself. The `get_gb_records` tool accepts a simple text file containing your information for MySQL as well as NCBI (*Database management configuration file*) using the '-c' flag:

```
> python3 -m pdm_utils get_gb_records Actino_Draft -o ./ -c config_file.txt
```

The `get_gb_records` tool will determine which accessions from the MySQL database are valid, will retrieve all live records from GenBank, and store them locally in GenBank-formatted flat file format.

Genome data retrieved from GenBank can also be retrieved in five-column feature table format for a more streamlined view of gene products, notes, and coordinates:

```
> python3 -m pdm_utils get_gb_records Actino_Draft -ft tbl
```

4.9 import: manage genome data and annotations

In general, data pertaining to a complete phage genome is managed within a MySQL database as a discrete unit, in which genome data (such as the PhageID, genome sequence, host data, etc.) is added, removed, and replaced concomitantly with all associated gene annotation data (primarily CDS features), such that any data pertaining to a particular phage in the database has been parsed from one external source, instead of added piecemeal and incrementally from separate sources or modified within the database after insertion. There are a few fields that are exceptions to this general practice (*update*)

`pdm_utils import` is used to manage the addition or replacement of genomes:

```
> python3 -m pdm_utils import Actino_Draft ./genomes/ ./import_table.csv -o ./ -p -c␣
↪config_file.txt
```

The first argument ('Actino_Draft') following 'import' indicates the database to be used. The next argument ('./genomes/') indicates the directory where the flat files are located. The next argument ('./import_table.csv') indicates the location of the import table. The optional '-o ./' argument indicates the directory where the import results should be stored (if omitted, the default is the working directory from where the pipeline is called). As with other pipelines, use of the *Database management configuration file* option can automate accessing MySQL.

This tool can be used to specifically update the Actino_Draft database, manage different MySQL database instances, and support the process of genome annotation because it:

1. relies on import tickets to substantially automate the import process.
2. performs evaluations to verify the quality of the incoming data.
3. provides an interactive environment for a more flexible process.

4.9.1 Parse and validate import table

The first step of `import` is to parse and prepare tickets from the import table (*Import tickets*). The structure of the table as well as data in each ticket is validated. For each ticket type, there are specific rules regarding how the ticket fields are populated to ensure that the ticket is implemented correctly. Additionally, the pipeline confirms that there are no duplicated tickets or tickets with conflicting data (such as an add and remove ticket for the same phage). Import tickets are automatically generated by `get_data`, but they can also be manually generated.

4.9.2 Process flat files

After preparing import tickets from the import table, flat files (*GenBank-formatted flat files*) are processed one at a time, matched to the corresponding import ticket, evaluated, and implemented. For replace tickets, the current genome data in the database is removed and the data from the flat file is parsed and inserted. Two types of data are parsed from the flat file and evaluated: genome-specific and gene-specific data.

Genome-specific data

Genome-specific data, such as the phage name, nucleotide sequence, host genus, accession, and annotation authorship is parsed and stored in the *phage* table. The data in the flat file is matched to the import ticket by the phage name parsed from the file. Subsequently, the data is evaluated and compared to data in the import ticket and in the database. After this, several fields in the *phage* table are populated from data derived from the import ticket or from the flat file.

Matching tickets to flat files requires that the phage names are spelled identically. Sometimes this is not the case, in which the desired spelling of the phage name in the database (and thus in the import ticket) is slightly differently than the spelling in the GenBank record. These conflicts can arise for several reasons that cannot be immediately corrected (e.g. different nomenclature constraints, such as how "LeBron" is spelled "Bron" in the GenBank record).

To account for these conflicts, `import` contains a pre-defined phage name dictionary that converts several GenBank phage names to the desired phage name stored in the Actino_Draft database. This list only contains about two dozen name conversions and does not change frequently. To avoid phage name discrepancies, the phage name can be parsed from different parts of the file (such as the filename itself). This allows for greater flexibility when parsing batches of flat files that may not adhere to default expectations, such as when new database instances are developed for phages that have not been annotated from disparate sources. This option can be implemented as a command line option.

Gene-specific data

The second type of data parsed from the flat file pertains to individual genes (and is stored in the *gene* table). After parsing the genome-specific information, the annotated features are processed. The Source, tRNA, tmRNA, and CDS features are evaluated, and all others are ignored.

Note: Currently, tRNA and tmRNA features are not dynamically parsed from flat files.

CDS features are parsed, evaluated, and stored in the *gene* table. The majority of data that `import` stores in the *gene* table are derived directly from the flat file. Several things to note:

1. GeneIDs represent the gene's unique identifier in the database, and are automatically generated during import, irrespective of data from within the flat file.
2. Gene descriptions are stored in the Notes field of the *gene* table. However, CDS features in flat files can contain descriptions in three different fields: PRODUCT, FUNCTION, and NOTE. The 'description_field' field in the import ticket indicates which of these three flat file fields are expected to contain gene description data in the flat file.
3. The LocusTag field in the *gene* table is populated directly from the LOCUS_TAG field in the CDS feature. It provides an unambiguous link to the original CDS feature in the GenBank record. This is valuable when reporting the gene information in a publication, and it is required when requesting GenBank to update information about specific CDS features (such as corrections to coordinates or gene descriptions).
4. In many GenBank records, CDS features may contain descriptions that are not informative (e.g. "hypothetical protein", "phage protein", etc). These generic descriptions are not retained.

4.9.3 Evaluations

For each flat file, `import` checks numerous fields for accuracy through a series of QC evaluations.

For some QC evaluations, an error is automatically logged when a problem is encountered. For other QC evaluations, a warning is reported when a problem is encountered, the data processing pauses, and the user is prompted to provide feedback about whether the evaluation should log a warning or an error.

Note: The prompt typically asks "Is this correct?" Replying "yes" indicates there is no true error, and no error will be logged. Replying "no" will log an error.

If a genome acquires one or more errors during import, the genome will not be imported, and no changes are made to the database for that genome. The success or failure of an import ticket has no impact on the success or failure of the next ticket. After all tickets are processed, `import` is completed.

4.9.4 Logging database changes

Several methods of tracking and managing tickets (and the associated genomes) as they pass or fail QC are implemented:

1. A summary of the import process is reported in the UNIX shell during import and after all tickets are processed.
2. The results of every ticket are recorded in a log file, including any errors and warnings that were generated. Searching for "warnings" or "errors" in the file can quickly highlight the potential problems.
3. Tickets and genome files are copied to new folders based on their 'success' or 'fail' import status. This enables quick reference to the specific tickets and genome files that need to be reviewed, modified, and repeated.
4. `import` can be run under 'test' or 'production' mode. During a production run, import tickets and genome files are processed and evaluated, and the database is updated as specified by the ticket if QC is passed. In contrast, during a test run, import tickets and genome files are processed and evaluated, but the database is not updated. The test run can determine if any particular group of tickets and flat files are ready to be imported without actually altering the database, allowing flat files to be repeatedly evaluated during the annotation process (*Reviewing genome annotations*).

4.10 phamerate: create gene phamilies

Phameration is the process whereby phamilies ("phams") are constructed from groups of putatively homologous protein-coding genes. This procedure allows us to distill the genetic diversity in a set of genomes down to its most discrete units, which in turn facilitates examination of the evolutionary relationships between very distantly related genomes.

The `pdm_utils phamerate` tool provides access to two pipelines for phameration. The first pipeline uses MMseqs2 (*Steinegger and Söding, 2017*), a homology detection suite purpose built for searching and clustering large genomics datasets. The second pipeline mimics a pipeline that seems to be popular in the phage genomics world, which uses blastp (*Altschul et al, 1990*) for homology detection and Markov Clustering (*van Dongen & Abreu-Goodger, 2012*) to interpret the blastp output into clusters.

To phamerate gene products using the MMseqs2 pipeline:

```
> python3 -m pdm_utils phamerate mmseqs Actino_Draft
```

The pipeline will crash if the mmseqs software suite has not been installed or is not globally executable

To phamerate gene products using the blastp –> MCL pipeline:

```
> python3 -m pdm_utils phamerate blast-mcl Actino_Draft
```

The pipeline will crash if the NCBI's blast+ package OR Markov Clustering have not been installed or are not globally executable

The argument 'Actino_Draft' indicates the name of the database whose gene products are to be phamerated.

Both pipelines have optional arguments that can be used to adjust the thresholds/algorithms used for clustering. The mmseqs pipeline has been optimized from the ground up to construct phamilies of globally homologous phage proteins. In general we expect that functions can be propagated across members within the same pham. The blast-mcl pipeline has not been optimized nearly as rigorously, but it uses parameters I've seen commonly published for similar pipelines used for phage genomics.

Regardless of which pipeline is used, new phams and their colors (for display in Phamerator) are inserted into the *pham* table of the database. Any phams that are unchanged (or now include one or more newly added genes) between rounds of phameration will have their pham designation and color preserved.

4.10.1 Notes for mmseqs pipeline

The mmseqs pipeline will run in two steps by default:

1. Sequence-sequence clustering to construct pre-phamily HMM profiles
2. Profile-consensus clustering to merge pre-phamilies into more sensitive phamilies

The second step can be skipped by using the –skip-hmm argument at the command line. Doing so without also adjusting the parameters used in the first step will result in phams with relatively weak sensitivity, but few (if any) false positives.

As previously mentioned, this pipeline's default parameters have been optimized to construct phams of globally homologous phage proteins. Aside from minor tweaks to these parameters, in general the only reason to deviate from the defaults would be to construct phams for a different use cases than comparing genomes on the basis of shared genes or propagating functions. For example, if one's goal is to examine intragenic mosaicism, the default coverage threshold is too high to identify most domain-linked sequences. In this case it's probably simpler to use the blast-mcl pipeline with a low (or no) coverage cutoff.

4.11 push: upload a database to a public server

Data can be uploaded to the Hatfull lab's public server using the `pdm_utils push` tool:

```
> python3 -m pdm_utils push -f Actino_Draft.sql
```

The '-f' flag indicates a specific file needs to be uploaded. Alternatively, a directory of files can be indicated using the '-d' flag:

```
> python3 -m pdm_utils push -d ./new_data/
```

As with other pipelines, use of the *Database management configuration file* option can automate accessing the server to upload data.

4.12 review: review data for consistency

This tool is used to generate files used to review and revise gene product annotations, focusing on genes with similar amino acid sequences that have discrepant product annotations.

Specifically, the review pipeline focuses on genes within the same pham that have discrepant product annotations in order to review and revise product annotation inaccuracies that are due to human error or outdated data.

To export a csv spreadsheet of phams with discrepant product annotations and other relevant data from a database:

```
> python3 -m pdm_utils pham_review Actino_Draft
```

Like many other pipelines that involve exporting data in the `pdm_utils` package, the output of these phams can be modified with command line `filter` module implementations. Filtering, grouping, and sorting can be done in the same manner as described in the `export` module docs:

```
> python3 -m pdm_utils pham_review Actino_Draft -w "phage.Cluster = 'A'" -g phage.
↪Subcluster -s gene.Name
```

For attempting to revise specific phams that do not necessarily have genes with discrepant product annotations, the process of 'review' can be toggled off with the command-line flag **-r** or **–review**:

```
> python3 -m pdm_utils pham_review Actino_Draft -nr
```

References to the database that the review was performed on, and the profile of the phages of the genes within the phams selected for review can be obtained by generating a summary report with the command-line flag **-sr** or **–summary_report**:

```
> python3 -m pdm_utils pham_review Actino_Draft -sr
```

In addition, other supplementary information files that might help with the review process can be generated. Translation data and data about the respective phage genomes are available in the gene reports generated with the command-line flag **-gr** or **–gene_reports**:

```
> python3 -m pdm_utils pham_review Actino_Draft -gr
```

A comprehensive profile of the pham including Conserved Domain Database data and the most common annotations of adjacent genes can be generated using the command-line flag **-psr** or **–pham_summary_reports**:

```
> python3 -m pdm_utils pham_review Actino_Draft -psr
```

A complete review with all reports included can be done with the command-line flag **-a** or **–all_reports**:

```
> python3 -m pdm_utils pham_review Actino_Draft -a
```

The `pham_review` pipeline can also be targetted for gene data in the Actino_Draft database that has been recorded as submitted to GenBank and accessible for change as part of a pipeline of exchanging/updating data with GenBank:

```
> python3 -m pdm_utils pham_review Actino_Draft -p
```

4.13 revise: revise data inconsistencies

This tool is to be used in combination with the `pham_review` pipeline to generate files formatted to suit a GenBank automated gene product annotation resubmission pipeline.

Specifically, the local `revise` pipeline reads in edits made to a review csv spreadsheet and searches through an indicated database to find instances of genes selected for review with product annotations dissimilar to the indicated product annotation. The `revise` pipeline marks these, and generates a formatted file of the changes required to correct them.

To export the resubmission csv file from a edited review file:

```
> python3 -m pdm_utils revise Actino_Draft local <path/to/review/file>
```

To run the `revise` pipeline in production mode, which takes into account the various flags used to denote submitted GenBank files, utilize the production flag at the command line:

```
>python3 -m pdm_utils revise Actino_Draft local <path/to/review/file> --production
```

Like many other pipelines that involve exporting data in the `pdm_utils` package, the range of the database entries inspected by the `revise` pipeline can be modified with command line `filter` module implementations. Filtering, grouping, and sorting can be done in the same manner as described in the `export` pipeline documentation:

```
> python3 pdm_utils revise Actino_Draft local FunctionReport.csv -w "gene.Cluster NOT IN␣
↪('A', 'B', 'K')" -g phage.Cluster -s phage.PhageID
```

The local `revise` pipeline can also translate review formatted data into update ticket tables that can be used to update the database:

```
>python3 pdm_utils revise Actino_Draft local FunctionReport.csv -ft ticket
```

The remote `revise` pipeline retrieves data from GenBank in five-column feature table format and searches through an indicated database to find discrepancies between the product annotations and starts in the local database and those stored at GenBank. The `revise` pipeline marks these, and edits the retrieved GenBank files to generate five-column feature tables with feature data consistent with the local data.

To generate the revised five-column feature table format files:

```
> python3 -m pdm_utils revise Actino_Draft remote
```

And again, the remote `revise` pipeline can be modified with command line `filter` module implementations in the same manner as described in the `export` pipeline documentation:

```
> python3 pdm_utils revise Actino_Draft remove -w "gene.Subcluster IN ('K1', 'K2', 'K6')
```

4.14 update: make updates to specific database fields

Sometimes it is necessary to modify or update specific fields for specific phages. This can be accomplished using the `pdm_utils update` tool that relies on a specifically structured update ticket (*Update tickets*):

```
> python3 -m pdm_utils update Actino_Draft -f ./update_table.csv -c config_file.txt
```

The argument 'Actino_Draft' indicates the name of the database in which the updates should be implemented. The './update_table.csv' indicates the CSV-formatted table containing the list of update tickets. Each ticket is implemented

with very little QC. As with other pipelines, use of the *Database management configuration file* option can automate accessing MySQL.

Additionally, `update` can be used to quickly increment the database version, which needs to be changed every time changes are made in the database:

```
> python3 -m pdm_utils update Actino_Draft -v
```

Table 1: pdm_utils tools

Tool	Description	Schema bound	SEA-PHAGES oriented
compare	Directly compare phage data between a database instance, PhagesDB, and GenBank	Yes	Yes
convert	Upgrade or downgrade a database instance to another schema version	No	No
export	Export data from a database	Yes/No	No
find_domains	Identify NCBI conserved domains in genes	Yes	No
freeze	Create a derivative database instance that is no longer routinely updated	Yes	No
get_db	Retrieve the most up-to-date version of the database	No	No
get_data	Retrieve new data that needs to be imported or updated from PhagesDB and GenBank	Yes	Yes
get_gb_records	Retrieve GenBank records associated with genomes in the database	Yes	No
import	Import new or replacement genome annotations	Yes	Yes
phamerate	Group phage genes into phamilies based on amino acid sequence similarity	Yes	No
push	Push an updated database to a public server	No	No
review	Review gene description data consistency of a database	Yes	No
revise	Revise inconsistent data and prepare for submission to GenBank for updating records	Yes	No
update	Update specific fields	No	No

The `pdm_utils` toolkit can be used to manage different database instances. However, some tools may only be relevant specifically to the primary instance, Actino_Draft.

4.15 Supplementary files

Supplementary files associated with different tools:

4.15.1 Database management configuration file

Every pipeline can accept a configuration file that contains user-specific information to enable the pipeline to be run automatically, including MySQL login credentials, NCBI credentials (for rapid retrieval of GenBank records), and server information and credentials (to retrieve data from, or upload data to, a server).

The configuration file is formatted to meet requirements for the configparser module. Below is an example of how it is structured with a brief description of each parameter:

```
[ncbi]                      # Settings for connecting to NCBI server
email=jane@acme.com         # Your email address
api_key=123456789           # Your NCBI api key
tool=DataRetrievalTool      # The name of your script

[mysql]                     # Settings for connecting to local MySQL
user=janedoe                # Your MySQL login username
password=random             # Your MySQL login password

[upload_server]             # Settings to upload file(s) to a server
host=abc.def.com            # Name of the host server to upload file(s)
dest=/path/to/folder/       # Server path to where the file(s) will be uploaded
user=janedoe                # Your server login username
password=random             # Your server login password

[download_server]           # Settings to download file(s) from a server
url=http://abc.def.com/     # URL hosting the file to download
```

Not every pipeline requires all information in the config file, so in general, individual sections or rows are optional. If a pipeline needs information not present in the config file, it will prompt the user for the requisite information. For NCBI, if you do not provide an email address or api key, your GenBank queries may be throttled or blocked (please refer to NCBI for more details). An example config file is available on the `pdm_utils` source code repository.

4.15.2 Import tickets

A structured, database ticketing system is used to automate several steps in the database management process, including importing new data, making updates to the database, and maintaining a record of changes. For most types of changes to the database, there needs to be a unique 'ticket' in a csv-formatted import table that provides instructions on how to implement the change.

Ticket structure

Within the import table, an individual row of data populating 11 columns constructs a unique ticket.

1. **type**: there are currently two types of tickets that the import script implements:
    - 'add' if a new phage genome needs to be added to the database.
    - 'replace' if a phage genome currently in the database needs to be replaced with a new phage genome.
2. **phage_id**: the name of the new phage genome that the ticket addresses. Ensure the spelling of the phage name in the ticket precisely matches the spelling of the name in the flat file.
3. **description_field**: indicates the field in the CDS feature annotations in the associated Genbank-formatted flat file that is expected to contain the gene descriptions ('product', 'function', 'note').
4. **eval_mode**: indicates the *evaluation mode* ('draft', 'final', 'auto', 'misc', 'custom'), determining which QC checks to implement and thus how the flat file is evaluated.
5. **host_genus**: the genus of the bacterial host that the phage infects.
6. **cluster**: the Cluster designation for the phage, which should be:
    - the assigned Cluster, if it is in a Cluster.
    - 'Singleton' if the phage is not in a Cluster.

- 'UNK' if the Cluster has not yet been determined.

7. **subcluster**: the Subcluster designation for the phage, which should be:
   - the assigned Subcluster, if it is in a Subcluster.
   - 'none' if the phage is not in a Subcluster, or if the Cluster has not yet been determined.
8. **accession**: the accession for the GenBank record, which should be:
   - the assigned accession, if available.
   - 'none' if there is not accession.
9. **annotation_author**: indicates whether the end-user has control of the annotations, which should be:
   - '1' if the end-user has control of the annotations.
   - '0' if the end-user does not have control of the annotations.
10. **retrieve_record**: indicates whether the genome should be automatically updated when the associated GenBank record (if available) is updated. This should be:
    - '1' if new versions of the GenBank record should be retrieved.
    - '0' if new versions of the GenBank record should NOT be retrieved.
11. **annotation_status**: the stage of gene annotations, which should be:
    - 'draft' if the annotations have been automatically generated.
    - 'final' if the annotations have been manually generated.
    - 'unknown' if the strategy of annotation is not known.

Import tickets are automatically generated by `pdm_utils get_data`, but they can also be manually generated, such as the following add and replace tickets:

type	phage_id	description_field	eval_mode	host_genus	cluster	subcluster	accession	annotation_author	retrieve_record	annotation_status
add	Trixie	product	draft	Mycobacterium	A	A2	none	1	1	draft
replace	Finch	function	final	Rhodococcus	Singleton	none	MG962361	1	1	final

Note: The first row in the import table SHOULD be the column headers exactly as indicated above. Each subsequent row should represent a unique import ticket.

Ticket field options

1. Some fields can be set to pre-defined keywords for automatic data acquisition:
    - 'retrieve': for genomes that are also in PhagesDB, the data should be retrieved from PhagesDB.
    - 'retain': for genomes that are being replaced, the data already present in the database should be retained.
    - 'parse': for data that should be automatically parsed from the flat file.
2. Some fields can be set to the pre-defined keyword 'none' if they are not applicable for the ticket.
3. Since some field settings are commonly shared between all tickets, they can be omitted from the import table and set as a `import` command line argument instead.

The table below indicates which of the above options can be used for each ticket field:

type	phage_id	description_field	eval_mode	host_genus	cluster	subcluster	accession	annotation_author	retrieve_record	annotation_status
				retrieve	retrieve	retrieve	retrieve			
				retain	retain	retain	retain	retain	retain	
				parse			parse			
		command	command							
						none	none			

Automatic ticket construction

A simplified 'minimal' ticket can be used for adding and replacing genomes in which several fields are automatically populated when `import` is run:

- the 'type' and 'phage_id' fields need to be manually indicated in the import table.
- the 'description_field' and 'eval_mode' field settings are determined from the default command line arguments ('product' and 'final', respectively).
- for replace tickets, the 'annotation_status' is set to 'final' if the current genome is set to 'draft', otherwise it is set to 'retain'.

The table below indicates default settings for each type of ticket:

type	phage_id	description_field	eval_mode	host_genus	cluster	subcluster	accession	annotation_author	retrieve_record	annotation_status
add	<manual>	<command>	<command>	retrieve	retrieve	retrieve	retrieve	1	1	draft
replace	<manual>	<command>	<command>	retain	retain	retain	retain	retain	retain	final or retain

4.15.3 Update tickets

Within an update table, an individual row of data populating 5 columns constructs a unique 'ticket'.

1. **table**: the name of the database table to update.
2. **field**: the name of the table column to update.
3. **value**: the new value to be inserted into the column.
4. **key_name**: the name of the table column by which MySQL can identify which rows will be updated.
5. **key_value**: the value of the conditional column by which MySQL can identify which rows will be updated.

For example, the update ticket below…

table	field	value	key_name	key_value
phage	Cluster	A	PhageID	Trixie

…will result in the following MySQL statement…

UPDATE phage SET Cluster = 'A' WHERE PhageID = 'Trixie';

…and only the Cluster data for Trixie will be updated.

4.15.4 Evaluation modes and evaluation flags

Evaluation flags

Many `import` QC steps need to be performed on every genome (such as confirming the nucleotide sequence is not already present in the database under a separate name). However, a MySQL database may store data for diverse types of genomes, and some evaluations are dependent on factors such as the annotation status, the authorship, or the data source. As a result, some QC steps can be turned on (yes) and off (no) depending on the type of genome being imported.

check_replace

Should unexpected genome replacements be reported?

import_locus_tag

Should CDS feature locus_tags be imported?

check_locus_tag

Should the structure of CDS feature locus_tags be evaluated?

check_description_tally

Should the number of descriptions be evaluated?

check_description_field

Should CDS descriptions in unexpected fields be reported?

check_description

Should unexpected CDS descriptions be reported?

check_trna

Should tRNA features be evaluated?

check_id_typo

Should genome ID typos be reported?

check_host_typo

Should host typos be reported?

check_author

Should unexpected authors be reported?

check_gene

Should the CDS 'gene' qualifier be evaluated?

check_seq

Should the nucleotide sequence be evaluated?

check_coords

Should feature duplication be evaluated?

Evaluations modes

In order to manage which evaluations are implemented, the evaluation mode is specified for each ticket:

draft

For new, automatically-generated 'draft' annotations, since they are likely to have several types of automatically-generated errors that can be ignored.

final

For new, manually reviewed 'final' annotations that are expected to have very few errors.

auto

For manually reviewed 'final' annotations that are automatically retrieved from GenBank. Some errors are ignored since they are already publicly available.

misc

For genome annotations created from an external source. Since it is not always known how they have been annotated, and since any errors may not be able to be changed, certain types of errors are ignored.

custom

For manual selection of the specific evaluation flags that should be performed if none of the other four preset run modes are appropriate.

4.15.5 GenBank-formatted flat files

GenBank-formatted flat files are a common data format used to represent an entire phage genome, and a detailed description of this data structure can be found at NCBI website GenBank-formatted flat file. This is a structured text file that systematically stores diverse types of information about the genome.

A flat file can be generated for any genome at any annotation stage using:

1. GenBank
2. DNA Master
3. PECAAN
4. Biopython (*Cock et al., 2009*)

Flat file fields, such as LOCUS, DEFINITION, and REFERENCE-AUTHORS provide information regarding the entire record, while others, such as FEATURES, provide information about particular regions of the sequence in the record, such as tRNA or CDS genes. Data from flat files are stored in the *phage* and *gene* tables.

```
LOCUS       KF861510               52974 bp    DNA     linear   PHG 05-DEC-2017
DEFINITION  Mycobacterium phage EagleEye, complete genome.
ACCESSION   KF861510
VERSION     KF861510.1
KEYWORDS    .
SOURCE      Mycobacterium phage EagleEye
  ORGANISM  Mycobacterium phage EagleEye
            Viruses; dsDNA viruses, no RNA stage; Caudovirales; Siphoviridae;
            L5virus; unclassified L5virus.
REFERENCE   1  (bases 1 to 52974)
  AUTHORS   Awa,H., Bernal,J.T., Coelho,R.E., Culpepper,S.C., Devaraju,V.S.,
            Higgins,R.T., Husein,A.J., Johnston,E.M., Jung,J.A.,
            Kanani-Hendijani,T.A., Knapp,R.E., Lepiocha,N., McCarter,A.J.,
            Merlau,P.R., Monfared,M.S., Olney,H.P., Pineda,M.R., Pizzini,S.E.,
            Roberson,D.J., Rodriguez,J., Simpson,N.A., Stevens,S.C.,
            Stroub-Tahmassi,C.A., Syed,N., Torres,S.E., Townsend,C.W.,
            White,X.E., Willette,C.E., Deming,K.E., Simon,S.E., Benjamin,R.C.,
            Hughes,L.E., Hale,R.H., Lamson-Kim,T., Visi,D.H., Allen,M.S.,
            Bradley,K.W., Clarke,D.Q., Lewis,M.F., Barker,L.P., Bailey,C.,
            Asai,D.J., Garber,M.L., Bowman,C.A., Russell,D.A., Pope,W.H.,
            Jacobs-Sera,D., Hendrix,R.W. and Hatfull,G.F.
  TITLE     Direct Submission
  JOURNAL   Submitted (15-NOV-2013) Pittsburgh Bacteriophage Institute and
            Department of Biological Sciences, University of Pittsburgh, 365
            Crawford Hall, Pittsburgh, PA 15260, USA
COMMENT     Phage Isolation, DNA preparation, annotation analysis, and
            sequencing by Ion Torrent to approximately 90x coverage was
            performed at University of North Texas, Denton, TX
            Assembly performed with Newbler and Consed software as of Feb,
            2013.
            Supported by Science Education Alliance, Howard Hughes Medical
            Institute, Chevy Chase, MD.

            ##Assembly-Data-START##
            Assembly Method       :: Newbler and Consed v. Feb-2013
            Coverage              :: 90x
            Sequencing Technology :: Ion Torrent
            ##Assembly-Data-END##
```

```
FEATURES             Location/Qualifiers
     source          1..52974
                     /organism="Mycobacterium phage EagleEye"
                     /mol_type="genomic DNA"
                     /isolation_source="soil"
                     /db_xref="taxon:1429759"
                     /lab_host="Mycobacterium smegmatis mc2 155"
                     /country="USA: Houston, TX"
                     /lat_lon="29.880211 N 95.462524 W"
                     /collection_date="01-Oct-2012"
                     /collected_by="J. Bernal and T. Kanani"
                     /identified_by="J. Bernal and T. Kanani"

     tRNA            5584..5657
                     /gene="9"
                     /locus_tag="PBI_EAGLEEYE_9"
                     /product="tRNA-Trp"
                     /note="tRNA Trp (cca)"

     CDS             complement(45372..45914)
                     /gene="81"
                     /locus_tag="PBI_EAGLEEYE_81"
                     /note="homology to L5 Immunity Repressor"
                     /function="DNA binding"
                     /codon_start=1
                     /transl_table=11
                     /product="immunity repressor"
                     /protein_id="AHG23861.1"
                     /translation="MSGKTQQSGSFRAPLIFSVIEDLRRKGYNQSEIADMHGVTRQAV
                     SWQKKTYGGRMTPRDIVREAWPFETTNLHGKSVPFQRLRDHGEFMATGGAGMSENKIR
                     RLKAWWRKLRDEDVVLEFDPNIPPTPGMAGGGFRYVHRDIVTDGPDLLIRVNEHTAPM
                     TEKSEMIWSFPHDIEEILQS"

ORIGIN
        1 tgctgccgaa ccatcgtgta cgggttttca agtcgatcag aagaagaggc ctgcactgga
       61 aggcctcgga atgggcctgt gggccctcct ggctgacgaa cacccgctcc cgcgggtatc
      121 tgtaccccca aactcgccag cgggcttctg ggcccgttgg ctgtaaaggt gaactacctc
      181 acatttacag tgagcacctt gcgatactcc cgtatatata ttatgagggg ctgaaggccc
      241 ctctgaagag cgcctttagg gcgctcacta agaactaaag accggccttc gagggccggt
      301 catagaactt ggaacccgca accgcgggtt ctctggcggc gccagtgcgc cgcctaagag
      361 atcgggcgcc cagtggcgcc ctctaagggg tgttcactcg tgtttggcac tcgctcgagt
      421 gtcaactggg acactcaacc ggggaagttc gacgttctga acctgcggat gcggttcgac
      481 agctcgtccg agcacgagat ccccgacctg gccgcgaccg acttcgtgcc ggccaacctc
      541 gcggcgtgga atatgccgcg acatcgcgaa tacgccgcca tttcgggcgg cgctctgcac
```

Fig. 1: Summary of how a GenBank-formatted flat file is parsed

CHAPTER

FIVE

APPLICATIONS

`pdm_utils` is designed with several applications in mind:

5.1 Reviewing genome annotations

The Actino_Draft database is routinely updated with new genomics data. When new genome annotations need to to be imported into the database, they are processed using `pdm_utils import` which reviews the quality of the annotations and how they relate to data already present in the database. The *import pipeline* processes GenBank-formatted flat files and checks for a variety of potential errors, including:

1. Changes to the genome sequence
2. Phage name typos
3. Host name typos
4. Missing CDS locus tags
5. Incorrect protein translation tables used
6. Missing protein translations
7. Incorrect field used to store the gene descriptions
8. Incorrect tRNA genes

Note: `pdm_utils import` checks the quality of flat files for the purpose of maintaining database consistency and integrity. Although it does check the quality of many aspects of the genome, it is not intended for comprehensive evaluation of the quality, validity, or biological accuracy of all data stored in the flat file.

After creating the GenBank-formatted flat file, annotators can follow the steps below to review their files using this pipeline to verify that it contains all the necessary information to be successfully imported into the Actino_Draft database:

1. Ensure that MySQL is installed. If using a Mac, also ensure that the MySQL server is turned ON (*installation*).
2. Open a Terminal window.
3. If Conda is used to manage dependencies, activate the Conda environment. If Conda is not used, ensure all dependencies are installed (*installation*).
4. Ensure that the newest version of `pdm_utils` is installed (*installation*).
5. Ensure you have the most recent version of the Actino_Draft database, using *get_db*.

6. Create a folder (such as 'validation') to work in and navigate to it:

```
> mkdir validation
> cd ./validation
```

7. Within this new folder, create a csv-formatted import table (such as 'import_table.csv') of *import tickets*. For routine review of flat files to replace auto-annotated 'draft' genomes in the database, a simplified import table can be used consisting only of the 'type' and 'phage_id' fields. A template table is provided on the `pdm_utils` source code repository on GitHub. The ticket table should contain one ticket per flat file, in which:

    1. 'type' is set to 'replace'.
    2. 'phage_id' should be changed for each flat file.

    Example ticket table with 3 tickets:

type	phage_id
replace	Trixie
replace	L5
replace	D29

8. Create a new folder (such as 'genomes') within the validation folder to contain all flat files to be checked:

```
> mkdir genomes
```

9. Manually move all flat files into that folder. No other files should be present.
10. Run `import`. The pipeline requires you to indicate the name of the database, the folder of flat files, the import table, and where to create the output folder. Below is an example of the command that executes the script, assuming you are still in the 'validation' folder:

```
> python3 -m pdm_utils import Actino_Draft ./genomes/ ./import_table.csv -
↪o ./
```

Note: By default, the pipeline does not run in 'production' mode, so it does not actually make any changes to the database.

---

11. When prompted, provide your MySQL username and password to access your local Actino_Draft database.
12. The file is automatically processed, generating a log file of errors.
13. After the evaluation is complete, review specific errors in the log file if needed.
14. Repeat process if needed. After any errors are identified, re-create the flat files with the appropriate corrections, and repeat the import process to ensure the corrected file now passes validation.
15. Once everything is correct, upload the flat file to PhagesDB for official import into the database.

5.2 Create customized phage genomics databases

Below is a description of how the `pdm_utils` package can be used to create customized phage genomics databases using the same tools and pipelines implemented in the SEA-PHAGES program.

First, create a new, empty database with the most current database schema:

```
> python3 -m pdm_utils get_db PhageDatabase new
```

A new MySQL database, PhageDatabase, has been created, and it has the same structure as the most current Actino_Draft database.

Note: You may see two MySQL notices appear during installation: Warning: 3090, "Changing sql mode 'NO_AUTO_CREATE_USER' is deprecated. It will be removed in a future release." Warning: 1305, "PROCEDURE test_empty.split_subcluster does not exist" These warnings can be ignored.

If a different schema version is needed, this can be selected at the time of installation:

```
> python3 -m pdm_utils get_db PhageDatabase new -s 7
```

Alternatively, the `convert` tool can be used to subsequently downgrade the schema to a prior version.

Next, use `import` to add new genomics data to PhageDatabase…

5.3 Compatibility

As the production phage database schema changes and improves, downstream pipelines and tools designed for earlier schema versions may no longer be compatible. These tools may still be used as long as the newest database is first downgraded to the schema version that is compatible with the code.

Conversely, databases created for specific projects or publications using older schema versions may no longer be accessible by newer data analysis pipelines and tools that were designed for newer schema versions. These newer tools may still be used on older databases as long as the older databases are first upgraded to the schema version that is compatible with the analysis code.

To achieve both of these goals, databases can be converted (i.e. either upgraded or downgraded) along an incremental, linear schema history path using the `pdm_utils convert` tool. Refer to a description of this *conversion pipeline*.

5.4 Library tutorial

`pdm_utils` provides a library of functions, classes, and methods that leverage SQLAlchemy, a highly-refined, powerful, well-supported 'Database Toolkit for Python'. Tailoring SQLAlchemy tools to the pre-defined pdm_utils MySQL database provide for diverse entry points to the database. This tutorial provides a brief introduction to how the library can be used.

5.4.1 Set up connection to MySQL

The primary tool to connect to MySQL and access databases is the `pdm_utils` AlchemyHandler. It stores login credentials, manages connections, and stores different types of information about the database and MySQL environment.

Create an AlchemyHandler object:

```
>>> from pdm_utils.classes import alchemyhandler
>>> alchemist = alchemyhandler.AlchemyHandler()
```

The AlchemyHandler can take credentials by setting its various attributes:

```
>>> alchemist.username = "user123"
>>> alchemist.password = "p@ssword"
```

Alternatively, the connect() method prompts for the credentials:

```
>>> alchemist.connect()
MySQL username:
MySQL password:
```

Similarly, a specific database can be directly set:

```
>>> alchemist.database = "Actino_Draft"
```

or indirectly using the connect() method:

```
>>> alchemist.connect(ask_database=True)
MySQL database:
```

Engine

If login credentials are valid, a SQLAlchemy engine is created and stored in the engine attribute. The engine object provides the core interface between Python and MySQL. It stores information about the database of interest, manages connections, and contains methods to directly interact with the database. With the engine, the database can be queried using pure SQL syntax:

```
>>> engine = alchemist.engine
>>> result = engine.execute("SELECT PhageID, HostGenus FROM phage WHERE Subcluster = 'A2'
↪")
>>> phages = result.fetchall()
>>> len(phages)
90
>>> dict(phages[0])
{'PhageID': '20ES', 'HostGenus': 'Mycobacterium'}
```

MySQL transactions can also be executed using the engine. It returns 0 if successful, or 1 if unsuccessful:

```
>>> engine.execute("UPDATE phage SET HostGenus = 'Arthrobacter' WHERE PhageID = '20ES'")
>>> result = engine.execute("SELECT PhageID,HostGenus FROM phage WHERE Subcluster = 'A2'
↪")
>>> phages = result.fetchall()
>>> dict(phages[0])
{'PhageID': '20ES', 'HostGenus': 'Arthrobacter'}
```

The pdm_utils 'mysqldb_basic' module provides several functions that rely on the engine, such as retrieving the list of tables in the database, or the list of columns in one of the tables:

```
>>> from pdm_utils.functions import mysqldb_basic
>>> tables = mysqldb_basic.get_tables(engine, alchemist.database)
>>> tables
{'gene_domain', 'phage', 'gene', 'pham', 'tmrna', 'version', 'trna', 'domain'}
>>> columns = mysqldb_basic.get_columns(engine, alchemist.database, 'phage')
>>> columns
{'Length', 'Notes', 'Subcluster', 'Sequence', 'HostGenus', 'DateLastModified',
↪'RetrieveRecord', 'Cluster', 'Accession', 'AnnotationAuthor', 'GC', 'Status', 'Name',
↪'PhageID'}
```

Metadata

A SQLAlchemy MetaData object can also be created from a AlchemyHandler object with a valid credentials and database. A SQLAlchemy MetaData object allows for access to SQLAlchemy Base Table and Object classes, where the Tables can be directly accessed from the metadata object and the Columns from a Table:

```
>>> metadata = alchemist.metadata
>>> phage = metadata.tables["phage"]
>>> type(phage)
<class 'sqlalchemy.sql.schema.Table'>

>>> PhageID = phage.columns.PhageID
>>> type(PhageID)
<class 'sqlalchemy.sql.schema.Column'>
```

Table objects retrieved this way can be used to retrieve information about the structure of a table in the database the AlchemyHandler is connected to:

```
>>> metadata = alchemist.metadata
>>> list(metadata.tables)
['domain', 'gene', 'phage', 'pham', 'gene_domain', 'tmrna', 'trna', 'version']

>>> phage = metadata.tables["phage"]
>>> phage.name
'phage'

>>> primary_keys = phage.primary_key
>>> type(primary_keys)
<class 'sqlalchemy.sql.schema.PrimaryKeyConstraint'>

>>> dict(primary_keys.columns).keys()
dict_keys(['PhageID'])
```

Column objects retrieved from Table objects can be used to retrieve information about the characteristics of a Column within the table represented in the connected database:

```
>>> PhageID = phage.columns.PhageID
>>> PhageID.name
'PhageID'
>>> str(PhageID)
```

(continues on next page)

(continued from previous page)

```
'phage.PhageID'

>>> PhageID.type
VARCHAR(length=25)
>>> PhageID.nullable
False
>>> PhageID.primary_key
True
```

These Column and Table objects can be used to manually select, insert, or update data in a more object-oriented way when paired with an Engine object:

```
>>> phage = alchemist.metadata.tables["phage"]
>>> HostGenus = phage.columns.HostGenus
>>> PhageID = phage.columns.PhageID
>>> Subcluster = phage.columns.Subcluster

>>> query = phage.select(Subcluster == 'A2')
>>> result = alchemist.engine.execute(query)
>>> phages = result.fetchall()
>>> len(phages)
90

>>> from sqlalchemy import select

>>> query = select([PhageID, Subcluster]).where(Subcluster == 'A2')
>>> result = alchemist.engine.execute(query)
>>> phages = result.fetchall()
>>> len(phages)
90
```

To query for information by indirect relationship conditionals, Tables and Columns can be used to join tables to select from:

```
>>> phage = alchemist.metadata.tables["phage"]
>>> gene = alchemist.metadata.tables["gene"]

>>> Cluster = phage.columns.Cluster
>>> PhamID = gene.columns.PhamID

>>> from sqlalchemy import join
>>> joined_table = join(phage, gene, isouter=True)

>>> from sqlalchemy import select
>>> query = select([Cluster.distinct()]).select_from(joined_tables).where(PhamID == 2002)
>>> result = alchemist.engine.execute(query)
>>> clusters = result.fetchall()
>>> dict(clusters[0])
{'Cluster' : 'A'}
```

Database graph

An AlchemyHandler also has the ability to generate and store a graphical representation of the SQLAlchemy MetaData object as a `NetworkX` Graph object. The graph object has access to the same Table and objects as the MetaData as well as similar basic information:

```
>>> db_graph = alchemist.graph
>>> list(db_graph.nodes)
['domain', 'gene', 'phage', 'pham', 'gene_domain', 'tmrna', 'trna', 'version']

>>> phage_node = db_graph.nodes["phage"]
>>> phage = phage_node["table"]
>>> phage.name
'phage'
```

The graph object also stores information about the relationships between two tables, specifically the foreign key constraints between tables (and if joining two tables is possible):

```
>>> from networkx import shortest_graph
>>> db_graph = alchemist.graph
>>> shortest_path(db_graph, 'phage', 'domain')
['phage', 'gene', 'gene_domain', 'domain']

>>> foreign_key_edge = db_graph['phage']['gene']
>>> foreign_key = foreign_key_edge["key"]

>>> type(foreign_key)
<class 'sqlalchemy.sql.schema.ForeignKey'>
>>> foreign_key
ForeignKey('phage.PhageID')
```

Mapper

The AlchemyHandler provides support for using the SQLAlchemy ORM module and SQLAlchemy ORM objects based on the schema of the connected database. Access to the SQLAlchemy ORM objects is possible though the Automap Base object generated by the AlchemyHandler:

```
>>> mapper = alchemist.mapper

>>> Phage = mapper.classes["phage"]
>>> type(Phage)
<class 'sqlalchemy.ext.declarative.api.DeclarativeMeta'>
```

SQLAlchemy ORM objects have attributes that directly correspond to columns in the table that they represent, and these columns can be used in a similar way to the Base SQLAlchemy Column objects:

```
>>> mapper = alchemist.mapper
>>> Phage = mapper.classes["phage"]
>>> conditional = Phage.Subcluster == 'A2'

>>> Phage = alchemist.metadata.tables["phage"]
>>> query = phage.select(conditional)
>>> result = alchemist.engine.execute(query)
```

(continues on next page)

(continued from previous page)

```
>>> phages = result.fetchall()
>>> len(phages)
90
```

Session

SQLAlchemy ORM objects are extremely powerful when used in combination with the SQLAlchemy session object. The session object can be used with basic queries to create objects that represent entries in the database that store information as attributes named after the columns in the table which the ORM object represents. In addition, the ORM object instances can be created, updated, or deleted in a python environment, and the session object will manage and track the changes:

```
>>> session = alchemist.session
>>> Phage = alchemist.mapper.classes["phage"]

>>> phages = session.query(Phage).filter(Phage.Subcluster == 'A2')
>>> len(phages)
90

>>> phage[0].PhageID
'20ES'
>>> phage[0].HostGenus
'Mycobacterium'
```

5.4.2 Back-end ORM

A collection of customized classes represent the 'back-end' biology-centric ORM that is used to parse, evaluate, and exchange biological data between different data sources, including GenBank, PhagesDB, and MySQL. The objects leverage the highly-refined and well-supported BioPython package to coordinate data exchange as well as perform biology-related tasks (sequence extraction, translation, etc.). In contrast to the SQLAlchemy 'front-end' ORM, this ORM is only compatible with the most current `pdm_utils` database schema.

For information on how different classes in the bio-centric ORM map to different types of data sources, including the MySQL database, refer to the *bio-centric object relational mappings*.

Genome data

Data for specific phage genomes can be retrieved in an object-oriented structure using the 'mysqldb' module. First, create a list of phages for which data should be retrieved. These are expected to be stored in the PhageID column of the *phage* table:

```
>>> from pdm_utils.functions import mysqldb
>>> phage_id_list = ['L5', 'Trixie', 'D29']
```

Construct the MySQL query to retrieve the specific types of data from the *phage* table and the *gene* table:

```
>>> phage_query = 'SELECT PhageID, Name, Sequence, Cluster, Subcluster, Status,␣
↪HostGenus FROM phage'
>>> gene_query = 'SELECT GeneID, Start, Stop, Orientation, Translation, Notes FROM gene'
```

The mysqldb.parse_genome_data() function retrieves the data and constructs pdm_utils Genome, Cds, Trna, and Tmrna objects from the data. In the example below, there are three Genome objects created, each corresponding to a different phage in phage_id_list:

```
>>> phage_data = mysqldb.parse_genome_data(engine, phage_id_list=phage_id_list, phage_
↪query=phage_query, gene_query=gene_query)
>>> len(phage_data)
3
```

Data for each phage can be directly accessed:

```
>>> phage_data[0].id
'D29'
>>> d29 = phage_data[0]
>>> d29.host_genus
'Mycobacterium'
>>> d29.cluster
'A'
>>> d29.subcluster
'A2'
>>> d29.annotation_status
'final'
```

The genome sequence is stored in the seq attribute as a Biopython Seq object, so Biopython Seq attributes and methods (such as 'lower' or 'reverse_complement') can also be directly accessed:

```
>>> len(d29.seq)
49136
>>> d29.seq[:10]
Seq('GGTCGGTTAT')
>>> d29.seq[:10].lower()
Seq('ggtcggttat')
>>> d29.seq[:10].reverse_complement()
Seq('ATAACCGACC')
```

Cds data

Data from the *gene* table is retrieved and parsed into Cds objects. For each phage, all Cds objects are stored in the Genome object's 'cds_features' attribute as a list. Data for each CDS feature can be directly accessed:

```
>>> len(d29.cds_features)
77
>>> cds54 = d29.cds_features[54]
>>> cds54.description
'DNA primase'
>>> cds54.start
38737
>>> cds54.stop
39127
>>> cds54.orientation
'R'
>>> cds54.coordinate_format
'0_half_open'
```

Similar to the nucleotide sequence in the Genome object, the CDS translation is stored in the translation attribute as a Biopython Seq object:

```
>>> cds54.translation
Seq('MTATGIAEVIQRYYPDWDPPPDHYEWNKCLCPFHGDETPSAAVSYDLQGFNCLA...PWS', IUPACProtein())
```

The nucleotide sequence for each Cds feature is not explicitly stored in the MySQL database. The sequence can be extracted from the parent genome, but this relies on the Cds object containing a Biopython SeqFeature object stored in the seqfeature attribute, but this is also empty at first:

```
>>> cds54.seq
Seq('', IUPACAmbiguousDNA())
>>> cds54.seqfeature
```

To extract the sequence, first construct the Biopython SeqFeature object:

```
>>> cds54.set_seqfeature()
>>> cds54.seqfeature
SeqFeature(FeatureLocation(ExactPosition(38737), ExactPosition(39127), strand=-1), type=
↪'CDS')
```

With the SeqFeature constructed, the 390 bp nucleotide sequence can be retrieved from the parent genome:

```
>>> cds54.set_nucleotide_sequence(parent_genome_seq=d29.seq)
>>> cds54.seq
Seq('TTGACAGCCACCGGCATCGCGGAGGTCATCCAGCGGTACTACCCGGACTGGGAT...TGA')
>>> len(cds54.seq)
390
```

Trna and Tmrna data

Similar to CDS data, data from the *trna* and *tmrna* tables are retrieved and parsed into Trna and Tmrna objects, and stored in the Genome.trna_features and Genome.tmrna_features attributes, respectively. Each class contains a variety of methods to validate and manipulate this type of data.

Source data

Similar to CDS data, data from source features in GenBank-formatted flat files are parsed into Source objects. There is no equivalent source table in the database, but the class contains a variety of methods to validate and manipulate this type of data from flat files.

5.4.3 Filtering and dynamic querying

By combining the SQLAlchemy Core Metadata object with graphing tools from the NetworkX package, `pdm_utils` can support dynamic querying of the database and filtering of data.

Dynamic MySQL joining and query construction

The Filter can dynamically accept a column and table inputs since it combines SQLAlchemy Metadata with a NetworkX graph. The graph describes the relationships and constraints between the tables of a database and allows for identifying the relational path between the tables. The graph thus acts as a guide for joining database tables during MySQL query construction. To create this graph, you can use a `pdm_utils` AlchemyHandler object connected to a MySQL database:

```
>>> from pdm_utils.classes import alchemyhandler
>>> alchemist = alchemyhandler.AlchemyHandler()
>>> alchemist.username = "user123"
>>> alchemist.password = "p@ssword"
>>> alchemist.database = "Actino_Draft"
>>> alchemist.connect()
>>> graph = alchemist.graph
```

The graph can be used in conjunction with SQLAlchemy Column objects to dynamically create SQLAlchemy Select objects that represent a MySQL statement. SQLAlchemy Column and Table objects can be retrieved from a SQLAlchemy Metadata object:

```
>>> metadata = alchemist.metadata
>>> phage_table = metadata.tables["phage"]
>>> phageid_column = phage_table.columns["PhageID"]
```

Alternatively, SQLAlchemy Columns can be retrieved from case-insensitive MySQL formatted inputs using the `pdm_utils` 'querying' module:

```
>>> from pdm_utils.functions import querying
>>> metadata = alchemist.metadata
>>> phageid = querying.get_column(metadata, "PHAGE.phageID")
```

Columns can be used with the `pdm_utils` 'querying' module and the graph object to create SQLAlchemy Select statements that dynamically join the required tables:

```
>>> from pdm_utils.functions import querying
>>> cluster_column = querying.get_column(alchemist.metadata, "phage.cluster")
>>> domain_name_column = querying.get_column(alchemist.metadata, "domain.name")
>>> gene_notes_column = querying.get_column(alchemist.metadata, "gene.notes")
>>> columns = [cluster_column, domain_name_column, gene_notes_column]
>>> select_statement = querying.build_select(alchemist.graph, columns)
>>> results_proxy = alchemist.engine.execute(select_statement)
```

Columns can be used to create conditionals to narrow the results of a select statement, populating the MySQL WHERE clause of the statement. The `pdm_utils` 'querying' module incorporates the tables of the columns used in the conditionals when joining tables to allow for easier addition of conditionals with indirect relationships:

```
>>> domain_name_column = querying.get_column(alchemist.metadata, "domain.name")
>>> domain_name_conditional = (domain_name_column == "HTH_26")
>>> columns = [cluster_column, gene_notes_column]
>>> select_statement = querying.build_select(alchemist.graph, columns, where=domain_name_
↪conditional)
>>> results_proxy = alchemist.engine.execute(select_statement)
```

Alternatively, SQLAlchemy ORM mapped objects can be retrieved from MySQL formatted conditional inputs using the `pdm_utils` 'querying' module build_where_clause() function and a graph object:

```
>>> domain_name_conditional = querying.build_where_clause(alchemist.graph, "domain.name␣
↪= 'HTH_26'")
>>> columns = [cluster_column, gene_notes_column]
>>> select_statement = querying.build_select(alchemist.graph, columns, where=domain_name_
↪conditional)
>>> results_proxy = alchemist.engine.execute(select_statement)
```

Select statements can be executed by the execute() function of the pdm_utils 'querying' module. This allows for input of values to condition the query on and automatically retrieves the results from the results proxy. populating the MySQL IN clause of the statement. The execute() comes with safer procedures for IN clause statement and automatically subqueries if the number of values exceeds the average query length limit for MySQL statements:

```
>>> domain_name_column = querying.get_column(alchemist.metadata, "domain.name")
>>> gene_notes_column = querying.get_column(alchemist.metadata, "gene.notes")
>>> phageid_column = querying.get_column(alchemist.metadata, "phage.PhageID")
>>> columns = [gene_notes_column, gene_notes_column]
>>> select_statement = querying.build_select(alchemist.graph, columns)
>>> results = querying.execute(alchemist.engine, select_statement, in_column=phageid_
↪column, values=["Trixie", "D29", "Myrna", "Alice"])
```

The pdm_utils Filter object wraps much of the querying functionality to add a layer of abstraction (see below for more details):

```
>>> db_filter.key = "phage.PhageID"
>>> db_filter.values = ["Trixie", "D29", "Myrna", "Alice"]
>>> results = db_filter.select(["domain.name", "gene.notes"])
```

Access subsets of data using a pdm_utils Filter

The pdm_utils Filter object enables iterative querying, sorting, and retrieving of subsets of data:

```
>>> from pdm_utils.classes import filter
>>> db_filter = filter.Filter()
```

A filter object can connect to a MySQL database by calling a method Filter.connect():

```
>>> db_filter = filter.Filter()
>>> db_filter.connect()
```

Alternatively, a valid AlchemyHandler can be provided when the filter is instantiated:

```
>>> db_filter = filter.Filter(alchemist=alchemy_handler_obj)
```

A filter object requires a key to retrieve values from, which can be set using the attribute Filter.key and passing in a MySQL formatted column:

```
>>> db_filter.key = "phage.PhageID"
```

Alternatively, a SQLAlchemy Metadata Column object can be passed:

```
>>> db_filter.key = PhageID_Column_obj
>>> db_filter.key
Column('PhageID', VARCHAR(length=25), table=<phage>, primary_key=True, nullable=False)
```

A filter object key can also be set by passing in the name of a table as a string, and the filter key will be the primary key of that table:

```
>>> db_filter.key = "phage"
>>> db_filter.key
Column('PhageID', VARCHAR(length=25), table=<phage>, primary_key=True, nullable=False)
```

The filter retrieves values from the database depending on previously-retrieved values and given conditionals. New conditionals can be applied using a method Filter.add() and MySQL syntax. Ex. Creating the Subcluster filter identifies 90 phages in Subcluster A2:

```
>>> db_filter.add("phage.Subcluster = 'A2'")
>>> db_filter.update()
>>> db_filter.hits()
90
```

The method Filter.add() can take a number of expressions with different operators including !=, >, <, >=, <=, LIKE, IS NOT, IN, and NOT IN:

```
>>> db_filter.add("phage.Subcluster IN ('A2')")
>>> db_filter.update()
>>> db_filter.hits()
90
```

The filter results are stored in the values attribute, and can be sorted and accessed:

```
>>> db_filter.sort("phage.PhageID")
>>> len(db_filter.values)
90
>>> db_filter.values[:10]
['20ES', 'AbbyPaige', 'Acolyte', 'Adzzy', 'AN3', 'AN9', 'ANI8', 'AnnaL29', 'Anselm',
↪'ArcherNM']
```

This list of PhageIDs can now be passed to other functions, such as mysqldb.parse_genome_data(). The filtered results can be subsequently filtered if needed. Suppose that only Subcluster A2 phages that contain at least one gene that is annotated as the 'repressor' are needed. This filter can be added, resulting in a list of only 4 phages:

```
>>> db_filter.add("gene.Notes = 'repressor'")
>>> db_filter.update()
>>> db_filter.hits()
4
>>> db_filter.values
['Pukovnik', 'RedRock', 'Odin', 'Adzzy']
```

The same list of PhageIDs can be retrieved by adding conjunctions to the added conditionals:

```
>>> db_filter.add("gene.Notes = 'repressor' AND phage.Subcluster = 'A2'")
>>> db_filter.update()
>>> db_filter.hits()
4
>>> db_filter.values
['Pukovnik', 'RedRock', 'Odin', 'Adzzy']
```

OR conjunctions can be used with these statements, and OR conjunctions work in the same way that MySQL OR's do: AND conjunctions take precedence and are processed first before OR conjunctions:

```
>>> db_filter.add("gene.Notes = 'repressor' AND phage.Subcluster = 'A2' OR gene.Notes =
↪'repressor' AND phage.Subcluster = 'C1'")
>>> db_filter.update()
>>> db_filter.hits()
5
>>> db_filter.values
['Pukovnik', 'RedRock', 'Odin', 'Adzzy', 'LRRHood']
```

To get the distinct set of values from another column related to the current set of values, the method Filter.transpose() accepts a column input and returns the distinct values in a list:

```
>>> db_filter.values
['Trixie', 'D29', 'Myrna', 'Alice']
>>> db_filter.transpose("phage.Cluster")
['A', 'C']
```

Filter.transpose() has an optional parameter that allow you to switch a Filter's key and values:

```
>>> db_filter.values
['Trixie', 'D29', 'Myrna', 'Alice']
>>> db_filter.key
Column('PhageID', VARCHAR(length=25), table=<phage>, primary_key=True, nullable=False)
>>> db_filter.transpose("phage.Cluster", set_values=True)
['A', 'C']
>>> db_filter.values
['A', 'C']
>>> db_filter.key
Column('Cluster', VARCHAR(length=5), table=<phage>)
```

To get the distinct set of values from multiple columns related to the current set of values, the method Filter.mass_transpose() accepts column input(s) and returns the values in a dictionary where the keys are the names of the entered columns, and the values are the respective distinct values in a list:

```
>>> db_filter.values
['Trixie', 'D29', 'Myrna', 'Alice']
>>> db_filter.mass_transpose(["phage.Cluster",  "phage.Subcluster"])
{'Cluster' : ['A', 'C'], 'Subcluster' : ['A2', 'C1', 'C2']}
```

To get the distinct set of values from multiple columns for each individual value in the set of values, the method Filter.retrieve() accepts a column input and returns a dictionary where the keys are the set of values and the values are dictionaries where the keys are the names of the entered columns, and the values are the respective distinct values in a list:

```
>>> db_filter.values
['Trixie', 'Myrna', 'Alice']
>>> db_filter.retrieve(["phage.Cluster", "phage.Subcluster"])
{'Trixie' : {'Cluster' : ['A'], 'Subcluster' : ['A2']}, 'Myrna' : {'Cluster' : ['C'],
↪'Subcluster' : ['C2']}, 'Alice' : {'Cluster' : ['C'], 'Subcluster' : ['C1']}}
```

To group the current set of values based on the distinct set of values related to the current set of values, the method Filter.group() accepts a column input and returns the values in a dictionary where the keys are the set of related distinct columns and the values are the respective subset of the current set of values in a list:

```
>>> db_filter.values
['Trixie', 'D29', 'Myrna', 'Alice']
>>> db_filter.group("phage.Cluster")
{'A' : ['Trixie', 'D29'], 'C' : ['Myrna', 'Alice']}
```

5.4.4 Front-end ORM

In contrast to the customized 'back-end' biology-centric ORM, an orthologous and distinct 'front-end' SQLAlchemy ORM is available through `pdm_utils`. This ORM is automatically generated from SQLAlchemy and consists of a collection of classes that have a direct relationship to the structure of the database, including its tables, columns, datatypes, and table relationships.

Classes in this ORM do not contain any special biology-related methods. However, in contrast to the bio-centric ORM, the 'front-end' ORM is completely agnostic to the schema version. This means that `pdm_utils` can be used to retrieve a database, convert the schema to any version in the schema history, and a perfectly consistent ORM is immediately available.

This ORM is automatically generated by SQLAlchemy and `pdm_utils` merely provides the lightweight, requisite, boilerplate code to connect this software to a specific phage database instance. As a result, `pdm_utils` serves as a thin wrapper for SQLAlchemy, and can get out of the way so that the end-user can solely rely on and leverage the powerful, highly-refined and well-supported SQLAlchemy toolkit to build novel pipelines.

Introduction to the SQLAlchemy ORM

The SQLAlchemy ORM is a powerful tool for modifying and exploring a database. Access to the ORM is available through the 'mapper' attribute of the `pdm_utils` AlchemyHandler object:

```
>>> from pdm_utils.classes import alchemyhandler
>>> alchemist = alchemyhandler.AlchemyHandler()
>>> alchemist.connect(ask_database=True)
>>> mapper = alchemist.mapper
>>> type(mapper)
<class 'sqlalchemy.ext.declarative.api.DeclarativeMeta'>
```

Get a list of classes representing all available tables in the database:

```
>>>  mapper.classes.keys()
['domain', 'gene', 'phage', 'pham', 'gene_domain', 'tmrna', 'trna', 'version']
```

Select the phage table class:

```
>>> Phage = mapper.classes["phage"]
>>> type(Phage)
<class 'sqlalchemy.ext.declarative.api.DeclarativeMeta'>
```

Create a new entry (e.g. "new_phage") for the *phage* table, assigning values to each valid column:

```
>>> new_phage = Phage(PhageID="NewPhage", Name="NewPhage_Draft", HostGenus="Mycobacterium
↪", Cluster="A", Subcluster="A2", Sequence="AAAA")
>>> type(new_phage)
<class 'sqlalchemy.ext.automap.phage'>
>>> new_phage.Cluster
'A'
```

The SQLAlchemy 'session' object uses these ORM classes to query for data, add new entries, update entries, and remove entries. A session object can be used to retrieve instances that are tracked by the session. A session can be created using the `pdm_utils` AlchemyHandler object:

```
>>> session = alchemist.session
>>> type(session)
<class 'sqlalchemy.orm.session.Session'>
```

Note: The session is a powerful object that manages access to the database. Please refer to SQLAlchemy documentation to learn more about what it does and how to properly implement it.

Query the database for phages in Cluster A:

```
>>> query = session.query(Phage).filter(Phage.Cluster=="A")
>>> phages = query.all()
>>> type(phages)
<class 'list'>
>>> len(phages)
643
>>> phage1 = phages[0]
>>> phage1.PhageID
'20ES'
>>> phage1.Subcluster
'A2'
```

SQLAlchemy automatically retrieves data from connected tables such as *gene* and *trna*:

```
>>> len(phage1.gene_collection)
96
>>> cds1 = phage1.gene_collection[0]
>>> cds1.GeneID
'20ES_CDS_1'
>>> cds1.Start
568
```

Now query the database for phages that are in Cluster A and that have CDS features annotated as "repressor" in the gene table:

```
>>> Gene = mapper.classes["gene"]
>>> query = session.query(Phage).join(Gene).filter(Phage.Cluster=="A").filter(Gene.
↪Notes=="repressor".encode("utf-8"))
>>> phages = query.all()
>>> len(phages)
38
```

When done with the session, commit any changes:

```
>>> session.commit()
```

pdm_utils tools based on the SQLAlchemy ORM

Several tools in pdm_utils can leverage the SQLAlchemy ORM to provide additional functionalities.

For instance, SQLAlchemy ORM mapped objects can be retrieved from case-insensitive MySQL formatted inputs using the pdm_utils 'cartography' module:

```
>>> from pdm_utils.functions import cartography
>>> phage = cartography.get_map(alchemist.mapper, "PHAGE")
```

Python objects, ORM map instances, that reflect entries in the database can be retrieved using the session object, and conditionals to filter the results of the query can be formed using attributes of the retrieved ORM map.:

```
>>> phage = cartography.get_map(alchemist.mapper, "phage")
>>> trixie = alchemist.session.query(phage).filter(phage.PhageID == 'Trixie').scalar()
```

The retrieved instance is a reflection of a single entry tied to the primary_key from the mapped table specified, and has attributes that reflect the related columns of that table:

```
>>> trixie.PhageID
'Trixie'
>>> trixie.Cluster
'A'
>>> trixie.Length
53526
```

The dynamic querying from pdm_utils 'querying' module can be applied to SQLAlchemy ORM queries using the query() function, and SQLAlchemy base objects and conditionals can be incorporated from the querying module into ORM queries to generate ORM objects:

```
>>> phage = cartography.get_map(alchemist.mapper, "phage")
>>> subcluster_conditional = phage.Subcluster == 'A2'
>>> notes_conditional = querying.build_where_clause(alchemist.graph, "phage.Notes =
↪'antirepressor'")
>>> conditionals = [subcluster_conditional, notes_conditional]
>>> mapped_obj_instances = querying.query(alchemist.session, alchemist.graph, phage,␣
↪where=conditionals)
>>> phage_instance = mapped_obj_instances[0]
>>> phage_instance.PhageID
'IronMan'
```

Additionally, the pdm_utils Filter object can be used to retrieve these mapped instances. The filter object can apply filters and retrieve a list of values that can be used to retrieve a similar set of mapped obj instances:

```
>>> phage = cartography.get_map(alchemist.mapper, "phage")
>>> db_filter.add("phage.Subcluster = 'A2' AND gene.Notes = 'antirepressor'")
>>> db_filter.update()
>>> mapped_obj_instances = db_filter.query(phage)
>>> phage_instance = mapped_obj_instances[0]
>>> phage_instance.PhageID
'IronMan'
```

The SQLAlchemy session tracks instances generated through these queries, and can be used to manually manage entries in the database:

```
>>> phage = cartography.get_map(alchemist.mapper, "phage")
>>> IronMan = alchemist.session.query(phage).filter(phage.PhageID == 'IronMan').scalar()
>>> IronMan.DateLastModified
datetime.datetime(2020, 3, 13, 0, 0)
>>> from datetime import datetime
>>> today = datetime(2020, 5, 22, 0, 0)
>>> IronMan.DateLastModified = today
>>> alchemist.session.commit()
>>> IronMan = alchemist.session.query(phage).filter(phage.PhageID == 'IronMan').scalar()
>>> IronMan.DateLastModified
datetime.datetime(2020, 5, 22, 0, 0)
```

Once references to instances have been acquired using the session, entries in the database can also be deleted:

```
>>> IronMan = alchemist.session.query(phage).filter(phage.PhageID == 'IronMan').scalar()
>>> alchemist.session.delete(IronMan)
>>> alchemist.session.query(phage).filter_by(PhageID='IronMan').count()
0
```

The SQLAlchemy map can also be used to instantiate new objects that are then added as entries to the database:

```
>>> phage = cartography.get_map(alchemist.mapper, "phage")
>>> Phabulous = phage(PhageID='Phabulous', Cluster='A', Subcluster='A2', Length=52342)
>>> alchemist.session.commit()
>>> alchemist.session.query(phage).filter_by(PhageID='Phabulous').count()
1
```

SQLAlchemy mapped instances generated from a session also have access to the data that the relevant entry has a relationship with:

```
>>> phage = cartography.get_map(alchemist.mapper, "phage")
>>> IronMan = alchemist.session.query(phage).filter_by(PhageID='IronMan').scalar()
>>> IronMan_genes = IronMan.gene_collection
>>> IronMan_gene1 = IronMan_genes[0]
>>> IronMan_gene1.PhageID
'IronMan'
>>> IronMan_gene1.Name
1
>>> IronMan_gene1.GeneID
IronMan_CDS_1
```

These instances retrieved from the relationship attributes of another mapped instance can likewise be updated or deleted with use of the SQLAlchemy session:

```
>>> IronMan = alchemist.session.query(phage).filter_by(PhageID='IronMan').scalar()
>>> IronMan_gene1 = IronMan.gene_collection[0]
>>> IronMan_gene1.PhamID
42415
>>> IronMan_gene1.PhamID = 54326
>>> alchemist.session.commit()
```

When all interaction with MySQL is complete, the DBAPI connections can be closed:

```
>>> engine.dispose()
```

Each section of the tutorial assumes a Python IDE has been initiated within a terminal environment with all dependencies available. In the shell terminal, activate the Conda environment containing the installed `pdm_utils` package (if needed) to ensure all dependencies are present. Then open a Python IDE:

```
> conda activate pdm_utils
(pdm_utils)>
(pdm_utils)> python3
>>>
```

CHAPTER

SIX

PDM_UTILS PACKAGE

6.1 Overview

The `pdm_utils` source code is structured within a general hierarchical strategy in mind:

- Submodules in the 'constants' subpackage define global variables for the entire package, and they do not import any other `pdm_utils` or third-party modules.
- Submodules in the 'classes' subpackage only import from the 'constants' subpackage, the 'basic' submodule in the 'functions' subpackage, or third-party modules. In general, each submodule defines one class.
- Submodules in the 'functions' subpackage (other than the 'basic' submodule) contain functions that are either stand-alone, that rely on third-party packages, or that manipulate `pdm_utils` objects.
- Submodules in the 'pipelines' subpackage contain programs that can be executed as command line tools that are meant to run from start to completion, and that can import all other submodules in the other subpackages. The most common entry point for these pipelines is the main() function. These submodules are functionalized such that custom Python tools can create alternative pipeline entry points or utilize individual functions.
- The 'run' module' controls the command line toolkit. It is not intended to be called from other tools.

6.2 Subpackages

6.2.1 constants

constants

Collection of constants and dictionaries used for maintaining the phage database.

db_schema_0

Empty database schema DDL for schema version 0.

eval_descriptions

Import evaluations descriptions.

pdm_utils.constants.eval_descriptions.**get_string**(*value_set*)

schema_conversions

Database schema conversion steps.

6.2.2 classes

Each submodule contains a specific class.

Database ORM

cds

Represents a collection of data about a CDS features that are commonly used to maintain and update SEA-PHAGES phage genomics data.

class pdm_utils.classes.cds.**Cds**
Bases: object

Class to hold data about a CDS feature.

check_amino_acids(*check_set={}*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check whether all amino acids in the translation are valid.

Parameters

- **check_set** (*set*) – Set of valid amino acids.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_attribute(*attribute*, *check_set*, *expect=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the attribute value is valid.

Parameters

- **attribute** (*str*) – Name of the CDS object attribute to evaluate.
- **check_set** (*set*) – Set of reference ids.
- **expect** (*bool*) – Indicates whether the attribute value is expected to be present in the check set.
- **eval_id** (*str*) – Unique identifier for the evaluation.
- **success** (*str*) – Default status if the outcome is a success.
- **fail** (*str*) – Default status if the outcome is not a success.
- **eval_def** (*str*) – Description of the evaluation.

check_compatible_gene_and_locus_tag(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if gene and locus_tag attributes contain identical numbers.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_description_field(*attribute='product'*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if there are CDS descriptions in unexpected fields.

Evaluates whether the indicated attribute is empty or generic, and other fields contain non-generic data.

Parameters

- **attribute** (*str*) – Indicates the reference attribute for the evaluation ('product', 'function', 'note').
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_gene_structure(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if the gene qualifier contains an integer.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_generic_data(*attribute=None*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if the indicated attribute contains generic data.

Parameters

- **attribute** (*str*) – Indicates the attribute for the evaluation ('product', 'function', 'note').
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_locus_tag_structure(*check_value=None*, *only_typo=False*, *prefix_set={}*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if the locus_tag is structured correctly.

Parameters

- **check_value** (*str*) – Indicates the genome id that is expected to be present. If None, the 'genome_id' parameter is used.

- **only_typo** (*bool*) – Indicates if only the genome id spelling should be evaluated.
- **prefix_set** (*set*) – Indicates valid common prefixes, if a prefix is expected.
- **case** (*bool*) – Indicates whether the locus_tag is expected to be capitalized.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_magnitude(*attribute*, *expect*, *ref_value*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the magnitude of a numerical attribute is valid.

Parameters

- **attribute** – same as for check_attribute().
- **expect** (*str*) – Comparison symbol indicating direction of magnitude (>, =, <).
- **ref_value** (*int, float, datetime*) – Numerical value for comparison.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_orientation(*format='fr_short'*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check if orientation is set appropriately.

Relies on the *reformat_strand* function to manage orientation data.

Parameters

- **format** (*str*) – Indicates how coordinates should be formatted.
- **case** (*bool*) – Indicates whether the orientation data should be cased.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_translation(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the current and expected translations match.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

create_seqfeature(*type*, *start*, *stop*, *strand*)

get_begin_end()
Get feature coordinates in transcription begin-end format.

Returns (Begin, End) Start and stop coordinates ordered by which coordinate indicates the transcriptional beginning and end of the feature.

Return type tuple

get_qualifiers(*type*)
Helper function that uses cds data to populate the qualifiers SeqFeature attribute

Returns qualifiers(dictionary) is a dictionary with the formating of BioPython's SeqFeature qualifiers attribute.

reformat_start_and_stop(*new_format*)
Convert start and stop coordinates to new coordinate format. This also updates the coordinate format attribute to reflect change.

Relies on the *reformat_coordinates* function.

Parameters **new_format** (*str*) – Indicates how coordinates should be formatted.

set_description(*value*)
Set the primary raw and processed description attributes.

Parameters **value** (*str*) – Indicates which reference attributes are used to set the attributes ('product', 'function', 'note').

set_description_field(*attr*, *description*, *delimiter=None*, *prefix_set=None*)
Set a description attribute parsed from a description.

Parameters

- **attr** (*str*) – Attribute to set the description.
- **description** (*str*) – Description data to parse. Also passed to set_num().
- **delimiter** (*str*) – Passed to set_num().
- **prefix_set** (*set*) – Passed to set_num().

set_eval(*eval_id*, *definition*, *result*, *status*)
Constructs and adds an Evaluation object to the evaluations list.

Parameters

- **eval_id** (*str*) – Unique identifier for the evaluation.
- **definition** (*str*) – Description of the evaluation.
- **result** (*str*) – Description of the outcome of the evaluation.
- **status** (*str*) – Outcome of the evaluation.

set_gene(*value*, *delimiter=None*, *prefix_set=None*)
Set the gene attribute.

Parameters

- **value** (*str*) – Gene data to parse. Also passed to set_num().
- **delimiter** (*str*) – Passed to set_num().
- **prefix_set** (*set*) – Passed to set_num().

set_location_id()
Create a tuple of feature location data.

For start and stop coordinates of the feature, it doesn't matter whether the feature is complex with a translational frameshift or not. Retrieving the "start" and "stop" boundary attributes return the very beginning and end of the feature, disregarding the inner "join" coordinates. If only the feature transcription "end" coordinate is used, orientation information is required. If transcription "begin" and "end" coordinates are used instead of "start" and "stop" coordinates, no orientation information is required.

set_locus_tag(*tag=''*, *delimiter='_'*, *check_value=None*)
Set locus tag and parse the locus_tag feature number.

Parameters

- **tag** (*str*) – Input locus_tag data.
- **delimiter** (*str*) – Value used to split locus_tag data.
- **check_value** (*str*) – Indicates genome name or other value that will be used to parse the locus_tag to identify the feature number. If no check_value is provided, the genome_id attribute is used.

set_name(*value=None*)
Set the feature name.

Ideally, the name of the CDS will be an integer. This information can be stored in multiple fields in the GenBank-formatted flat file. The name is derived from one of several qualifiers.

Parameters **value** (*str*) – Indicates a value that should be used to directly set the name regardless of the 'gene' and '_locus_tag_num' attributes.

set_nucleotide_length(*seq=False*, *translation=False*)
Set the length of the nucleotide sequence.

Nucleotide length can be computed several different ways, including from the difference of the start and stop coordinates, the length of the transcribed nucleotide sequence, or the length of the translation. For compound features, using either the nucleotide or translation sequence is the accurate way to determine the true length of the feature, but 'length' may mean different things in different contexts.

Parameters

- **seq** (*bool*) – Use the nucleotide sequence from the 'seq' attribute to compute the length.
- **translation** (*bool*) – Use the translation sequence from the 'translation' attribute to compute the length.

set_nucleotide_sequence(*value=None*, *parent_genome_seq=None*)
Set the nucleotide sequence of the feature.

This method can directly set the attribute from a supplied 'value', or it can retrieve the sequence from the parent genome using Biopython. In this latter case, it relies on a Biopython SeqFeature object for the sequence extraction method and coordinates. If this object was generated from a Biopython-parsed GenBank-formatted flat file, the coordinates are by default '0-based half-open', the object contains coordinates for every part of the feature (e.g. if it is a compound feature) and fuzzy locations. As a result, the length of the retrieved sequence may not exactly match the length indicated from the 'start' and 'stop' coordinates. If the nucleotide sequence 'value' is provided, the 'parent_genome_seq' does not impact the result.

Parameters

- **value** (*str of Seq*) – Input nucleotide sequence
- **parent_genome_seq** (*Seq*) – Input parent genome nucleotide sequence.

set_num(*attr*, *description*, *delimiter=None*, *prefix_set=None*)
Set a number attribute from a description.

Parameters

- **attr** (*str*) – Attribute to set the number.
- **description** (*str*) – Description data from which to parse the number.
- **delimiter** (*str*) – Value used to split the description data.
- **prefix_set** (*set*) – Valid possible delimiters in the description.

set_orientation(*value*, *format*, *case=False*)
Sets orientation based on indicated format.

Relies on the *reformat_strand* function to manage orientation data.

Parameters

- **value** (*misc.*) – Input orientation value.
- **format** (*str*) – Indicates how the orientation data should be formatted.
- **case** (*bool*) – Indicates whether the output orientation data should be cased.

set_seqfeature(*type='CDS'*)
Set the 'seqfeature' attribute.

The 'seqfeature' attribute stores a Biopython SeqFeature object, which contains methods valuable to extracting sequence data relevant to the feature.

set_translation(*value=None*, *translate=False*)
Set translation and its length.

The translation is coerced into a Biopython Seq object. If no input translation value is provided, the translation is generated from the parent genome nucleotide sequence. If an input translation value is provided, the 'translate' parameter has no impact.

Parameters

- **value** (*str or Seq*) – Amino acid sequence
- **translate** (*bool*) – Indicates whether the translation should be generated from the parent genome nucleotide sequence.

set_translation_table(*value*)
Set translation table integer.

Parameters **value** (*int*) – Translation table that should be used to generate the translation.

translate_seq()
Translate the CDS nucleotide sequence.

Use Biopython to translate the nucleotide sequece. The method expects the nucleotide sequence to be a valid CDS sequence in which:

1. it begins with a valid start codon,
2. it ends with a stop codon,
3. it contains only one stop codon,
4. its length is divisible by 3,
5. it translates non-standard start codons to methionine.

If these criteria are not met, an empty Seq object is returned.

Returns Amino acid sequence

Return type Seq

genome

Represents a collection of data about a genome that are commonly used to maintain and update SEA-PHAGES phage genomics data.

class pdm_utils.classes.genome.**Genome**
Bases: object

Class to hold data about a phage genome.

check_attribute(*attribute*, *check_set*, *expect=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the attribute value is valid.

Parameters

- **attribute** (*str*) – Name of the Genome object attribute to evaluate.
- **check_set** (*set*) – Set of reference ids.
- **expect** (*bool*) – Indicates whether the attribute value is expected to be present in the check set.
- **eval_id** (*str*) – Unique identifier for the evaluation.
- **success** (*str*) – Default status if the outcome is a success.
- **fail** (*str*) – Default status if the outcome is not a success.
- **eval_def** (*str*) – Description of the evaluation.

check_authors(*check_set={}*, *expect=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check author list.

Evaluates whether at least one author in the in the list of authors is present in a set of reference authors.

Parameters

- **check_set** (*set*) – Set of reference authors.
- **expect** (*bool*) – Indicates whether at least one author in the list of authors is expected to be present in the check set.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_cds_end_orient_ids(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if there are any duplicate transcription end-orientation coordinates.

Duplicated transcription end-orientation coordinates may represent unintentional duplicate CDS features with slightly different start coordinates.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().

- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_cds_start_end_ids(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if there are any duplicate start-end coordinates.

Duplicated start-end coordinates may represent unintentional duplicate CDS features.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_cluster_structure(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check whether the cluster attribute is structured appropriately.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_compatible_cluster_and_subcluster(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check compatibility of cluster and subcluster attributes.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_feature_coordinates(*use_cds=False*, *use_trna=False*, *use_tmrna=False*, *other=None*, *strand=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Identify nested, duplicated, or partially-duplicated features.

Parameters

- **use_cds** (*bool*) – Indicates whether ids for CDS features should be generated.
- **use_trna** (*bool*) – Indicates whether ids for tRNA features should be generated.
- **use_tmrna** (*bool*) – Indicates whether ids for tmRNA features should be generated.
- **other** (*list*) – List of features that should be included.
- **strand** (*bool*) – Indicates if feature orientation should be included.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().

- **eval_def** – same as for check_attribute().

check_magnitude(*attribute*, *expect*, *ref_value*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the magnitude of a numerical attribute is valid.

Parameters

- **attribute** – same as for check_attribute().
- **expect** (*str*) – Comparison symbol indicating direction of magnitude (>, =, <).
- **ref_value** (*int, float, datetime*) – Numerical value for comparison.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_nucleotides(*check_set={}*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check if all nucleotides in the sequence are expected.

Parameters

- **check_set** (*set*) – Set of reference nucleotides.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_subcluster_structure(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check whether the subcluster attribute is structured appropriately.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

clear_locus_tags()

Resets locus_tags to empty string.

compare_two_attributes(*attribute1*, *attribute2*, *expect_same=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Determine if two attributes are the same.

Parameters

- **attribute1** (*str*) – First attribute to compare.
- **attribute2** (*str*) – Second attribute to compare.
- **expect_same** (*bool*) – Indicates whether the two attribute values are expected to be the same.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().

- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

parse_description()
Retrieve the name and host_genus from the 'description' attribute.

parse_organism()
Retrieve the name and host_genus from the 'organism' attribute.

parse_source()
Retrieve the name and host_genus from the 'source' attribute.

set_accession(*value*, *format='empty_string'*)
Set the accession.

The Accession field in the MySQL database defaults to ''. Some flat file accessions have the version number suffix, so discard the version number.

Parameters

- **value** (*str*) – GenBank accession number.
- **format** (*misc.*) – indicates the format of the data if it is not a valid accession. Default is ''.

set_annotation_author(*value*)
Convert annotation_author to integer value if possible.

Parameters **value** (*str, int*) – Numeric value.

set_cds_descriptions(*value*)
Set each CDS processed description as indicated.

Parameters **value** (*str*) – Name of the description field.

set_cds_features(*value*)
Set and tally the CDS features.

Parameters **value** (*list*) – list of Cds objects.

set_cds_id_list()
Creates lists of CDS feature identifiers.

The first identifier is derived from the start and end coordinates. The second identifier is derived from the transcription end coordinate and orientation.

set_cluster(*value*)
Set the cluster and modify singleton if needed.

Parameters **value** (*str*) – Cluster designation of the genome.

set_date(*value*, *format='empty_datetime_obj'*)
Set the date attribute.

Parameters

- **value** (*misc*) – Date
- **format** (*str*) – Indicates the format if the value is empty.

set_eval(*eval_id*, *definition*, *result*, *status*)
Constructs and adds an Evaluation object to the evaluations list.

Parameters

- **eval_id** (*str*) – Unique identifier for the evaluation.

- **definition** (*str*) – Description of the evaluation.
- **result** (*str*) – Description of the outcome of the evaluation.
- **status** (*str*) – Outcome of the evaluation.

set_feature_genome_ids(*use_cds=False*, *use_trna=False*, *use_tmrna=False*, *use_source=False*, *value=None*)
Sets the genome_id of each feature.

Parameters

- **use_cds** (*bool*) – Indicates whether genome_id for CDS features should be set.
- **use_trna** (*bool*) – Indicates whether genome_id for tRNA features should be set.
- **use_tmrna** (*bool*) – Indicates whether genome_id for tmRNA features should be set.
- **use_source** (*bool*) – Indicates whether genome_id for source features should be set.
- **value** (*str*) – Genome identifier.

set_feature_ids(*use_type=False*, *use_cds=False*, *use_trna=False*, *use_tmrna=False*, *use_source=False*)
Sets the id of each feature.

Lists of features can be added to this method. The method assumes that all elements in all lists contain 'id', 'start', and 'stop' attributes. This feature attribute is processed within the Genome object because and not within the feature itself since the method sorts all features and generates systematic IDs based on feature order in the genome.

Parameters

- **use_type** (*bool*) – Indicates whether the type of object should be added to the feature id.
- **use_cds** (*bool*) – Indicates whether ids for CDS features should be generated.
- **use_trna** (*bool*) – Indicates whether ids for tRNA features should be generated.
- **use_tmrna** (*bool*) – Indicates whether ids for tmRNA features should be generated.
- **use_source** (*bool*) – Indicates whether ids for source features should be generated.

set_filename(*filepath*)
Set the filename. Discard the path and file extension.

Parameters **filepath** (*Path*) – name of the file reference.

set_host_genus(*value=None*, *attribute=None*, *format='empty_string'*)
Set the host_genus from a value parsed from the indicated attribute.

The input data is split into multiple parts, and the first word is used to set host_genus.

Parameters

- **value** (*str*) – the host genus of the phage genome
- **attribute** (*str*) – the name of the genome attribute from which the host_genus attribute will be set
- **format** (*str*) – the default format if the input is an empty/null value.

set_id(*value=None*, *attribute=None*)
Set the id from either an input value or an indicated attribute.

Parameters

- **value** (*str*) – unique identifier for the genome.

- **attribute** (*str*) – name of a genome object attribute that stores a unique identifier for the genome.

set_retrieve_record(*value*)
Convert retrieve_record to integer value if possible.

Parameters **value** (*str, int*) – Numeric value.

set_sequence(*value*)
Set the nucleotide sequence and compute the length.

This method coerces sequences into a Biopython Seq object.

Parameters **value** (*str or Seq*) – the genome's nucleotide sequence.

set_source_features(*value*)
Set and tally the source features.

Parameters **value** (*list*) – list of Source objects.

set_subcluster(*value*)
Set the subcluster.

Parameters **value** (*str*) – Subcluster designation of the genome.

set_tmrna_features(*value*)
Set and tally the tmRNA features. :param value: list of Tmrna objects. :type value: list

set_trna_features(*value*)
Set and tally the tRNA features.

Parameters **value** (*list*) – list of Trna objects.

set_unique_cds_end_orient_ids()
Identify CDS features contain unique transcription end-orientation coordinates.

set_unique_cds_start_end_ids()
Identify CDS features contain unique start-end coordinates.

tally_cds_descriptions()
Tally the non-generic CDS descriptions.

update_name_and_id(*value*)
Update the genome name and id in all locations in a Genome object.

Parameters

- **gnm** (*Genome*) – A pdm_utils Genome object.
- **value** (*str*) – Value used to update the Genome id and name.

tmrna

Represents a collection of data about a tmRNA feature that are commonly used to maintain and update SEA-PHAGES phage genomics data.

class pdm_utils.classes.tmrna.**Tmrna**
Bases: object

check_attribute(*attribute*, *check_set*, *expect=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks whether the indicated feature attribute is present in the given check_set. Uses expect to determine whether the presence (or lack thereof) is an error, or correct. :param attribute: the gene feature attribute to

evaluate :type attribute: str :param check_set: set of reverence values :type check_set: set :param expect: whether the attribute's value is expected to be in the reference set :type expect: bool :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_compatible_gene_and_locus_tag(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that gene and locus_tag attributes contain identical numbers :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_gene_structure(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the gene qualifier contains an integer. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_locus_tag_structure(*check_value=None*, *only_typo=False*, *prefix_set={}*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if the locus_tag is structured correctly.

Parameters

- **check_value** (*str*) – Indicates the genome id that is expected to be present. If None, the 'genome_id' parameter is used.
- **only_typo** (*bool*) – Indicates if only the genome id spelling should be evaluated.
- **prefix_set** (*set*) – Indicates valid common prefixes, if a prefix is expected.
- **case** (*bool*) – Indicates whether the locus_tag is expected to be capitalized.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_magnitude(*attribute*, *expect*, *ref_value*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the magnitude of a numerical attribute meets expectations. :param attribute: the gene feature attribute to evaluate :type attribute: str :param expect: symbol designating direction of magnitude (>=<) :type expect: str :param ref_value: numerical value for comparison :type ref_value: int, float, datetime :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_orientation(*fmt='fr_short'*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the orientation is set appropriately. :param fmt: indicates how coordinates should be formatted :type fmt: str :param case: indicates whether orientation data should be cased :type case: bool :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_orientation_correct(*fmt='fr_short'*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the orientation agrees with the Aragorn and/or tRNAscan-SE predicted orientation. If

Aragorn/tRNAscan-SE report a forward orientation, it means they agree with the annotated orientation. If they report reverse orientation, they think the annotation is backwards. :param fmt: indicates how coordinates should be formatted :type fmt: str :param case: indicates whether orientation data should be cased :type case: bool :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_parts(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Makes sure only one region exists for this tRNA. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_peptide_tag_correct(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks whether the annotated peptide tag matches the Aragorn output. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_peptide_tag_valid(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks whether the annotated peptide tag contains any letters not strictly within the protein alphabet. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

get_begin_end()
Accesses feature coordinates in transcription begin-end format. :return: (begin, end)

get_qualifiers()
Helper function that uses tRNA data to populate the qualifiers attribute of *seqfeature*. :return: qualifiers OrderedDict()

parse_peptide_tag()
Parse the *peptide_tag* attribute out of the note field. :return:

reformat_start_and_stop(*fmt*)
Convert existing start and stop coordinates to the indicated new format; also updates the coordinate format attribute to reflect any change. :param fmt: the new desired coordinate format :type fmt: str :return:

run_aragorn()
Uses an AragornHandler object to negotiate the flow of information between this object and Aragorn. :return:

set_eval(*eval_id*, *definition*, *result*, *status*)
Constructs and adds and Evaluation object to this feature's list of evaluations. :param eval_id: unique identifier for the evaluation :type eval_id: str :param definition: description of the evaluation :type definition: str :param result: description of the evaluation outcome :type result: str :param status: overall outcome of the evaluation :type status: str :return:

set_gene(*value*, *delimiter=None*, *prefix_set=None*)
Set the gene attribute.

Parameters

- **value** (*str*) – Gene data to parse. Also passed to set_num().
- **delimiter** (*str*) – Passed to set_num().
- **prefix_set** (*set*) – Passed to set_num().

set_location_id()
Create identifier tuples containing feature location data. For this method we only care about gene boundaries and will ignore any multi-part elements to the gene. :return:

set_locus_tag(*tag=''*, *delimiter='_'*, *check_value=None*)
Populate the locus_tag and parse the locus_tag number. :param tag: Input locus_tag data :type tag: str :param delimiter: Value used to split locus_tag data :type delimiter: str :param check_value: Genome name or other value that will be used to parse the locus_tag to identify the feature number :type check_value: str

set_name(*value=None*)
Set the feature name. Ideally, the name of the CDS will be an integer. This information can be stored in multiple fields in the GenBank-formatted flat file. The name is derived from one of several qualifiers. :param value: Indicates a value that should be used to directly set the name regardless of the 'gene' and '_locus_tag_num' attributes. :type value: str

set_nucleotide_length(*use_seq=False*)
Set the nucleotide length of this gene feature. :param use_seq: whether to use the Seq feature to calculate nucleotide length of this feature :type use_seq: bool :return:

set_nucleotide_sequence(*value=None*, *parent_genome_seq=None*)
Set this feature's nucleotide sequence :param value: sequence :type value: str or Seq :param parent_genome_seq: parent genome sequence :type parent_genome_seq: Seq :raise: ValueError :return:

set_num(*attr*, *description*, *delimiter=None*, *prefix_set=None*)
Set a number attribute from a description. :param attr: Attribute to set the number. :type attr: str :param description: Description data from which to parse the number. :type description: str :param delimiter: Value used to split the description data. :type delimiter: str :param prefix_set: Valid possible delimiters in the description. :type prefix_set: set

set_orientation(*value*, *fmt*, *case=False*)
Set the orientation based on the indicated format. :param value: orientation value :type value: int or str :param fmt: how orientation should be formatted :type fmt: str :param case: whether to capitalize the first letter of orientation :type case: bool :return:

set_seqfeature()
Create a SeqFeature object with which to populate the *seqfeature* attribute. :return:

trna

Represents a collection of data about a tRNA feature that are commonly used to maintain and update SEA-PHAGES phage genomics data.

class pdm_utils.classes.trna.**Trna**
Bases: object

check_amino_acid_correct(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks that the amino acid that has been annotated for this tRNA agrees with the Aragorn and/or tRNAscan-SE prediction(s). :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_amino_acid_valid(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks that the amino acid that has been annotated for this tRNA is in the set of amino acids that we have opted to allow in the MySQL database. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_anticodon_correct(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks that the annotated anticodon agrees with the prediction by Aragorn or tRNAscan-SE. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_anticodon_valid(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks that the anticodon conforms to the expected length (2-4) and alphabet ("a", "c", "g", "t") or is "nnn". :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_attribute(*attribute*, *check_set*, *expect=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks whether the indicated feature attribute is present in the given check_set. Uses expect to determine whether the presence (or lack thereof) is an error, or correct. :param attribute: the gene feature attribute to evaluate :type attribute: str :param check_set: set of reverence values :type check_set: set :param expect: whether the attribute's value is expected to be in the reference set :type expect: bool :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_compatible_gene_and_locus_tag(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that gene and locus_tag attributes contain identical numbers. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_coordinates(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Parameters

- **eval_id** (*str*) – unique identifier for the evaluation
- **success** (*str*) – status if the outcome is successful
- **fail** (*str*) – status if the outcome is unsuccessful
- **eval_def** (*str*) – description of the evaluation

Returns

check_gene_structure(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the gene qualifier contains an integer. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_length(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Checks that the tRNA is in the expected range of lengths. The average tRNA gene is 70-90bp in length, but it is not uncommon to identify well-scoring tRNAs in the 60-100bp range. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_locus_tag_structure(*check_value=None*, *only_typo=False*, *prefix_set={}*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if the locus_tag is structured correctly.

Parameters

- **check_value** (*str*) – Indicates the genome id that is expected to be present. If None, the 'genome_id' parameter is used.
- **only_typo** (*bool*) – Indicates if only the genome id spelling should be evaluated.
- **prefix_set** (*set*) – Indicates valid common prefixes, if a prefix is expected.
- **case** (*bool*) – Indicates whether the locus_tag is expected to be capitalized.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_magnitude(*attribute*, *expect*, *ref_value*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the magnitude of a numerical attribute meets expectations. :param attribute: the gene feature attribute to evaluate :type attribute: str :param expect: symbol designating direction of magnitude (>=<) :type expect: str :param ref_value: numerical value for comparison :type ref_value: int, float, datetime :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_note_structure(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Checks that the note field is formatted properly.

Genbank does not enforce any standard for the note field. This means that a note does not have to exist.

SEA-PHAGES note fields should look like 'tRNA-Xxx(nnn)'. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_orientation(*fmt='fr_short'*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the orientation is set appropriately. :param fmt: indicates how coordinates should be formatted :type fmt: str :param case: indicates whether orientation data should be cased :type case: bool :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_orientation_correct(*fmt='fr_short'*, *case=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the orientation agrees with the Aragorn and/or tRNAscan-SE predicted orientation. If Aragorn/tRNAscan-SE report a forward orientation, it means they agree with the annotated orientation. If they report reverse orientation, they think the annotation is backwards. :param fmt: indicates how coordinates should be formatted :type fmt: str :param case: indicates whether orientation data should be cased :type case: bool :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_product_structure(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Checks that the product field is formatted properly, and that the annotated amino acid is valid. Genbank enforces that all tRNA annotations that have a product field have it annotated as either 'tRNA-Xxx', where Xxx is one of the 20 standard amino acids, or 'tRNA-OTHER' for those tRNAs which decode a non-standard amino acid (e.g. SeC, Pyl, fMet). SEA-PHAGES may also append the anticodon parenthetically for a product field such as 'tRNA-Xxx(nnn)'. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the

outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_sources(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that this tRNA's DNA sequence can successfully turn up a tRNA when run through Aragorn and tRNAscan-SE. :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

check_terminal_nucleotides(*eval_id=None*, *success='correct'*, *fail='warning'*, *eval_def=None*)
Checks that the tRNA ends with "CCA" or "CC" or "C". :param eval_id: unique identifier for the evaluation :type eval_id: str :param success: status if the outcome is successful :type success: str :param fail: status if the outcome is unsuccessful :type fail: str :param eval_def: description of the evaluation :type eval_def: str :return:

create_seqfeature(*type*, *start*, *stop*, *strand*)

get_begin_end()
Accesses feature coordinates in transcription begin-end format. :return: (begin, end)

get_qualifiers(*type*)
Helper function that uses tRNA data to populate the qualifiers attribute of *seqfeature*. :return: qualifiers OrderedDict()

parse_amino_acid()
Attempts to parse the *amino_acid* attribute from the *product* and *note* attributes. :return:

parse_anticodon()
Attempts to parse the *anticodon* attribute from the *note* attribute. :return:

reformat_start_and_stop(*fmt*)
Convert existing start and stop coordinates to the indicated new format; also updates the coordinate format attribute to reflect any change. :param fmt: the new desired coordinate format :type fmt: str :return:

run_aragorn()
Uses an AragornHandler object to negotiate the flow of information between this object and Aragorn. :return:

run_trnascanse()
Uses a TRNAscanSEHandler object to negotiate the flow of information between this object and tRNAscan-SE. :return:

set_amino_acid(*value*)
Sets the *amino_acid* attribute using the indicated value. :param value: the Amino acid to be used :type value: str :raise: ValueError :return:

set_anticodon(*value*)
Sets the *anticodon* attribute using the indicated value. :param value: the anticodon to use for this tRNA :type value: str :return:

set_eval(*eval_id*, *definition*, *result*, *status*)
Constructs and adds and Evaluation object to this feature's list of evaluations. :param eval_id: unique identifier for the evaluation :type eval_id: str :param definition: description of the evaluation :type definition: str :param result: description of the evaluation outcome :type result: str :param status: overall outcome of the evaluation :type status: str :return:

set_gene(*value*, *delimiter=None*, *prefix_set=None*)
Set the gene attribute. :param value: Gene data to parse. Also passed to set_num(). :type value: str :param delimiter: Passed to set_num(). :type delimiter: str :param prefix_set: Passed to set_num(). :type prefix_set: set

set_location_id()
Create identifier tuples containing feature location data. For this method we only care about gene boundaries and will ignore any multi-part elements to the gene. :return:

set_locus_tag(*tag=''*, *delimiter='_'*, *check_value=None*)
Populate the locus_tag and parse the locus_tag number. :param tag: Input locus_tag data :type tag: str :param delimiter: Value used to split locus_tag data :type delimiter: str :param check_value: Genome name or other value that will be used to parse the locus_tag to identify the feature number :type check_value: str

set_name(*value=None*)
Set the feature name. Ideally, the name of the CDS will be an integer. This information can be stored in multiple fields in the GenBank-formatted flat file. The name is derived from one of several qualifiers. :param value: Indicates a value that should be used to directly set the name regardless of the 'gene' and '_locus_tag_num' attributes. :type value: str

set_nucleotide_length(*use_seq=False*)
Set the nucleotide length of this gene feature. :param use_seq: whether to use the Seq feature to calculate nucleotide length of this feature :type use_seq: bool :return:

set_nucleotide_sequence(*value=None*, *parent_genome_seq=None*)
Set this feature's nucleotide sequence :param value: sequence :type value: str or Seq :param parent_genome_seq: parent genome sequence :type parent_genome_seq: Seq :raise: ValueError :return:

set_num(*attr*, *description*, *delimiter=None*, *prefix_set=None*)
Set a number attribute from a description. :param attr: Attribute to set the number. :type attr: str :param description: Description data from which to parse the number. :type description: str :param delimiter: Value used to split the description data. :type delimiter: str :param prefix_set: Valid possible delimiters in the description. :type prefix_set: set

set_orientation(*value*, *fmt*, *case=False*)
Set the orientation based on the indicated format. :param value: orientation value :type value: int or str :param fmt: how orientation should be formatted :type fmt: str :param case: whether to capitalize the first letter of orientation :type case: bool :return:

set_seqfeature(*type=None*)
Create a SeqFeature object with which to populate the *seqfeature* attribute. :return:

set_structure(*value*)
Set the secondary structure string so downstream users can easily display the predicted fold of this tRNA. :param value: the string to use as the secondary structure :type value: str :return:

source

Represents a collection of data about a Source feature that is commonly used to maintain and update SEA-PHAGES phage genomics data.

class pdm_utils.classes.source.**Source**
Bases: object

check_attribute(*attribute*, *check_set*, *expect=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the attribute value is valid.

Parameters

- **attribute** (*str*) – Name of the Source feature object attribute to evaluate.
- **check_set** (*set*) – Set of reference ids.

- **expect** (*bool*) – Indicates whether the attribute value is expected to be present in the check set.
- **eval_id** (*str*) – Unique identifier for the evaluation.
- **success** (*str*) – Default status if the outcome is a success.
- **fail** (*str*) – Default status if the outcome is not a success.
- **eval_def** (*str*) – Description of the evaluation.

parse_host()
Retrieve the host_genus name from the 'host' field.

parse_lab_host()
Retrieve the host_genus name from the 'lab_host' field.

parse_organism()
Retrieve the phage and host_genus names from the 'organism' field.

set_eval(*eval_id*, *definition*, *result*, *status*)
Constructs and adds an Evaluation object to the evaluations list.

Parameters

- **eval_id** (*str*) – Unique identifier for the evaluation.
- **definition** (*str*) – Description of the evaluation.
- **result** (*str*) – Description of the outcome of the evaluation.
- **status** (*str*) – Outcome of the evaluation.

Data processing

alchemyhandler

class pdm_utils.classes.alchemyhandler.**AlchemyHandler**(*database=None*, *username=None*, *password=None*, *dialect='mysql'*, *driver=None*)

Bases: object

property URI
Returns a SQLAlchemy URI string from stored credentials.

ask_credentials()
Ask for username and password input to store in AlchemyHandler.

ask_database()
Ask for database input to store in AlchemyHandler.

build_all()
Create and store all relevant SQLAlchemy objects.

build_engine()
Create and store SQLAlchemy Engine object.

build_graph()
Create and store SQLAlchemy MetaData related NetworkX Graph object.

build_mapper()
Create and store SQLAlchemy automapper Base object.

build_metadata()
Create and store SQLAlchemy MetaData object.

build_session()
Create and store SQLAlchemy Session object.

clear()
Clear properties tied to MySQL credentials/database.

connect(*ask_database=False*, *login_attempts=5*, *pipeline=False*)
Ask for input to connect to MySQL and MySQL databases.

Parameters

- **ask_database** (*Boolean*) – Toggle whether to connect to a database.
- **login_attempts** (*int*) – Set number of total login attempts.

construct_engine_string(*dialect='mysql'*, *driver='pymysql'*, *username=''*, *password=''*, *database=''*)
Construct a SQLAlchemy engine URL.

Parameters

- **dialect** (*str*) – Type of SQL database.
- **driver** (*str*) – Name of the Python DBAPI used to connect.
- **username** (*str*) – Username to login to SQL database.
- **password** (*str*) – Password to login to SQL database.
- **database** (*str*) – Name of the database to connect to.

Returns URI string to create SQLAlchemy engine.

Return type str

property database
Returns the AlchemyHandler's set database.

Returns Returns a copy of the database attribute.

Return type str

property databases
Returns a copy of the databases available to the current credentials

Returns Returns the AlchemyHandler's available databases.

Return type list[str]

property engine
Returns the AlchemyHandler's stored engine object.

Returns Returns the AlchemyHandler's stored engine object.

Return type Engine

extract_engine_credentials(*engine*)
Extract username, password, and/or database from a SQLAlchemy engine.

get_map(*table*)
Get SQLAlchemy ORM map object.

get_mysql_dbs()
Retrieve database names from MySQL.

Returns List of database names.

Return type list

property graph
Returns the AlchemyHandler's stored graph object.

Returns Returns the AlchemyHandler's stored metadata graph object.

Return type Graph

property login_attempts
Returns the AlchemyHandler's number of login attempts for login.

Returns Returns the number of login attempts for login.

Return type str

property mapper
Returns the AlchemyHandler's stored automapper object.

Returns Returns the AlchemyHandler's stored mapper object.

property metadata
Returns the AlchemyHandler's stored metadata object.

Returns Returns the AlchemyHandler's stored engine object.

Return type MetaData

property password
Returns the AlchemyHandler's set password.

Returns Returns a copy of the password attribute.

Return type str

property session
Returns the AlchemyHandler's stored session object.

property username
Returns the AlchemynHandler's set username.

Returns Returns a copy of the username attribute.

Return type str

validate_database()
Validate access to database using stored SQL credentials.

exception pdm_utils.classes.alchemyhandler.**MySQLDatabaseError**
Bases: Exception

exception pdm_utils.classes.alchemyhandler.**SQLCredentialsError**
Bases: Exception

exception pdm_utils.classes.alchemyhandler.**SQLiteDatabaseError**
Bases: Exception

aragornhandler

class pdm_utils.classes.aragornhandler.**AragornHandler**(*identifier*, *sequence*)
Bases: object

parse_determinate_trnas()
Searches *out_str* for matches to a regular expression for Aragorn tRNAs of determinate isotype. :return:

parse_indeterminate_trnas()
Searches *out_str* for matches to a regular expression for Aragorn tRNAs of indeterminate isotype. :return:

parse_tmrnas()
Searches *out_str* for matches to a regular expression for Aragorn tmRNAs. :return:

parse_trnas()
Calls two helper methods to parse determinate and indeterminate tRNAs. :return:

read_output()
Reads the Aragorn output file and joins the lines into a single string which it populates into *out_str*. :return:

run_aragorn(*c=False*, *d=True*, *m=False*, *t=True*)
Set up Aragorn command, then run it. Default arguments will assume linear sequence to be scanned on both strands for tRNAs only (no tmRNAs). :param c: treat sequence as circular :type c: bool :param d: search both strands of DNA :type d: bool :param m: search for tmRNAs :type m: bool :param t: search for tRNAs :type t: bool :return:

write_fasta()
Writes the search sequence to input file in FASTA format. :return:

bundle

Represents a structure to directly compare data between two or more genomes.

class pdm_utils.classes.bundle.**Bundle**
Bases: object

check_for_errors()
Check evaluation lists of all objects contained in the Bundle and determine how many errors there are.

check_genome_dict(*key*, *expect=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if a genome is present in the genome dictionary.

Parameters

- **key** (*str*) – The value to be evaluated if it is a valid key in the genome dictionary.
- **expect** (*bool*) – Indicates whether the key is expected to be a valid key in the genome dictionary.
- **eval_id** (*str*) – Unique identifier for the evaluation.
- **success** (*str*) – Default status if the outcome is a success.
- **fail** (*str*) – Default status if the outcome is not a success.
- **eval_def** (*str*) – Description of the evaluation.

check_genome_pair_dict(*key*, *expect=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check if a genome_pair is present in the genome_pair dictionary.

Parameters

- **key** – same as for check_genome_dict().
- **expect** – same as for check_genome_dict().
- **eval_id** – same as for check_genome_dict().
- **success** – same as for check_genome_dict().
- **fail** – same as for check_genome_dict().
- **eval_def** – same as for check_genome_dict().

check_statements(*execute_result*, *execute_msg*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check if MySQL statements were successfully executed.

Parameters

- **execute_result** (*int*) – Indication if MySQL statements were successfully execute.
- **execute_msg** (*str*) – Description of MySQL statement execution result.
- **eval_id** – same as for check_genome_dict().
- **success** – same as for check_genome_dict().
- **fail** – same as for check_genome_dict().
- **eval_def** – same as for check_genome_dict().

check_ticket(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check for whether a Ticket object is present.

Parameters

- **eval_id** – same as for check_genome_dict().
- **success** – same as for check_genome_dict().
- **fail** – same as for check_genome_dict().
- **eval_def** – same as for check_genome_dict().

get_evaluations()

Get all evaluations for all objects stored in the Bundle.

Returns Dictionary of evaluation lists for each feature.

Return type dict

set_eval(*eval_id*, *definition*, *result*, *status*)

Constructs and adds an Evaluation object to the evaluations list.

Parameters

- **eval_id** (*str*) – Unique identifier for the evaluation.
- **definition** (*str*) – Description of the evaluation.
- **result** (*str*) – Description of the outcome of the evaluation.
- **status** (*str*) – Outcome of the evaluation.

set_genome_pair(*genome_pair*, *key1*, *key2*)

Pair two genomes and add to the paired genome dictionary.

Parameters

- **genome_pair** (*GenomePair*) – An empty GenomePair object to stored paried genomes.

- **key1** (*str*) – A valid key in the Bundle object's 'genome_dict' that indicates the first genome to be paired.
- **key2** (*str*) – A valid key in the Bundle object's 'genome_dict' that indicates the second genome to be paired.

cdspair

evaluation

Represents a structure to contain results of an evaluation.

class pdm_utils.classes.evaluation.**Evaluation**(*id=''*, *definition=''*, *result=''*, *status=''*)
Bases: object

filter

Object to provide a formatted filtering query for retrieving data from a SQL database.

class pdm_utils.classes.filter.**Filter**(*alchemist=None*, *key=None*)
Bases: object

add(*filter_string*)
Add a MySQL where filter(s) to the Filter object class.

Parameters **filter** (*str*) – Formatted MySQL WHERE clause.

and_(*filter*)
Add an and conditional to the Filter object class.

Param_filter Formatted MySQL WHERE clause.

Type_filter str

build_values(*where=None*, *column=None*, *raw_bytes=False*, *limit=8000*)
Queries for values from stored WHERE clauses and Filter key.

Parameters

- **where** (*list*) – MySQL WHERE clause_related SQLAlchemy object(s).
- **order_by** (*list*) – MySQL ORDER BY clause-related SQLAlchemy object(s).
- **column** (*str*) – SQLAlchemy Column object or object name.
- **limit** (*int*) – SQLAlchemy IN clause query length limiter.

Returns Distinct values fetched from given and innate constraints.

Return type list

build_where_clauses()
Builds BinaryExpression objects from stored Filter object filters.

Returns A list of SQLAlchemy WHERE conditionals.

Return type list

check()
Check Filter object contains valid essential objects. Filter object requires a SQLAlchemy Engine and Column as well as a NetworkX Graph.

connect(*alchemist=None*)
Connect Filter object to a database with an AlchemyHandler.

Parameters **alchemist** (*AlchemyHandler*) – An AlchemyHandler object.

property connected

copy()
Returns a copy of a Filter object.

copy_filters()
Returns a copy of a Filter object's filter dictionary.

property engine

property filters

get_column(*raw_column*)
Converts a column input, string or Column, to a Column.

Parameters **raw_column** (*str*) – SQLAlchemy Column object or object name.

get_columns(*raw_columns*)
Converts a column input list, string or Column, to a list of Columns.

Parameters **raw_column** (*list[str]*) – SQLAlchemy Column object or object name.

Returns Returns SQLAlchemy Columns

Return type list[Column]

property graph

group(*raw_column*, *raw_bytes=False*, *filter=False*)
Queries and separates Filter object's values based on a Column.

Parameters **raw_column** (*str*) – SQLAlchemy Column object or object name.

hits()
Gets the number of a Filter object's values.

property key

link(*alchemist*)
Connect Filter object to a database with an existing AlchemyHandler.

Parameters **alchemist** (*AlchemyHandler*) – An AlchemyHandler object.

property mapper

mass_transpose(*raw_columns*, *raw_bytes=False*, *filter=False*)
Queries for sets of distinct values, using self.transpose()

Parameters **columns** (*list*) – SQLAlchemy Column object(s)

Returns Distinct values fetched from given and innate restraints.

Return type dict

new_or_()
Create a new conditional block to the Filter object class.

property or_index

parenthesize()
Condense current filters into an isolated clause

print_results()
Prints the Filter object's values in a formatted way.

query(*table_map*)
Queries for ORM object instances conditioned on Filter values.

Parameters **table_map** (*DeclarativeMeta*) – SQLAlchemy ORM map object.

Returns List of mapped object instances.

Return type list

refresh()
Re-queries for the Filter's values.

remove(*filter*)
Remove an and filter from the current block of and conditionals.

Parameters **filter** (*str*) – Formatted MySQL WHERE clause.

reset()
Resets all filters, values, and Filter state conditions.

reset_filters()
Resets all filters and relevant Filter state condition.

retrieve(*raw_columns*, *raw_bytes=False*, *filter=False*)
Queries for distinct data for each value in the Filter object.

Parameters **columns** (*list[str]*) – SQLAlchemy Column object(s)

Returns Distinct values for each Filter value.

Return type dict{dict}

select(*raw_columns*, *return_dict=True*)
Queries for data conditioned on the values in the Filter object.

Parameters

- **columns** (*list[str]*) – SQLAlchemy Column object(s)
- **return_dict** (*Boolean*) – Toggle whether to return data as a dictionary.

Returns SELECT data conditioned on the values in the Filter object.

Return type dict

Return type list[RowProxy]

property session

sort(*raw_columns*)
Re-queries for the Filter's values, applying a ORDER BY clause.

Parameters **raw_column** – SQLAlchemy Column object(s) or object name(s).

transpose(*raw_column*, *return_dict=False*, *set_values=False*, *raw_bytes=False*, *filter=False*)
Queries for distinct values from stored values and a MySQL Column.

Parameters

- **raw_column** (*str*) – SQLAlchemy Column object or object name.
- **return_dict** (*Boolean*) – Toggle whether to return data as a dictionary.
- **set_values** (*Boolean*) – Toggle whether to replace Filter key and values.

Returns Distinct values fetched from given and innate constraints.

Return type list

Return type dict

`update()`
Queries using the Filter's key and its stored BinaryExpressions.

`property updated`

`property values`

`property values_valid`

genomepair

Represents a structure to pair two Genome objects and perform comparisons between them to identify inconsistencies.

`class pdm_utils.classes.genomepair.GenomePair`
Bases: `object`

`compare_attribute(attribute, expect_same=False, eval_id=None, success='correct', fail='error', eval_def=None)`
Compare values of the specified attribute in each genome.

Parameters

- **attribute** (*str*) – Name of the GenomePair object attribute to evaluate.
- **expect_same** (*bool*) – Indicates whether the two attribute values are expected to be the same.
- **eval_id** (*str*) – Unique identifier for the evaluation.
- **success** (*str*) – Default status if the outcome is a success.
- **fail** (*str*) – Default status if the outcome is not a success.
- **eval_def** (*str*) – Description of the evaluation.

`compare_date(expect, eval_id=None, success='correct', fail='error', eval_def=None)`
Compare the date of each genome.

Parameters

- **expect** (*str*) – Is the first genome expected to be "newer", "equal", or "older" than the second genome.
- **eval_id** – same as for compare_attribute().
- **success** – same as for compare_attribute().
- **fail** – same as for compare_attribute().
- **eval_def** – same as for compare_attribute().

`set_eval(eval_id, definition, result, status)`
Constructs and adds an Evaluation object to the evaluations list.

Parameters

- **eval_id** (*str*) – Unique identifier for the evaluation.
- **definition** (*str*) – Description of the evaluation.

- **result** (*str*) – Description of the outcome of the evaluation.
- **status** (*str*) – Outcome of the evaluation.

randomfieldupdatehandler

class pdm_utils.classes.randomfieldupdatehandler.**RandomFieldUpdateHandler**(*connection*)
Bases: object

execute_ticket()
This function checks whether the ticket is valid. If it is not valid, the function returns with code 0, indicating failure to execute the ticket. If the ticket is valid, request input from the user to verify that they actually want to proceed with the update they've proposed. If response is in the affirmative, the ticket is executed. Otherwise, indicate that this ticket will be skipped, and return 0 as the ticket was not executed. If an error is encountered during execution of the ticket, print error message and return 0. If the ticket is executed without issue, return 1 indicating success. :return:

validate_field()
This function attempts to validate the replacement field by checking whether it's on the list of fields in the indicated table. :return:

validate_key_name()
This function attempts to validate the selection key by checking whether it's a field in the table marked as any kind of key. :return:

validate_key_value()
This function attempts to validate the selection key's value by querying the database for the data associated with that key and value on the indicated table :return:

validate_table()
This function attempts to validate the table by simply querying for the table's description. :return:

validate_ticket()
This function runs all 4 of the object's built-in ticket validation methods, and checks whether any of the ticket inputs were invalid. If any are invalid, reject the ticket. If none are invalid, accept the ticket. :return:

ticket

Represents a structure to contain directions for how to parse and import genomes into a MySQL database.

class pdm_utils.classes.ticket.**ImportTicket**
Bases: object

check_attribute(*attribute*, *check_set*, *expect=False*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)
Check that the id is valid.

Parameters

- **attribute** (*str*) – Name of the ImportTicket object attribute to evaluate.
- **check_set** (*set*) – Set of reference ids.
- **expect** (*bool*) – Indicates whether the attribute value is expected to be present in the check set.
- **eval_id** (*str*) – Unique identifier for the evaluation.
- **success** (*str*) – Default status if the outcome is a success.

- **fail** (*str*) – Default status if the outcome is not a success.
- **eval_def** (*str*) – Description of the evaluation.

check_compatible_type_and_data_retain(*eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check if the ticket type and data_retain are compatible.

If the ticket type is 'add', then the data_retain set is not expected to have any data.

Parameters

- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_eval_flags(*expect=True*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the eval_flags is valid.

Parameters

- **expect** (*bool*) – Indicates whether the eval_flags is expected to contain data.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

check_valid_data_source(*ref_set_attr*, *check_set*, *eval_id=None*, *success='correct'*, *fail='error'*, *eval_def=None*)

Check that the values in the specified attribute are valid.

Parameters

- **ref_set_attr** (*str*) – Name of the data_dict in the ticket to be evaluated (data_add, data_retain, data_retrieve, data_parse)
- **check_set** (*set*) – Set of valid field names.
- **eval_id** – same as for check_attribute().
- **success** – same as for check_attribute().
- **fail** – same as for check_attribute().
- **eval_def** – same as for check_attribute().

set_description_field(*value*)

Set the description_field.

Parameters **value** (*str*) – Value to be set as the description_field.

set_eval(*eval_id*, *definition*, *result*, *status*)

Constructs and adds an Evaluation object to the evaluations list.

Parameters

- **eval_id** (*str*) – Unique identifier for the evaluation.
- **definition** (*str*) – Description of the evaluation.
- **result** (*str*) – Description of the outcome of the evaluation.

- **status** (*str*) – Outcome of the evaluation.

set_eval_mode(*value*)
Set the eval_mode.

Parameters **value** (*str*) – Value to be set as the eval_mode.

set_type(*value*)
Set the ticket type.

Parameters **value** (*str*) – Value to be set as the type.

trnascansehandler

class pdm_utils.classes.trnascansehandler.**TRNAscanSEHandler**(*identifier*, *sequence*)
Bases: object

parse_trnas()
Searches *out_str* for matches to a regular expression for tRNAscan-SE tRNAs. :return:

read_output()
Reads the Aragorn output file and joins the lines into a single string which it populates into *out_str*. :return:

run_trnascanse(*x=10*)
Set up tRNAscan-SE command, then run it. Explanation of arguments: :param x: score cutoff for tRNAscan-SE :type x: int :return:

write_fasta()
Writes the search sequence to input file in FASTA format. :return:

6.2.3 functions

basic

Misc. base/simple functions. These should not require import of other modules in this package to prevent circular imports.

pdm_utils.functions.basic.**ask_yes_no**(*prompt=''*, *response_attempt=1*)
Function to get the user's yes/no response to a question.

Accepts variations of yes/y, true/t, no/n, false/f, exit/quit/q.

Parameters

- **prompt** (*str*) – the question to ask the user.
- **response_attempt** (*int*) – The number of the number of attempts allowed before the function exits. This prevents the script from getting stuck in a loop.

Returns The default is False (e.g. user hits Enter without typing anything else), but variations of yes or true responses will return True instead. If the response is 'exit' or 'quit', the loop is exited and None is returned.

Return type bool, None

pdm_utils.functions.basic.**check_empty**(*value*, *lower=True*)
Checks if the value represents a null value.

Parameters

- **value** (*misc.*) – Value to be checked against the empty set.

- **lower** (*bool*) – Indicates whether the input value should be lowercased prior to checking.

Returns Indicates whether the value is present in the empty set.

Return type bool

pdm_utils.functions.basic.**check_value_expected_in_set**(*value*, *set1*, *expect=True*)
Check if a value is present within a set and if it is expected.

Parameters

- **value** (*misc.*) – The value to be checked.
- **set1** (*set*) – The reference set of values.
- **expect** (*bool*) – Indicates if 'value' is expected to be present in 'set1'.

Returns The result of the evaluation.

Return type bool

pdm_utils.functions.basic.**check_value_in_two_sets**(*value*, *set1*, *set2*)
Check if a value is present within two sets.

Parameters

- **value** (*misc.*) – The value to be checked.
- **set1** (*set*) – The first reference set of values.
- **set2** (*set*) – The second reference set of values.

Returns

The result of the evaluation, indicating whether the value is present within:

1. only the 'first' set
2. only the 'second' set
3. 'both' sets
4. 'neither' set

Return type str

pdm_utils.functions.basic.**choose_from_list**(*options*)
Iterate through a list of values and choose a value.

Parameters **options** (*list*) – List of options to choose from.

Returns the user select option of None

Return type option or None

pdm_utils.functions.basic.**choose_most_common**(*string*, *values*)
Identify most common occurrence of several values in a string.

Parameters

- **string** (*str*) – String to search.
- **values** (*list*) – List of string characters. The order in the list indicates preference, in the case of a tie.

Returns Value from values that occurs most.

Return type str

pdm_utils.functions.basic.**clear_screen**()
Brings the command line to the top of the screen.

pdm_utils.functions.basic.**compare_cluster_subcluster**(*cluster*, *subcluster*)
Check if a cluster and subcluster designation are compatible.

Parameters

- **cluster** (*str*) – The cluster value to be compared. 'Singleton' and 'UNK' are lowercased.
- **subcluster** (*str*) – The subcluster value to be compared.

Returns The result of the evaluation, indicating whether the two values are compatible.

Return type bool

pdm_utils.functions.basic.**compare_sets**(*set1*, *set2*)
Compute the intersection and differences between two sets.

Parameters

- **set1** (*set*) – The first input set.
- **set2** (*set*) – The second input set.

Returns tuple (set_intersection, set1_diff, set2_diff) WHERE set_intersection(set) is the set of shared values. set1_diff(set) is the set of values unique to the first set. set2_diff(set) is the set of values unique to the second set.

Return type tuple

pdm_utils.functions.basic.**convert_empty**(*input_value*, *format*, *upper=False*)
Converts common null value formats.

Parameters

- **input_value** (*str, int, datetime*) – Value to be re-formatted.
- **format** (*str*) – Indicates how the value should be edited. Valid format types include: 'empty_string' = '' 'none_string' = 'none' 'null_string' = 'null' 'none_object' = None 'na_long' = 'not applicable' 'na_short' = 'na' 'n/a' = 'n/a' 'zero_string' = '0' 'zero_num' = 0 'empty_datetime_obj' = datetime object with arbitrary date, '1/1/0001'
- **upper** (*bool*) – Indicates whether the output value should be uppercased.

Returns The re-formatted value as indicated by 'format'.

Return type str, int, datetime

pdm_utils.functions.basic.**convert_list_to_dict**(*data_list*, *key*)
Convert list of dictionaries to a dictionary of dictionaries

Parameters

- **data_list** (*list*) – List of dictionaries.
- **key** (*str*) – key in each dictionary to become the returned dictionary key.

Returns Dictionary of all dictionaries. Returns an empty dictionary if all intended keys are not unique.

Return type dict

pdm_utils.functions.basic.**convert_to_decoded**(*values*)
Converts a list of strings to utf-8 encoded values.

Parameters **values** (*list[bytes]*) – Byte values from MySQL queries to be decoded.

Returns List of utf-8 decoded values.

Return type list[str]

pdm_utils.functions.basic.**convert_to_encoded**(*values*)
Converts a list of strings to utf-8 encoded values.

Parameters **values** (*list[str]*) – Strings for a MySQL query to be encoded.

Returns List of utf-8 encoded values.

Return type list[bytes]

pdm_utils.functions.basic.**create_indices**(*input_list*, *batch_size*)
Create list of start and stop indices to split a list into batches.

Parameters

- **input_list** (*list*) – List from which to generate batch indices.
- **batch_size** (*int*) – Size of each batch.

Returns List of 2-element tuples (start index, stop index).

Return type list

pdm_utils.functions.basic.**edit_suffix**(*value*, *option*, *suffix='_Draft'*)
Adds or removes the indicated suffix to an input value.

Parameters

- **value** (*str*) – Value that will be edited.
- **option** (*str*) – Indicates what to do with the value and suffix ('add', 'remove').
- **suffix** (*str*) – The suffix that will be added or removed.

Returns The edited value. The suffix is not added if the input value already has the suffix.

Return type str

pdm_utils.functions.basic.**expand_path**(*input_path*)
Convert a non-absolute path into an absolute path.

Parameters **input_path** (*str*) – The path to be expanded.

Returns The expanded path.

Return type str

pdm_utils.functions.basic.**find_expression**(*expression*, *list_of_items*)
Counts the number of items with matches to a regular expression.

Parameters

- **expression** (*re*) – Regular expression object
- **list_of_items** (*list*) – List of items that will be searched with the regular expression.

Returns Number of times the regular expression was identified in the list.

Return type int

pdm_utils.functions.basic.**get_user_pwd**(*user_prompt='Username: '*, *pwd_prompt='Password: '*)
Get username and password.

Parameters

- **user_prompt** (*str*) – Displayed description when prompted for username.

- **pwd_prompt** (*str*) – Displayed description when prompted for password.

Returns tuple (username, password) WHERE username(str) is the user-supplied username. password(str) is the user-supplied password.

Return type tuple

pdm_utils.functions.basic.**get_values_from_dict_list**(*list_of_dicts*)
Convert a list of dictionaries to a set of the dictionary values.

Parameters **list_of_dicts** (*list*) – List of dictionaries.

Returns Set of values from all dictionaries in the list.

Return type set

pdm_utils.functions.basic.**get_values_from_tuple_list**(*list_of_tuples*)
Convert a list of tuples to a set of the tuple values.

Parameters **list_of_tuples** (*list*) – List of tuples.

Returns Set of values from all tuples in the list.

Return type set

pdm_utils.functions.basic.**identify_contents**(*path_to_folder*, *kind=None*, *ignore_set={}*)
Create a list of filenames and/or folders from an indicated directory.

Parameters

- **path_to_folder** (*Path*) – A valid directory path.
- **kind** (*str*) – ("file", "dir"), corresponding with paths to be checked as either files or directories.
- **ignore_set** (*set*) – A set of strings representing file or folder names to ignore.

Returns List of valid contents in the directory.

Return type list

pdm_utils.functions.basic.**identify_nested_items**(*complete_list*)
Identify nested and non-nested two-element tuples in a list.

Parameters **complete_list** (*list*) – List of tuples that will be evaluated.

Returns tuple (not_nested_set, nested_set) WHERE not_nested_set(set) is a set of non-nested tuples. nested_set(set) is a set of nested tuples.

Return type tuple

pdm_utils.functions.basic.**identify_one_list_duplicates**(*item_list*)
Identify duplicate items within a list.

Parameters **item_list** (*list*) – The input list to be checked.

Returns The set of non-unique/duplicated items.

Return type set

pdm_utils.functions.basic.**identify_two_list_duplicates**(*item1_list*, *item2_list*)
Identify duplicate items between two lists.

Parameters

- **item1_list** (*list*) – The first input list to be checked.
- **item2_list** (*list*) – The second input list to be checked.

Returns The set of non-unique/duplicated items between the two lists (but not duplicate items within each list).

Return type set

pdm_utils.functions.basic.**identify_unique_items**(*complete_list*)
Identify unique and non-unique items in a list.

Parameters **complete_list** (*list*) – List of items that will be evaluated.

Returns tuple (unique_set, duplicate_set) WHERE unique_set(set) is a set of all unique/non-duplicated items. duplicate_set(set) is a set of non-unique/duplicated items. non-informative/generic data is removed.

Return type tuple

pdm_utils.functions.basic.**increment_histogram**(*data*, *histogram*)
Increments a dictionary histogram based on given data.

Parameters

- **data** (*list*) – Data to be used to index or create new keys in the histogram.
- **histogram** (*dict*) – Dictionary containing keys whose values contain counts.

pdm_utils.functions.basic.**invert_dictionary**(*dictionary*)
Inverts a dictionary, where the values and keys are swapped.

Parameters **dictionary** (*dict*) – A dictionary to be inverted.

Returns Returns an inverted dictionary of the given dictionary.

Return type dict

pdm_utils.functions.basic.**is_float**(*string*)
Check if string can be converted to float.

pdm_utils.functions.basic.**join_strings**(*input_list*, *delimiter=' '*)
Open file and retrieve a dictionary of data.

Parameters

- **input_list** (*list*) – List of values to join.
- **delimiter** (*str*) – Delimiter used between values.

Returns Concatenated values, excluding all None and '' values.

Return type str

pdm_utils.functions.basic.**lower_case**(*value*)
Return the value lowercased if it is within a specific set of values.

Parameters **value** (*str*) – The value to be checked.

Returns The lowercased value if it is equivalent to 'none', 'retrieve', or 'retain'.

Return type str

pdm_utils.functions.basic.**make_new_dir**(*output_dir*, *new_dir*, *attempt=1*, *mkdir=True*)
Make a new directory.

Checks to verify the new directory name is valid and does not already exist. If it already exists, it attempts to extend the name with an integer suffix.

Parameters

- **output_dir** (*Path*) – Full path to the directory where the new directory will be created.

- **new_dir** (*Path*) – Name of the new directory to be created.
- **attempt** (*int*) – Number of attempts to create the directory.

Returns If successful, the full path of the created directory. If unsuccessful, None.

Return type Path, None

pdm_utils.functions.basic.**make_new_file**(*output_dir*, *new_file*, *ext*, *attempt=1*)
Make a new file.

Checks to verify the new file name is valid and does not already exist. If it already exists, it attempts to extend the name with an integer suffix.

Parameters

- **output_dir** (*Path*) – Full path to the directory where the new directory will be created.
- **new_file** (*Path*) – Name of the new file to be created.
- **ext** (*str*) – Name of the file extension to be used.
- **attempt** (*int*) – Number of attempts to create the file.

Returns If successful, the full path of the created file. If unsuccessful, None.

Return type Path, None

pdm_utils.functions.basic.**match_items**(*list1*, *list2*)
Match values of two lists and return several results.

Parameters

- **list1** (*list*) – The first input list.
- **list2** (*list*) – The second input list.

Returns tuple (matched_unique_items, set1_unmatched_unique_items, set2_unmatched_unique_items, set1_duplicate_items, set2_duplicate_items) WHERE matched_unique_items(set) is the set of matched unique values. set1_unmatched_unique_items(set) is the set of unmatched unique values from the first list. set2_unmatched_unique_items(set) is the set of unmatched unique values from the second list. set1_duplicate_items(set) is the the set of duplicate values from the first list. set2_duplicate_items(set) is the set of unmatched unique values from the second list.

Return type tuple

pdm_utils.functions.basic.**merge_set_dicts**(*dict1*, *dict2*)
Merge two dictionaries of sets.

Parameters

- **dict1** (*dict*) – First dictionary of sets.
- **dict2** (*dict*) – Second dictionary of sets.

Returns Merged dictionary containing all keys from both dictionaries, and for each shared key the value is a set of merged values.

Return type dict

pdm_utils.functions.basic.**parse_flag_file**(*flag_file*)
Parse a file to an evaluation flag dictionary.

Parameters **flag_file** (*str*) – A two-column csv-formatted file WHERE 1. evaluation flag 2. 'True' or 'False'

Returns A dictionary WHERE keys (str) are evaluation flags values (bool) indicate the flag setting Only flags that contain boolean values are returned.

Return type dict

pdm_utils.functions.basic.**parse_names_from_record_field**(*description*)
Attempts to parse the phage/plasmid/prophage name and host genus from a given string. :param description: the input string to be parsed :type description: str :return: name, host_genus

pdm_utils.functions.basic.**partition_list**(*data_list*, *size*)
Chunks list into a list of lists with the given size.

Parameters

- **data_list** (*list*) – List to be split into equal-sized lists.
- **size** – Length of the resulting list chunks.
- **size** – int

Returns Returns list of lists with length of the given size.

Return type list[list]

pdm_utils.functions.basic.**prepare_filepath**(*folder_path*, *file_name*, *folder_name=None*)
Prepare path to new file.

Parameters

- **folder_path** (*Path*) – Path to the directory to contain the file.
- **file_name** (*str*) – Name of the file.
- **folder_name** (*Path*) – Name of sub-directory to create.

Returns Path to file in directory.

Return type Path

pdm_utils.functions.basic.**reformat_coordinates**(*start*, *stop*, *current*, *new*)
Converts common coordinate formats.

The type of coordinate formats include:

'0_half_open':

0-based half-open intervals that is the common format for BAM files and UCSC Browser database. This format seems to be more efficient when performing genomics computations.

'1_closed':

1-based closed intervals that is the common format for the MySQL Database, UCSC Browser, the Ensembl genomics database, VCF files, GFF files. This format seems to be more intuitive and used for visualization.

The function assumes coordinates reflect the start and stop boundaries (where the start coordinates is smaller than the stop coordinate), instead of transcription start and stop coordinates.

Parameters

- **start** (*int*) – Start coordinate
- **stop** (*int*) – Stop coordinate
- **current** (*str*) – Indicates the indexing format of the input coordinates.
- **new** (*str*) – Indicates the indexing format of the output coordinates.

Returns The re-formatted start and stop coordinates.

Return type int

pdm_utils.functions.basic.**reformat_description**(*raw_description*)
Reformat a gene description.

Parameters **raw_description** (*str*) – Input value to be reformatted.

Returns tuple (description, processed_description) WHERE description(str) is the original value stripped of leading and trailing whitespace. processed_description(str) is the reformatted value, in which non-informative/generic data is removed.

Return type tuple

pdm_utils.functions.basic.**reformat_strand**(*input_value*, *format*, *case=False*)
Converts common strand orientation formats.

Parameters

- **input_value** (*str, int*) – Value that will be edited.
- **format** (*str*) – Indicates how the value should be edited. Valid format types include: 'fr_long' ('forward', 'reverse') 'fr_short' ('f', 'r') 'fr_abbrev1' ('for', 'rev') 'fr_abbrev2' ('fwd', 'rev') 'tb_long' ('top', 'bottom') 'tb_short' ('t', 'b') 'wc_long' ('watson', 'crick') 'wc_short' ('w','c') 'operator' ('+', '-') 'numeric' (1, -1).
- **case** (*bool*) – Indicates whether the output value should be capitalized.

Returns The re-formatted value as indicated by 'format'.

Return type str, int

pdm_utils.functions.basic.**select_option**(*prompt*, *valid_response_set*)
Select an option from a set of options.

Parameters

- **prompt** (*str*) – Message to display before displaying option.
- **valid_response_set** (*set*) – Set of valid options to choose.

Returns option

Return type str, int

pdm_utils.functions.basic.**set_path**(*path*, *kind=None*, *expect=True*)
Confirm validity of path argument.

Parameters

- **path** (*Path*) – path
- **kind** (*str*) – ("file", "dir"), corresponding with paths to be checked as either files or directories.
- **expect** (*bool*) – Indicates if the path is expected to the indicated kind.

Returns Absolute path if valid, otherwise sys.exit is called.

Return type Path

pdm_utils.functions.basic.**sort_histogram**(*histogram*, *descending=True*)
Sorts a dictionary by its values and returns the sorted histogram.

Parameters **histogram** (*dict*) – Dictionary containing keys whose values contain counts.

Returns An ordered dict from items from the histogram sorted by value.

Return type OrderedDict

pdm_utils.functions.basic.**sort_histogram_keys**(*histogram*, *descending=True*)
Sorts a dictionary by its values and returns the sorted histogram.

Parameters **histogram** (*dict*) – Dictionary containing keys whose values contain counts.

Returns A list from keys from the histogram sorted by value.

Return type list

pdm_utils.functions.basic.**split_string**(*string*)
Split a string based on alphanumeric characters.

Iterates through a string, identifies the first position in which the character is a float, and creates two strings at this position.

Parameters **string** (*str*) – The value to be split.

Returns tuple (left, right) WHERE left(str) is the left portion of the input value prior to the first numeric character and only contains alphabetic characters (or will be ‘’). right(str) is the right portion of the input value after the first numeric character and only contains numeric characters (or will be ‘’).

Return type tuple

pdm_utils.functions.basic.**trim_characters**(*string*)
Remove leading and trailing generic characters from a string.

Parameters **string** (*str*) – Value that will be trimmed. Characters that will be removed include: ‘.’, ‘,’, ‘;’, ‘-’, ‘_’.

Returns Edited value.

Return type str

pdm_utils.functions.basic.**truncate_value**(*value*, *length*, *suffix*)
Truncate a string.

Parameters

- **value** (*str*) – String that should be truncated.
- **length** (*int*) – Final length of truncated string.
- **suffix** (*str*) – String that should be appended to truncated string.

Returns the truncated string

Return type str

pdm_utils.functions.basic.**verify_path**(*filepath*, *kind=None*)
Verifies that a given path exists.

Parameters

- **filepath** (*str*) – full path to the desired file/directory.
- **kind** (*str*) – (“file”, “dir”), corresponding with paths to be checked as either files or directories.

Return Boolean True if path is verified, False otherwise.

pdm_utils.functions.basic.**verify_path2**(*path*, *kind=None*, *expect=True*)
Verifies that a given path exists.

Parameters

- **path** (*Path*) – path
- **kind** (*str*) – ("file", "dir"), corresponding with paths to be checked as either files or directories.
- **expect** (*bool*) – Indicates if the path is expected to the indicated kind.

Returns tuple (result, message) WHERE result(bool) indicates if the expectation was satisfied. message(str) is a description of the result.

Return type tuple

eval_modes

Evaluation mode functions and dictionaries.

pdm_utils.functions.eval_modes.**get_eval_flag_dict**(*eval_mode*)
Get a dictionary of evaluation flags.

Parameters **eval_mode** (*str*) – Valid evaluation mode (base, draft, final, auto, misc, custom)

Returns Dictionary of boolean values.

Return type dict

flat_files

Functions to interact with, use, and parse genomic data from GenBank-formatted flat files.

pdm_utils.functions.flat_files.**cds_to_seqrecord**(*cds*, *parent_genome*, *gene_domains=[]*, *desc_type='gb'*)
Creates a SeqRecord object from a Cds and its parent Genome.

Parameters

- **cds** (*Cds*) – A populated Cds object.
- **phage_genome** – Populated parent Genome object of the Cds object.
- **domains** (*list*) – List of domain objects populated with column attributes
- **desc_type** (*str*) – Inteneded format of the CDS SeqRecord description.

Returns Filled Biopython SeqRecord object.

Return type SeqRecord

pdm_utils.functions.flat_files.**create_fasta_seqrecord**(*header*, *sequence_string*)
Create a fasta-formatted Biopython SeqRecord object.

Parameters

- **header** (*str*) – Description of the sequence.
- **sequence_string** (*str*) – Nucleotide sequence.

Returns Biopython SeqRecord containing the nucleotide sequence.

Return type SeqRecord

pdm_utils.functions.flat_files.**create_seqfeature_dictionary**(*seqfeature_list*)
Create a dictionary of Biopython SeqFeature objects based on their type.

From a list of all Biopython SeqFeatures derived from a GenBank-formatted flat file, create a dictionary of SeqFeatures based on their 'type' attribute.

Parameters

- **seqfeature_list** (*list*) – List of Biopython SeqFeatures
- **genome_id** (*str*) – An identifier for the genome in which the seqfeature is defined.

Returns A dictionary of Biopython SeqFeatures: Key: SeqFeature type (source, tRNA, CDS, other) Value: SeqFeature

Return type dict

pdm_utils.functions.flat_files.**format_cds_seqrecord_CDS_feature**(*cds_feature*, *cds*, *parent_genome*)

pdm_utils.functions.flat_files.**genome_to_seqrecord**(*phage_genome*)
Creates a SeqRecord object from a pdm_utils Genome object.

Parameters **phage_genome** (*Genome*) – A pdm_utils Genome object.

Returns A BioPython SeqRecord object

Return type SeqRecord

pdm_utils.functions.flat_files.**get_cds_seqrecord_annotations**(*cds*, *parent_genome*)
Function that creates a Cds SeqRecord annotations attribute dict. :param cds: A populated Cds object. :type cds: Cds :param phage_genome: Populated parent Genome object of the Cds object. :type phage_genome: Genome :returns: Formatted SeqRecord annotations dictionary. :rtype: dict{str}

pdm_utils.functions.flat_files.**get_cds_seqrecord_annotations_comments**(*cds*)
Function that creates a Cds SeqRecord comments attribute tuple.

Parameters **cds** –

pdm_utils.functions.flat_files.**get_cds_seqrecord_regions**(*gene_domains*, *cds*)

pdm_utils.functions.flat_files.**get_genome_seqrecord_annotations**(*phage_genome*)
Helper function that uses Genome data to populate the annotations SeqRecord attribute

Parameters **phage_genome** (*genome*) – Input a Genome object.

Returns annotations(dictionary) is a dictionary with the formatting of BioPython's SeqRecord annotations attribute

pdm_utils.functions.flat_files.**get_genome_seqrecord_annotations_comments**(*phage_genome*)
Helper function that uses Genome data to populate the comment annotation attribute

Parameters **phage_genome** (*genome*) – Input a Genome object.

Returns cluster_comment, auto_generated_comment annotation_status_comment, qc_and_retrieval values (tuple) is a tuple with the formatting of BioPython's SeqRecord annotations comment attribute

pdm_utils.functions.flat_files.**get_genome_seqrecord_description**(*phage_genome*)
Helper function to construct a description SeqRecord attribute.

Parameters **phage_genome** (*genome*) – Input a Genome object.

Returns description is a formatted string parsed from genome data

pdm_utils.functions.flat_files.**get_genome_seqrecord_features**(*phage_genome*)
Helper function that uses Genome data to populate the features SeqRecord atribute

Parameters **phage_genome** (*genome*) – Input a Genome object.

Returns features is a list of SeqFeature objects parsed from cds objects

pdm_utils.functions.flat_files.**parse_cds_seqfeature**(*seqfeature*)
Parse data from a Biopython CDS SeqFeature object into a Cds object.

Parameters

- **seqfeature** (*SeqFeature*) – Biopython SeqFeature
- **genome_id** (*str*) – An identifier for the genome in which the seqfeature is defined.

Returns A pdm_utils Cds object

Return type Cds

pdm_utils.functions.flat_files.**parse_coordinates**(*seqfeature*)
Parse the boundary coordinates from a GenBank-formatted flat file.

The functions takes a Biopython SeqFeature object containing data that was parsed from the feature in the flat file. Parsing these coordinates can be tricky. There can be more than one set of coordinates if it is a compound location. Only features with 1 or 2 open reading frames (parts) are correctly parsed. Also, the boundaries may not be precise; instead they may be open or fuzzy. Non-precise coordinates are converted to '-1'. If the strand is undefined, the coordinates are converted to '-1' and parts is set to '0'. If an incorrect data type is provided, coorindates are set to '-1' and parts is set to '0'.

Parameters **seqfeature** (*SeqFeature*) – Biopython SeqFeature

Returns tuple (start, stop, parts) WHERE start(int) is the first coordinate, regardless of strand. stop(int) is the second coordinate, regardless of strand. parts(int) is the number of open reading frames that define the feature.

pdm_utils.functions.flat_files.**parse_genome_data**(*seqrecord*, *filepath=PosixPath('.')*, *translation_table=11*, *genome_id_field='_organism_name'*, *gnm_type=''*, *host_genus_field='_organism_host_genus'*)
Parse data from a Biopython SeqRecord object into a Genome object.

All Source, CDS, tRNA, and tmRNA features are parsed into their associated Source, Cds, Trna, and Tmrna objects.

Parameters

- **seqrecord** (*SeqRecord*) – A Biopython SeqRecord object.
- **filepath** (*Path*) – A filename associated with the returned Genome object.
- **translation_table** (*int*) – The applicable translation table for the genome's CDS features.
- **genome_id_field** (*str*) – The SeqRecord attribute from which the unique genome identifier/name is stored.
- **host_genus_field** (*str*) – The SeqRecord attribute from which the unique host genus identifier/name is stored.
- **gnm_type** (*str*) – Identifier for the type of genome.

Returns A pdm_utils Genome object.

Return type Genome

pdm_utils.functions.flat_files.**parse_source_seqfeature**(*seqfeature*)
Parses a Biopython Source SeqFeature.

Parameters

- **seqfeature** (*SeqFeature*) – Biopython SeqFeature

- **genome_id** (*str*) – An identifier for the genome in which the seqfeature is defined.

Returns A pdm_utils Source object

Return type Source

pdm_utils.functions.flat_files.**parse_tmrna_seqfeature**(*seqfeature*)
Parses data from a BioPython tmRNA SeqFeature object into a Tmrna object. :param seqfeature: BioPython SeqFeature :type seqfeature: SeqFeature :return: pdm_utils Tmrna object :rtype: Tmrna

pdm_utils.functions.flat_files.**parse_trna_seqfeature**(*seqfeature*)
Parse data from a Biopython tRNA SeqFeature object into a Trna object. :param seqfeature: Biopython SeqFeature :type seqfeature: SeqFeature :returns: a pdm_utils Trna object :rtype: Trna

pdm_utils.functions.flat_files.**retrieve_genome_data**(*filepath*)
Retrieve data from a GenBank-formatted flat file.

Parameters **filepath** (*Path*) – Path to GenBank-formatted flat file that will be parsed using Biopython.

Returns If there is only one record, a Biopython SeqRecord of parsed data. If the file cannot be parsed, or if there are multiple records, None value is returned.

Return type SeqRecord

pdm_utils.functions.flat_files.**sort_seqrecord_features**(*seqrecord*)
Function that sorts and processes the seqfeature objects of a seqrecord.

Parameters **seqrecord** (*SeqRecord*) – Phage genome Biopython seqrecord object

mysqldb

Functions to interact with MySQL.

pdm_utils.functions.mysqldb.**change_version**(*engine*, *amount=1*)
Change the database version number.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object able to connect to a MySQL database.
- **amount** (*int*) – Amount to increment/decrement version number.

pdm_utils.functions.mysqldb.**check_schema_compatibility**(*engine*, *pipeline*, *code_version=None*)
Confirm database schema is compatible with code.

If schema version is not compatible, sys.exit is called.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object able to connect to a MySQL database.
- **pipeline** (*str*) – Description of the pipeline checking compatibility.
- **code_version** (*int*) – Schema version on which the pipeline operates. If no schema version is provided, the package-wide schema version value is used.

pdm_utils.functions.mysqldb.**create_delete**(*table*, *field*, *data*)
Create MySQL DELETE statement.

"'DELETE FROM <table> WHERE <field> = '<data>'."

Parameters

- **table** (*str*) – The database table to insert information.

- **field** (*str*) – The column upon which the statement is conditioned.
- **data** (*str*) – The value of 'field' upon which the statement is conditioned.

Returns A MySQL DELETE statement.

Return type str

pdm_utils.functions.mysqldb.**create_gene_table_insert**(*cds_ftr*)
Create a MySQL gene table INSERT statement.

Parameters **cds_ftr** (*Cds*) – A pdm_utils Cds object.

Returns A MySQL statement to INSERT a new row in the 'gene' table with data for several fields.

Return type str

pdm_utils.functions.mysqldb.**create_genome_statements**(*gnm*, *tkt_type=''*)
Create list of MySQL statements based on the ticket type.

Parameters

- **gnm** (*Genome*) – A pdm_utils Genome object.
- **tkt_type** (*str*) – 'add' or 'replace'.

Returns List of MySQL statements to INSERT all data from a genome into the database (DELETE FROM genome, INSERT INTO phage, INSERT INTO gene, ...).

Return type list

pdm_utils.functions.mysqldb.**create_phage_table_insert**(*gnm*)
Create a MySQL phage table INSERT statement.

Parameters **gnm** (*Genome*) – A pdm_utils Genome object.

Returns A MySQL statement to INSERT a new row in the 'phage' table with data for several fields.

Return type str

pdm_utils.functions.mysqldb.**create_seq_set**(*engine*)
Create set of genome sequences currently in a MySQL database.

Parameters **engine** (*Engine*) – SQLAlchemy Engine object able to connect to a MySQL database.

Returns A set of unique values from phage.Sequence.

Return type set

pdm_utils.functions.mysqldb.**create_tmrna_table_insert**(*tmrna_ftr*)

Parameters **tmrna_ftr** –

Returns

pdm_utils.functions.mysqldb.**create_trna_table_insert**(*trna_ftr*)
Create a MySQL trna table INSERT statement. :param trna_ftr: a pdm_utils Trna object :type trna_ftr: Trna :returns: a MySQL statement to INSERT a new row in the 'trna' table with all of trna_ftr's relevant data :rtype: str

pdm_utils.functions.mysqldb.**create_update**(*table*, *field2*, *value2*, *field1*, *value1*)
Create MySQL UPDATE statement.

"'UPDATE <table> SET <field2> = '<value2' WHERE <field1> = '<data1>'."

When the new value to be added is 'singleton' (e.g. for Cluster fields), or an empty value (e.g. None, "none", etc.), the new value is set to NULL.

Parameters

- **table** (*str*) – The database table to insert information.
- **field1** (*str*) – The column upon which the statement is conditioned.
- **value1** (*str*) – The value of 'field1' upon which the statement is conditioned.
- **field2** (*str*) – The column that will be updated.
- **value2** (*str*) – The value that will be inserted into 'field2'.

Returns A MySQL UPDATE statement.

Return type set

pdm_utils.functions.mysqldb.**execute_transaction**(*engine*, *statement_list=[]*)
Execute list of MySQL statements within a single defined transaction.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object able to connect to a MySQL databas.
- **statement_list** – a list of any number of MySQL statements with no expectation that anything will return

Returns tuple (result, message) WHERE result (int) is 0 or 1 status code. 0 means no problems, 1 means problems message(str) is a description of the result.

Return type tuple

pdm_utils.functions.mysqldb.**get_schema_version**(*engine*)
Identify the schema version of the database_versions_list.

Schema version data has not been persisted in every schema version, so if schema version data is not found, it is deduced from other parts of the schema.

Parameters **engine** (*Engine*) – SQLAlchemy Engine object able to connect to a MySQL database.

Returns The version of the pdm_utils database schema.

Return type int

pdm_utils.functions.mysqldb.**parse_feature_data**(*engine*, *ftr_type*, *column=None*, *phage_id_list=None*, *query=None*)
Returns Cds objects containing data parsed from a MySQL database.

Parameters

- **engine** (*Engine*) – This parameter is passed directly to the 'retrieve_data' function.
- **query** (*str*) – This parameter is passed directly to the 'retrieve_data' function.
- **ftr_type** (*str*) – Indicates the type of features retrieved.
- **column** (*str*) – This parameter is passed directly to the 'retrieve_data' function.
- **phage_id_list** (*list*) – This parameter is passed directly to the 'retrieve_data' function.

Returns A list of pdm_utils Cds objects.

Return type list

pdm_utils.functions.mysqldb.**parse_gene_table_data**(*data_dict*, *trans_table=11*)
Parse a MySQL database dictionary to create a Cds object.

Parameters

- **data_dict** (*dict*) – Dictionary of data retrieved from the gene table.

- **trans_table** (*int*) – The translation table that can be used to translate CDS features.

Returns A pdm_utils Cds object.

Return type Cds

pdm_utils.functions.mysqldb.**parse_genome_data**(*engine*, *phage_id_list=None*, *phage_query=None*, *gene_query=None*, *trna_query=None*, *tmrna_query=None*, *gnm_type=''*)

Returns a list of Genome objects containing data parsed from a MySQL database.

Parameters

- **engine** (*Engine*) – This parameter is passed directly to the 'retrieve_data' function.
- **phage_query** (*str*) – This parameter is passed directly to the 'retrieve_data' function to retrieve data from the phage table.
- **gene_query** (*str*) – This parameter is passed directly to the 'parse_feature_data' function to retrieve data from the gene table. If not None, pdm_utils Cds objects for all of the phage's CDS features in the gene table will be constructed and added to the Genome object.
- **trna_query** (*str*) – This parameter is passed directly to the 'parse_feature_data' function to retrieve data from the trna table. If not None, pdm_utils Trna objects for all of the phage's tRNA features in the trna table will be constructed and added to the Genome object.
- **tmrna_query** (*str*) – This parameter is passed directly to the 'parse_feature_data' function to retrieve data from the tmrna table. If not None, pdm_utils Tmrna objects for all of the phage's tmRNA features in the tmrna table will be constructed and added to the Genome object.
- **phage_id_list** (*list*) – This parameter is passed directly to the 'retrieve_data' function. If there is at at least one valid PhageID, a pdm_utils genome object will be constructed only for that phage. If None, or an empty list, genome objects for all phages in the database will be constructed.
- **gnm_type** (*str*) – Identifier for the type of genome.

Returns A list of pdm_utils Genome objects.

Return type list

pdm_utils.functions.mysqldb.**parse_phage_table_data**(*data_dict*, *trans_table=11*, *gnm_type=''*)

Parse a MySQL database dictionary to create a Genome object.

Parameters

- **data_dict** (*dict*) – Dictionary of data retrieved from the phage table.
- **trans_table** (*int*) – The translation table that can be used to translate CDS features.
- **gnm_type** (*str*) – Identifier for the type of genome.

Returns A pdm_utils genome object.

Return type genome

pdm_utils.functions.mysqldb.**parse_tmrna_table_data**(*data_dict*)

Parse a MySQL database dictionary to create a Tmrna object.

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from the gene table.

Returns A pdm_utils Tmrna object.

Return type Tmrna

pdm_utils.functions.mysqldb.**parse_trna_table_data**(*data_dict*)
Parse a MySQL database dictionary to create a Trna object.

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from the gene table.

Returns A pdm_utils Trna object.

Return type Trna

ncbi

Misc. functions to interact with NCBI databases.

pdm_utils.functions.ncbi.**get_accessions_to_retrieve**(*summary_records*)
Extract accessions from summary records.

Parameters **summary_records** (*list*) – List of dictionaries, where each dictionary is a record summary.

Returns List of accessions.

Return type list

pdm_utils.functions.ncbi.**get_data_handle**(*accession_list*, *db='nucleotide'*, *rettype='gb'*, *retmode='text'*)

pdm_utils.functions.ncbi.**get_records**(*accession_list*, *db='nucleotide'*, *rettype='gb'*, *retmode='text'*)
Retrieve records from NCBI from a list of active accessions.

Uses NCBI efetch implemented through BioPython Entrez.

Parameters

- **accession_list** (*list*) – List of NCBI accessions.
- **db** (*str*) – Name of the database to get summaries from (e.g. 'nucleotide').
- **rettype** (*str*) – Type of record to retrieve (e.g. 'gb').
- **retmode** (*str*) – Format of data to retrieve (e.g. 'text').

Returns List of BioPython SeqRecords generated from GenBank records.

Return type list

pdm_utils.functions.ncbi.**get_summaries**(*db=''*, *query_key=''*, *webenv=''*)
Retrieve record summaries from NCBI for a list of accessions.

Uses NCBI esummary implemented through BioPython Entrez.

Parameters

- **db** (*str*) – Name of the database to get summaries from.
- **query_key** (*str*) – Identifier for the search. This can be directly generated from run_esearch().
- **webenv** (*str*) – Identifier that can be directly generated from run_esearch()

Returns List of dictionaries, where each dictionary is a record summary.

Return type list

pdm_utils.functions.ncbi.**get_verified_data_handle**(*acc_id_dict*, *ncbi_cred_dict={}*, *batch_size=200*, *file_type='gb'*)
Retrieve genomes from GenBank.

output_folder = Path to where files will be saved. acc_id_dict = Dictionary where key = Accession and value = List[PhageIDs]

pdm_utils.functions.ncbi.**run_esearch**(*db=''*, *term=''*, *usehistory=''*)
Search for valid records in NCBI.

Uses NCBI esearch implemented through BioPython Entrez.

Parameters

- **db** (*str*) – Name of the database to search.
- **term** (*str*) – Search term.
- **usehistory** (*str*) – Indicates if prior searches should be used.

Returns Results of the search for each valid record.

Return type dict

pdm_utils.functions.ncbi.**set_entrez_credentials**(*tool=None*, *email=None*, *api_key=None*)
Set BioPython Entrez credentials to improve speed and reliability.

Parameters

- **tool** (*str*) – Name of the software/tool being used.
- **email** (*str*) – Email contact information for NCBI.
- **api_key** (*str*) – Unique NCBI-issued identifier to enhance retrieval speed.

parallelize

Functions to parallelize of processing of a list of inputs. Adapted from https://docs.python.org/3/library/multiprocessing.html

pdm_utils.functions.parallelize.**count_processors**(*inputs*, *num_processors*)
Programmatically determines whether the specified num_processors is appropriate. There's no need to use more processors than there are inputs, and it's impossible to use fewer than 1 processor or more than exist on the machine running the code. :param inputs: list of inputs :param num_processors: specified number of processors :return: num_processors (optimized)

pdm_utils.functions.parallelize.**parallelize**(*inputs*, *num_processors*, *task*, *verbose=True*)
Parallelizes some task on an input list across the specified number of processors :param inputs: list of inputs :param num_processors: number of processor cores to use :param task: name of the function to run :param verbose: updating progress bar output? :return: results

pdm_utils.functions.parallelize.**start_processes**(*inputs*, *num_processors*, *verbose*)
Creates input and output queues, and runs the jobs :param inputs: jobs to run :param num_processors: optimized number of processors :param verbose: updating progress bar output? :return: results

pdm_utils.functions.parallelize.**worker**(*input_queue*, *output_queue*)

parsing

pdm_utils.functions.parsing.**check_operator**(*operator*, *column_object*)
Validates an operator's application on a MySQL column.

Parameters

- **operator** (*str*) – Accepted MySQL operator.
- **column_object** (*Column*) – A SQLAlchemy Column object.

pdm_utils.functions.parsing.**create_filter_key**(*unparsed_filter*)
Creates a standardized filter string from a valid unparsed_filter.

Parameters **unparsed_filter** – Formatted MySQL WHERE clause.

Returns Standardized MySQL conditional string.

Return type str

pdm_utils.functions.parsing.**parse_cmd_list**(*unparsed_string_list*)
Recognizes and parses MySQL WHERE clause structures from cmd lists.

Parameters **unparsed_string_list** (*list[str]*) – Formatted MySQL WHERE clause arguments.

Returns 2-D array containing lists of statements joined by ORs.

Return type list[list]

pdm_utils.functions.parsing.**parse_cmd_string**(*unparsed_cmd_string*)
Recognizes and parses MySQL WHERE clause structures.

Parameters **unparsed_cmd_string** (*str*) – Formatted MySQL WHERE clause string.

Returns 2-D array containing lists of statements joined by ORs.

Return type list[list]

pdm_utils.functions.parsing.**parse_column**(*unparsed_column*)
Recognizes and parses a MySQL structured column.

Parameters **unparsed_column** (*str*) – Formatted MySQL column.

Returns List containing segments of a MySQL column.

Return type list[str]

pdm_utils.functions.parsing.**parse_filter**(*unparsed_filter*)
Recognizes and parses a MySQL structured WHERE clause.

Parameters **unparsed_filter** – Formatted MySQL WHERE clause.

Returns List containing segments of a MySQL WHERE clause.

Return type list[str]

pdm_utils.functions.parsing.**parse_in_spaces**(*unparsed_string_list*)
Convert a list of strings to a single space separated string.

Parameters **unparsed_string_list** (*list[str]*) – String list to be concatenated

Returns String with parsed in whitespace.

Return type str

pdm_utils.functions.parsing.**parse_out_ends**(*unparsed_string*)
Parse and remove beginning and end whitespace of a string.

Parameters **unparsed_string** (*str*) – String with variable terminal whitespaces.

Returns String with parsed and removed beginning and ending whitespace.

Return type str

pdm_utils.functions.parsing.**parse_out_spaces**(*unparsed_string*)
Parse and remove beginning and internal white space of a string.

Parameters **unparsed_string** (*str*) – String with variable terminal whitespaces.

Returns String with parsed and removed beginning and ending whitespace.

Return type str

pdm_utils.functions.parsing.**translate_column**(*metadata*, *raw_column*)
Converts a case-insensitve {table}.{column} str to a case-sensitive str.

Parameters

- **metadata** (*MetaData*) – Reflected SQLAlchemy MetaData object.
- **raw_column** (*str*) – Case-insensitive {table}.{column}.

Returns Case-sensitive column name.

Return type str

pdm_utils.functions.parsing.**translate_table**(*metadata*, *raw_table*)
Converts a case-insensitive table name to a case-sensitive str.

Parameters

- **metadata** (*MetaData*) – Reflected SQLAlchemy MetaData object.
- **raw_table** – Case-insensitive table name.

Type_table str

Returns Case-sensitive table name.

Return type str

phagesdb

Functions to interact with PhagesDB

pdm_utils.functions.phagesdb.**construct_phage_url**(*phage_name*)
Create URL to retrieve phage-specific data from PhagesDB.

Parameters **phage_name** (*str*) – Name of the phage of interest.

Returns URL pertaining to the phage.

Return type str

pdm_utils.functions.phagesdb.**create_cluster_subcluster_sets**(*url='https://phagesdb.org/api/clusters/'*)
Create sets of clusters and subclusters currently in PhagesDB.

Parameters **url** (*str*) – A URL from which to retrieve cluster and subcluster data.

Returns tuple (cluster_set, subcluster_set) WHERE cluster_set(set) is a set of all unique clusters on PhagesDB. subcluster_set(set) is a set of all unique subclusters on PhagesDB.

Return type tuple

pdm_utils.functions.phagesdb.**create_host_genus_set**(*url='https://phagesdb.org/api/host_genera/'*)
Create a set of host genera currently in PhagesDB.

Parameters **url** (*str*) – A URL from which to retrieve host genus data.

Returns All unique host genera listed on PhagesDB.

Return type set

pdm_utils.functions.phagesdb.**get_genome**(*phage_id*, *gnm_type=''*, *seq=False*)
Get genome data from PhagesDB.

Parameters

- **phage_id** (*str*) – The name of the phage to be retrieved from PhagesDB.
- **gnm_type** (*str*) – Identifier for the type of genome.
- **seq** (*bool*) – Indicates whether the genome sequence should be retrieved.

Returns A pdm_utils Genome object with the parsed data. If not genome is retrieved, None is returned.

Return type Genome

pdm_utils.functions.phagesdb.**get_phagesdb_data**(*url*)
Retrieve all sequenced genome data from PhagesDB.

Parameters **url** (*str*) – URL to connect to PhagesDB API.

Returns List of dictionaries, where each dictionary contains data for each phage. If a problem is encountered during retrieval, an empty list is returned.

Return type list

pdm_utils.functions.phagesdb.**get_unphamerated_phage_list**(*url*)
Retreive list of unphamerated phages from PhagesDB.

Parameters **url** (*str*) – A URL from which to retrieve a list of PhagesDB genomes that are not in the most up-to-date instance of the Actino_Draft MySQL database.

Returns List of PhageIDs.

Return type list

pdm_utils.functions.phagesdb.**parse_accession**(*data_dict*)
Retrieve Accession from PhagesDB.

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from PhagesDB.

Returns Accession of the phage.

Return type str

pdm_utils.functions.phagesdb.**parse_cluster**(*data_dict*)
Retrieve Cluster from PhagesDB.

If the phage is clustered, 'pcluster' is a dictionary, and one key is the Cluster data (Cluster or 'Singleton'). If for some reason no Cluster info is added at the time the genome is added to PhagesDB, 'pcluster' may automatically be set to NULL, which gets converted to "Unclustered" during retrieval. In the MySQL database NULL means Singleton, and the long form "Unclustered" is invalid due to its character length, so this value is converted to 'UNK' ('Unknown').

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from PhagesDB.

Returns Cluster of the phage.

Return type str

pdm_utils.functions.phagesdb.**parse_fasta_data**(*fasta_data*)
Parses data returned from a fasta-formatted file.

Parameters **fasta_data** (*str*) – Data from a fasta file.

Returns tuple (header, sequence) WHERE header(str) is the first line parsed from the parsed file. sequence(str) is the nucleotide sequence parsed from the file.

Return type tuple

pdm_utils.functions.phagesdb.**parse_fasta_filename**(*data_dict*)
Retrieve fasta filename from PhagesDB.

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from PhagesDB.

Returns Name of the fasta file for the phage.

Return type str

pdm_utils.functions.phagesdb.**parse_genome_data**(*data_dict*, *gnm_type=''*, *seq=False*)
Parses a dictionary of PhagesDB genome data into a pdm_utils Genome object.

Parameters

- **data_dict** (*dict*) – Dictionary of data retrieved from PhagesDB.
- **gnm_type** (*str*) – Identifier for the type of genome.
- **seq** (*bool*) – Indicates whether the genome sequence should be retrieved.

Returns A pdm_utils Genome object with the parsed data.

Return type Genome

pdm_utils.functions.phagesdb.**parse_genomes_dict**(*data_dict*, *gnm_type=''*, *seq=False*)
Returns a dictionary of pdm_utils Genome objects

Parameters

- **data_dict** (*dict*) – Dictionary of dictionaries. Key = PhageID. Value = Dictionary of genome data retrieved from PhagesDB.
- **gnm_type** (*str*) – Identifier for the type of genome.
- **seq** (*bool*) – Indicates whether the genome sequence should be retrieved.

Returns Dictionary of pdm_utils Genome object. Key = PhageID. Value = Genome object.

Return type dict

pdm_utils.functions.phagesdb.**parse_host_genus**(*data_dict*)
Retrieve host_genus from PhagesDB.

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from PhagesDB.

Returns Host genus of the phage.

Return type str

pdm_utils.functions.phagesdb.**parse_phage_name**(*data_dict*)
Retrieve Phage Name from PhagesDB.

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from PhagesDB.

Returns Name of the phage.

Return type str

pdm_utils.functions.phagesdb.**parse_subcluster**(*data_dict*)
Retrieve Subcluster from PhagesDB.

If for some reason no cluster info is added at the time the genome is added to PhagesDB, 'psubcluster' may automatically be set to NULL, which gets returned as None. If the phage is a Singleton, 'psubcluster' is None. If the phage is clustered but not subclustered, 'psubcluster' is None. If the phage is clustered and subclustered, 'psubcluster' is a dictionary, and one key is the Subcluster data.

Parameters **data_dict** (*dict*) – Dictionary of data retrieved from PhagesDB.

Returns Subcluster of the phage.

Return type str

pdm_utils.functions.phagesdb.**retrieve_data_list**(*url*)
Retrieve list of data from PhagesDB.

Parameters **url** (*str*) – A URL from which to retrieve data.

Returns A list of data retrieved from the URL.

Return type list

pdm_utils.functions.phagesdb.**retrieve_genome_data**(*phage_url*)
Retrieve all data from PhagesDB for a specific phage.

Parameters **phage_url** (*str*) – URL for data pertaining to a specific phage.

Returns Dictionary of data parsed from the URL.

Return type dict

pdm_utils.functions.phagesdb.**retrieve_url_data**(*url*)
Retrieve fasta file from PhagesDB.

Parameters **url** (*str*) – URL for data to be retrieved.

Returns Data from the URL.

Return type str

phameration

Functions that are used in the phameration pipeline

pdm_utils.functions.phameration.**blastp**(*index*, *chunk*, *tmp*, *db_path*, *evalue*, *query_cov*)
Runs 'blastp' using the given chunk as the input gene set. The blast output is an adjacency matrix for this chunk. :param index: chunk index being run :type index: int :param chunk: the translations to run right now :type chunk: tuple of 2-tuples :param tmp: path where I/O can go on :type tmp: str :param db_path: path to the target blast database :type db_path: str :param evalue: e-value cutoff to report hits :type evalue: float

pdm_utils.functions.phameration.**chunk_translations**(*translation_groups*, *chunksize=500*)
Break translation_groups into a dictionary of chunksize-tuples of 2-tuples where each 2-tuple is a translation and its corresponding geneid. :param translation_groups: translations and their geneids :type translation_groups: dict :param chunksize: how many translations will be in a chunk? :type chunksize: int :return: chunks :rtype: dict

pdm_utils.functions.phameration.**create_blastdb**(*fasta*, *db_name*, *db_path*)
Runs 'makeblastdb' to create a BLAST-searchable database. :param fasta: FASTA-formatted input file :type fasta: str :param db_name: BLAST sequence database :type db_name: str :param db_path: BLAST sequence database path :type db_path: str

pdm_utils.functions.phameration.**fix_colored_orphams**(*engine*)
Find any single-member phams which are colored as though they are multi-member phams (not #FFFFFF in pham.Color). :param engine: sqlalchemy Engine allowing access to the database :return:

pdm_utils.functions.phameration.**fix_white_phams**(*engine*)
Find any phams with 2+ members which are colored as though they are orphams (#FFFFFF in pham.Color). :param engine: sqlalchemy Engine allowing access to the database :return:

pdm_utils.functions.phameration.**get_geneids_and_translations**(*engine*)
Constructs a dictionary mapping all geneids to their translations. :param engine: the Engine allowing access to the database :return: gs_to_ts

pdm_utils.functions.phameration.**get_new_geneids**(*engine*)
Queries the database for those genes that are not yet phamerated. :param engine: the Engine allowing access to the database :return: new_geneids

pdm_utils.functions.phameration.**get_pham_colors**(*engine*)
Queries the database for the colors of existing phams :param engine: the Engine allowing access to the database :return: pham_colors

pdm_utils.functions.phameration.**get_pham_geneids**(*engine*)
Queries the database for those genes that are already phamerated. :param engine: the Engine allowing access to the database :return: pham_geneids

pdm_utils.functions.phameration.**get_translation_groups**(*engine*)
Constructs a dictionary mapping all unique translations to their groups of geneids that share them :param engine: the Engine allowing access to the database :return: ts_to_gs

pdm_utils.functions.phameration.**markov_cluster**(*adj_mat_file*, *inflation*, *tmp_dir*)
Run 'mcl' on an adjacency matrix to cluster the blastp results. :param adj_mat_file: 3-column file with blastp resultant queries, subjects, and evalues :type adj_mat_file: str :param inflation: mcl inflation parameter :type inflation: float :param tmp_dir: file I/O directory :type tmp_dir: str :return: outfile :rtype: str

pdm_utils.functions.phameration.**merge_pre_and_hmm_phams**(*hmm_phams*, *pre_phams*, *consensus_lookup*)
Merges the pre-pham sequences (which contain all nr sequences) with the hmm phams (which contain only hmm consensus sequences) into the full hmm-based clustering output. Uses consensus_lookup dictionary to find the pre-pham that each consensus belongs to, and then adds each pre-pham geneid to a full pham based on the hmm phams. :param hmm_phams: clustered consensus sequences :type hmm_phams: dict :param pre_phams: clustered sequences (used to generate hmms) :type pre_phams: dict :param consensus_lookup: reverse-mapped pre_phams :type consensus_lookup: dict :return: phams :rtype: dict

pdm_utils.functions.phameration.**mmseqs_clust**(*consensus_db*, *align_db*, *cluster_db*)
Runs 'mmseqs clust' to cluster an MMseqs2 consensus database using an MMseqs2 alignment database, with results being saved to an MMseqs2 cluster database. :param consensus_db: MMseqs sequence database :type consensus_db: str :param align_db: MMseqs2 alignment database :type align_db: str :param cluster_db: MMseqs2 cluster database :type cluster_db: str

pdm_utils.functions.phameration.**mmseqs_cluster**(*sequence_db*, *cluster_db*, *args*)
Runs 'mmseqs cluster' to cluster an MMseqs2 sequence database. :param sequence_db: MMseqs2 sequence database :type sequence_db: str :param cluster_db: MMseqs2 clustered database :type cluster_db: str :param args: parsed command line arguments :type args: dict

pdm_utils.functions.phameration.**mmseqs_createdb**(*fasta*, *sequence_db*)
Runs 'mmseqs createdb' to convert a FASTA file into an MMseqs2 sequence database. :param fasta: path to the FASTA file to convert :type fasta: str :param sequence_db: MMseqs2 sequence database :type sequence_db: str

pdm_utils.functions.phameration.**mmseqs_createseqfiledb**(*sequence_db*, *cluster_db*, *seqfile_db*)
Runs 'mmseqs createseqfiledb' to create the intermediate to the FASTA-like parseable output. :param sequence_db: MMseqs2 sequence database :type sequence_db: str :param cluster_db: MMseqs2 clustered

database :type cluster_db: str :param seqfile_db: MMseqs2 seqfile database :type seqfile_db: str

pdm_utils.functions.phameration.**mmseqs_profile2consensus**(*profile_db*, *consensus_db*)
Runs 'mmseqs profile2consensus' to extract consensus sequences from an MMseqs2 profile database, and creates an MMseqs2 sequence database from the consensuses. :param profile_db: MMseqs2 profile database :type profile_db: str :param consensus_db: MMseqs2 sequence database :type consensus_db: str

pdm_utils.functions.phameration.**mmseqs_result2flat**(*query_db*, *target_db*, *seqfile_db*, *outfile*)
Runs 'mmseqs result2flat' to create FASTA-like parseable output. :param query_db: MMseqs2 sequence or profile database :type query_db: str :param target_db: MMseqs2 sequence database :type target_db: str :param seqfile_db: MMseqs2 seqfile database :type seqfile_db: str :param outfile: FASTA-like parseable output :type outfile: str

pdm_utils.functions.phameration.**mmseqs_result2profile**(*sequence_db*, *cluster_db*, *profile_db*)
Runs 'mmseqs result2profile' to convert clusters from one MMseqs2 clustered database into a profile database. :param sequence_db: MMseqs2 sequence database :type sequence_db: str :param cluster_db: MMseqs2 clustered database :type cluster_db: str :param profile_db: MMseqs2 profile database :type profile_db: str

pdm_utils.functions.phameration.**mmseqs_search**(*profile_db*, *consensus_db*, *align_db*, *args*)
Runs 'mmseqs search' to search profiles against their consensus sequences and save the alignment results to an MMseqs2 alignment database. The profile_db and consensus_db MUST be the same size. :param profile_db: MMseqs2 profile database :type profile_db: str :param consensus_db: MMseqs2 sequence database :type consensus_db: str :param align_db: MMseqs2 alignment database :type align_db: str :param args: parsed command line arguments :type args: dict

pdm_utils.functions.phameration.**parse_mcl_output**(*outfile*)
Parse the mci output into phams :param outfile: mci output file :type outfile: str :return: phams :rtype: dict

pdm_utils.functions.phameration.**parse_mmseqs_output**(*outfile*)
Parses the indicated MMseqs2 FASTA-like file into a dictionary of integer-named phams. :param outfile: FASTA-like parseable output :type outfile: str :return: phams :rtype: dict

pdm_utils.functions.phameration.**preserve_phams**(*old_phams*, *new_phams*, *old_colors*, *new_genes*)
Attempts to keep pham numbers consistent from one round of pham building to the next :param old_phams: the dictionary that maps old phams to their genes :param new_phams: the dictionary that maps new phams to their genes :param old_colors: the dictionary that maps old phams to colors :param new_genes: the set of previously unphamerated genes :return:

pdm_utils.functions.phameration.**reintroduce_duplicates**(*new_phams*, *trans_groups*, *genes_and_trans*)
Reintroduces into each pham ALL GeneIDs that map onto the set of translations in the pham. :param new_phams: the pham dictionary for which duplicates are to be reintroduced :param trans_groups: the dictionary that maps translations to the GeneIDs that share them :param genes_and_trans: the dictionary that maps GeneIDs to their translations :return:

pdm_utils.functions.phameration.**update_gene_table**(*phams*, *engine*)
Updates the gene table with new pham data :param phams: new pham gene data :type phams: dict :param engine: sqlalchemy Engine allowing access to the database :return:

pdm_utils.functions.phameration.**update_pham_table**(*colors*, *engine*)
Populates the pham table with the new PhamIDs and their colors. :param colors: new pham color data :type colors: dict :param engine: sqlalchemy Engine allowing access to the database :return:

pdm_utils.functions.phameration.**write_fasta**(*translation_groups*, *outfile*)
Writes a FASTA file of the non-redundant protein sequences to be assorted into phamilies. :param translation_groups: groups of genes that share a translation :type translation_groups: dict :param outfile: FASTA filename :type outfile: str :return:

querying

pdm_utils.functions.querying.**append_group_by_clauses**(*executable*, *group_by_clauses*)
Add GROUP BY SQLAlchemy Column objects to a Select object.

Parameters

- **executable** (*Select*) – SQLAlchemy executable query object.
- **order_by_clauses** (*list*) – MySQL GROUP BY clause-related SQLAlchemy object(s)

Returns MySQL expression-related SQLAlchemy exectuable.

Return type Select

pdm_utils.functions.querying.**append_having_clauses**(*executable*, *having_clauses*)
Add HAVING SQLAlchemy Column objects to a Select object.

Parameters

- **executable** (*Select*) – SQLAlchemy executable query object.
- **having_clauses** – MySQL HAVING clause-related SQLAlchemy object(s).

:returns MySQL expression-related SQLAlchemy executable. :rtype: Select

pdm_utils.functions.querying.**append_order_by_clauses**(*executable*, *order_by_clauses*)
Add ORDER BY SQLAlchemy Column objects to a Select object.

Parameters

- **executable** (*Select*) – SQLAlchemy executable query object.
- **order_by_clauses** (*list*) – MySQL ORDER BY clause-related SQLAlchemy object(s)

Returns MySQL expression-related SQLAlchemy exectuable.

Return type Select

pdm_utils.functions.querying.**append_where_clauses**(*executable*, *where_clauses*)
Add WHERE SQLAlchemy BinaryExpression objects to a Select object.

Parameters

- **executable** (*Select*) – SQLAlchemy executable query object.
- **where_clauses** (*list*) – MySQL WHERE clause-related SQLAlchemy object(s).

Returns MySQL expression-related SQLAlchemy exectuable.

Return type Select

pdm_utils.functions.querying.**build_count**(*db_graph*, *columns*, *where=None*, *add_in=None*)
Get MySQL COUNT() expression SQLAlchemy executable.

Parameters

- **db_graph** (*Graph*) – SQLAlchemy structured NetworkX Graph object.
- **columns** (*list*) – SQLAlchemy Column object(s).
- **where** (*list*) – MySQL WHERE clause-related SQLAlchemy object(s).
- **add_in** (*list*) – MySQL Column-related inputs to be considered for joining.

Returns MySQL COUNT() expression-related SQLAlchemy executable.

Return type Select

pdm_utils.functions.querying.**build_distinct**(*db_graph*, *columns*, *where=None*, *order_by=None*, *add_in=None*)

Get MySQL DISTINCT expression SQLAlchemy executable.

Parameters

- **db_graph** (*Graph*) – SQLAlchemy structured NetworkX Graph object.
- **columns** (*list*) – SQLAlchemy Column object(s).
- **where** (*list*) – MySQL WHERE clause-related SQLAlchemy object(s).
- **order_by** (*list*) – MySQL ORDER BY clause-related SQLAlchemy object(s).
- **add_in** (*list*) – MySQL Column-related inputs to be considered for joining.

Returns MySQL DISTINCT expression-related SQLAlchemy executable.

Return type Select

pdm_utils.functions.querying.**build_fromclause**(*db_graph*, *columns*)

Get a joined table from pathing instructions for joining MySQL Tables. :param db_graph: SQLAlchemy structured NetworkX Graph object. :type db_graph: Graph :param columns: SQLAlchemy Column object(s). :type columns: Column :type columns: list :returns: SQLAlchemy Table object containing left outer-joined tables. :rtype: Table

pdm_utils.functions.querying.**build_graph**(*metadata*)

Get a NetworkX Graph object populated from a SQLAlchemy MetaData object.

Parameters **metadata** (*MetaData*) – Reflected SQLAlchemy MetaData object.

Returns Populated and structured NetworkX Graph object.

Return type Column

pdm_utils.functions.querying.**build_onclause**(*db_graph*, *source_table*, *adjacent_table*)

Creates a SQLAlchemy BinaryExpression object for a MySQL ON clause expression

Parameters

- **db_graph** (*Graph*) – SQLAlchemy structured NetworkX Graph object.
- **source_table** (*str*) – Case-insensitive MySQL table name.
- **adjacent_table** – Case-insensitive MySQL table name.

Returns MySQL foreign key related SQLAlchemy BinaryExpression object.

Return type BinaryExpression

pdm_utils.functions.querying.**build_select**(*db_graph*, *columns*, *where=None*, *order_by=None*, *add_in=None*, *having=None*, *group_by=None*)

Get MySQL SELECT expression SQLAlchemy executable.

Parameters

- **db_graph** (*Graph*) – SQLAlchemy structured NetworkX Graph object.
- **columns** (*list*) – SQLAlchemy Column object(s).
- **where** (*list*) – MySQL WHERE clause-related SQLAlchemy object(s).
- **order_by** (*list*) – MySQL ORDER BY clause-related SQLAlchemy object(s).
- **add_in** (*list*) – MySQL Column-related inputs to be considered for joining.

- **having** (*list*) – MySQL HAVING clause-related SQLAlchemy object(s).
- **group_by** (*list*) – MySQL GROUP BY clause-related SQLAlchemy object(s).

Returns MySQL SELECT expression-related SQLAlchemy executable.

Return type Select

pdm_utils.functions.querying.**build_where_clause**(*db_graph*, *filter_expression*)

Creates a SQLAlchemy BinaryExpression object from a MySQL WHERE clause expression.

Parameters

- **db_graph** (*Graph*) – SQLAlchemy structured NetworkX Graph object.
- **filter_expression** (*str*) – MySQL where clause expression.

Returns MySQL expression-related SQLAlchemy BinaryExpression object.

Return type BinaryExpression

pdm_utils.functions.querying.**execute**(*engine*, *executable*, *in_column=None*, *values=[]*, *limit=8000*, *return_dict=True*)

Use SQLAlchemy Engine to execute a MySQL query.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object used for executing queries.
- **executable** (*str*) – Input a executable MySQL query.
- **return_dict** (*Boolean*) – Toggle whether execute returns dict or tuple.

Returns Results from execution of given MySQL query.

Return type list[dict]

Return type list[tuple]

pdm_utils.functions.querying.**execute_value_subqueries**(*engine*, *executable*, *in_column*, *source_values*, *return_dict=True*, *limit=8000*)

Query with a conditional on a set of values using subqueries.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object used for executing queries.
- **executable** (*str*) – Input a executable MySQL query.
- **in_column** (*Column*) – SQLAlchemy Column object.
- **source_values** (*list[str]*) – Values from specified MySQL column.
- **return_dict** (*Boolean*) – Toggle whether to return data as a dictionary.
- **limit** (*int*) – SQLAlchemy IN clause query length limiter.

Returns List of grouped data for each value constraint.

Return type list

pdm_utils.functions.querying.**extract_column**(*column*, *check=None*)

Get a column from a supported SQLAlchemy Column-related object.

Parameters

- **column** (*UnaryExpression*) – SQLAlchemy Column-related object.

- **check** (*<type BinaryExpression>*) – SQLAlchemy Column-related object type.

Returns Corresponding SQLAlchemy Column object.

Return type Column

pdm_utils.functions.querying.**extract_columns**(*columns*, *check=None*)
Get a column from a supported SQLAlchemy Column-related object(s).

Parameters

- **column** (*UnaryExpression*) – SQLAlchemy Column-related object.
- **check** (*<type BinaryExpression>*) – SQLAlchemy Column-related object type.

Returns List of SQLAlchemy Column objects.

Return type list[Column]

pdm_utils.functions.querying.**first_column**(*engine*, *executable*, *in_column=None*, *values=[]*, *limit=8000*)
Use SQLAlchemy Engine to execute and return the first column of fields.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object used for executing queries.
- **executable** (*str*) – Input an executable MySQL query.

Returns A column for a set of MySQL values.

Return type list[str]

pdm_utils.functions.querying.**first_column_value_subqueries**(*engine*, *executable*, *in_column*, *source_values*, *limit=8000*)
Query with a conditional on a set of values using subqueries.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object used for executing queries.
- **executable** (*str*) – Input a executable MySQL query.
- **in_column** (*Column*) – SQLAlchemy Column object.
- **source_values** (*list[str]*) – Values from specified MySQL column.
- **return_dict** (*Boolean*) – Toggle whether to return data as a dictionary.
- **limit** (*int*) – SQLAlchemy IN clause query length limiter.

Returns Distinct values fetched from value constraints.

Return type list

pdm_utils.functions.querying.**get_column**(*metadata*, *column*)
Get a SQLAlchemy Column object, with a case-insensitive input. Input must be formatted {Table_name}.{Column_name}.

Parameters

- **metadata** (*MetaData*) – Reflected SQLAlchemy MetaData object.
- **table** (*str*) – Case-insensitive column name.

Returns Corresponding SQLAlchemy Column object.

Return type Column

pdm_utils.functions.querying.**get_table**(*metadata*, *table*)
Get a SQLAlchemy Table object, with a case-insensitive input.

Parameters

- **metadata** (*MetaData*) – Reflected SQLAlchemy MetaData object.
- **table** (*str*) – Case-insensitive table name.

Returns Corresponding SQLAlchemy Table object.

Return type Table

pdm_utils.functions.querying.**get_table_list**(*columns*)
Get a nonrepeating list SQLAlchemy Table objects from Column objects.

Parameters **columns** (*list*) – SQLAlchemy Column object(s).

Returns List of corresponding SQLAlchemy Table objects.

Return type list

pdm_utils.functions.querying.**get_table_pathing**(*db_graph*, *table_list*, *center_table=None*)
Get pathing instructions for joining MySQL Table objects.

Parameters

- **db_graph** (*Graph*) – SQLAlchemy structured NetworkX Graph object.
- **table_list** (*list[Table]*) – List of SQLAlchemy Table objects.
- **center_table** (*Table*) – SQLAlchemy Table object to begin traversals from.

Returns 2-D list containing the center table and pathing instructions.

Return type list

pdm_utils.functions.querying.**join_pathed_tables**(*db_graph*, *table_pathing*)
Get a joined table from pathing instructions for joining MySQL Tables.

Parameters

- **db_graph** (*Graph*) – SQLAlchemy structured NetworkX Graph object.
- **table_pathing** (*list*) – 2-D list containing a Table and pathing lists.

Returns SQLAlchemy Table object containing left outer-joined tables.

Return type Table

pdm_utils.functions.querying.**query**(*session*, *db_graph*, *table_map*, *where=None*)
Use SQLAlchemy session to retrieve ORM objects from a mapped object.

Parameters

- **session** (*Session*) – Bound and connected SQLAlchemy Session object.
- **table_map** – SQLAlchemy ORM map object.
- **where** (*list*) – MySQL WHERE clause-related SQLAlchemy object(s).
- **order_by** (*list*) – MySQL ORDER BY clause-related SQLAlchemy object(s).

Returns List of mapped object instances.

Return type list

server

Misc. functions to utilizes server.

pdm_utils.functions.server.**get_transport**(*host*)
Create paramiko Transport with the server name.

Parameters **host** (*str*) – Server to connect to.

Returns Paramiko Transport object. If the server is not available, None is returned.

Return type Transport

pdm_utils.functions.server.**set_log_file**(*filepath*)
Set the filepath used to stored the Paramiko output.

This is a soft requirement for compliance with Paramiko standards. If it is not set, paramiko throws an error.

Parameters **filepath** (*Path*) – Path to file to log Paramiko results.

pdm_utils.functions.server.**setup_sftp_conn**(*transport*, *user=None*, *pwd=None*, *attempts=1*)
Get credentials and setup connection to the server.

Parameters

- **transport** (*Transport*) – Paramiko Transport object directed towards a valid server.
- **attempts** (*int*) – Number of attempts to connect to the server.

Returns Paramiko SFTPClient connection. If no connection can be made, None is returned.

Return type SFTPClient

pdm_utils.functions.server.**upload_file**(*sftp*, *local_filepath*, *remote_filepath*)
Upload a file to the server.

Parameters

- **sftp** (*SFTPClient*) – Paramiko SFTPClient connection to a server.
- **local_filepath** (*str*) – Absoluate path to file to be uploaded.
- **remote_filepath** (*str*) – Absoluate path to server destination.

Returns Indicates whether upload was successful.

Return type bool

tickets

Misc. functions to manipulate tickets.

pdm_utils.functions.tickets.**construct_tickets**(*list_of_data_dict*, *eval_data_dict*, *description_field*, *required_keys*, *optional_keys*, *keywords*)
Construct pdm_utils ImportTickets from parsed data dictionaries.

Parameters

- **list_of_data_dict** (*list*) – List of import ticket data dictionaries.
- **eval_data_dict** (*dict*) – Dictionary of boolean evaluation flags.
- **description_field** (*str*) – Default value to set ticket.description_field attribute if not present in the data dictionary.
- **required_keys** (*set*) – Set of keys required to be in the data dictionary.

- **optional_keys** (*set*) – Set of optional keys that are not required to be in the data dictionary.
- **keywords** (*set*) – Set of valid keyword values that are handled differently than other values.

Returns List of pdm_utils ImportTicket objects.

Return type list

pdm_utils.functions.tickets.**get_genome**(*tkt*, *gnm_type=''*)
Construct a pdm_utils Genome object from a pdm_utils ImportTicket object.

Parameters

- **tkt** (*ImportTicket*) – A pdm_utils ImportTicket object.
- **gnm_type** (*str*) – Identifier for the type of genome.

Returns A pdm_utils Genome object.

Return type Genome

pdm_utils.functions.tickets.**identify_duplicates**(*list_of_tickets*, *null_set={}*)
Compare all tickets to each other to identify ticket conflicts.

Identifies if the same id, PhageID, and Accession is present in multiple tickets.

Parameters

- **list_of_tickets** (*list*) – A list of pdm_utils ImportTicket objects.
- **null_set** (*set*) – A set of values that may be expected to be duplicated, that should not throw errors.

Returns tuple (tkt_id_dupes, phage_id_dupes) WHERE tkt_id_dupes(set) is a set of duplicate ticket ids. phage_id_dupes(set) is a set of duplicate PhageIDs.

Return type tuple

pdm_utils.functions.tickets.**modify_import_data**(*data_dict*, *required_keys*, *optional_keys*, *keywords*)
Modifies ticket data to conform to requirements for an ImportTicket object.

Parameters

- **data_dict** (*dict*) – Dictionary of import ticket data.
- **required_keys** (*set*) – Set of keys required to be in the data dictionary.
- **optional_keys** (*set*) – Set of optional keys that are not required to be in the data dictionary.
- **keywords** (*set*) – Set of valid keyword values that are handled differently than other values.

Returns Indicates if the ticket is structured properly.

Return type bool

pdm_utils.functions.tickets.**parse_import_ticket_data**(*data_dict*)
Converts import ticket data to a ImportTicket object.

Parameters **data_dict** (*dict*) – A dictionary of data with the following keys:

0. Import action type
1. Primary PhageID
2. Host
3. Cluster
4. Subcluster

5. Status
6. Annotation Author (int)
7. Feature field
8. Accession
9. Retrieve Record (int)
10. Evaluation mode

Returns A pdm_utils ImportTicket object.

Return type ImportTicket

pdm_utils.functions.tickets.**set_dict_value**(*data_dict*, *key*, *first*, *second*)
Set the value for a specific key based on 'type' key-value.

Parameters

- **data_dict** (*dict*) – Dictionary of import ticket data.
- **key** (*str*) – Dictionary key to change value of.
- **first** (*str*) – Value to assign to 'key' if 'type' == 'add'.
- **second** (*str*) – Value to assign to 'key' if 'type' != 'add'.

pdm_utils.functions.tickets.**set_empty**(*data_dict*)
Convert None values to an empty string.

Parameters **data_dict** (*dict*) – Dictionary of import ticket data.

pdm_utils.functions.tickets.**set_keywords**(*data_dict*, *keywords*)
Convert specific values in a dictionary to lowercase.

Parameters

- **data_dict** (*dict*) – Dictionary of import ticket data.
- **keywords** (*set*) – Set of valid keyword values that are handled differently than other values.

pdm_utils.functions.tickets.**set_missing_keys**(*data_dict*, *expected_keys*)
Add a list of keys-values to a dictionary if it doesn't have those keys.

Parameters

- **data_dict** (*dict*) – Dictionary of import ticket data.
- **expected_keys** (*set*) – Set of keys expected to be in the dictionary.

6.2.4 pipelines

compare

convert

Pipeline to upgrade or downgrade the schema of a MySQL database.

pdm_utils.pipelines.convert_db.**convert_schema**(*engine*, *actual*, *dir*, *steps*, *verbose=False*)
Iterate through conversion steps and convert database schema.

pdm_utils.pipelines.convert_db.**get_conversion_direction**(*actual*, *target*)
Determine needed conversion direction and steps.

pdm_utils.pipelines.convert_db.**get_step_data**(*step_name*)
Get dictionary of conversion step data.

pdm_utils.pipelines.convert_db.**get_step_name**(*dir*, *step*)
Generates the name of the script conversion filename.

pdm_utils.pipelines.convert_db.**main**(*unparsed_args_list*)
Run main conversion pipeline.

pdm_utils.pipelines.convert_db.**parse_args**(*unparsed_args_list*)
Verify the correct arguments are selected for converting database.

pdm_utils.pipelines.convert_db.**print_summary**(*summary*)
Print summary of data that could be lost or inaccurate.

export

Pipeline for exporting database information into files.

pdm_utils.pipelines.export_db.**append_database_version**(*genome_seqrecord*, *version_data*)
Function that appends the database version to the SeqRecord comments.

Parameters

- **genome_seqrecord** – Filled SeqRecord object.
- **version_data** (*dict*) – Dictionary containing database version information.

pdm_utils.pipelines.export_db.**decode_results**(*results*, *columns*, *verbose=False*)
Function that decodes encoded results from SQLAlchemy generated data.

Parameters

- **results** (*list[dict]*) – List of data dictionaries from a SQLAlchemy results proxy.
- **columns** (*list[Column]*) – SQLAlchemy Column objects.

pdm_utils.pipelines.export_db.**execute_csv_export**(*db_filter*, *export_path*, *folder_path*, *columns*, *csv_name*, *data_cache=None*, *sort=[]*, *raw_bytes=False*, *verbose=False*, *dump=False*)
Executes csv export of a MySQL database table with select columns.

Parameters

- **db_filter** (*Filter*) – A connected and fully built Filter object.
- **export_path** (*Path*) – Path to a dir for file creation.
- **folder_path** (*Path*) – Path to a top-level dir.
- **table** (*str*) – MySQL table name.
- **conditionals** (*list[BinaryExpression]*) – MySQL WHERE clause-related SQLAlchemy objects.
- **sort** (*list[Column]*) – A list of SQLAlchemy Columns to sort by.
- **values** (*list[str]*) – List of values to fitler database results.
- **verbose** (*bool*) – A boolean value to toggle progress print statements.
- **dump** (*bool*) – A boolean value to toggle dump in current working dir.

pdm_utils.pipelines.export_db.**execute_export**(*alchemist*, *pipeline*, *folder_path=None*, *folder_name='20220119_export'*, *values=None*, *verbose=False*, *dump=False*, *force=False*, *table='phage'*, *filters=''*, *groups=[]*, *sort=[]*, *include_columns=[]*, *exclude_columns=[]*, *sequence_columns=False*, *raw_bytes=False*, *concatenate=False*, *db_name=None*, *phams_out=False*, *threads=1*)

Executes the entirety of the file export pipeline.

Parameters

- **alchemist** (*AlchemyHandler*) – A connected and fully built AlchemyHandler object.
- **pipeline** (*str*) – File type that dictates data processing.
- **folder_path** (*Path*) – Path to a valid dir for new dir creation.
- **folder_name** (*str*) – A name for the export folder.
- **force** (*bool*) – A boolean to toggle aggresive building of directories.
- **values** (*list[str]*) – List of values to filter database results.
- **verbose** (*bool*) – A boolean value to toggle progress print statements.
- **dump** (*bool*) – A boolean value to toggle dump in current working dir.
- **table** (*str*) – MySQL table name.
- **filters** (*str*) – A list of lists with filter values, grouped by ORs.
- **groups** (*list[str]*) – A list of supported MySQL column names to group by.
- **sort** (*list[str]*) – A list of supported MySQL column names to sort by.
- **include_columns** (*list[str]*) – A csv export column selection parameter.
- **exclude_columns** (*list[str]*) – A csv export column selection parameter.
- **sequence_columns** (*bool*) – A boolean to toggle inclusion of sequence data.
- **concatenate** – A boolean to toggle concaternation for SeqRecords.
- **threads** (*int*) – Number of processes/threads to spawn during the pipeline

pdm_utils.pipelines.export_db.**execute_ffx_export**(*alchemist*, *export_path*, *folder_path*, *values*, *file_format*, *db_version*, *table*, *concatenate=False*, *data_cache=None*, *verbose=False*, *dump=False*, *threads=1*, *export_name=None*)

Executes SeqRecord export of the compilation of data from a MySQL entry.

Parameters

- **alchemist** (*AlchemyHandler*) – A connected and fully build AlchemyHandler object.
- **export_path** (*Path*) – Path to a dir for file creation.
- **folder_path** (*Path*) – Path to a top-level dir.
- **file_format** (*str*) – Biopython supported file type.
- **db_version** (*dict*) – Dictionary containing database version information.
- **table** (*str*) – MySQL table name.
- **values** (*list[str]*) – List of values to fitler database results.

- **conditionals** (*list[BinaryExpression]*) – MySQL WHERE clause-related SQLAlchemy objects.
- **sort** (*list[Column]*) – A list of SQLAlchemy Columns to sort by.
- **concatenate** – A boolean to toggle concatenation of SeqRecords.
- **verbose** (*bool*) – A boolean value to toggle progress print statements.

pdm_utils.pipelines.export_db.**execute_sql_export**(*alchemist*, *export_path*, *folder_path*, *db_version*, *db_name=None*, *dump=False*, *force=False*, *phams_out=False*, *threads=1*, *verbose=False*)

pdm_utils.pipelines.export_db.**filter_csv_columns**(*alchemist*, *table*, *include_columns=[]*, *exclude_columns=[]*, *sequence_columns=False*)

Function that filters and constructs a list of Columns to select.

Parameters

- **alchemist** (*AlchemyHandler*) – A connected and fully built AlchemyHandler object.
- **table** (*str*) – MySQL table name.
- **include_columns** (*list[str]*) – A list of supported MySQL column names.
- **exclude_columns** (*list[str]*) – A list of supported MySQL column names.
- **sequence_columns** (*bool*) – A boolean to toggle inclusion of sequence data.

Returns A list of SQLAlchemy Column objects.

Return type list[Column]

pdm_utils.pipelines.export_db.**get_cds_seqrecords**(*alchemist*, *values*, *data_cache=None*, *nucleotide=False*, *verbose=False*, *file_format=None*)

pdm_utils.pipelines.export_db.**get_genome_seqrecords**(*alchemist*, *values*, *data_cache=None*, *verbose=False*)

pdm_utils.pipelines.export_db.**get_single_genome**(*alchemist*, *phageid*, *get_features=False*, *data_cache=None*)

pdm_utils.pipelines.export_db.**get_sort_columns**(*alchemist*, *sort_inputs*)

Function that converts input for sorting to SQLAlchemy Columns.

Parameters

- **alchemist** (*AlchemyHandler*) – A connected and fully build AlchemyHandler object.
- **sort_inputs** (*list[str]*) – A list of supported MySQL column names.

Returns A list of SQLAlchemy Column objects.

Return type list[Column]

pdm_utils.pipelines.export_db.**main**(*unparsed_args_list*)

Uses parsed args to run the entirety of the file export pipeline.

Parameters **unparsed_args_list** (*list[str]*) – Input a list of command line args.

pdm_utils.pipelines.export_db.**parse_export**(*unparsed_args_list*)

Parses export_db arguments and stores them with an argparse object.

Parameters **unparsed_args_list** (*list[str]*) – Input a list of command line args.

Returns ArgParse module parsed args.

pdm_utils.pipelines.export_db.**parse_feature_data**(*alchemist*, *values=[]*, *limit=8000*)
Returns Cds objects containing data parsed from a MySQL database.

Parameters

- **alchemist** (*AlchemyHandler*) – A connected and fully built AlchemyHandler object.
- **values** (*list[str]*) – List of GeneIDs upon which the query can be conditioned.

find_domains

pdm_utils.pipelines.find_domains.**clear_domain_data**(*engine*)
Delete all domain data stored in the database.

pdm_utils.pipelines.find_domains.**construct_domain_stmt**(*data_dict*)
Construct the SQL statement to insert data into the domain table.

pdm_utils.pipelines.find_domains.**construct_gene_domain_stmt**(*data_dict*, *gene_id*)
Construct the SQL statement to insert data into the gene_domain table.

pdm_utils.pipelines.find_domains.**construct_gene_update_stmt**(*gene_id*)
Construct the SQL statement to update data in the gene table.

pdm_utils.pipelines.find_domains.**construct_sql_txn**(*gene_id*, *rps_data_list*)
Map domain data back to gene_id and create SQL statements for one transaction.

rps_data_list is a list of dictionaries, where each dictionary reflects a significat rpsblast domain hit.

pdm_utils.pipelines.find_domains.**construct_sql_txns**(*cds_trans_dict*, *rpsblast_results*)
Construct the list of SQL transactions.

pdm_utils.pipelines.find_domains.**create_cds_translation_dict**(*cdd_genes*)
Create a dictionary of genes and translations.

Returns a dictionary, where: key = unique translation value = set of GeneIDs with that translation.

pdm_utils.pipelines.find_domains.**create_results_dict**(*search_results*)
Create a dictionary of search results

Input is a list of dictionaries, one dict per translation, where: keys = "Translation" and "Data", where key = "Translation" has value = translation, key = "Data"" has value = list of rpsblast results, where Each result element is a dictionary containing domain and gene_domain data.

Returns a dictionary, where: key = unique translation, value = list of dictionaries, each dictionary a unique rpsblast result

pdm_utils.pipelines.find_domains.**execute_statement**(*connection*, *statement*)

pdm_utils.pipelines.find_domains.**execute_transaction**(*connection*, *statement_list=[]*)

pdm_utils.pipelines.find_domains.**get_rpsblast_command**()
Determine rpsblast+ command based on operating system.

pdm_utils.pipelines.find_domains.**get_rpsblast_path**(*command*)
Determine rpsblast+ binary path.

pdm_utils.pipelines.find_domains.**insert_domain_data**(*engine*, *results*)
Attempt to insert domain data into the database.

pdm_utils.pipelines.find_domains.**learn_cdd_name**(*cdd_dir*)

pdm_utils.pipelines.find_domains.**log_gene_ids**(*cdd_genes*)
Record names of the genes processed for reference.

pdm_utils.pipelines.find_domains.**main**(*argument_list*)

Parameters **argument_list** –

Returns

pdm_utils.pipelines.find_domains.**make_tempdir**(*tmp_dir*)
Uses pdm_utils.functions.basic.expand_path to expand TMP_DIR; then checks whether tmpdir exists - if it doesn't, uses os.makedirs to make it recursively. :param tmp_dir: location where I/O should take place :return:

pdm_utils.pipelines.find_domains.**process_align**(*align*)
Process alignment data.

Returns description, domain_id, and name.

pdm_utils.pipelines.find_domains.**process_rps_output**(*filepath*, *evalue*)
Process rpsblast output and return list of dictionaries.

pdm_utils.pipelines.find_domains.**search_and_process**(*rpsblast*, *cdd_name*, *tmp_dir*, *evalue*, *translation_id*, *translation*)
Uses rpsblast to search indicated gene against the indicated CDD :param rpsblast: path to rpsblast binary :param cdd_name: CDD database path/name :param tmp_dir: path to directory where I/O will take place :param evalue: evalue cutoff for rpsblast :param translation_id: unique identifier for the translation sequence :param translation: protein sequence for gene to query :return: results

pdm_utils.pipelines.find_domains.**search_summary**(*rolled_back*)
Print search results.

pdm_utils.pipelines.find_domains.**search_translations**(*rpsblast*, *cdd_name*, *tmp_dir*, *evalue*, *threads*, *engine*, *unique_trans*, *cds_trans_dict*)
Search for conserved domains in a list of unique translations.

pdm_utils.pipelines.find_domains.**setup_argparser**()
Builds argparse.ArgumentParser for this script :return:

freeze

Pipeline to freeze a database.

pdm_utils.pipelines.freeze_db.**add_filters**(*filter_obj*, *filters*)
Add filters from command line to filter object.

pdm_utils.pipelines.freeze_db.**construct_count_query**(*table*, *primary_key*, *phage_id_set*)
Construct SQL query to determine count.

pdm_utils.pipelines.freeze_db.**construct_delete_stmt**(*table*, *primary_key*, *phage_id_set*)
Construct SQL query to determine count.

pdm_utils.pipelines.freeze_db.**construct_set_string**(*phage_id_set*)
Convert set of phage_ids to string formatted for MySQL.

e.g. set: {'Trixie', 'L5', 'D29'} returns: "('Trixie', 'L5', 'D29')""

pdm_utils.pipelines.freeze_db.**get_prefix**()
Allow user to select appropriate prefix for the new database.

pdm_utils.pipelines.freeze_db.**main**(*unparsed_args_list*)
Run main freeze database pipeline.

pdm_utils.pipelines.freeze_db.**parse_args**(*unparsed_args_list*)
Verify the correct arguments are selected.

get_data

Pipeline to gather new data to be imported into a MySQL database.

pdm_utils.pipelines.get_data.**check_record_date**(*record_list*, *accession_dict*)
Check whether the GenBank record is new.

pdm_utils.pipelines.get_data.**compare_data**(*gnm_pair*)
Compare data and create update tickets.

pdm_utils.pipelines.get_data.**compute_genbank_tallies**(*results*)
Tally results from GenBank retrieval.

pdm_utils.pipelines.get_data.**convert_tickets_to_dict**(*list_of_tickets*)
Convert list of tickets to list of dictionaries.

pdm_utils.pipelines.get_data.**create_accession_sets**(*genome_dict*)
Generate set of unique and non-unique accessions.

Input is a dictionary of pdm_utils genome objects.

pdm_utils.pipelines.get_data.**create_draft_ticket**(*name*)
Create ImportTicket for draft genome.

pdm_utils.pipelines.get_data.**create_genbank_ticket**(*gnm*)
Create ImportTicket for GenBank record.

pdm_utils.pipelines.get_data.**create_phagesdb_ticket**(*phage_id*)
Create ImportTicket for PhagesDB genome.

pdm_utils.pipelines.get_data.**create_results_dict**(*gnm*, *genbank_date*, *result*)
Create a dictionary of data summarizing NCBI retrieval status.

pdm_utils.pipelines.get_data.**create_ticket_table**(*tickets*, *output_folder*)
Save tickets associated with retrieved from GenBank files.

pdm_utils.pipelines.get_data.**create_update_ticket**(*field*, *value*, *key_value*)
Create update ticket.

pdm_utils.pipelines.get_data.**get_accessions_to_retrieve**(*summary_records*, *accession_dict*)
Review GenBank summary to determine which records are new.

pdm_utils.pipelines.get_data.**get_draft_data**(*output_path*, *phage_id_set*)
Run sub-pipeline to retrieve auto-annotated 'draft' genomes.

pdm_utils.pipelines.get_data.**get_final_data**(*output_folder*, *matched_genomes*)
Run sub-pipeline to retrieve 'final' genomes from PhagesDB.

pdm_utils.pipelines.get_data.**get_genbank_data**(*output_folder*, *genome_dict*, *ncbi_cred_dict={}*, *genbank_results=False*, *force=False*)
Run sub-pipeline to retrieve genomes from GenBank.

pdm_utils.pipelines.get_data.**get_matched_drafts**(*matched_genomes*)
Generate a list of matched 'draft' genomes.

pdm_utils.pipelines.get_data.**get_update_data**(*output_folder*, *matched_genomes*)
Run sub-pipeline to retrieve field updates from PhagesDB.

pdm_utils.pipelines.get_data.**main**(*unparsed_args_list*)
Run main retrieve_updates pipeline.

pdm_utils.pipelines.get_data.**match_genomes**(*dict1*, *dict2*)
Match MySQL database genome data to PhagesDB genome data.

Both dictionaries: Key = PhageID Value = pdm_utils genome object

pdm_utils.pipelines.get_data.**output_genbank_summary**(*output_folder*, *results*)
Save summary of GenBank retrieval results to file.

pdm_utils.pipelines.get_data.**parse_args**(*unparsed_args_list*)
Verify the correct arguments are selected for getting updates.

pdm_utils.pipelines.get_data.**print_genbank_tallies**(*tallies*)
Print results of GenBank retrieval.

pdm_utils.pipelines.get_data.**print_match_results**(*dict*)
Print results of genome matching.

pdm_utils.pipelines.get_data.**process_failed_retrieval**(*accession_list*, *accession_dict*)
Create list of dictionaries for records that could not be retrieved.

pdm_utils.pipelines.get_data.**retrieve_drafts**(*output_folder*, *phage_list*)
Retrieve auto-annotated 'draft' genomes from PECAAN.

pdm_utils.pipelines.get_data.**retrieve_records**(*accession_dict*, *ncbi_folder*, *batch_size=200*)
Retrieve GenBank records.

pdm_utils.pipelines.get_data.**save_and_tickets**(*record_list*, *accession_dict*, *output_folder*)
Save flat files retrieved from GenBank and create import tickets.

pdm_utils.pipelines.get_data.**save_genbank_file**(*seqrecord*, *accession*, *name*, *output_folder*)
Save retrieved record to file.

pdm_utils.pipelines.get_data.**save_pecaan_file**(*response*, *name*, *output_folder*)
Save data retrieved from PECAAN.

pdm_utils.pipelines.get_data.**save_phagesdb_file**(*data*, *gnm*, *output_folder*)
Save file retrieved from PhagesDB.

pdm_utils.pipelines.get_data.**set_phagesdb_gnm_date**(*gnm*)
Set the date of a PhagesDB genome object.

pdm_utils.pipelines.get_data.**set_phagesdb_gnm_file**(*gnm*)
Set the filename of a PhagesDB genome object.

pdm_utils.pipelines.get_data.**sort_by_accession**(*genome_dict*, *force=False*)
Sort genome objects based on their accession status.

Only retain data if genome is set to be automatically updated, there is a valid accession, and the accession is unique.

get_db

Pipeline to install a pdm_utils MySQL database.

pdm_utils.pipelines.get_db.**execute_get_file_db**(*alchemist*, *database*, *filename*, *config_file=None*, *schema_version=None*, *verbose=False*)

pdm_utils.pipelines.get_db.**execute_get_new_db**(*alchemist*, *database*, *schema_version*, *config_file=None*, *verbose=False*)

pdm_utils.pipelines.get_db.**execute_get_server_db**(*alchemist*, *database*, *url*, *folder_path=None*, *folder_name='20220119_get_db'*, *db_name=None*, *config_file=None*, *verbose=False*, *subdirectory=None*, *download_only=False*, *get_fastas=False*, *get_alns=False*, *force_pull=False*, *get_version=False*, *schema_version=None*)

pdm_utils.pipelines.get_db.**install_db**(*alchemist*, *database*, *db_filepath=None*, *config_file=None*, *schema_version=None*, *verbose=False*, *pipeline=False*)

Install database. If database already exists, it is first removed. :param database: Name of the database to be installed :type database: str :param db_filepath: Directory for installation :type db_filepath: Path

pdm_utils.pipelines.get_db.**main**(*unparsed_args_list*)

Run the get_db pipeline.

The database data can be retrieved from three places: The server, which needs to be downloaded to a new folder. A local file, in which no download and no new folder are needed. The empty schema stored within pdm_utils, in which no download, new folder, or local file are needed.

Parameters **unparsed_args_list** (*list*) – list of arguments to run the pipeline unparsed

pdm_utils.pipelines.get_db.**parse_args**(*unparsed_args_list*)

Verify the correct arguments are selected for getting a new database. :param unparsed_args_list: arguments in sys.argv format :type unparsed_args_list: list :returns: A parsed list of arguments :rtype: argparse.Namespace

pdm_utils.pipelines.get_db.**prepare_download**(*local_folder*, *url_folder*, *db_name*, *extension*, *verbose=False*)

Construct filepath and check if it already exists, then download. :param local_folder: Working directory where the database is downloaded :type local_folder: Path :param url_folder: Base url where db_files are located. :type url_folder: str :param db_name: Name of the database to be downloaded :type db_name: str :param extension: file extension for the database :type extension: str :returns: Path to the destination directory and the status of the download :rtype: Path, bool

get_gb_records

Pipeline to retrieve GenBank records using accessions stored in the MySQL database.

pdm_utils.pipelines.get_gb_records.**copy_gb_data**(*ncbi_handle*, *acc_id_dict*, *records_path*, *file_type*, *verbose=False*)

Save retrieved records to file.

pdm_utils.pipelines.get_gb_records.**execute_get_gb_records**(*alchemist*, *file_type*, *folder_path=None*, *folder_name='20220119_gb_records'*, *config=None*, *values=None*, *verbose=False*, *force=False*, *filters=''*, *groups=[]*)

Executes the entirety of the get_gb_records pipeline

Parameters

- **alchemist** (*AlchemyHandler*) – A connected and fully build AlchemyHandler object.
- **folder_path** (*Path*) – Path to a valid dir for new dir creation.
- **folder_name** (*str*) – A name for the export folder.
- **file_type** (*str*) – File type to be exported.
- **config** (*ConfigParser*) – ConfigParser object containing NCBI credentials.
- **force** (*bool*) – A boolean to toggle aggresive building of directories.

- **values** (*list[str]*) – List of values to filter database results.
- **verbose** (*bool*) – A boolean value to toggle progress print statemtns.
- **filters** – A List of lists with filter value,grouped by ORs.
- **groups** (*list[str]*) – A list of supported MySQL column names to goup by.

pdm_utils.pipelines.get_gb_records.**main**(*unparsed_args_list*)
Run main get_gb_records pipeline.

pdm_utils.pipelines.get_gb_records.**parse_args**(*unparsed_args_list*)
Parses export_db arguments and stores them with an argparse object.

Parameters **unparsed_args_list** (*list[str]*) – Input a list of command line args.

Returns ArgParse module parsed args.

import

Primary pipeline to process and evaluate data to be imported into the MySQL database.

pdm_utils.pipelines.import_genome.**check_bundle**(*bndl*, *ticket_ref=''*, *file_ref=''*, *retrieve_ref=''*, *retain_ref=''*)
Check a Bundle for errors.

Evaluate whether all genomes have been successfully grouped, and whether all genomes have been paired, as expected. Based on the ticket type, there are expected to be certain types of genomes and pairs of genomes in the bundle.

Parameters

- **bndl** – same as for run_checks().
- **ticket_ref** – same as for prepare_bundle().
- **file_ref** – same as for prepare_bundle().
- **retrieve_ref** – same as for prepare_bundle().
- **retain_ref** – same as for prepare_bundle().

pdm_utils.pipelines.import_genome.**check_cds**(*cds_ftr*, *eval_flags*, *description_field='product'*)
Check a Cds object for errors.

Parameters

- **cds_ftr** (*Cds*) – A pdm_utils Cds object.
- **eval_flags** (*dicts*) – Dictionary of boolean evaluation flags.
- **description_field** (*str*) – Description field to check against.

pdm_utils.pipelines.import_genome.**check_genome**(*gnm*, *tkt_type*, *eval_flags*, *phage_id_set={}*, *seq_set={}*, *host_genus_set={}*, *cluster_set={}*, *subcluster_set={}*, *accession_set={}*)
Check a Genome object parsed from file for errors.

Parameters

- **gnm** (*Genome*) – A pdm_utils Genome object.
- **tkt** (*Ticket*) – A pdm_utils Ticket object.
- **eval_flags** (*dicts*) – Dictionary of boolean evaluation flags.

- **phage_id_set** (*set*) – Set of PhageIDs to check against.
- **seq_set** (*set*) – Set of genome sequences to check against.
- **host_genus_set** (*set*) – Set of host genera to check against.
- **cluster_set** (*set*) – Set of clusters to check against.
- **subcluster_set** (*set*) – Set of subclusters to check against.
- **accession_set** (*set*) – Set of accessions to check against.

pdm_utils.pipelines.import_genome.**check_retain_genome**(*gnm*, *tkt_type*, *eval_flags*)
Check a Genome object currently in database for errors.

Parameters

- **gnm** (*Genome*) – A pdm_utils Genome object.
- **tkt_type** (*str*) – ImportTicket type
- **eval_flags** (*dicts*) – Dictionary of boolean evaluation flags.

pdm_utils.pipelines.import_genome.**check_source**(*src_ftr*, *eval_flags*, *host_genus=''*)
Check a Source object for errors.

Parameters

- **src_ftr** (*Source*) – A pdm_utils Source object.
- **eval_flags** (*dicts*) – Dictionary of boolean evaluation flags.
- **host_genus** (*str*) – Host genus to check against.

pdm_utils.pipelines.import_genome.**check_ticket**(*tkt*, *type_set={}*, *description_field_set={}*, *eval_mode_set={}*, *id_dupe_set={}*, *phage_id_dupe_set={}*, *retain_set={}*, *retrieve_set={}*, *add_set={}*, *parse_set={}*)
Evaluate a ticket to confirm it is structured appropriately.

The assumptions for how each field is populated varies depending on the type of ticket.

Parameters

- **tkt** – same as for set_cds_descriptions().
- **type_set** (*set*) – Set of ImportTicket types to check against.
- **description_field_set** (*set*) – Set of description fields to check against.
- **eval_mode_set** (*set*) – Set of evaluation modes to check against.
- **id_dupe_set** (*set*) – Set of duplicated ImportTicket ids to check against.
- **phage_id_dupe_set** (*set*) – Set of duplicated ImportTicket PhageIDs to check against.
- **retain_set** (*set*) – Set of retain values to check against.
- **retrieve_set** (*set*) – Set of retrieve values to check against.
- **add_set** (*set*) – Set of add values to check against.
- **parse_set** (*set*) – Set of parse values to check against.

pdm_utils.pipelines.import_genome.**check_tmrna**(*tmrna_ftr*, *eval_flags*)
Check a Tmrna object for errors.

Parameters

- **tmrna_ftr** (*Tmrna*) – A pdm_utils Cds object.
- **eval_flags** (*dicts*) – Dictionary of boolean evaluation flags.

pdm_utils.pipelines.import_genome.**check_trna**(*trna_ftr*, *eval_flags*)
Check a Trna object for errors.

Parameters

- **trna_ftr** (*Trna*) – A pdm_utils Trna object.
- **eval_flags** (*dicts*) – Dictionary of boolean evaluation flags.

pdm_utils.pipelines.import_genome.**compare_genomes**(*genome_pair*, *eval_flags*)
Compare two genomes to identify discrepancies.

Parameters

- **genome_pair** (*GenomePair*) – A pdm_utils GenomePair object.
- **eval_flags** (*dicts*) – Dictionary of boolean evaluation flags.

pdm_utils.pipelines.import_genome.**data_io**(*engine=None*, *genome_folder=PosixPath('.')*, *import_table_file=PosixPath('.')*, *genome_id_field=''*, *host_genus_field=''*, *prod_run=False*, *description_field=''*, *eval_mode=''*, *output_folder=PosixPath('.')*, *interactive=False*, *accept_warning=False*)
Set up output directories, log files, etc. for import.

Parameters

- **engine** (*Engine*) – SQLAlchemy Engine object able to connect to a MySQL database.
- **genome_folder** (*Path*) – Path to the folder of flat files.
- **import_table_file** (*Path*) – Path to the import table file.
- **genome_id_field** (*str*) – The SeqRecord attribute that stores the genome identifier/name.
- **host_genus_field** (*str*) – The SeqRecord attribute that stores the host genus identifier/name.
- **prod_run** (*bool*) – Indicates whether MySQL statements will be executed.
- **description_field** (*str*) – The SeqFeature attribute that stores the feature's description.
- **eval_mode** (*str*) – Name of the evaluation mode to evaluation genomes.
- **output_folder** (*Path*) – Path to the folder to store results.
- **interactive** (*bool*) – Indicates whether user is able to interact with genome evaluations at run time
- **accept_warning** (*bool*) – Toggles whether the import pipeline will accept warnings without interactivity.

pdm_utils.pipelines.import_genome.**get_logfile_path**(*bndl*, *paths_dict=None*, *filepath=None*, *file_ref=None*)
Choose the path to output the file-specific log.

Parameters

- **bndl** – same as for run_checks().
- **paths_dict** (*dict*) – Dictionary indicating paths to success and fail folders.
- **filepath** (*Path*) – Path to flat file.

- **file_ref** – same as for prepare_bundle().

Returns Path to log file to store flat-file-specific evaluations. If paths_dict is set to None, then None is returned instead of a path.

Return type Path

pdm_utils.pipelines.import_genome.**get_mysql_reference_sets**(*engine*)
Get multiple sets of data from the MySQL database for reference.

Parameters **engine** – same as for data_io().

Returns Dictionary of unique PhageIDs, clusters, subclusters, host genera, accessions, and sequences stored in the MySQL database.

Return type dict

pdm_utils.pipelines.import_genome.**get_phagesdb_reference_sets**()
Get multiple sets of data from PhagesDB for reference.

Returns Dictionary of unique clusters, subclusters, and host genera stored on PhagesDB.

Return type dict

pdm_utils.pipelines.import_genome.**get_result_string**(*object*, *attr_list*)
Construct string of values from several object attributes.

Parameters

- **object** (*misc*) – A object from which to retrieve values.
- **attr_list** (*list*) – List of strings indicating attributes to retrieve from the object.

Returns A concatenated string representing values from all attributes.

Return type str

pdm_utils.pipelines.import_genome.**import_into_db**(*bndl*, *engine=None*, *gnm_key=''*, *prod_run=False*)
Import data into the MySQL database.

Parameters

- **bndl** – same as for run_checks().
- **engine** – same as for data_io().
- **gnm_key** (*str*) – Identifier for the Genome object in the Bundle's genome dictionary.
- **prod_run** – same as for data_io().

pdm_utils.pipelines.import_genome.**log_and_print**(*msg*, *terminal=False*)
Print message to terminal in addition to logger if needed.

Parameters

- **msg** (*str*) – Message to print.
- **terminal** (*bool*) – Indicates if message should be printed to terminal.

pdm_utils.pipelines.import_genome.**log_evaluations**(*dict_of_dict_of_lists*, *logfile_path=None*)
Export evaluations to log.

Parameters

- **dict_of_dict_of_lists** (*dict*) – Dictionary of evaluation dictionaries. Key1 = Bundle ID. Value1 = dictionary for each object in the Bundle. Key2 = object type ('bundle', 'ticket', etc.) Value2 = List of evaluation objects.

- **logfile_path** (*Path*) – Path to the log file.

pdm_utils.pipelines.import_genome.**main**(*unparsed_args_list*)
Runs the complete import pipeline.

This is the only function of the pipeline that requires user input. All other functions can be implemented from other scripts.

Parameters **unparsed_args_list** (*list*) – List of strings representing command line arguments.

pdm_utils.pipelines.import_genome.**parse_args**(*unparsed_args_list*)
Verify the correct arguments are selected for import new genomes.

Parameters **unparsed_args_list** (*list*) – List of strings representing command line arguments.

Returns ArgumentParser Namespace object containing the parsed args.

Return type Namespace

pdm_utils.pipelines.import_genome.**prepare_bundle**(*filepath=PosixPath('.')*, *ticket_dict={}*, *engine=None*, *genome_id_field=''*, *host_genus_field=''*, *id=None*, *file_ref=''*, *ticket_ref=''*, *retrieve_ref=''*, *retain_ref=''*, *id_conversion_dict={}*, *interactive=False*)
Gather all genomic data needed to evaluate the flat file.

Parameters

- **filepath** (*Path*) – Name of a GenBank-formatted flat file.
- **ticket_dict** (*dict*) – A dictionary of pdm_utils ImportTicket objects.
- **engine** – same as for data_io().
- **genome_id_field** – same as for data_io().
- **host_genus_field** – same as for data_io().
- **id** (*int*) – Identifier to be assigned to the Bundle object.
- **file_ref** (*str*) – Identifier for Genome objects derived from flat files.
- **ticket_ref** (*str*) – Identifier for Genome objects derived from ImportTickets.
- **retrieve_ref** (*str*) – Identifier for Genome objects derived from PhagesDB.
- **retain_ref** (*str*) – Identifier for Genome objects derived from MySQL.
- **id_conversion_dict** (*dict*) – Dictionary of PhageID conversions.
- **interactive** – same as for data_io().

Returns A pdm_utils Bundle object containing all data required to evaluate a flat file.

Return type Bundle

pdm_utils.pipelines.import_genome.**prepare_tickets**(*import_table_file=PosixPath('.')*, *eval_data_dict=None*, *description_field=''*, *table_structure_dict={}*)
Prepare dictionary of pdm_utils ImportTickets.

Parameters

- **import_table_file** – same as for data_io().
- **description_field** – same as for data_io().
- **eval_data_dict** (*dict*) – Evaluation data dictionary Key1 = "eval_mode" Value1 = Name of the eval_mode Key2 = "eval_flag_dict" Value2 = Dictionary of evaluation flags.

- **table_structure_dict** (*dict*) – Dictionary describing structure of the import table.

Returns Dictionary of pdm_utils ImportTicket objects. If a problem was encountered parsing the import table, None is returned.

Return type dict

pdm_utils.pipelines.import_genome.**process_files_and_tickets**(*ticket_dict*, *files_in_folder*, *engine=None*, *prod_run=False*, *genome_id_field=''*, *host_genus_field=''*, *interactive=False*, *log_folder_paths_dict=None*, *accept_warning=False*)

Process GenBank-formatted flat files and import tickets.

Parameters

- **ticket_dict** (*dict*) – A dictionary WHERE key (str) = The ticket's phage_id value (Ticket) = The ticket
- **files_in_folder** (*list*) – A list of filepaths to be parsed.
- **engine** – same as for data_io().
- **prod_run** – same as for data_io().
- **genome_id_field** – same as for data_io().
- **host_genus_field** – same as for data_io().
- **interactive** – same as for data_io().
- **accept_warning** – same as for data_io().
- **log_folder_paths_dict** (*dict*) – Dictionary indicating paths to success and fail folders.

Returns tuple of five objects WHERE [0] success_ticket_list (list) is a list of successful ImportTickets. [1] failed_ticket_list (list) is a list of failed ImportTickets. [2] success_filepath_list (list) is a list of successfully parsed flat files. [3] failed_filepath_list (list) is a list of unsuccessfully parsed flat files. [4] evaluation_dict (dict): dictionary from each Bundle, containing dictionaries for each bundled object, containing lists of evaluation objects.

Return type tuple

pdm_utils.pipelines.import_genome.**review_bundled_objects**(*bndl*, *interactive=False*, *accept_warning=False*)

Review all evaluations of all bundled objects.

Iterate through all objects stored in the bundle. If there are warnings, review whether status should be changed.

Parameters

- **bndl** – same as for run_checks().
- **interactive** – same as for data_io().
- **accept_warnring** – same as for data_io().

pdm_utils.pipelines.import_genome.**review_cds_descriptions**(*feature_list*, *description_field*)

Iterate through all CDS features and review descriptions.

Parameters

- **feature_list** (*list*) – A list of pdm_utils Cds objects.

- **description_field** – same as for data_io().

Returns Name of the primary description_field after review.

Return type str

pdm_utils.pipelines.import_genome.**review_evaluation**(*evl*, *interactive=False*, *accept_warning=False*)
Review an evaluation object.

Parameters

- **evl** (*Evaluation*) – A pdm_utils Evaluation object.
- **interactive** – same as for data_io().
- **accept_warning** – same as for data_io().

Returns tuple (exit, message) WHERE exit (bool) indicates whether user selected to exit the review process. correct(bool) indicates whether the evalution status is accurate.

Return type tuple

pdm_utils.pipelines.import_genome.**review_evaluation_list**(*evaluation_list*, *interactive=False*, *accept_warning=False*)
Iterate through all evaluations and review 'warning' results.

Parameters

- **evaluation_list** (*list*) – List of pdm_utils Evaluation objects.
- **interactive** – same as for data_io().
- **accept_warning** – same as for data_io().

Returns Indicates whether user selected to exit the review process.

Return type bool

pdm_utils.pipelines.import_genome.**review_object_list**(*object_list*, *object_type*, *attr_list*, *interactive=False*, *accept_warning=False*)
Determine if evaluations are present and record results.

Parameters

- **object_list** (*list*) – List of pdm_utils objects containing evaluations.
- **object_type** (*str*) – Name of the pdm_utils object.
- **attr_list** (*list*) – List of attributes used to log data about the object instance.
- **interactive** – same as for data_io().
- **accept_warning** – same as for data_io().

pdm_utils.pipelines.import_genome.**run_checks**(*bndl*, *accession_set={}*, *phage_id_set={}*, *seq_set={}*, *host_genus_set={}*, *cluster_set={}*, *subcluster_set={}*, *file_ref=''*, *ticket_ref=''*, *retrieve_ref=''*, *retain_ref=''*)
Run checks on the different types of data in a Bundle object.

Parameters

- **bndl** (*Bundle*) – A pdm_utils Bundle object containing bundled data.
- **accession_set** (*set*) – Set of accessions to check against.
- **phage_id_set** (*set*) – Set of PhageIDs to check against.
- **seq_set** (*set*) – Set of nucleotide sequences to check against.

- **host_genus_set** (*set*) – Set of host genera to check against.
- **cluster_set** (*set*) – Set of Clusters to check against.
- **subcluster_set** (*set*) – Set of Subclusters to check against.
- **file_ref** – same as for prepare_bundle().
- **ticket_ref** – same as for prepare_bundle().
- **retrieve_ref** – same as for prepare_bundle().
- **retain_ref** – same as for prepare_bundle().

pdm_utils.pipelines.import_genome.**set_cds_descriptions**(*gnm*, *tkt*, *interactive=False*)
Set the primary CDS descriptions.

Parameters

- **gnm** (*Genome*) – A pdm_utils Genome object.
- **tkt** (*ImportTicket*) – A pdm_utils ImportTicket object.
- **interactive** – same as for data_io().

phamerate

Program to group related gene products into phamilies using either MMseqs2 for both similarity search and clustering, or blastp for similarity search and mcl for clustering.

pdm_utils.pipelines.phamerate.**main**(*argument_list*)

pdm_utils.pipelines.phamerate.**refresh_tempdir**(*tmpdir*)
Recursively deletes tmpdir if it exists, otherwise makes it :param tmpdir: directory to refresh :return:

pdm_utils.pipelines.phamerate.**setup_argparser**()
Builds argparse.ArgumentParser for this script :return:

push

Pipeline to push files to a server using SFTP.

pdm_utils.pipelines.push_db.**get_files**(*directory*, *file*, *ignore*)
Get the list of file(s) that need to be uploaded.

Parameters

- **directory** – (optional) directory containing files for upload
- **file** (*pathlib.Path*) – (optional) file to upload
- **ignore** (*set*) – file(s) to ignore during upload process

Type directory: pathlib.Path

Returns file_list

pdm_utils.pipelines.push_db.**main**(*unparsed_args*)
Driver function for the push pipeline.

Parameters **unparsed_args** (*list*) – the command-line arguments given to this pipeline's caller (likely pdm_utils.__main__)

pdm_utils.pipelines.push_db.**parse_args**(*unparsed_args*)
Verify the correct arguments are selected for uploading to the server.

pdm_utils.pipelines.push_db.**upload**(*sftp_client*, *destination*, *files*)
Try to upload the file(s).

Parameters

- **sftp_client** (*paramiko.SFTPClient*) – an open SFTPClient
- **destination** (*pathlib.Path*) – remote file directory to upload to
- **files** (*list of pathlib.Path*) – the file(s) to upload

Returns successes, failures

update

Pipeline to update specific fields in a MySQL database.

pdm_utils.pipelines.update_field.**main**(*unparsed_args*)
Runs the complete update pipeline.

pdm_utils.pipelines.update_field.**parse_args**(*unparsed_args_list*)
Verify the correct arguments are selected for getting updates.

pdm_utils.pipelines.update_field.**update_field**(*alchemist*, *update_ticket*)
Attempts to update a field using information from an update_ticket.

Parameters

- **alchemist** (*AlchemyHandler*) – A connected and fully build AlchemyHandler object.
- **update_ticket** (*dict*) – Dictionary with instructions to update a field.

6.3 Submodules

6.3.1 run

Use this script to run all pipelines within the pipelines folder. It verifies a valid pipeline is selected, then passes all command line arguments to the main pipeline module.

pdm_utils.run.**main**(*unparsed_args*)
Run a pdm_utils pipeline.

pdm_utils.run.**parse_args**(*unparsed_args*)
Use argparse to verify pipeline argument only.

Parameters **unparsed_args** (*list*) – raw command line args

6.4 Bio-centric object relational mapping

A subset of pdm_utils classes represents the 'back-end' bio-centric ORM that can be used to exchange data between different data sources and perform biology-related tasks with the data:

6.4.1 The bio-centric ORM

The `pdm_utils` bio-centric ORM is designed to process data that has been stored/retrieved in a MySQL database, GenBank-formatted flat files, PhagesDB, etc. Data is parsed from GenBank-formatted flat files using BioPython. Data is parsed from PhagesDB using the PhagesDB API. As a result, some `pdm_utils` objects contain attributes that directly map to each of these different data structures.

Refer to the brief introductory *library tutorial* to coding with the `pdm_utils` library.

Below is a map of how genome-level data is stored within these data structures:

pdm_utils object.attribute	MySQL table.column	GenBank flat file BioPython object.attribute	PhagesDB API dictionary
Genome.id	phage.PhageID	SeqRecord.id OR user-selected	dictionary[“phage_name”]
Genome.name	phage.Name	SeqRecord.name OR user-selected	dictionary[“phage_name”]
Genome.seq	phage.Sequence	SeqRecord.seq	dictionary[“fasta_file”]
Genome.length	phage.Length	SeqRecord.seq	dictionary[“fasta_file”]
Genome.gc	phage.GC	SeqRecord.seq	dictionary[“fasta_file”]
Genome.cluster	phage.Cluster	N/A	dictionary[“pcluster”][“cluster”]
Genome.subcluster	phage.Subcluster	N/A	dictionary[“psubcluster”][“subcluster”]
Genome.accession	phage.Accession	SeqRecord.annotations[“accessions”][0]	dictionary[“genbank_accession”]
Genome.host_genus	phage.HostGenus	SeqRecord.annotations[“source”] OR [“organism”] OR [“description”] OR user-selected	dictionary[“isolation_host”][“genus”]
Genome.date	phage.DateLastModified	SeqRecord.annotations[“date”]	N/A
Genome.annotation_status	phage.Status	N/A	N/A
Genome.retrieve_record	phage.RetrieveRecord	N/A	N/A
Genome.annotation_author	phage.AnnotationAuthor	N/A	N/A
Genome.description	N/A	SeqRecord.description	dictionary[“fasta_file”]
Genome.source	N/A	SeqRecord.annotations[“source”]	N/A
Genome.organism	N/A	SeqRecord.annotations[“organism”]	N/A
Genome.authors	N/A	Reference.authors (from SeqRecord.annotations[“references”])	N/A
Genome.filename	N/A	N/A	dictionary[“fasta_file”]

Below is a map of how CDS-level data is stored within these data structures:

pdm_utils object.attribute	MySQL table.column	GenBank flat file BioPython object.attribute	PhagesDB API dictionary
Cds.id	gene.GeneID	N/A	N/A
Cds.name	gene.Name	SeqFeature qualifiers	N/A
Cds.genome_id	gene.PhageID	N/A	N/A
Cds.start	gene.Start	SeqFeature.location.start OR end	N/A
Cds.stop	gene.Stop	SeqFeature.location.start OR end	N/A
Cds.coordinate_format	N/A	N/A	N/A
Cds.orientation	gene.Orientation	SeqFeature.strand	N/A
Cds.parts	N/A	SeqFeature.location.parts	N/A
Cds.seq	N/A	N/A	N/A
Cds.length	gene.Length	SeqFeature.location.start AND end	N/A
Cds.translation	gene.Translation	SeqFeature.qualifiers[“translation”][0]	N/A
Cds.translation_length	N/A	SeqFeature.qualifiers[“translation”][0]	N/A
Cds.translation_table	N/A	SeqFeature.qualifiers[“transl_table”][0]	N/A
Cds.locus_tag	gene.LocusTag	SeqFeature.qualifiers[“locus_tag”][0]	N/A
Cds.description	gene.Notes	SeqFeature.qualifiers[“product”] OR [“function”] OR [“note”]	N/A
Cds.gene	N/A	SeqFeature.qualifiers[“gene”][0]	N/A
Cds.product	N/A	SeqFeature.qualifiers[“product”][0]	N/A
Cds.function	N/A	SeqFeature.qualifiers[“function”][0]	N/A
Cds.note	N/A	SeqFeature.qualifiers[“note”][0]	N/A
Cds.seqfeature	N/A	SeqFeature	N/A
N/A	gene.DomainStatus	N/A	N/A

Below is a map of how tRNA-level data is stored within these data structures:

pdm_utils object.attribute	MySQL table.column	GenBank flat file BioPython object.attribute	PhagesDB API dictionary
Trna.id	trna.GeneID	N/A	N/A
Trna.name	trna.Name	SeqFeature qualifiers	N/A
Trna.genome_id	trna.PhageID	N/A	N/A
Trna.start	trna.Start	SeqFeature.location.start OR end	N/A
Trna.stop	trna.Stop	SeqFeature.location.start OR end	N/A
Trna.coordinate_format	N/A	N/A	N/A
Trna.orientation	trna.Orientation	SeqFeature.strand	N/A
Trna.parts	N/A	SeqFeature.location.parts	N/A
Trna.length	trna.Length	SeqFeature.location.start AND end	N/A
Trna.locus_tag	trna.LocusTag	SeqFeature.qualifiers[“locus_tag”][0]	N/A
Trna.note	trna.Note	SeqFeature.qualifiers[“note”][0]	N/A
Trna.seqfeature	N/A	SeqFeature	N/A
Trna.amino_acid	trna.AminoAcid	SeqFeature.qualifiers[“product”][0]	N/A
Trna.anticodon	trna.Anticodon	SeqFeature.qualifiers[“note”][0]	N/A
Trna.structure	trna.Structure	N/A	N/A
Trna.use	trna.Source	N/A	N/A
Trna.product	N/A	SeqFeature.qualifiers[“product”][0]	N/A
Trna.gene	N/A	SeqFeature.qualifiers[“gene”][0]	N/A

Below is a map of how tmRNA-level data is stored within these data structures:

pdm_utils object.attribute	MySQL table.column	GenBank flat file BioPython object.attribute	PhagesDB API dictionary
Tmrna.id	tmrna.GeneID	N/A	N/A
Tmrna.name	tmrna.Name	SeqFeature qualifiers	N/A
Tmrna.genome_id	tmrna.PhageID	N/A	N/A
Tmrna.start	tmrna.Start	SeqFeature.location.start OR end	N/A
Tmrna.stop	tmrna.Stop	SeqFeature.location.start OR end	N/A
Tmrna.coordinate_format	N/A	N/A	N/A
Tmrna.orientation	tmrna.Orientation	SeqFeature.strand	N/A
Tmrna.parts	N/A	SeqFeature.location.parts	N/A
Tmrna.length	tmrna.Length	SeqFeature.location.start AND end	N/A
Tmrna.locus_tag	tmrna.LocusTag	SeqFeature.qualifiers[“locus_tag”][0]	N/A
Tmrna.note	tmrna.Note	SeqFeature.qualifiers[“note”][0]	N/A
Tmrna.seqfeature	N/A	SeqFeature	N/A
Tmrna.gene	N/A	SeqFeature.qualifiers[“gene”][0]	N/A
Tmrna.peptide_tag	tmrna.PeptideTag	SeqFeature.qualifiers[“note”][0]	N/A

GenBank-formatted flat files contain a Source feature. Although this data is not stored within the MySQL database, it is parsed and evaluated for quality when the genome is imported into the database. Below is a map of how Source-level data is stored within these data structures:

pdm_utils object.attribute	MySQL table.column	GenBank flat file BioPython object.attribute	PhagesDB API dictionary
Source.id	N/A	N/A	N/A
Source.name	N/A	N/A	N/A
Source.seqfeature	N/A	SeqFeature	N/A
Source.start	N/A	SeqFeature.location.start OR end	N/A
Source.stop	N/A	SeqFeature.location.start OR end	N/A
Source.organism	N/A	SeqFeature.qualifiers[“organism”][0]	N/A
Source.host	N/A	SeqFeature.qualifiers[“host”][0]	N/A
Source.lab_host	N/A	SeqFeature.qualifiers[“lab_host”][0]	N/A

6.5 Tutorial

Refer to the brief introductory *library tutorial* to coding with the `pdm_utils` library.

CHAPTER

SEVEN

HOW TO CONTRIBUTE

If you find an issue or bug with the package, feel free to submit an issue on the GitHub page for the project.

If you would like to provide a bug fix or suggest an improvement, create a new fork or branch of the repository, add your improvements, and submit a pull request on GitHub.

CHAPTER

EIGHT

BIBLIOGRAPHY

Altschul et al, 1990. https://pubmed.ncbi.nlm.nih.gov/2231712/

Camacho et al., 2009. https://www.ncbi.nlm.nih.gov/pubmed/20003500.

Chan & Lowe, 2019. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6768409.

Cock et al., 2009. https://www.ncbi.nlm.nih.gov/pubmed/19304878.

Cresawn et al., 2011. https://www.ncbi.nlm.nih.gov/pubmed/21991981.

Hanauer et al., 2017. https://www.ncbi.nlm.nih.gov/pubmed/29208718.

Laslett & Canback, 2004. https://www.ncbi.nlm.nih.gov/pubmed/14704338.

Lu et al., 2020. https://www.ncbi.nlm.nih.gov/pubmed/31777944.

Nawrocki & Eddy, 2013. https://www.ncbi.nlm.nih.gov/pubmed/24008419.

Steinegger and Söding, 2017. https://www.ncbi.nlm.nih.gov/pubmed/29035372.

van Dongen & Abreu-Goodger, 2012. https://pubmed.ncbi.nlm.nih.gov/22144159/

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W